Graphics Hardware (2003)
M. Doggett, W. Heidrich, W. Mark, A. Schilling (Editors)

An Optimized Soft Shadow Volume Algorithm with Real-Time Performance

Ulf Assarsson,[1] Michael Dougherty,[2] Michael Mounier,[2] and Tomas Akenine-Möller[1]

[1] Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden
[2] Xbox Advanced Technology Group, Microsoft

Abstract

In this paper, we present several optimizations to our previously presented soft shadow volume algorithm. Our optimizations include tighter wedges, heavily optimized pixel shader code for both rectangular and spherical light sources, a frame buffer blending technique to overcome the limitation of 8-bit frame buffers, and a simple culling algorithm. These together give real-time performance, and for simple models we get frame rates of over 150 fps. For more complex models 50 fps is normal. In addition to optimizations, two simple techniques for improving the visual quality are also presented.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, Shading, Shadowing, and Texture

1. Introduction

In the 1990's, most real-time computer generated images did not contain shadows. However, this started to change in the late 1990's, and games begun to use shadows as an important ingredient in their game play. For example, shadows were often used to help the player orient herself. Furthermore, shadows also naturally increase the level of realism. Today, the majority of games have dynamic hard shadows implemented as a standard component. If one were to remove the shadows from an application that used to have shadows, it would immediately be much harder to determine spatial relationships, and the images would get a more "flat" feeling. After working on dynamic soft shadows for a few years,[2, 4, 5] it is our experience that removing the softness of shadows causes almost as great decrease in image quality as when one removes hard shadows. Therefore, we conclude that dynamic soft shadows are very important for real-time computer graphics.

Our work here focusses on substantially increasing the performance of our previous soft shadow volume algorithms.[2, 4, 5] In that work, we lacked the hardware needed to fully accelerate our algorithm. However, after obtaining graphics hardware, we found that several optimizations were needed in order to get real-time performance, and those are described in this paper. More specifically, we now create tighter wedges around the penumbra volume generated by a silhouette edge. Furthermore, the pixel shader code has been made significantly shorter for both spherical and rectangular light sources. To overcome the 8-bit limitation of the frame buffer, we present a technique that allows for higher precision in the frame buffer, where we generate the soft shadow mask. Finally, a simple culling technique is presented that further improves performance. Besides these optimizations, we also present two methods for decreasing the artifacts that can appear.

The paper is organized as follows. First, some previous work is reviewed, a brief presentation of the soft shadow volume algorithm, and then follows a section with all our optimizations described. Section 5 describes how two artifacts can be suppressed. Then follows results, conclusion and future work.

2. Previous Work

Shadow generation has become a well-documented topic within computer graphics, and the amount of literature is vast. Therefore, we will only cover the papers that are most relevant to our work. For an overview, consult Woo's et al's survey,[16] and for real-time algorithms, consult Akenine-Möller and Haines.[1]

The two most widely used real-time shadow algorithms are shadow mapping and shadow volumes. The shadow mapping algorithm by Williams,[20] renders a depth image, called the shadow map, as seen from the light source. To create shadows, the scene is rendered from the eye, and for each pixel, its corresponding depth with respect to the light source is compared to the shadow map depth value. This determines whether the point is in shadow. To alleviate resolution problems in the shadow maps, Fernanado et al.[11] presented an algorithm that increased the shadow map resolution where it was needed the most, and Stamminger and Drettakis presented perspective shadow maps for the same reason.[19] Heidrich et al.[10] presented a soft version of the shadow map algorithm. It could handle linear light sources by interpolating visibility using more than one shadow map. Recently, Brabec and Seidel presented a more general soft shadow map algorithm.[17] Their work was inspired by Parker et al's[15] soft shadow generation technique for ray tracing. In that work, "soft-edged" objects were ray traced at only one sample per pixel using a parallel ray tracer. Thus, Brabec and Seidel presented a hardware-accelerated version of Parker et al's algorithm.

The shadow volume algorithm by Crow[7] is often implemented using a stencil buffer on commodity graphics hardware.[9] The algorithm first renders the scene using ambient lighting. In a second pass, each silhouette edge as seen from the light source creates a shadow volume quadrilateral which is rendered from the eye. Note that all that is required for these silhouette edges is that two polygons share that edge, and one of the polygons is frontfacing, and the other is backfacing as seen from the light source. The generated quads are rendered as seen from the eye. Frontfacing quads that pass the depth test add one to the stencil buffer, and backfacing quads subtract one. Therefore, at the end of this pass, the stencil buffer contains a mask where a zero indicate no shadow, and anything else indicates that the pixel is in shadow. The third pass is rendered with full lighting where the stencil buffer is zero. Everitt and Kilgard have presented techniques to make the shadow volume robust, especially for cases when the eye is inside shadow.[12]

Our work has focused on extending the hard shadow volume algorithm so that area and/or volume light sources can be used.[2,4,5] Our first paper presented an algorithm that could render shadow at interactive rates on arbitrary surfaces.[2] However, the set of shadow casting objects was severely limited. Recently, we have presented a much improved algorithm[5] that overcomes the limitations of our first attempt. Arbitrary shadow casters can be used, and we presented an implementation using graphics hardware. To speed up computations, a 4D texture lookup was used to quickly compute the coverage of silhouette edges onto light sources. We have also presented a version that can handle the eye-in-shadow problem, and a speed-up technique targeted for hardware.[4] After we obtained graphics hardware that was needed for an implementation of our most recent algorithm,[5] we realized that several optimizations were needed in order to get real-time performance.

