Thirteenth Eurographics Workshop on Rendering (2002)
P. Debevec and S. Gibson (Editors)

Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges

Tomas Akenine-Möller and Ulf Assarsson

Department of Computer Engineering, Chalmers University of Technology, Sweden

Abstract

Shadow generation has been subject to serious investigation in computer graphics, and many clever algorithms have been suggested. However, previous algorithms cannot render high quality soft shadows onto arbitrary, animated objects in real time. Pursuing this goal, we present a new soft shadow algorithm that extends the standard shadow volume algorithm by replacing each shadow quadrilateral with a new primitive, called the penumbra wedge. For each silhouette edge as seen from the light source, a penumbra wedge is created that approximately models the penumbra volume that this edge gives rise to. Together the penumbra wedges can render images that often are remarkably close to more precisely rendered soft shadows. Furthermore, our new primitive is designed so that it can be rasterized efficiently. Many real-time algorithms can only use planes as shadow receivers, while ours can handle arbitrary shadow receivers. The proposed algorithm can be of great value to, e.g., 3D computer games, especially since it is highly likely that this algorithm can be implemented on programmable graphics hardware coming out within the next year, and because games often prefer perceptually convincing shadows.

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

Keywords: *soft shadows, graphics hardware, shadow volumes.*

1. Introduction

Shadows in computer graphics are important, both for the viewer to determine spatial relationships, and for the level of realism. When rendering shadows on arbitrary receivers in real time using commodity graphics hardware, the only currently feasible solution is to render hard shadows. A hard shadow consists only of a fully shadowed region, called the *umbra*. Therefore, a hard shadow edge can sometimes be misinterpreted for a geometric feature. However, in the real world, there is no such thing as a true point light source, as every light source occupies an area or volume. Area and volume light sources generate soft shadows that consist of an umbra, and a smoother transition, called the *penumbra*. Thus, soft shadows are more realistic in comparison to hard shadows, and they also avoid possible misinterpretations. Therefore, it is desirable to be able to render soft shadows in real time as well. However, currently no algorithm can handle all the following goals:

I. The softness of the penumbra should increase linearly with distance from the occluder, starting at zero at the occluder.[13]

II. The umbra region should disappear given that a light source is large enough.

III. Typical sampling artifacts should be avoided. For example, often a number of superpositioned hard shadows can be discerned.[17] The result should be visually smooth.[13]

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 algorithm, which is an extension of the shadow volume (SV) algorithm (see Section 3), achieves these goals with some limitations on the type of scenes that can be used.

Instead of creating a shadow quadrilateral (quad) for each silhouette edge (as seen from the light source), a penumbra wedge is created. Each such wedge represents the penumbra volume that a silhouette edge gives rise to. See Figure 2. Together these shadow wedges represent an approximation of the soft shadow volume with more or less correct characteristics (see Section 7). For example, the results often look remarkably close to those of Heckbert and Herf.[9] Some approximations are introduced, but still the results are plausible (as can be seen in Figure 12). In addition to the new algorithm, an important contribution is a technique for efficiently rasterizing wedges. Our software implementation of the algorithm runs at interactive rates on a standard PC. Assuming that the algorithm can be implemented using graphics hardware that comes out within a year, which is very likely, the algorithm will reach real-time speeds. Our focus has therefore been on generating soft shadows that approximate true soft shadows well, and that can be rendered rapidly, instead of a slow and accurate algorithm. This is a significant step forward for shadow generation in, e.g., games.

Next, some related work is reviewed, followed by a description of the standard shadow volume algorithm,[1] which is the foundation of our new algorithm. In Section 4, our algorithm is described. Then follows optimizations, implementation notes, and results. In Section 8, we discuss limitations of our work, and finally we offer some ideas for future work, and a conclusion.

2. Related Work

In this section, the most relevant work for soft shadow generation at interactive rates is presented. Consult Woo et al.[20] for an excellent survey on shadow algorithms in general, and Haines and Möller[6] for a survey on real-time shadows.

By averaging a number of hard shadows, each generated by a different sample point on an extended light source, soft shadows can be generated as presented by Heckbert and Herf.[9] This is mostly suitable for pre-generation of textures containing soft shadows, because a high number of samples (64–256) is needed so that a soft shadow edge does not look like a number of superpositioned hard shadows. These types of algorithms can normally only get $n+1$ different levels of shadow intensities for n samples.[11] Once the soft shadow textures have been generated, they can be rendered in real time for a static scene. Such algorithms only apply to planar shadow receivers. Gooch et al.[5] also project hard shadows onto planes and compute the average of these. Light source samples are taken from a line parallel to the normal of the receiver. This creates approximately concentric hard shadows, which in general look better than the method by Heckbert and Herf,[9] and so fewer samples can be used.

Haines[7] presents a novel technique for generating planar shadows. The idea is to use a hard shadow from the center of a light source. Then a cone is "drawn" from each silhouette (as seen from the point light source) vertex, with shadow intensity decreasing from full (in the center) to zero (at the border of the cone). Between two such cones, inner and outer Coons patches are drawn, with similar shadow intensity settings. These geometrical objects are then drawn to the Z-buffer to generate the soft shadow. Our algorithm can be seen as an extension of Haines' method and the SV algorithm. Haines' algorithm produces umbra regions that are equal to a hard shadow generated from one point on the light source, and thus the umbra region is too large.[7] Our algorithm overcomes this limitation and also allows soft shadows to be cast on arbitrary receiving geometry. The only requirement is that it should be possible to render the receiving geometry to the Z-buffer.

