A Geometry-based Soft Shadow Volume Algorithm using Graphics Hardware

Ulf Assarsson   Tomas Akenine-Möller

Chalmers University of Technology
Sweden

Abstract

Most previous soft shadow algorithms have either suffered from aliasing, been too slow, or could only use a limited set of shadow casters and/or receivers. Therefore, we present a strengthened soft shadow volume algorithm that deals with these problems. Our critical improvements include robust penumbra wedge construction, geometry-based visibility computation, and also simplified computation through a four-dimensional texture lookup. This enables us to implement the algorithm using programmable graphics hardware, and it results in images that most often are indistinguishable from images created as the average of 1024 hard shadow images. Furthermore, our algorithm can use both arbitrary shadow casters and receivers. Also, one version of our algorithm completely avoids sampling artifacts which is rare for soft shadow algorithms. As a bonus, the four-dimensional texture lookup allows for small textured light sources, and, even video textures can be used as light sources. Our algorithm has been implemented in pure software, and also using the GeForce FX emulator with pixel shaders. Our software implementation renders soft shadows at 0.5–5 frames per second for the images in this paper. With actual hardware, we expect that our algorithm will render soft shadows in real time. An important performance measure is bandwidth usage. For the same image quality, an algorithm using the accumulated hard shadow images uses almost two orders of magnitude more bandwidth than our algorithm.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, Shading, Shadowing, and Texture

Keywords: soft shadows, graphics hardware, pixel shaders

1 Introduction

Soft shadow generation is a fundamental and inherently difficult problem in computer graphics. In general, shadows not only increase the level of realism in the rendered images, but also help the user to determine spatial relationships between objects. In the real world, shadows are often soft since most light sources have an area or volume. A soft shadow consists of an umbra, which is a region where no light can reach directly from the light source, and a penumbra, which is a smooth transition from no light to full light. In contrast, point light sources generate shadows without the penumbra region, so the transition from no light to full light is abrupt. Therefore, this type of shadow is often called a hard shadow. However, point light sources rarely exist in the real world. In addition to that, the hard-edged look can also be misinterpreted for geometric features, which clearly is undesirable. For these reasons, soft shadows in computer-generated imagery are in general preferred over hard shadows.

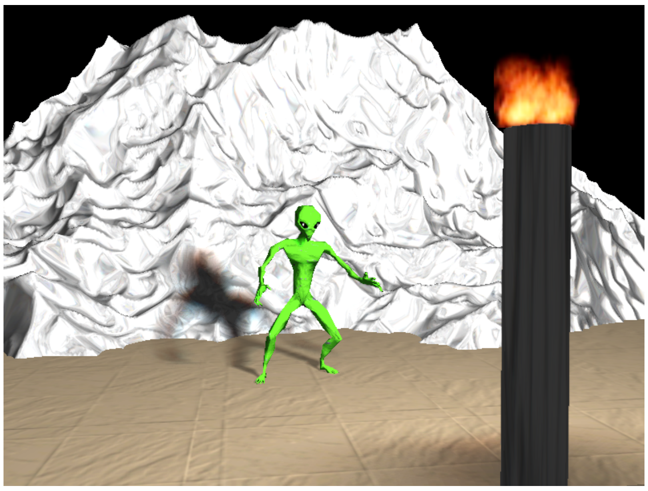


Figure 1: An image texture of fire is used as a light source, making the alien cast a soft shadow onto the fractal landscape. The soft shadow pass of this scene was rendered at 0.8 frames per second.

Previous algorithms for soft shadow generation have either been too slow for real-time purposes, or have suffered from aliasing problems due to the algorithm's image-based nature, or only allowed a limited set of shadow receivers and/or shadow casters. We overcome most of these problems by introducing a set of new and critical improvements over a recently introduced soft shadow volume algorithm [Akenine-Möller and Assarsson 2002]. This algorithm used penumbra wedge primitives to model the penumbra volume. Both the construction of the penumbra wedges and the visibility computation inside the penumbra wedges were empirical, and this severely limited the set of shadow casting objects that could be used, as pointed out in that paper. Also, the quality of the soft shadows only matched a high-quality rendering for a small set of scenes. Our contributions include the following:

1. geometry-based visibility computation,
2. a partitioning of the algorithm that allows for implementation using programmable shaders,
3. robust penumbra wedge computation, and
4. textured and video-textured light sources.