There are also several algorithms that only can handle planar soft shadows. For example, Haines presents shadow plateaus, where a hard shadow, which is used to model the umbra, is drawn from the center of the light source.[13] The penumbra is rendered by letting each silhouette vertex, as seen from the light source, generate a cone, which is drawn into the Z-buffer. The light intensity in a cone varies from 1.0, in the center, to 0.0, at the border. A Coons patch is drawn between two neighboring cones, and similar light intensities are used. Heckbert and Herf use the average of 64–256 hard shadows into an accumulation buffer.[14] These images can then be used as textures on the planar surfaces. Radiance transfer can be precomputed, as proposed by Sloan et al.[18], and then used to render several difficult light transport situations in low-frequency environments. Real-time soft shadows are included there.

3. Soft Shadow Volume Algorithm

In this section, a brief recap of the soft shadow algorithm[5] will be presented. The first pass renders the entire scene with specular and diffuse lighting into the frame buffer. The second pass computes a visibility mask into a visibility-buffer (V-buffer), which is used to modulate the image of the first pass. Finally, ambient lighting is added in a third pass. The computation of the visibility mask renders the hard shadow quads for silhouette edges into the V-buffer. This is done using an ordinary shadow volume algorithm for hard shadows, and that pass ensures that the umbra regions receive full shadow. Each silhouette edge is then used to create a penumbra wedge, which contains the penumbra volume generated by that edge. The frontfacing triangles of each wedge is then rendered with depth writing disabled. For each rasterized pixel (x,y) with z as a depth value obtained from the first rendering pass, a pixel shader is executed. Note that z is made available by creating a texture that contains the depth buffer of the first rendering pass.

The hard shadow quad from a silhouette edge splits the wedge corresponding to the same edge, in an inner and outer half (see Figure 2). For points $\mathbf{p} = (x,y,z)$ located in the inner half of the wedge, the pixel shader computes how much of the light source $\mathbf{p}$ can "see" with respect to the silhouette edge of the wedge. This percentage value will be added to the V-buffer and compensates for the full shadow given in the umbra pass by the hard shadow quad. For pixels in the outer half of the wedge, the pixel shader will compute how much of the light source that is covered with respect to the silhouette edge, and this percentage value will instead be subtracted from the V-buffer. For pixels outside the wedge, the light source will be fully visible or covered, and the modification value will be zero in both cases. The accumulated effect of all this, is that a visibility mask, which represent the soft shadow, is computed.

4. Optimizations

In this section, several optimizations are presented in order to obtain real-time rendering using the soft shadow volume algorithm.

4.1. Tighter Wedges for Rectangular Light Sources

Our previously presented method created wedges from bounding spheres surrounding the light source.[5] The penumbra volume corresponding to an edge and a spherical light source is the swept volume of the circular cone created by reflecting the light source through the sweeping point that moves from one edge end point to the other (see Figure 1a). To handle robustness issues and avoid the hyperbolic[13] front and back wedge surfaces along the edge, the edge vertex furthest from the light center is temporarily moved straight towards the light center to a new location at the same distance from the light center as the other edge vertex. Then, this new edge is used for sweeping the wedge volume. This will make the front and back wedge surfaces planar, and the resulting wedge will completely enclose the true penumbra volume (Figure 1c). This process also simplifies the computation of the wedge polygons.

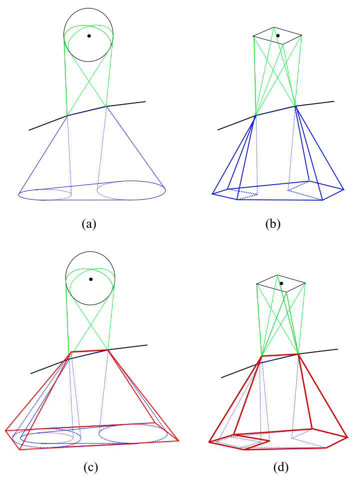

Figure 1: *a) and b) show the penumbra volume for an edge for a spherical and a rectangular light source respectively. c) and d) show the corresponding wedges, which enclose the penumbra volumes. In c) and d) the inner left cone is the reflected cone through the left silhouette vertex. The outer left cone is generated from the relocated edge vertex.*

Tighter wedges can be constructed if we exploit that the light source is rectangular. The penumbra volume is created as described above, with the exception that the cones now have a rectangular base, as shown in Figure 1b.

For a spherical light source, a left and a right plane was used to close the wedge on the sides. Now instead, we use the two leftmost and two rightmost triangles respectively of the two end pyramids that are formed by the reflection of the light source through the edge end points. This will result in a more tight fitting wedge at the sides. Along the edge, this wedge will typically be thinner than one created from a spherical light source (see Figure 1d). A bottom plane may be added for the z-fail algorithm and culling (see section 4.4).