For real-time work, there are two dominating shadow algorithms that cast shadows on arbitrary surfaces. One is the shadow volume algorithm (Section 3), and the other is shadow mapping. The shadow mapping algorithm[19] renders an image, called the shadow map, from the point of the light source. This shadow map captures the depth of the scene at each pixel from the point of view of the light. When rendering from the eye, each pixel's depth is tested against the depth in the shadow map, which determines whether the point is in shadow. Reeves et al.[15] improve upon this by introducing *percentage closer filtering*, which reduces aliasing along shadow edges. Segal et al.[16] describe a hardware implementation of shadow mapping. Today, shadow mapping with percentage closer filtering is implemented in commodity graphics hardware, such as the GeForce3. Heidrich et al.[11] extend the shadow mapping to deal with linear light sources, where two shadow maps are created; one for each endpoint of the line segment. Visibility is then interpolated across the light source into a visibility map used at rendering. For dynamic scenes, the process of creating the visibility map is quite expensive (may take up to two seconds per frame). All shadow mapping algorithms have biasing problems, which occur due to numerical imprecisions in the Z-buffer, and the problem of choosing a reasonable shadow map size to avoid aliasing. One notable exception is the adaptive shadow map algorithm, which iteratively refines the shadow map resolution where needed.[4]

Parker et al.[13] extends ray tracing so that only one sample is used for soft shadow generation. This is done by using a "soft-edged" object, and using the intersection location with this object as an indicator of where in the shadow region a point is located. This was used in a real-time ray tracer. In 1998, Soler and Sillion[17] presented an algorithm based on convolution. Their ingenious insight was that for parallel configurations (a limited class of scenes), a hard shadow image can be convolved with an image of the light source to form the soft shadow image. They also present a hierarchical error-driven algorithm for arbitrary configurations by using approximations. Hart et al.[8] present a lazy evaluation algorithm for accurately computing direct illumination from extended light sources. They report rendering times of several minutes, even for relatively simple scenes. Stark and Riesenfeld [18] present a shadow algorithm based on vertex tracing. Their algorithm computes exact illumination for scenes consisting of polygons, and is based on the vertex behavior of the polygons.

There are also several algorithms that use back projection to compute a discontinuity mesh, which can be used to capture soft shadows. However, these are often very geometrically complex algorithms. See, for example, the work by Drettakis and Fiume.[2]

3. Shadow Volumes

In 1977, Crow presented an algorithm for generating hard shadows.[1] By using a stencil buffer, an implementation is possible that uses commodity graphics hardware.[10] That implementation of Crow's algorithm is called the *shadow volume* (SV) algorithm. It will be briefly described here, as it is the foundation for our new algorithm. The SV algorithm builds volumes that bound the shadow. This is done by taking each silhouette edge (as seen from the light source) of the shadow casting object, and creating a shadow quad. A shadow quad is formed from a silhouette edge, and then extending lines from the edge end points in the direction from the light source to the edge end points. The shadow volume is illustrated in Figure 1. In theory, the shadow quad is extended infinitely. The SV algorithm is a multipass algorithm.

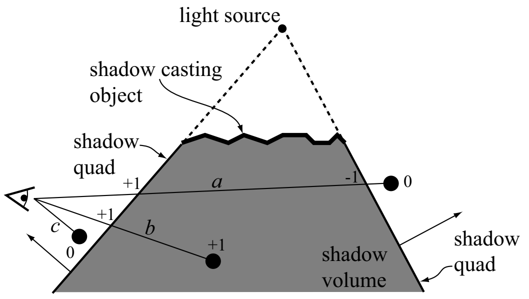

Figure 1: *The standard shadow volume algorithm. Ray b is in shadow, since the stencil buffer has been incremented once, and the stencil buffer values thus is* $+1$. *Rays a and b are not in shadow, because their stencil buffer values are zero.*

First, the scene is rendered from the camera's view, with only ambient lighting. Then the front facing shadow quads are rasterized without writing to the color and Z-buffer. For each fragment that passes the depth test, i.e., that is visible, the stencil buffer is incremented. Backfacing shadow quads are rendered next, and the stencil buffer is decremented for visible fragments. This means that the stencil buffer will hold a mask (after all shadow quads have been rendered), where zeroes indicate fragments not in shadow. The final pass renders with full shading where the stencil buffer is zero.

See Everitt and Kilgard's paper for a robust implementation of shadow volumes. [3]

4. New Algorithm

Our new algorithm replaces the shadow quads of the SV algorithm with penumbra wedges (Section 4.1), as illustrated in Figure 2. For the rest of this description, we assume that the light source is a sphere. The light intensity (LI), s, in a point $\mathbf{p}$, is a number in $[0,1]$ that describes how much of a light source the point $\mathbf{p}$ can "see." A point is in *full shadow* (in the umbra) when $s=0$, and *fully lit* when $s=1$, and otherwise in a penumbra region. The LI varies inside a wedge, and our goal is to approximate a physically-correct value as well as possible, while at the same time obtaining fast rendering.

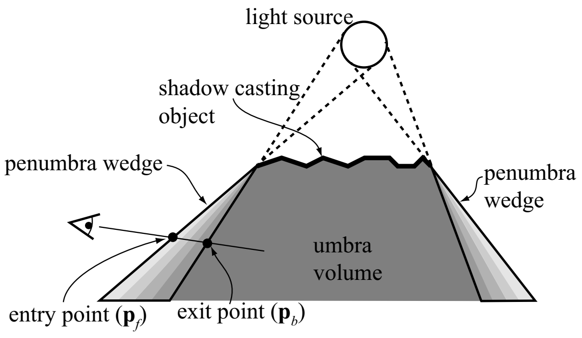

Figure 2: *The new algorithm uses penumbra wedges to capture the soft region in the shadow.*

The wedges that model the penumbra regions also implicitly model the umbra volume. The difference between our algorithm and the standard SV algorithm is that for our algorithm, one need to pass through an entire wedge (or a combination of wedges) before entering the umbra volume.

For a visually appealing result, the light intensity interpolation must be continuous between adjacent wedges. Thus, the idea of our algorithm is to introduce a new rendering primitive, namely, the penumbra wedge, that can be rasterized quickly and that achieves continuous light intensity. The details of this interpolation are given in Section 4.2.