All this results in a robust algorithm with the ability to use arbitrary shadow casters and receivers. Furthermore, spectacular effects are obtained, such as light sources with textures on them, where each texel acts as a small rectangular light source. A sequence of textures, here called a video texture, can also be used as a light source. For example, images of animated fire can be used as seen in Figure 1. In addition to that, the quality of the shadows is, in the majority of cases, extremely close to that of a high-quality rendering (using 1024 point samples on the area light source) of the same scene.

The rest of the paper is organized as follows. First, some previous work is reviewed, and then our algorithm is presented, along with implementation details. In Section 5, results are presented together with a discussion, and the paper ends with a conclusion and suggestions for future work.

2 Previous Work

The research on shadow generation is vast, and here, only the most relevant papers will be referenced. For general information about the classical shadow algorithms, consult Woo et al.'s survey [Woo et al. 1990]. A more recent presentation covering real-time algorithms is also available [Haines and Möller 2001].

There are several algorithms that generate soft shadows on planar surfaces. Heckbert and Herf average hard shadows into an accumulation buffer from a number of point samples on area light sources [Heckbert and Herf 1997]. These images can then be used as textures on the planar surfaces. Often between 64 and 256 samples are needed, and therefore the algorithm is not perfectly suited for animated scenes. Haines presents a drastically different algorithm, where a hard shadow is drawn from the center of the light source [Haines 2001]. Each silhouette vertex, as seen from the light source, then generates 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. Between two such neighboring cones, a Coons patch is "drawn" with similar light intensities. Haines notes that the umbra region is overstated.

For real-time rendering of hard shadows onto curved surfaces, shadow mapping [Williams 1978] and shadow volumes [Crow 1977] are probably the two most widely used algorithms. The shadow mapping (SM) algorithm generates a depth buffer, the shadow map, as seen from the light source, and then, during rendering from the eye, this depth buffer is used to determine if a point is in shadow or not. Reeves et al. presented percentage-closer filtering, which reduces aliasing along shadow boundaries [Reeves et al. 1987]. A hardware implementation of SM has been presented [Segal et al. 1992], and today most commodity graphics hardware (e.g., NVIDIA GeForce3) has SM with percentage-closer filtering implemented.

To reduce resolution problems with SM algorithms, both adaptive shadow maps [Fernando et al. 2001] and perspective shadow maps have been proposed [Stamminger and Drettakis 2002]. By using more than one shadow map, and interpolating visibility, soft shadows can be generated as well [Heidrich et al. 2000]. Linear lights were used, and more shadow maps had to be generated in complex visibility situations to guarantee a good result. Recently, another soft version of shadow mapping has been presented [Brabec and Seidel 2002], which adapts Parker et al's algorithm [1998] for ray tracing soft shadows so that graphics hardware could be used. The neighborhood of the sample in the depth map is searched until a blocker or a maximum radius is found. This gives an approximate penumbra level. The rendered images suffered from aliasing. Another image-based soft shadow algorithm uses layered attenuation maps [Agrawala et al. 2000]. Interactive frame rates (5–10 fps) for static scenes were achieved after seconds (5-30) of precomputation, and for higher-quality images, a coherent ray tracer was presented.

The other classical real-time hard shadow algorithm is shadow volumes [Crow 1977], which can be implemented using the stencil buffer on commodity graphics hardware [Heidmann 1991]. We refer to this algorithm as the hard shadow volume algorithm. In a first pass, the scene is rendered using ambient lighting. The second pass generates a shadow quadrilateral (quads) for each silhouette edge, as seen from the light source. The definition of a silhouette edge is that one of the triangles that are connected to it must be backfacing with respect to the light source, and the other must be frontfacing. Those shadow quads are rendered as seen from the eye, where front facing shadow quads increment the stencil buffer, and back facing decrement. After rendering all quads, the stencil buffer holds a mask, where zeroes indicate no shadow, and numbers larger than zero indicate shadow. A third pass, then renders the scene with full lighting where there is not shadow. Everitt and Kilgard have presented algorithms to make the shadow volume robust, especially for cases when the eye is inside shadow [Everitt and Kilgard 2002].