If an edge intersects the light source, that edge should be clipped against the light source. In this case, all wedge planes are coplanar, and the wedge will enclose everything on one side of the plane. This is correct behavior, but is inconvenient for real-time applications, since rasterization of such a wedge may be very expensive.

4.2. Optimized Pixel Shaders

We have implemented two optimized pixel shaders that handle both spherical and rectangular light sources.

The shaders were tested on an ATI 9700 Pro. The test program first generates shadow volumes and wedge geometry on the CPU. The plane for the hard shadow quad separates the wedge in an inner and outer half. It is worth noting that on the ATI 9700 Pro, this polygon needed to match the shadow volume polygon used to determine the shadow approximation exactly (including culling order) for the stencil operations explained below to align correctly. The shadow volume quad itself does not extend all the way out to the wedge sides. Therefore, one more polygon on either side of the hard shadow quad is added that lies in the separating plane and closes the wedge halves (see Figure 2).

The test program then renders the world space per-pixel positions of shadow receiving objects to a 16 bit per channel float texture that is used by the wedge pixel shaders. This is to avoid computing the screen space to world space transformation of the pixel position in the pixel shader, which is more expensive. It should be noted that if the screen space position were used, the z-coordinates would have to be available through a depth texture, which means that a texture lookup is necessary anyhow. Next, the shadow volumes are rendered and the V-buffer is incremented and decremented in the usual fashion to determine the shadow approximation. Then, the wedges are rendered to produce the soft shadows in the V-buffer. Each wedge side is rendered separately and stencil operations are used in order to separate positive and negative V-buffer contributions and to avoid rendering pixels that do not intersect a wedge. This is done by applying the culling method, described in Section 4.4, on each wedge half. A final pass uses the resulting V-buffer in a per-pixel diffuse and specular lighting calculation.

Also, hardware user defined clipping planes were used since the default guard band clipping on the ATI 9700 Pro introduced enough error in the clipped texture coordinates to cause visual artifacts.

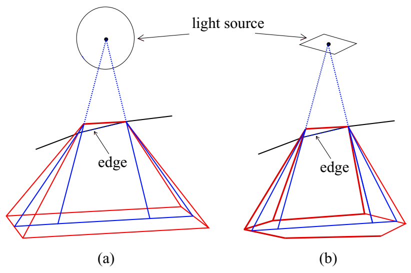

Figure 2: *The wedge is split by the hard shadow quad in an outer and inner half. Since the hard shadow quad does not extend all the way out to the left and right wedge sides, two triangles are used on either side to achieve the split. Together, these three polygons constitute the wedge center polygons. a) shows an example for a spherical light source, and b) for a rectangular light source.*

4.2.1. Spherical Light Source Shader

The spherical light source shader uses the cone defined by the point to be shaded and the spherical light source as shown in Figure 3 to clip the silhouette edge. The clipped silhouette edge is then projected to the plane defined by the light source center and the ray from the light source center to the point to be shaded. The projected points are used to determine the coverage.

The shader first transforms the silhouette edge from world space into light space where the light center is located on the z-axis at $z = 1$, the light radius is one, and the point to be shaded is at the origin. The line described by the transformed points is then clipped against the light cone which is now described by the cone equation $\mathbf{x}^2 + \mathbf{y}^2 = \mathbf{z}^2$ (see Figure 4). Solving the quadratic resulting from the equation of the intersection of the line and cone, with intersection points below the xy plane being rejected, does the clipping. The clipped points $\mathbf{c}_0$ and $\mathbf{c}_1$ are projected on the $z = 1$ plane by a simple division by $\mathbf{z}$.

The resulting 2D point values ($\mathbf{p}_0$ and $\mathbf{p}_1$ in Figure 5) are used in two separate texture lookups into a cube-map that implements **atan2**(y,x) to obtain θ_0 and θ_1. **atan2**(y,x) returns the arctangent of $\frac{y}{x}$ in the range -π to π radians. The two angles characterize the minor arc defined by the intersections of the rays from the light center to $\mathbf{p}_0$ and $\mathbf{p}_1$ and the unit circle. θ_0 and θ_1 are then used in another 2D texture lookup (Figure 5b) to obtain the area defined by the circle center and the minor arc. The area derived from the cross product of $\mathbf{p}_0$ and $\mathbf{p}_1$ is subtracted from this value and divided by the area of the unit circle to give the final coverage (shaded in gray on Figure 5a).

4.2.2. Rectangular Light Source Shader

To compute coverage it is necessary to project the edge onto the light source and clip it to the border of the light source.

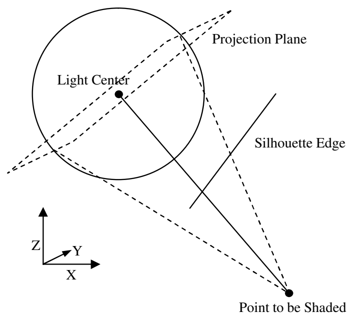

Figure 3: *The spherical light source clipping cone and projection plane shown in world space.*

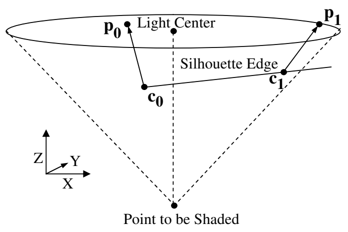

Figure 4: *The clipping of the silhouette edge with the light cone in light space. The clipped edge end points* $\mathbf{c}_0$ *and* $\mathbf{c}_1$ *are then projected onto the plane* $z = 1$*, which gives* $\mathbf{p}_0$ *and* $\mathbf{p}_1$.