Just as the SV algorithm requires a stencil buffer to rapidly render shadows using graphics hardware, so does our algorithm. However, the presence of penumbra regions makes the precision demands on the buffer higher. For this, we use a signed 16-bit buffer, which we call the light intensity (LI) buffer. So the LI buffer is just a stencil buffer with more precision. It is likely that the LI buffer can be implemented by rendering to a HILO texture, where the two components are 16 bits each. For certain scenes, a 12-bit buffer may be sufficient, and another implementation could use the an 8-bit stencil buffer, at the cost of fewer shades in the penumbra region.

By multiplying each LI value with k, it is possible to get k different gray shade levels in the penumbra region. We use $k=255$ since color buffers typically are eight bits per component. This choice allows for at least 256 overlapping (e.g., in screen-space) penumbra wedges, which is more than sufficient for most applications. It is also worth noting that this is similar to commodity graphics hardware that often has a 8-bit stencil buffer, which thus also allows for 256 overlapping objects, using the the SV algorithm. The penumbra wedges add or subtract from the LI buffer. For example, when a ray from through a wedge (from light to umbra), 255 will be subtracted.

The algorithm starts by clearing the LI buffer to 255, which implies that the viewer is outside shadow. Then the entire scene is rendered with only diffuse and specular lighting. Penumbra wedges are then rendered independently of each other to the LI buffer using the conceptual pseudocode (not optimized for hardware) below, where the entry and exit points are illustrated in Figure 2. See also Figure 3 for an example of the $\mathbf{p}_i$ value used in the code below.

```
1 :  rasterizeWedge()
2 :  foreach visible fragment$(x,y)$...
3 :    ...on front facing triangles of wedge
4 :    $\mathbf{p}_f$ = computeEntryPointOnWedge$(x,y)$;
5 :    $\mathbf{p}_b$ = computeExitPointOnWedge$(x,y)$;
6 :    $\mathbf{p}$ = point$(x,y,z)$; $-z$ is the $Z$-buffer value at $(x,y)$
7 :    $\mathbf{p}_i$ = choosePointClosestToEye$(\mathbf{p},\mathbf{p}_b)$;
8 :    $s_f$ = computeLightIntensity$(\mathbf{p}_f)$;
9 :    $s_i$ = computeLightIntensity$(\mathbf{p}_i)$;
10 :   addToLIBuffer(round$(255*(s_i - s_f))$);
11 : end;
```

Lines 4 and 5 compute the points on the wedge where

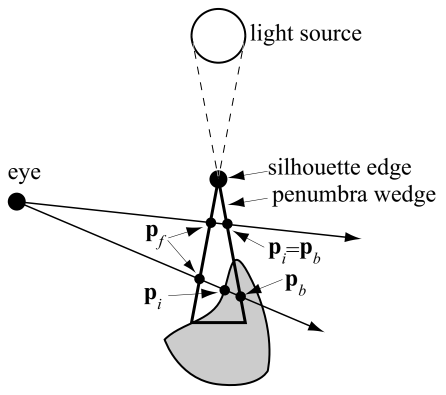

Figure 3: *Illustration of the* $\mathbf{p}_f$*,* $\mathbf{p}_b$*, and* $\mathbf{p}_i$ *values for two rays.*

the ray through the pixel at (x,y) enters (first intersection) and exits (second intersection) the wedge. A point is formed from (x,y,z), where z is the depth at (x,y) in the Z-buffer (line 6). If this point, transformed to world-space, is determined to be inside the wedge, then $\mathbf{p}_i$ is set equal to that point, as this is a point that is in the penumbra region. Otherwise, $\mathbf{p}_i$ is set to $\mathbf{p}_b$. This is done on line 7. Lines 8-9 compute the light intensity $[0,1]$ at the points, $\mathbf{p}_f$ and $\mathbf{p}_i$, and finally, the difference between these values are scaled with 255 and added to the LI buffer.

After all wedges have been rasterized to the LI buffer, the resulting image in the LI buffer is clamped to $[0,255]$, and used to modulate the rendered image (using diffuse and specular shading). This correctly avoids highlights in shadows. In a final pass, ambient lighting is added.

The clamping of the LI buffer is needed because it is possible to have overlapping penumbra wedges, e.g., it is possible to enter the umbra volume more than once. This would result in a negative LI value—clamping this to zero is correct, as the umbra volume cannot be darker than zero. LI values larger than 255 implies that we have gone out of shadow more than once—this is possible when the viewer is inside shadow to start with. Again clamping to 255 just means it cannot be lighter than being totally outside shadow.

In the following subsections, we discuss how penumbra wedges are constructed, and how light intensity interpolation is done.

4.1. Constructing Penumbra Wedges

In two dimensions, creation of penumbra wedges is trivial. In three dimensions it is more difficult. We approximate the penumbra volume that a silhouette edge gives rise to with a wedge defined by four planes: the *front*, *back*, *left side*, and *right side* planes. As Haines point out, a more correct shape would be a cone at each silhouette edge vertex, and two Coons patches connecting these.[7] The creation of the front and back planes is illustrated to the right in Figure 4, where the corresponding SV quad is shown the left.

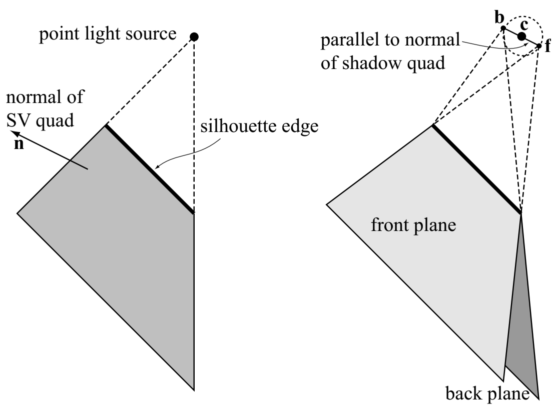

Figure 4: *Left: shadow volume quad. Right: front and back planes of a wedge.*

Assuming a spherical light with center $\mathbf{c}$ and radius r, two points are created as $\mathbf{b} = \mathbf{c} + r\mathbf{n}$ and $\mathbf{f} = \mathbf{c} - r\mathbf{n}$, where $\mathbf{n}$ is the normal of the SV quad. The front plane is then defined by $\mathbf{f}$ and the silhouette edge; and similarly for the back plane. Two adjacent wedges share one side plane, and it is created from these two wedges' front and back planes. See Figure 5. More specifically, a side plane is constructed from two adjacent wedges by finding the line of intersection of the two front planes. The same is done for the two back planes, and these two lines define the side plane between these wedges. An example of a wedge is shown to the left in Figure 9.