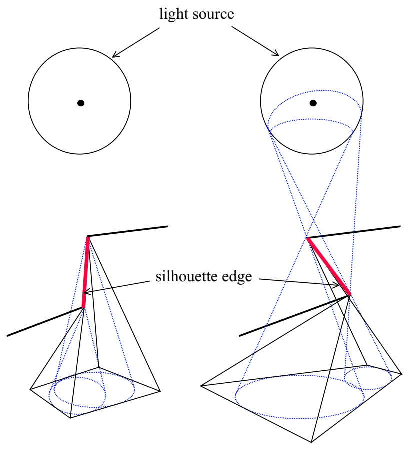


Figure 2: Difficult situations for the previous wedge generation algorithm. Left: The edge nearly points towards the light center, resulting in a non-convex wedge. Right: The edge is shadowed by one adjacent edge that is closer to the light source, making the left and right planes intersect inside the wedge. Unfortunately, the two tops of the cones cannot be swapped to make a better-behaved wedge, because that results in a discontinuity of the penumbra wedges between this wedge and the adjacent wedges.

Recently, a soft shadow algorithm has been proposed that builds on the shadow volume framework [Akenine-Möller and Assarsson 2002]. Each silhouette edge as seen from the light source gives rise to a penumbra wedge, and such a penumbra wedge empirically models the visibility with respect to the silhouette edge. However, as pointed out in that paper, the algorithm had several severe limitations. Only objects that had really simple silhouettes could be used as shadow casters. The authors also pointed out that robustness problems could occur in their penumbra wedge construction because adjacent wedges must share side planes. Furthermore, robustness issues occurred for the edge situations depicted in Figure 2. The latter problems were handled by eliminating such edges from the silhouette edge loop, making it better-shaped. The drawback is that the silhouette then no longer is guaranteed to follow the geometry correctly, with visual artifacts as a result. Their visibility computation was also empirical. All these problems are eliminated with our algorithm. Our work builds upon that penumbra wedge based algorithm. The hard shadow volume algorithm has also been combined with a depth map [Brotman and Badler 1984], where 100 point samples were used to generate shadow volumes. The overlap of these were computed using a depth map, and produced soft shadows.

Soft shadows can also be generated by back projection algorithms. However, such algorithms are often very geometrically complex. See Drettakis and Fiume [1994] for an overview of existing work. By convolving an image of the light source shape with a hard shadow, a soft shadow image can be generated for planar configurations (a limited class of scenes) [Soler and Sillion 1998]. An error-driven hierarchical algorithm is presented based on this observation. Hart et al. presented an algorithm for computing direct illumination base on lazy evaluation [1999], with rendering times of several minutes even for relatively simple scenes. Parker et al. rendered soft shadows at interactive rates in a parallel ray tracer using "soft-edged" objects at only one sample per pixel [1998]. Sloan et al. precompute radiance transfer and then renders several difficult light transport situations in low-frequency environments [2002]. This includes real-time soft shadows.

3 New Algorithm

Our algorithm first renders the scene using specular and diffuse lighting, and then a *visibility pass* computes a soft visibility mask in a visibility buffer (V-buffer), which modulates the image of the first pass. In a final pass, ambient lighting is added. This is illustrated in Figure 3.

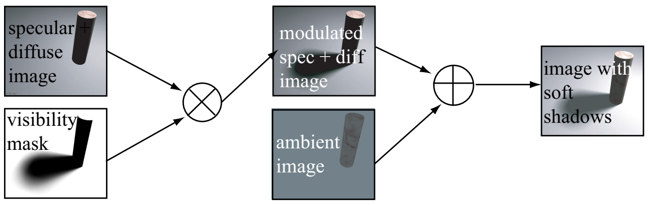


Figure 3: Overview of how the soft shadow algorithm works. Our work focuses on rapidly computing a visibility mask using a V-buffer, as seen from the eye point.

The V-buffer stores a *visibility factor*, $\bar{v}$, per pixel (x,y). If the point $\mathbf{p} = (x,y,z)$, where z is the Z-buffer value at pixel (x,y), can "see" all points on a light source, i.e., without occlusion, then $\bar{v} = 1$. This is in contrast to a point that is fully occluded, and thus has $\bar{v} = 0$. A point that can see x percent of a light source has $\bar{v} = x/100 \in [0,1]$. Thus, if a point has $0 < \bar{v} < 1$, then that point is in the penumbra region.