For robustness it is better to clip the edge first and then project it; otherwise points behind the origin of the projection will be inverted. Both clipping and projection can be done efficiently using homogenous coordinates. The endpoints of the edge are initially transformed so that the point to be shaded is at the origin and the normal to the light source plane is parallel to the z-axis. The matrix for the projection is computed with the point to be shaded as the origin of the projection and the near plane as the rectangular light source.

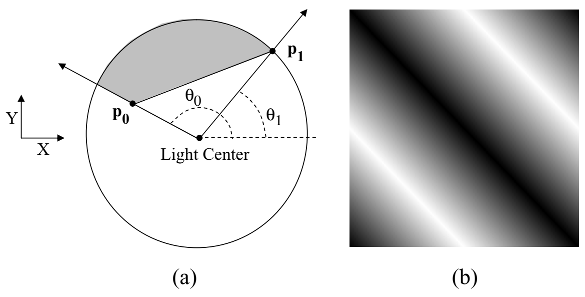

Figure 5: *a) The coverage is computed from the projected 2D points* $\mathbf{p}_0$ *and* $\mathbf{p}_1$ *in projected space. b) Area lookup texture.*

The endpoints of the edge are then transformed into clip space using the projection transform, followed by homogenous clipping to each side of the rectangular light source. After clipping, the endpoints are projected by dividing by the homogenous w-coordinate. The endpoints can then be used to look up the area in a 4D coverage texture or be used for analytic computation of the area.

For non-textured rectangular light sources, the coverage of the projected silhouette edge quadrilateral can be computed analytically instead of being computed using a 4D coverage texture. The advantage of computing the coverage analytically is that higher accuracy is possible using less texture space. The area covered by the projected edge quadrilateral is equal to the light source area between the two vectors of infinite length from the center of the light source through the projected clipped endpoints of the edge minus the area of the triangle defined by the center of the light source and the projected clipped endpoints of the edge (Figure 6a). The area of the triangle defined by the center of the light source and the projected clipped endpoints is computed using a 2D cross product. The light source area between the two vectors of infinite length is looked up in a 2D texture (Figure 6b) based on the angles of elevation for the two vectors (θ_0 and θ_1). A cube-map that implements **atan2**(y,x) is used to look up the angles to the two vectors.

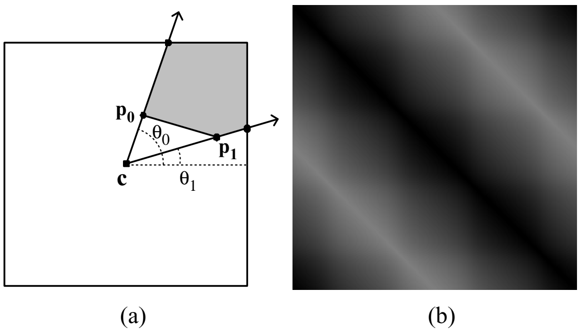

Figure 6: *a) The area covered by the silhouette edge ($\mathbf{p}_1\mathbf{p}_0$) projected onto the light source is equal to the area of the light source covered by the triangle defined by the light center* $\mathbf{c}$ *and the two vectors ($\mathbf{c},\mathbf{p}_0$) and ($\mathbf{c},\mathbf{p}_1$) extended to infinity minus the area of the triangle ($\mathbf{c},\mathbf{p}_0,\mathbf{p}_1$). b) Area lookup texture.*

4.3. Frame Buffer Blending

Current generation consumer graphics hardware (GeForceFX and Radeon 9700) can only blend to 8-bit per component frame buffers. The previous implementation of the soft shadow algorithm either 1) used 32-bit float buffers and circumvented the blending by rendering a wedge to a temporary buffer, using the frame buffer as a texture, and a following copy-back pass to the frame buffer, or 2) used the lower 6 bits of single 8-bit component while reserving the upper 2 bits for overflow. It is desirable to have at least 8 bits of precision when the final results are displayed using 8-bit RGB-components to avoid precision artifacts. Additive frame buffer blending can be accomplished with greater than 8-bit precision by splitting values across multiple 8-bit components of the frame buffer. A number of the most significant bits of each component are reserved to allow for overflow. The number of bits reserved is based on the expected maximum number of overlapping wedge polygons. n bits allows for up to 2^n levels of overlap.