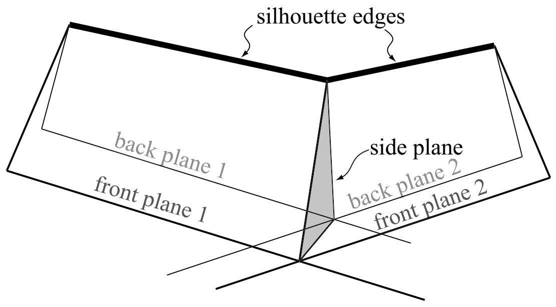

Figure 5: *Two adjacent wedges in general configuration. Their front and back planes define their shared side plane.*

For very large light sources, or sufficiently far away from the silhouette edge, the two side planes of a wedge may intersect. In such cases, the wedge is defined as shown in Figure 6.

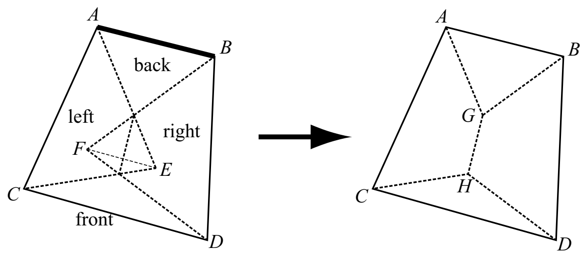

Figure 6: *Left: ABDC define the front plane's quadrilateral, and ABFE the back plane's quadrilateral, ACE the left side plane, and BFD the right side plane. The wedge on the right is used when rendering soft shadow, in cases where the side planes overlap.*

It should be noted that by simply setting the light source radius to zero, hard shadows can be rendered with our algorithm in the same way as the SV algorithm.

In Section 4.2, a ray direction that lies in each side plane is needed to make the interpolation across adjacent wedges continuous. This direction is shared by two adjacent wedges, and it is computed by taking the average of the two SV quad normals (whose corresponding silhouette edges share side plane), projecting it into the side plane, and then normalizing the resulting vector.

When two adjacent silhouette edges form an acute angle, the difference between our algorithm and Heckbert/Herf shadows is more obvious. However, those cases can easily be detected, and extra wedges around such vertices can be introduced, as in Figure 7, to create a better approximation. The number of extra wedges should depend on the angle between two adjacent silhouette edges: the smaller angle, the more extra wedges are introduced. It is worth noting that often the performance drop from using extra wedges around acute angles only was about 20 percent. This is because those wedges often are long and thin, and do not contribute much to the image, and are therefore cheap to render.

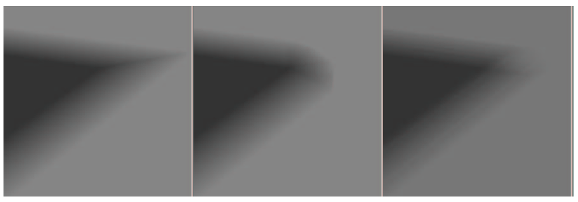

Figure 7: *A (partial) soft shadow of a triangle with an acute angle. Left: one wedge per silhouette edge. Middle: one wedge per silhouette edge plus 6 extra wedges around each vertex. Right: Heckbert/Herf shadows. Also, when comparing images on screen, a stepping effect of Heckbert/Herf shadows is apparent, while our algorithm inherently avoids stepping effects.*

4.2. Light Intensity Interpolation

In this section, we describe how the light intensity, s, for a point, $\mathbf{p}$, inside a penumbra wedge is computed. Recall that $\mathbf{p}$ is a point formed from the pixel coordinates, (x,y), and the depth, z, in the Z-buffer at that pixel. This is shown in Figure 8.

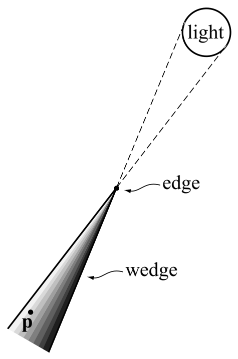

Figure 8: *The point* $\mathbf{p}$ *is in the penumbra wedge volume. The rationale for our interpolation scheme is that s should approximate how much the point* $\mathbf{p}$ *"sees" of the light source.*

Clearly, the minimal level of continuity of s between two adjacent wedges should be C^0. Our first attempt created a ray from $\mathbf{p}$ with the same direction as the normal of the SV quad. Then, the positive intersection distances, t_f and t_b, were found by computing the intersections between the ray and the front and the back plane, respectively. The light intensity was then computed as:

$$s = t_b/(t_f + t_b) \tag{1}$$

However, this does not guarantee C^0 continuity of the light intensity across adjacent wedges. Instead, the following approach is used. Two intermediate light intensities, s_l and s_r, are computed (similarly to the above) using **p** as the ray origin, and ray directions that lie in the left and right side plane, respectively (see Section 4.1 on how to construct these directions). See Figure 9. The computations are:

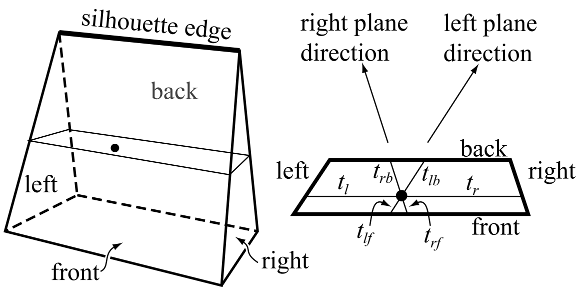

Figure 9: *Light intensity interpolation inside a penumbra wedge. Left: penumbra wedge. Right: cross-section of the wedge, where the positive intersection distances, t's, from the point (black dot) to the planes are shown.*

$$s_l = \frac{t_{lb}}{t_{lf} + t_{lb}}, \quad s_r = \frac{t_{rb}}{t_{rf} + t_{rb}} \tag{2}$$