Next, we describe our algorithm in more detail, and the first part uses arbitrary light sources. However, in Section 3.2.2 and in the rest of the paper, we focus only on using rectangular light sources, since these allow for faster computations.

3.1 Construction of Penumbra Wedges

One approximation in our algorithm is that we only use the shadow casting objects' silhouette edges as seen from a single point, often the center, of the light source. This approximation has been used before [Akenine-Möller and Assarsson 2002], and its limitations are discussed in Section 5. Here, a silhouette edge is connected to two triangles; one frontfacing and the other backfacing. Such silhouettes can be found by a brute-force algorithm that tests the edges of all triangles. Alternatively, one can use a more efficient algorithm, such as the one presented by Markosian et al. [1997]. This part of the algorithm is seldom a bottleneck, but may be for high density meshes.

For an arbitrary light source, the *exact penumbra volume* generated by a silhouette edge is the swept volume of a general cone from one vertex of the edge to the other. The cone is created by reflecting the light source shape through the sweeping point on the edge. This can be seen in Figure 4a. Computing exact penumbra volumes is not feasible for real-time applications with dynamic

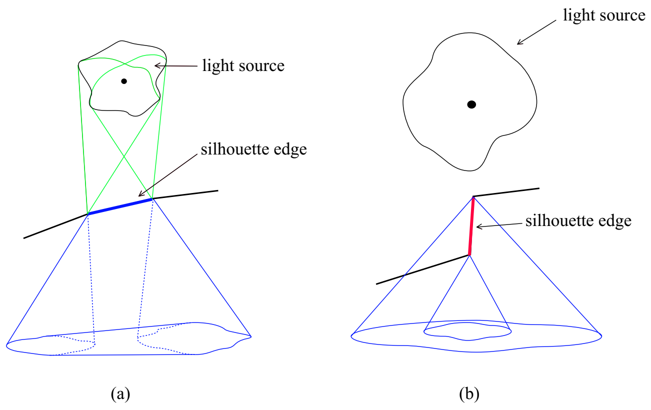
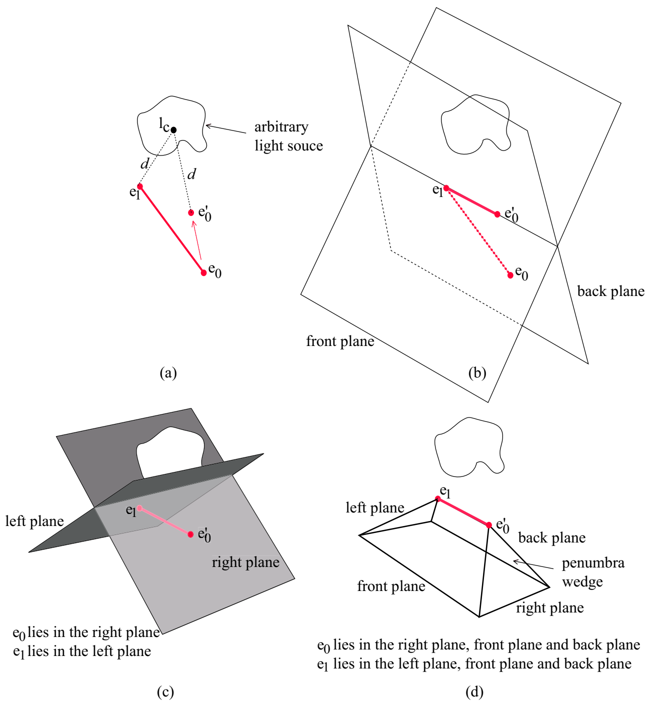


Figure 4: a) The penumbra volume generated by an edge. b) The penumbra volume can degenerate to a single cone, i.e., one end cone completely encloses the other end cone.

environments. However, we do not need the exact volume. In Section 3.2 we show that the computations can be arranged so that the visibility of a point inside a wedge can be computed independently of other wedges. It is then sufficient to create a bounding volume that fully encloses the exact penumbra volume. We chose a penumbra wedge defined by four planes (front, back, right, left) as our bounding volume [Akenine-Möller and Assarsson 2002] as seen in Figure 5d. It is worth noting that the penumbra volume will degenerate to a single cone, when one of the end cones completely enclose the other end cone (see Figure 4b).