In our implementation, two ordinary 8-bit per component rgba buffers are used for the V-buffer. One of the buffers contains the additive contribution and one of the buffers contains the subtractive contribution. The additive contributions are computed by drawing the back half of all wedges into the additive buffer and the subtractive contributions are computed by drawing the front half of all wedges into the subtractive buffer. An alternative implementation could use the multiple render target support of current generation graphics hardware to draw into both buffers simultaneously. We have found that on complex models more than 16 levels of overlap occur, requiring that 5 bits be reserved for carry, so a 12-bit coverage value is split across the four components of each buffer. For each of the four 8-bit components, the upper 5 bits are reserved for overflow and the lower 3 bits contain 3 bits of the 12-bit value.

Future generations of graphics hardware may be able to blend to 16-bit per component frame buffers making splitting up values unnecessary.

4.4. Culling

A consequence of the soft shadow volume algorithm is that the rendering of the wedges only affects those pixels whose corresponding points (formed as (x,y,z), where (x,y) are the pixel coordinates, and z is the depth at (x,y)) are located inside the wedges. Put differently, the rendering of a wedge can only affect a point if it is inside the penumbra region. Still, the wedges normally cover many more pixels whose corresponding points are not inside wedges. For these points, it is unnecessary to execute the rather expensive pixel shader.

Therefore, ideally, the pixel shader should only be executed for points inside the wedge, and culling should reject all other pixels whose corresponding points are outside the wedges. This culling can be done using two passes and combining the depth-test and the stencil test.

The culling is also used as the mechanism to separate the inner and outer wedge half contributions from each other, since points in an inner wedge half should give positive V-buffer contribution and points in an outer half should give negative contribution. In our first approach, when rendering a wedge, the plane equation for the hard shadow quad was used in the pixel shader to classify a point as being in the inner or outer wedge half. However, that approach can lead to precision errors for points that lie in or very close to the hard shadow quad plane, resulting in visual artifacts. Instead, each wedge half (depicted in Figure 2) is rendered separately, as follows.

First, the frontfacing triangles of the wedge half are rendered into the stencil buffer, while setting the stencil buffer elements to one for each fragment that passes the depth test. The depth test is set to `GL_GREATER`, i.e., only passing for fragments with points farther away than the frontfacing triangles. Then, the backfacing wedge half triangles are rendered into the V-buffer using the pixel shader and with both the stencil test and the depth test enabled. The stencil test is set to only pass for stencil values equal to one, and the depth test is set to `GL_LESS`. i.e., only passing for fragments with points closer to the eye than the backfacing triangle planes. If either of these tests fail, the point at that pixel is located outside the wedge half, and the fragment is culled from rendering. Early rejection in the hardware can then avoid executing the shader for culled fragments.

It should be noted that for this culling to be successful, it is required that the hardware is capable of doing early depth rejection and early stencil rejection. In general, a pixel shader may affect the depth value which will affect the outcome of the depth test, which in turn may affect the outcome of the stencil test, depending on what stencil function that is used. In our case, the pixel shader does not affect the depth values, and thus, the depth and stencil tests can be done before executing the shader.

5. Improving the Visual Results

We have reported that the soft shadow volume algorithm can suffer from two types of artifacts.[5] The first is that the soft shadows are created incorrectly for overlapping geometry, and the second is due to that only a single silhouette is used for the shadow casting objects. In this section, two very simple techniques, which can improve the visual results, are presented.

5.1. Overlap Approximation

The soft shadow volume algorithm can accurately render soft shadows for a single closed loop. However, the combination of soft shadows from several objects is more difficult, as can be seen in Figure 7.

The left part shows two pieces of gray-shaded geometry projected onto a square light source. The geometry does not overlap on the light source. Our algorithm handles this case correctly. However, it always assumes that the geometry is non-overlapping, so for the situation in the right part of Figure 7, the same geometry with different positions should create another coverage. Unfortunately, our algorithm treats the situation to the right in the same way as to the left, i.e., the result is the same.

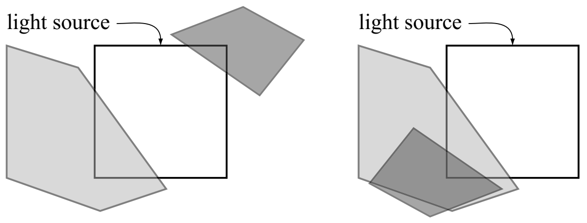

Figure 7: *Overlap problems due incorrect combination of coverage. The light gray shadow caster covers 16 percent of the light source, while the darker gray shadow caster cover 4 percent. To the left, the shadow casters together covers 20 percent, while to the right they cover 16 percent.*

To ameliorate this, a probabilistic approach is taken. Each silhouette loop is rendered separately, so that a visibility mask is created for each silhouette loop. Next, these two visibility mask images should be combined per pixel. Now assume, that A covers c_A percent of the light source, and B covers c_B percent. Since no information on how the geometry overlaps is available, it appears to be advantageous to produce a result that is in between the two extremes. Therefore, the following combined result is used per pixel:

$$c = \max(c_A, c_B) + \frac{1}{2}\min(c_A, c_B), \qquad (1)$$

which implies that a result in between the two extremes shown in Figure 7 is obtained.