The light intensity is linearly interpolated as below, where t_l and t_r are the positive intersection distances from **p** to the left and right side planes. The ray direction used for this is parallel to the silhouette edge.

$$s = \frac{t_r}{t_r + t_l} s_l + \frac{t_l}{t_r + t_l} s_r \tag{3}$$

Since the side directions are shared between adjacent wedges, this equation gives C^0 light intensity continuity. Also, we avoid any form of discretization (such as using a number of point samples on a light source) here, so the penumbra will always be smooth inside a wedge no matter how close to the shadow the viewer is. This choice of light intensity interpolation also has the added advantage that reciprocal dot products, used in ray/plane intersection to find the different t-values, can be precomputed at setup of the wedge rasterization in order to avoid divisions. Also, by simplifying and using the least common denominator in Equation 3, the number of divisions can be reduced to one per evaluation instead of four.

Parker et al.[13] report that the attenuation factor is a sinusoidal for spherical lights, and approximate it by $s' = 3s^2 - 2s^3$. This can easily be incorporated into our model as well.

5. Optimizations

In this section, several optimizations of the algorithm will be presented. As can be seen in the pseudocode in Section 4, a value of $s_i - s_f$ is added to the LI buffer for each rasterized fragment. The most expensive calculation in computing s_i and s_f is when Equation 3 needs to be evaluated. For points, (x, y, z), inside a wedge, this evaluation must be done. Here, we will present several other cases where this evaluation can be avoided.

When a ray enters (exits) a side plane, it will also exit (enter) a side plane on an adjacent wedge, and their LI values, s, will cancel out, and thus the LI values need not be computed. This is illustrated in Figure 10. Also, when entering

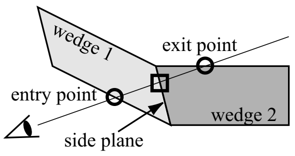

Figure 10: *A cross-section view through two adjacent wedges. The square shows where the ray intersects the shared side plane of the wedges. The LI values for wedge 1 and 2 in the shared side plane cancel each other.*

or exiting points are on front or back planes of the wedge, then we can simply use a value of 0 or 255, depending on entering/exiting and front/back planes. Using these two optimizations, we only evaluate Equation 3 for points inside the penumbra wedge, that is, where the computations contribute to the final image, which is minimal. Also, before rasterization of a wedge starts, we precompute several reciprocal dot products that are constant for the entire wedge, and used in Equation 3. The above optimizations gave about 50% faster wedge rasterization.

Visibility culling can also be done on the wedges. For each 8×8 Z-buffer region, the largest z-value, z_{max}, could be stored in a cache as presented by Morein.[12] Fragments on a front facing wedge triangle can thus be culled if the z-values are larger than z_{max}. This type of technique is implemented in commodity graphics hardware, such as ATI's Radeon and NVIDIA's GeForce3. Wedge rasterization (both hardware and software) can gain performance from using this technique.

All optimizations work for dynamic scenes as well, however, the wedges and the side direction vectors need to be recomputed when light sources or shadow casting geometry moves.

6. Implementation

The main objective of our current implementation was to prove that the algorithm generates plausible soft shadows reasonably fast. Since pretty large vertex and pixel shader programs are needed in order to implement this using graphics hardware, we need to await the next-generation graphics hardware before true real-time performance can be obtained.

Our current implementation works as follows. First, the scene is rendered using hardware-accelerated OpenGL. Wedge rasterization is implemented in software (SW), and therefore the Z-buffer is read out before rasterization starts. The front facing triangles of a wedge are rasterized using Pineda's edge function algorithm.[14] Since it thus is known which plane the rasterized wedge triangle belongs to, the plane of the entry point is known. The exit point is found by computing the intersection of the ray with all back facing planes, and picking the closest. The z-value is read, and a point in world space is formed by applying the (precomputed) screen-to-world transform. Thereafter, that point is inserted into all plane equations to determine whether the point is inside the wedge. If the point is inside the wedge, Equation 3 is evaluated by computing intersection distances from the point to the planes along the directions discussed in Section 4.2. We also implement the optimizations presented in Section 5, except for the culling techniques.

7. Results

In Figures 12 and 14, the major strength of our algorithm is shown, namely that soft shadows can be cast on arbitrarily complex shadow receivers. Note that only the spheres and the EG logo are casting shadows for the first figure, and only the "@" is casting shadow in the second figure. In Figure 12, a rather complex object is casting shadows on a complex receiver formed from several teapots, while the light source size is increased. As can be seen, the rendered images exhibit typical characteristics of soft shadows: the shadows are softer the farther away the occluder is from the receiver, and they are hard where the occluder is near the receiver. Furthermore, the umbra region becomes smaller and smaller with increasing light source size. At 512×512, those render at about 1.8 frames per seconds (fps).

To test the quality of our algorithm, we have compared it to both Heckbert/Herf (HH) shadows[9] with 128 samples, and Soler/Sillion (SS) shadows.[17] HH shadows are more precise given sufficiently many samples, and the ultimate goal is to render images like that in real time. The SS shadow algorithm is interesting to compare to, because it is targeted for real-time soft shadows. Some results are shown in Figure 12. The motivation for choosing such a simple scene is that we know what to expect, and that it still includes the most important effects of soft shadows (increasing penumbra width, etc). Despite the approximations introduced by our algorithm, the results are here remarkably similar to Heckbert and Herf's more precisely generated soft shadows. Our algorithm rendered those images at about 2 fps (in software), while HH shadows were rendered at about 20 fps (using hardware). Note, however, that there are two reasons why HH shadows are not really a feasible solution for real-time applications with dynamic objects. First, shadows can only be cast on planar surfaces. It is worth noting here that a soft shadow texture (generated on a plane) that is projected onto a curved surface cannot produce correct results. This is because the penumbra and umbra regions change in space in such a way that it does not correspond to a simple projection. Second, the rendering of 128 passes per frame consumes a lot of capacity of a graphics system that could be used for better tasks.