To efficiently and robustly compute a wedge enclosing the exact penumbra volume, we do as follows. A silhouette edge is defined by two vertices, $\mathbf{e}_0$ and $\mathbf{e}_1$. First, we find the edge's vertex that is closest to the light source. Assume that this is $\mathbf{e}_1$, without loss of generality. The other edge vertex is moved along the direction towards the light center until it is at the same distance as the first vertex. This vertex is denoted $\mathbf{e}'_0$. These two vertices form a new edge which will be the top of the wedge. See Figure 5a. Note that this newly formed edge is created to guarantee that the wedge contains the entire penumbra volume of the original edge, and that the original edge still is used for visibility computation. As we will see in the next subsection, points inside the wedge but outside the real penumbra volume will not affect visibility as can be expected. Second, the front plane and back plane are defined as containing the new edge, and both these planes are rotated around that edge so that they barely touch the light source on each side. This is shown in Figure 5b. The right plane contains $\mathbf{e}'_0$ and the vector that is perpendicular to both vector $\mathbf{e}_1\mathbf{e}'_0$ and the vector from $\mathbf{e}'_0$ to the light center. The left plane is defined similarly, but on the other side. Furthermore, both planes should also barely touch the light source on each side. Finally, polygons covering the faces on the penumbra wedge are extracted from the planes. These polygons will be used for rasterization of the wedge, as described in Section 3.2. See Figure 6 for examples of constructed wedges from a simple shadow casting object.

An advantageous property of this procedure is that the wedges are created independently of each other, which is key to making the algorithm robust, simple, and fast. Also, note that when a silhouette edge's vertices are significantly different distances from the light source, then the bounding volume will not be a tight fit. While this still will result in a correct image, unnecessarily many pixels will be rasterized by the wedge. However, the majority of time is spent on the points inside the exact penumbra volume generated by the silhouette edge, and more such points are not included by making the wedge larger. It should be pointed out that if the front and back



l_c
arbitrary light souce
d
d
e_1
e'_0
e_0
(a)
back plane
front plane
(b)
left plane
right plane
e_0 lies in the right plane
e_1 lies in the left plane
(c)
left plane
back plane
penumbra wedge
front plane
right plane
e_0 lies in the right plane, front plane and back plane
e_1 lies in the left plane, front plane and back plane
(d)

Figure 5: Wedge construction steps. a) Move the vertex furthest from the light center l_c towards l_c to the same distance as the other vertex. b) Create the front and back planes. c) Create the left and right planes. d) The final wedge is the volume inside the front, back, left, and right planes.

planes are created so that they pass through both $\mathbf{e}_0$ and $\mathbf{e}_1$, the wedge would, in general, not fully enclose the penumbra volume. That is why we have to use an adjusted vertex, as described above. Currently, we assume that geometry does not intersect light sources. However, a geometrical objects may well surround the light source.

3.2 Visibility Computation

The visibility computation is divided into two passes. First, the hard shadow quads, as used by the hard shadow volume algorithm [Crow 1977], are rendered into the V-buffer in order to overestimate the umbra region, and to detect entry/exit events into the shadows. Secondly, the penumbra wedges are rendered to compensate for the overstatement of the umbra. Together these passes render the soft shadows. In the following two subsections, these two passes are described in more detail. However, first, some two-dimensional examples of how these two passes cooperate are given, as this simplifies the rest of the presentation.

In Figure 7, an area light source, a shadow casting object, and two penumbra wedges are shown. The visibility for the points **a**, **b**, **c**, and **d**, are computed as follows. For points **a** and **b**, the algorithm works exactly as the hard shadow volume algorithm, since both these points lie outside both penumbra wedges. For point **a**, both the left and the right hard shadow quads are rendered, and since the left is front facing, and the right is back facing, point **a** will be outside shadow. Point **b** is only behind the left hard shadow quad, and is therefore fully in shadow (umbra). For point **c**, the left hard shadow quad is rendered, and then during wedge rendering, **c** is found to be inside the left wedge. Therefore, **c** is projected onto the light source through the left silhouette point to find out how much to compensate in order to compute a more correct visibility factor. Point **d** is in front of all hard shadow quads, but it is inside