Splitting the silhouettes into single silhouette loops is easy to do in real time. It is worth mentioning that a vertex of a silhouette edge always is connected to an even number of silhouette edges [3], which simplifies the task. The algorithm will have to render each silhouette loop separately and merge the visibility result with the result from the other rendered loops. To avoid using several buffers, the soft shadow contribution of a silhouette loop can be rendered into, for instance, the r- and g-component of the frame buffer. Then the result for the affected pixels could be merged, according to Equation 1, with the total result residing in the b- and α-component. This merging can be done by an intermediate pass between rendering each silhouette loop. The pass should also clear the values in the r- and g-components.

5.2. Single Silhouette Approximation

The silhouette edges are generated from only one sample location on the light source. This is obviously not physically accurate, since the silhouette edges, as seen from the sample location, may vary for different samples on the light source.

It is possible to reduce the effect of both the single silhouette artifact and the overlap approximation by splitting a large area light source into some smaller. The rationale for this is that the quality of the soft shadows are good for small light sources, but gets worse for larger. For a large light source, the probability is higher that several independent silhouette loops contribute to the soft shadow at pixel, than for a small light source. Since more sample points for the silhouettes are added, the single silhouette artifact will be reduced as well.

It is possible to achieve a correct result by increasing the number of splits to infinity or to a point where there is no longer any visual change in the result. This is closely related to the technique that uses multisampling of hard shadows to get a soft result. However, for a visually pleasing result, our penumbra wedge method typically requires two orders of magnitude fewer sample locations. Figure 8 shows a worst case scenario for the single silhouette artifact, now improved by splitting one large area light source into four smaller of equal size, together covering the same area as the larger. The reference image to the right shows the result of using 1024 samples points and blending hard shadows.

It should be emphasized that using n small light sources, covering the same area as one larger light source, is not necessarily n times as expensive than using the single larger light source. The cost of the soft shadow algorithm is proportional to the number of pixels with points located inside the wedges, and the wedges generated from each smaller light source will be significantly thinner than those generated from the larger light source.

Figure 9 shows an example of a more complex scene. Near the center of the shadow in the leftmost image the overlap artifact is pronounced. There are a lot of silhouette edges that are in shadow and falsely give shadow contribution. This results in an overly dark shadow, which can be seen when comparing to the more correct result in the rightmost reference image. Here, it can clearly be seen that, typically, very good quality is achieved by splitting the larger light source into only a few smaller ones.

6. Results

The pixel shader for spherical light sources requires 59 arithmetical and 4 texture load instructions. The optimized pixel shader program for a textured light source, using the 4D coverage texture lookup, consists of 61 arithmetic instructions and 2 texture load instructions. The optimized shader for a non-textured light, using the analytic coverage texture, consists of 60 arithmetic instructions and 5 texture load instructions. Code for all the three shaders are available online at *http://www.ce.chalmers.se/staff/uffe/NonTexturedRect.txt*, *http://www.ce.chalmers.se/staff/uffe/TexturedRect.txt*, and *http://www.ce.chalmers.se/staff/uffe/Sphere.txt*.

Note that our original code for rectangular light sources consisted of 250 instructions.[5] Figure 10 and Figure 11 shows two scenes with a comparison of image quality and frame rate between using hard shadows, soft shadows from a spherical light, square light and large rectangular light. Our original implementation renders the cylinder scene in about 8–10 fps and the alien scene in 3–4 fps for both spherical and rectangular light sources. With our optimized algorithm, the frame rate is 15-20 times higher, as can be seen in the two figures.

The optimized wedge generation method for rectangular light sources creates tighter wedges that typically improve the overall frame rate with 1.2-2 times, which, for Figure 9a) gives an overall speedup of 1.5 times. Regarding the culling optimization (see Section 4.4), we have only observed a small performance increase of about 5%, but since this is scene and hardware dependent, we believe that there are situations when it can perform better. Worth noting is that culling comes for free since it is the mechanism to separate the contributions from the inner and outer wedge halves.

The optimized shader for the non-textured spherical light source uses about 324KB of texture memory in total. A six-face cube-map of 128×128 16 bit values per face is used for the angle lookups. A 256×256 single channel 16-bit texture is used for the area lookup. A 1024 1D texture of four 8-bit channels is used to convert the coverage value into a subtractive or additive visibility value split over the r,b,g,a components.

For the non-textured rectangular light source shader, about 270KB of texture memory is used. A six-faced cube-map of 128×128 16-bit values per face is used for the angle lookup. A 256×256 single channel 8-bit per texel texture is used for area lookup, and the same 1024 entries texture as for the spherical light is used for splitting the visibility contribution over the rgba-components. A textured rectangular light source requires 1MB of texture memory for the 4D coverage texture for a single-colored light source, and 3MB if the light source is rgb-colored, as before.[5]

7. Conclusions

We have presented several optimizations to our original soft shadow algorithm that greatly improves the performance. The old algorithm typically rendered the scenes shown in this paper in 1-10 fps. With the optimizations presented here, frame rates of up to 150 fps are achieved (see Figure 10). The main improvements consist of three modified fragment shaders; one for spherical and two for rectangular light sources, that lowers the number of shader instructions from 250 to 63, 63 and 65 respectively. The fragment shaders also utilize an ordinary frame buffer with 8 bits per rgba-component to get 12 bits of precision for the soft shadow contribution. This circumvents the problem that, on current hardware, blending typically cannot be done to a frame buffer with 16 or 32 bits per rgba-component. The old algorithm had to use extra rendering passes or lower the precision to 5 bits for the soft shadows.