The SS shadows fail to produce believable results. This is because it only produces correct results for parallel configurations, and scenes (including this one) are in general not configured like that. To their advantage, both SS and HH shadows are image-based and therefore quite independent of shadow generating geometry, and they can also handle arbitrarily shaped light sources. Also, the SS shadow algorithm could split up the object into different cylinders to better capture the soft shadows, but it is highly likely that this would give rise to discontinuities in the shadows.

We have also implemented an approximation of our algorithm using current graphics hardware. See Figure 15. Each wedge is discretized with a number of quads sharing the silhouette edge and dividing the space between the front and back plane into different constant LI regions. This implementation render approximately concentric shadows, but a stepping effect can still be seen as for other sampling methods, and also a large amount of rasterization work is done. Everitt and Kilgard [3] implement a similar algorithm, but put samples on the light source in the Heckbert/Herf manner, and let each sample point add in shadow contribution without the need for an accumulation buffer.

Two lights are used in the test scene of Figure 16. The only modification we made to our algorithm was to multiply the light intensities, s, by $255/n$ instead of 255, where n is the number of lights. All test results are from our software implementation using a standard PC with an AMD Athlon 1.5 GHz, and a GeForce3 graphics card.

8. Discussion

Here we will discuss the limitations and possible artifacts of our algorithm.

In this paper, we have restricted the light source to be a sphere. Approximations of arbitrary, convex light sources are possible: when creating the front and back planes (which must pass through the silhouette edge), rotate these until they touch opposite sides of the light source. Our choice of light source shape restricts the number of applications, but certain applications, e.g., games, will most likely be satisfied. Also, the SV algorithm cannot handle non-polygonal shadow casting geometry, such as N-patches or textures with alpha, and neither can our algorithm. It is also worth noting that no shadow volume-based algorithm can handle transparent surfaces in a proper manner.

For all shadow volume algorithms, one must be careful when the viewer is in shadow. For hard shadows, this can be solved with the Z-fail technique. See Everitt and Kilgard[3] for a presentation on this. We have very recently solved this problem for our algorithm. Briefly, capping of the soft shadow volumes is needed, together with the Z-fail method, and with a restructured rendering algorithm. That technique will be described elsewhere due to space constraints.

One approximation is that we, as do Haines[7] and the classic SV algorithm, use the same silhouette for the entire volume light source. Since soft shadows are generated by area or volume light sources, the silhouette cannot in general be the same for all points on such a light source. Errors are visible, but only for very large light sources, and in practice, we have not found this to be a problem. The cost of the SV algorithm, Haines', and ours is to first find the silhouette edges of the model, and the rendering of the shadows is proportional to the number of silhouette edges and the area of the shadow primitives (e.g., wedges).

A silhouette edge is an edge that is connected to two triangles, where one triangle is facing toward the light, and the other facing away. Such silhouette edges form closed loops. Our algorithm can render shadows of geometry whose vertices in the silhouette edge lists only connects to two silhouette edges. However, this is not always the case. A vertex may connect to more than two silhouette edges. Currently, we do not handle this problem, and this limits the types of scenes that we can render. It may be possible to construct the wedges around such problematic vertices in other ways, or to interpolate shading differently there. We leave this for future work.

There may also be rays that pierce through a face on the wedge, but that do not exit through a wedge face. This occurs, for example, when the viewer is located close to the position of the light source. However, such rays do not pose any problem. The reason for this is that for any shadow volume algorithm to work properly, the shadow quads must penetrate the geometry of the scene to be rendered. The same holds for penumbra wedges: they must also intersect the geometry of the scene. This implies that rays that enter a wedge, must either hit geometry inside the wedge, or exit the wedge through one of the four wedge planes.

If a silhouette edge is nearly parallel or parallel to the direction of the incoming light, another problem may arise: the side plane construction will not be robust. To avoid this, we remove such edges, and shorten & connect its neighbors. This may give shadow artifacts near the shadow generating object.

When two objects overlap, as seen from a light source, it is very likely that wedges from these two objects also will overlap. Our algorithm automatically subtracts the light intensities from both wedges. This is not always correct. Sometimes it may be more correct to multiply their contributions, and sometimes it may be more correct to subtract only the contribution from one wedge (when wedges coincide). There does not seem to be a straightforward way to solve this. However, even though it is possible to see differences in images, it is often very hard to see which is correct. See Figure 11.

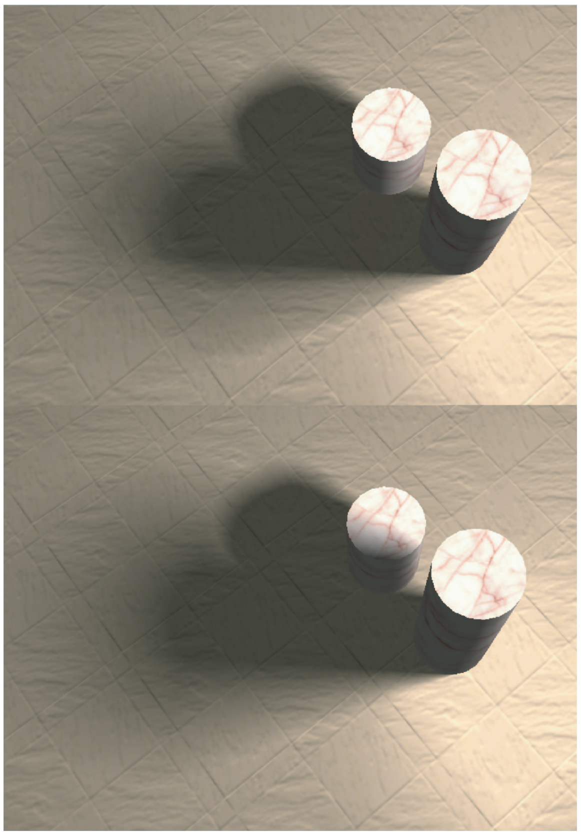

Figure 11: *Overlapping soft shadows. Top: rendered with Heckbert/Herf's algorithm with 128 samples. Bottom: result produced with our algorithm.*

As can be seen, there are several limitations of our algorithm. However, it should be noted that it is only recently that the standard shadow volume algorithm has matured so that it can handle all cases,[3] and a maturing process can be expected for our algorithm as well. Next, some ideas for future work, and some early initial results are presented.