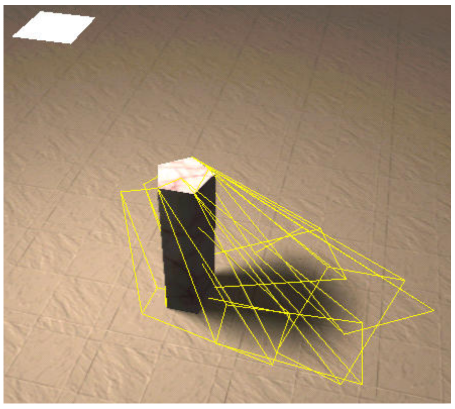


Figure 6: The wedges for a simple scene. At 512 × 512 resolution, this image was rendered at 5 frames per second using our software implementation. Note that there are wedges whose adjusted top edges differ a lot from the original silhouette edge. This can especially be seen to the left of the cylinder. The reason for this is that the vertices of the original silhouette edge are positioned with largely different distances to the light source.

the left wedge, and therefore **d** is projected onto the light source as well. Finally, its visibility factor compensation is computed, and added to the V-buffer.

3.2.1 Visibility Pass 1

Initially, the V-buffer is cleared to 1.0, which indicates that the viewer is outside any shadow regions. The hard shadow quads used by the hard shadow volume algorithm are then rendered exactly as in that algorithm, i.e., for front facing quads 1.0 is subtracted per pixel from the V-buffer, and for back facing quads, 1.0 is added.

An extremely important property of using the hard shadow quads, is that the exact surface between the penumbra and the umbra volumes is not needed. As mentioned in Section 3.1, computing this surface in three dimensions is both difficult and time-consuming. Our algorithm simplifies this task greatly by rendering the hard shadow volume, and then letting the subsequent pass compensate for the penumbra region by rendering the penumbra wedges. It should be emphasized that this first pass must be included, otherwise one cannot detect whether a point is inside or outside shadows, only whether a point is in the penumbra region or not. The previous algorithm [Akenine-Möller and Assarsson 2002] used an overly simplified model of the penumbra/umbra surface,

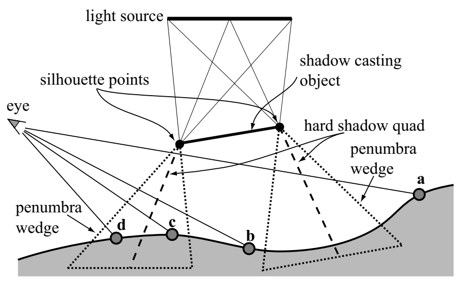


Figure 7: A two-dimensional example of how the two passes in computing the visibility cooperate. The two penumbra wedges, generated by the two silhouette points of the shadow casting object, are outlined with dots.

which was approximated by a quad per silhouette edge. This limitation is removed by our two-pass algorithm.

3.2.2 Visibility Pass 2

In this pass, our goal is to compensate for the overstatement of the umbra region from pass 1, and to compute visibility for all points, $\mathbf{p} = (x, y, z)$, where z is the z-buffer value at pixel (x, y), inside each wedge. In the following we assume that a rectangular light source, L, is used, and that the hard shadow quads used in pass 1, were generated using a point in the middle of the rectangular light source. To compute the visibility of a point, $\mathbf{p}$, with respect to the set of silhouette edges of a shadow casting object, imagine that a viewer is located at $\mathbf{p}$ looking at L. The visibility of $\mathbf{p}$ is then the area of the light source that the viewer can see, divided by total light source area [Drettakis and Fiume 1994].

Assume that we focus on a single penumbra wedge generated by a silhouette edge, $\mathbf{e}_0\mathbf{e}_1$, and a point, $\mathbf{p}$, inside that wedge. Here, we will explain how the visibility factor for $\mathbf{p}$ is computed with respect to $\mathbf{e}_0\mathbf{e}_1$, and then follows an explanation of how the collective visibility of all wedges gives the appearance of soft shadows. First, the semi-infinite hard shadow quad, Q, through the edge is projected, as seen from $\mathbf{p}$, onto the light source. This projection consists of the projected edge, and from each projected edge endpoint an infinite edge, parallel with the vector from the light source center to the projected edge endpoint, is extended outwards. This can be seen to the left in Figure 8. Second, the area of the intersection between the light source and the projected hard shadow quad is computed and divided by the total light source area. We call this the *coverage*, which is dark gray in the figure. For exact calculations, a simple clipping algorithm can be used. However, as shown in Section 3.3, a more efficient implementation is possible.