Furthermore, a method is presented for creating tighter wedges, which typically improves the overall frame rate with 1.2 to 2 times. A culling technique is also described that can increase performance a bit. Finally, we show how to improve the visual quality by reducing the effects of the overlap and single silhouette approximations. With the improvements presented in this paper, real-time soft shadows with very good quality can now be used in, for instance, games and virtual reality applications.

8. Future Work

The wedge generated for a small silhouette edge is often quite large. A silhouette edge simplification algorithm could be implemented to save a significant amount of wedge overdraw. One easy win would be to collapse two connected silhouette edges which are roughly parallel into a single edge.

By using vertex shaders for the wedge generation, the CPU will be offloaded so that it can do other more useful work (game logic, collision detection, etc). We will implement this shortly.

Acknowledgements

We wish to give a great thanks to Greg James and Gary King for sharing their technique of combining buffer channels with few bits to achieve a virtual buffer channel with many bits. Their technique is similar to the one described in this paper, which was developed independently by Michael Mounier. We also want to thank Randy Fernando, Mark Kilgard, and Chris Seitz at NVIDIA.

References

1. T. Akenine-Möller and E Haines, *Real-Time Rendering*, 2nd edition, June 2002.
2. T. Akenine-Möller and U. Assarsson, "Rapid Soft Shadows on Arbitrary Surfaces using Penumbra Wedges," *Eurographics Workshop on Rendering 2002*, pp. 309–318, June 2002.
3. T. Akenine-Möller and U. Assarsson, "On Shadow Volume Silhouettes," submitted to *Journal of Graphics Tools*, 2003.
4. U. Assarsson and T. Akenine-Möller, "Interactive Rendering of Soft Shadows using an Optimized and Generalized Penumbra Wedge Algorithm", submitted to the *Visual Computer*, 2002.
5. U. Assarsson and T. Akenine-Möller, "A Geometry-Based Soft Shadow Volume Algorithm using Graphics Hardware", to appear in *SIGGRAPH 2003*, July 2003.
6. P. Bergeron, "A General Version of Crow's Shadow Volumes," *IEEE Computer Graphics and Applications*, **6**(9):17–28, September 1986.
7. F. Crow, "Shadow Algorithms for Computer Graphics," *Computer Graphics (Proceedings of ACM SIGGRAPH 77)*, pp. 242–248, July 1977.
8. D. Eberly, "Intersection of a Line and a Cone," *http://www.magic-software.com*, Magic Software Inc., October 2000.
9. T. Heidmann, "Real shadows, real time," *Iris Universe*, no. 18, pp. 23–31, November 1991.
10. W. Heidrich, S. Brabec, and H-P. Seidel, "Soft Shadow Maps for Linear Lights", *11th Eurographics Workshop on Rendering*, pp. 269–280, 2000.
11. R. Fernando, S. Fernandez, K. Bala, and D. P. Greenberg, "Adaptive Shadow Maps", *Proceedings of ACM SIGGRAPH 2001*, pp. 387–390, August 2001.
12. C. Everitt and M. Kilgard, "Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering", *http://developer.nvidia.com/*, 2002.
13. E. Haines, "Soft Planar Shadows Using Plateaus", *Journal of Graphics Tools*, **6**(1):19–27, 2001.
14. P. Heckbert and M. Herf, *Simulating Soft Shadows with Graphics Hardware*, Carnegie Mellon University, Technical Report CMU-CS-97-104, January, 1997.
15. S. Parker, P. Shirley, and B. Smits, *Single Sample Soft Shadows*, University of Utah, Technical Report UUCS-98-019, October 1998.
16. A. Woo, P. Poulin, and A. Fournier, "A Survey of Shadow Algorithms", *IEEE Computer Graphics and Applications*, **10**(6):13–32, November 1990.
17. S. Brabec, and H-P. Seidel, "Single Sample Soft Shadows using Depth Maps", *Graphics Interface 2002*, pp. 219–228, 2002.
18. P-P. Sloan, J. Kautz, and J. Snyder, "Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments", *ACM Transactions on Graphics*, (21)(3):527–536, July 2002.
19. M. Stamminger, and G. Drettakis, "Perspective Shadow Maps", *ACM Transactions on Graphics*, **21**(3):557–562, July 2002.
20. L. Williams, "Casting Curved Shadows on Curved Surfaces", *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, pp. 270–274, August 1978.