9. Future Work

We are continuing to explore our algorithm, and the most valuable contribution to make in the future, would be to increase the complexity of geometrical models that can cast soft shadows. Currently, we are exploring several new ways of interpolating inside a wedge, and initial results show that several of the limitations from Section 8 can be overcome using different light intensity interpolation techniques. It remains to unify these in a single technique, and make it render rapidly.

Another avenue for future research is also to make more, and more accurate, comparisons to more algorithms, and to stress all algorithms. Finally, it will be interesting to implement this on graphics hardware that comes out within a year, which is expected to be massively programmable.

10. Conclusions

We have presented a new soft shadow algorithm that is an extension of the classical shadow volume algorithm. The shadow penumbra wedge is a new primitive that we have introduced, and that can be rasterized efficiently. The generated soft shadow images have been shown to often give similar results to the algorithm of Heckbert and Herf,[9] despite the approximations that we introduce. It is important to note that our algorithm can render soft shadows on arbitrary geometry. Also, the performance is independent of the receiving geometry since the contents of the *Z*-buffer is used as a receiver. The software implementation of our algorithm gives interactive rates on a standard PC. Thus, it seems highly likely that next-generation hardware would give real-time performance, which would increase the quality of real-time games and other applications. Therefore, we believe that this algorithm is a major leap forward for soft shadows in real time.

Acknowledgement

Thanks to Eric Haines, Kasper Høy Nielsen, and Jacob Ström for many good suggestions, and for improving our description.

References

1. Crow, Franklin C., "Shadow Algorithms for Computer Graphics," *SIGGRAPH 77 Proceedings*, pp. 242–248, July 1977. 2, 3

2. Drettakis, George, and Eugene Fiume, "A Fast Shadow Algorithm for Area Light Sources Using Back Projection," *SIGGRAPH 94 Proceedings*, pp. 223–230, July 1994. 3

3. Everitt, Cass, and Mark Kilgard, "Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering," `http://developer.nvidia.com/view.asp?IO=robust_shadow_volumes` 3, 7, 8

4. Fernando, R., S. Fernandez, L. Bala, and D. P. Greenberg, "Adaptive Shadow Maps," *SIGGRAPH 2001 Proceedings*, pp. 387–390, August 2001. 2

5. Gooch, Bruce, Peter-Pike J. Sloan, Amy Gooch, Peter Shirley, and Richard Riesenfeld, "Interactive Technical Illustration," *Proceedings 1999 Symposium on Interactive 3D Graphics*, pp. 31–38, April 1999. 2

6. Haines, Eric, and Tomas Möller, "Real-Time Shadows," *Game Developers Conference*, March 2001. 2

7. Haines, Eric, "Soft Planar Shadows Using Plateaus," *Journal of Graphics Tools*, vol. 6, no. 1, pp. 19–27, 2001. 2, 4, 8

8. Hart, David, Philip Dutre, and Donald P. Greenberg, "Direct Illumination with Lazy Visbility Evaluation," *SIGGRAPH 99 Proceedings*, pp. 147–154, August 1999. 2

9. Heckbert, P., and M. Herf, *Simulating Soft Shadows with Graphics Hardware,* Technical Report CMU-CS-97-104, Carnegie Mellon University, January 1997. 1, 2, 7, 9

10. Heidmann, Tim, "Real shadows, real time," *Iris Universe*, No. 18, pp. 23–31, Silicon Graphics Inc., November 1991. 3

11. Heidrich, W., S. Brabec, and H-P. Seidel, "Soft Shadow Maps for Linear Lights," *11th Eurographics Workshop on Rendering*, pp. 269–280, June 2000. 2

12. Morein, Steve, "ATI Radeon—HyperZ Technology," *SIGGRAPH/Eurographics Graphics Hardware Workshop 2000*, Hot3D session, 2000. 6

13. Parker, S., Shirley, P., and Smits, B., *Single Sample Soft Shadows*, TR UUCS-98-019, Computer Science Department, University of Utah, October 1998. 1, 2, 6

14. Pineda, Juan, "A Parallel Algorithm for Polygon Rasterization," *SIGGRAPH 88 Proceedings*, pp. 17–20, August 1988. 7

15. Reeves, William T., David H. Salesin, and Robert L. Cook, "Rendering Antialiased Shadows with Depth Maps," *SIGGRAPH 87 Proceedings*, pp. 283–291, July 1987. 2

16. Segal, M., C. Korobkin, R. van Widenfelt, J. Foran, P. and Haeberli, "Fast Shadows and Lighting Effects Using Texture Mapping," *SIGGRAPH 92 Proceedings*, pp. 249–252, July 1992. 2

17. Soler, Cyril, and François X. Sillion, "Fast Calculation of Soft Shadow Textures Using Convolution," *SIGGRAPH 98 Proceedings*, pp. 321–332, July 1998. 1, 2, 7

18. Stark, Michael M., and Richard F. Riesenfeld, "Exact Illumination in Polygonal Environments using Vertex Tracing," *Rendering Techniques 2000*, pp. 149–160, June 2000. 2

19. Williams, Lance, "Casting Curved Shadows on Curved Surfaces," *SIGGRAPH 78 Proceedings*, pp. 270–274, August 1978. 2

20. Woo, A., P. Poulin, and A. Fournier, "A Survey of Shadow Algorithms," *IEEE Computer Graphics and Applications*, vol. 10, no. 6, pp. 13–32, November 1990. 2