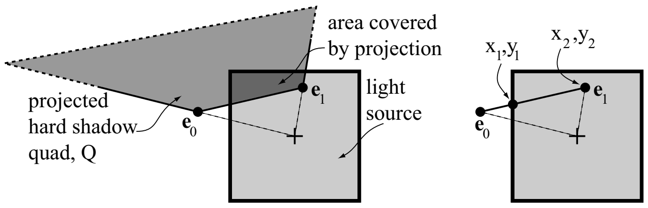


Figure 8: Left: Computation of coverage (dark gray area divided by total area) of a point $\mathbf{p}$ with respect to the edge $\mathbf{e}_0\mathbf{e}_1$. A three-dimensional view is shown, where $\mathbf{p}$ looks at the light source center. It can also be thought of as the projection of the hard shadow quad, Q, onto the light source as seen from $\mathbf{p}$. Note that Q, in theory, should be extended infinitely outwards from $\mathbf{e}_0\mathbf{e}_1$, and this is shown as dashed in the figure. Right: the edge is clipped against the border of the light source. This produces the 4-tuple (x_1, y_1, x_2, y_2) which is used as an index into the four-dimensional coverage texture.

The pseudo code for rasterizing a wedge becomes quite simple as shown below.

```
 1 : rasterizeWedge(wedge $W$, hard shadow quad $Q$, light $L$)
 2 : for each pixel $(x,y)$ covered by front facing triangles of wedge
 3 :   $\mathbf{p}$ = point$(x,y,z)$;  // $z$ is depth buffer value
 4 :   if $\mathbf{p}$ is inside the wedge
 5 :     $v_\mathbf{p}$ = projectQuadAndComputeCoverage$(W,\mathbf{p},Q)$;
 6 :     if $\mathbf{p}$ is in positive half space of Q
 7 :       $\overline{v}(x,y) = \overline{v}(x,y) - v_\mathbf{p}$;  // update V−buffer
 8 :     else
 9 :       $\overline{v}(x,y) = \overline{v}(x,y) + v_\mathbf{p}$;  // update V−buffer
10 :     end;
11 :   end;
12 : end;
```

When this code is used for all silhouettes, the visible area of the light source is essentially computed using Green's theorem. If line 4 is true, then $\mathbf{p}$ might be in the penumbra region, and more computations must be made. Line 5 computes the coverage, i.e., how much of the area light source that the projected quad covers. This corresponds to the dark region in Figure 8 divided by the area of the light source. The plane of Q divides space into a negative half space, and a positive half space. The negative half space is defined to be the part that includes the umbra. This information is needed in line 6 to determine what operation ($+$ or $-$) should be used in order to evaluate Green's theorem. An example of how this works can be seen in Figure 9, which shows the view from point $\mathbf{p}$ looking towards the light source. The gray area is an occluder, i.e., a shadow

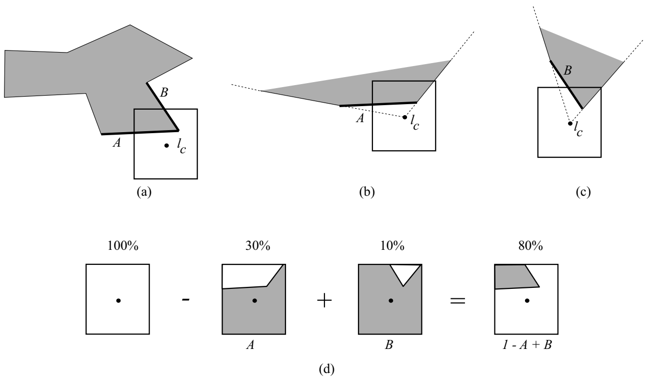


Figure 9: Splitting of shadow contribution for each wedge for a point $\mathbf{p}$. A and B are two silhouette edges of a shadow casting object.

casting object, as seen from $\mathbf{p}$. Both edge A and B contribute to the visibility of $\mathbf{p}$. By setting the contributions from A and B to be those of the virtual occluders depicted in Figure 9b and c, using the technique illustrated in Figure 8, the visible area can be computed without global knowledge of the silhouette.

3.3 Rapid Visibility Computation using 4D Textures

The visibility computation for rectangular light sources presented in Section 3.2 can be implemented efficiently using precomputed textures and pixel shaders.