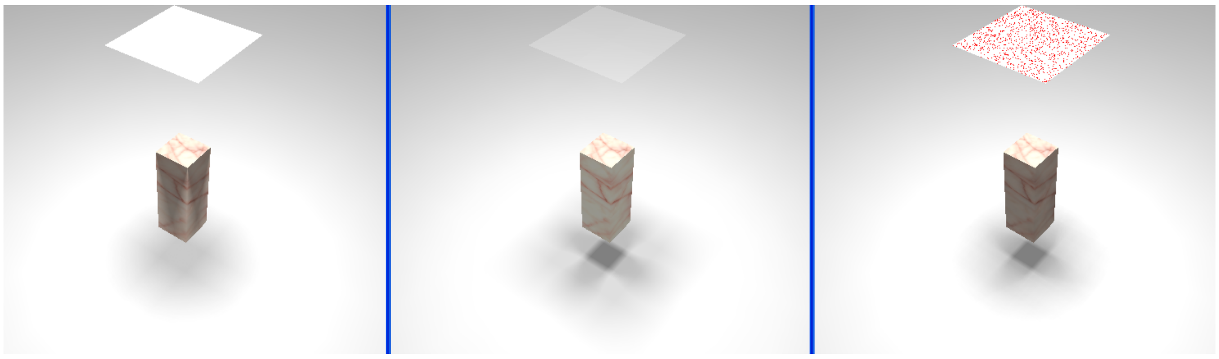

Figure 8: *One area light source,* 2×2 *area light sources, 1024 point light sources.*

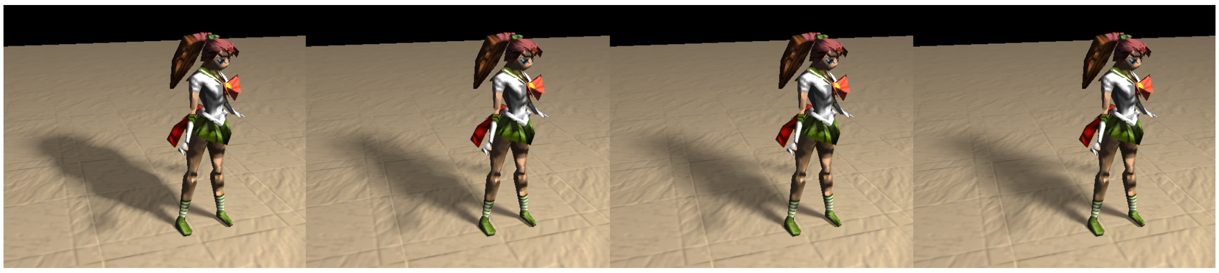

Figure 9: *One area light source,* 2×2 *area light sources,* 3×3 *area light sources, 1024 point light sources.*

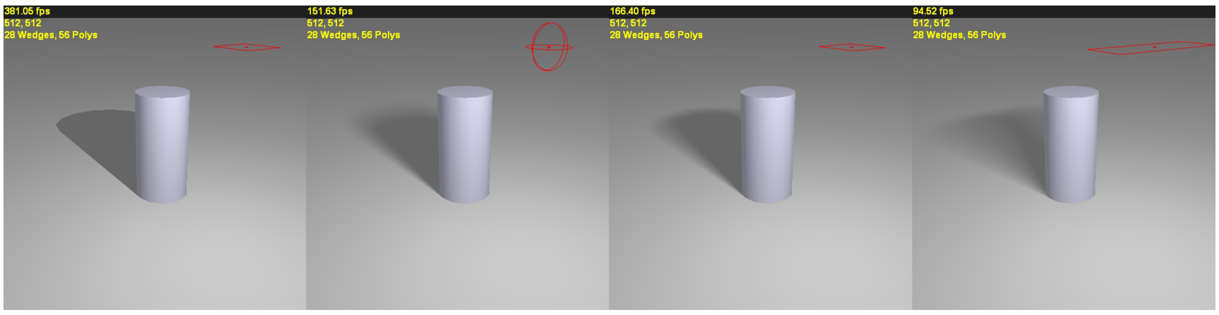

Figure 10: *Comparison of appearance and frame rate for a cylinder with hard shadow, soft shadow from a spherical light source, soft shadow from a square light source, and soft shadow from a wide rectangular light source .*

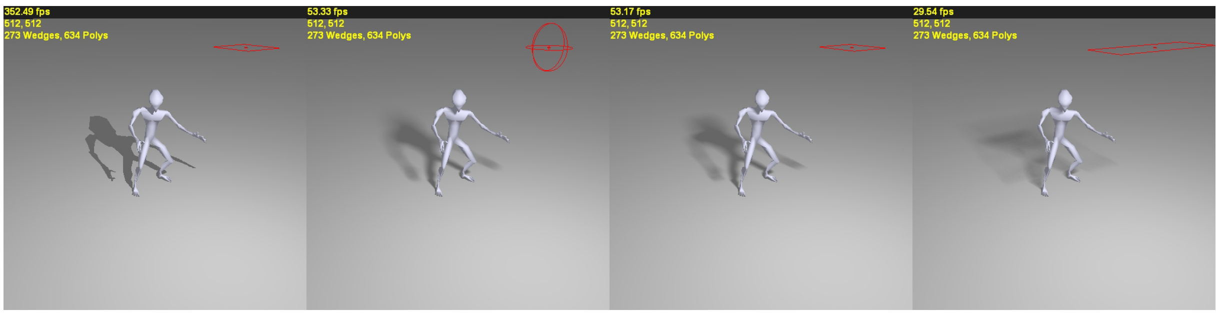

Figure 11: *Same situations as for Figure 10, but for a more complex shadow caster.*