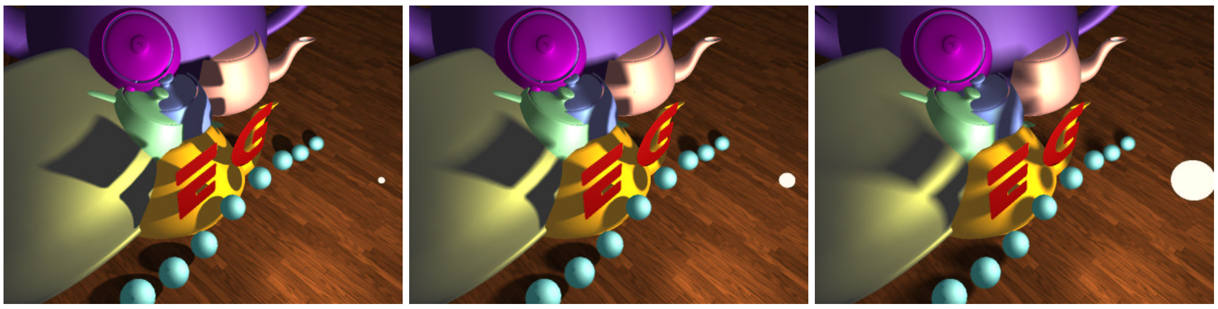

Figure 12: *Increasing light source size from left to right. Only the EG logo, and the spheres are casting shadows. Notice that the umbra region correctly gets smaller and smaller with increasing light source.*

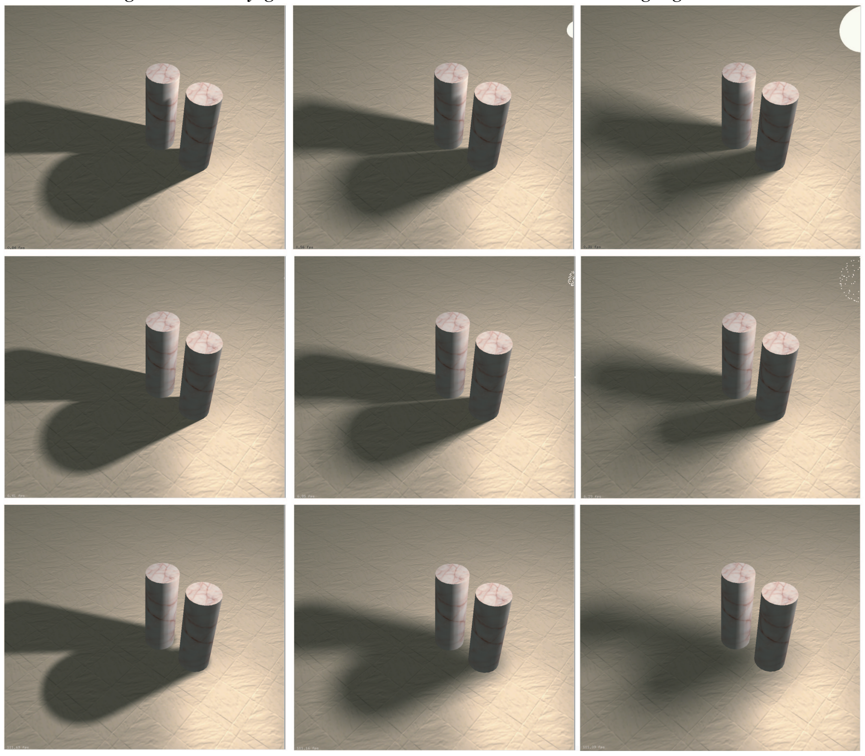

Figure 13: *Comparison of our algorithm (top), Heckbert/Herf (middle), and Soler/Sillion (bottom). Our algorithm provides the accuracy of the much more expensive Heckbert/Herf algorithm. In addition, our algorithm handles all surfaces, and so casts a shadow from the right cylinder onto the left, which the other two algorithms cannot do.*

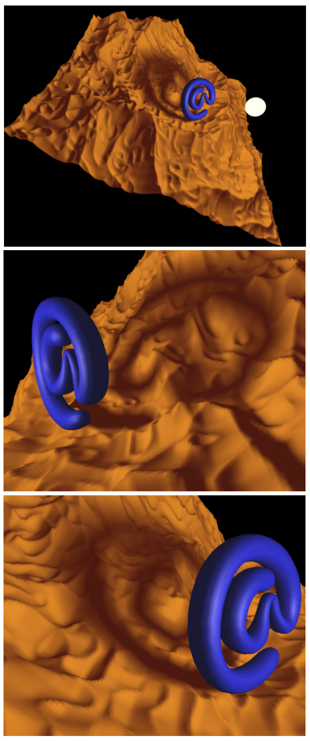

Figure 14: *A fractal landscape with 100k triangles is used as a complex shadow receiver from different viewpoints.*

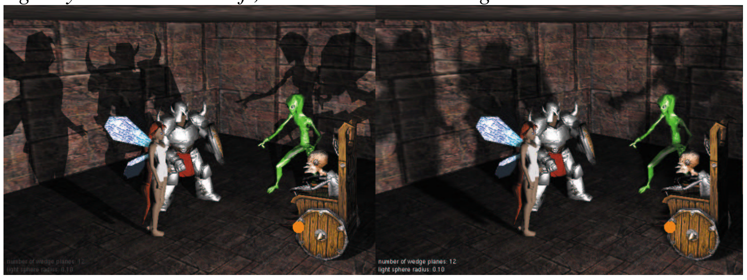
number of wedge planes: 12
light sphere radius: 0.10
number of wedge planes: 12
light sphere radius: 0.10

Figure 15: *Rendered at 5 fps on a 1.5 GHz PC with a Geforce3. We modified nVidia's shadow volume demo (left) to render soft shadows (right).*

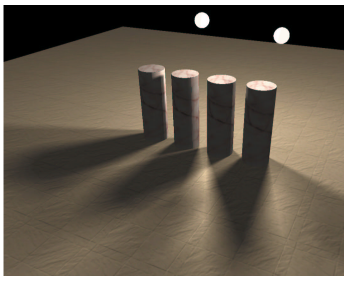

Figure 16: *Two light sources are used in this simple test scene.*