The visibility value for a wedge and a point $\mathbf{p}$ depends on how the edge is projected onto the light source. Furthermore, it only depends on the part of the projected edge that lies inside the light source region (left part of Figure 8). Therefore, we start by projecting the edge onto the light source and clipping the projected edge against the light source borders, keeping the part that is inside.

The two end points of the clipped projected edge, (x_1, y_1) and (x_2, y_2), can together be used to index a four-dimensional lookup table. See the right part of Figure 8. That is, $f(x_1, y_1, x_2, y_2)$ returns the coverage with respect to the edge. This can be implemented using dependent texture reads if we discretize the function f. We strongly believe that this is the "right" place to introduce discetization, since this function varies slowly.

Now, assume that the light source is discretized into $n \times n$ texel positions, and that the first edge end point coincides with one of these positions, say $(x_1 = a, y_1 = b)$, where a and b are integers. The next step creates an $n \times n$ *subtexture* where each texel position represents the coordinates of the second edge end point, (x_2, y_2). In each of these texels, we precompute the actual coverage with respect to $(x_1 = a, y_1 = b)$ and (x_2, y_2). This can be done with exact clipping as described in Section 3.2.2. We precompute $n \times n$ such $n \times n$ subtextures, and store these in a single two-dimensional texture, called a *coverage texture*, as shown in Figure 10. At runtime, we compute (x_1, y_1) and round to the nearest texel centers, which

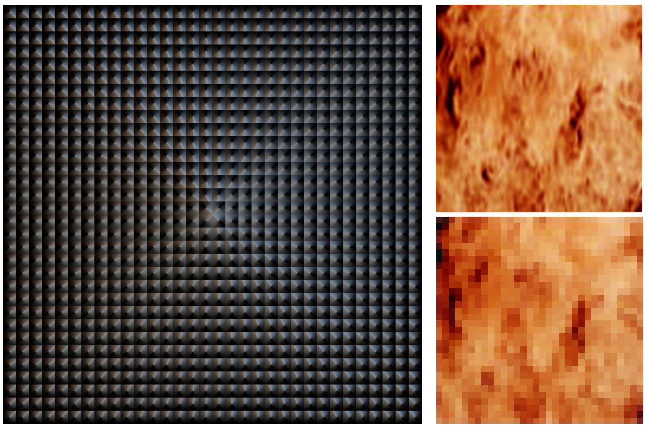


Figure 10: Left: example of precomputed coverage texture with $n = 32$. Top right: the original fire image. Bottom right: an undersampled 32×32 texel version of the original fire texture, used as light texture when computing the coverage texture. Each of the small 32×32 squares in the coverage texture are identified by the first edge end point, (x_1, y_1), and each texel in such a square corresponds to the coordinate of the second edge end point, (x_2, y_2).

is used to identify which of the $n \times n$ subtextures that should be looked up. The second edge end point is then used to read the coverage from that subtexture. To improve smoothness, we have experimented with using bilinear filtering while doing this lookup. We also implemented bilinear filtering for (x_1, y_1) in the pixel shader. This means that the four texel centers closest to (x_1, y_1) are computed, and that four different subtextures are accessed using bilinear filtering. Then, these four coverage values are filtered, again, using bilinear filtering. This results in quadlinear filtering. However, our experience is that for normal scenes, this filtering is not necessary. It could potentially be useful for very large light sources, but we have not verified this yet.

In practice, we use $n = 32$, which results in a 1024×1024 texture, which is reasonable texture usage. This also results in high quality images as can be seen in Section 5. With a Pentium4 1.7 GHz processor, the precomputation of one such coverage texture takes less than 3 minutes with a naive implementation.

Our technique using precomputed four-dimensional coverage textures can easily be extended to handle light sources with textures on them. In fact, even a sequence of textures, here called a video texture, can be used. Assume that the light source is a rectangle with an $n \times n$ texture on it. This two-dimensional texture can act as a light source, where each texel is a colored rectangular light. Thus, the texture defines the colors of the light source, and since a black texel implies absence of light, the texture also indirectly determines the shape of the light source. For instance, the image of fire can be used. To produce the coverage texture for a colored light source texture, we do as follows. Assume, we compute only, say, the red component. For each texel in the coverage texture, the sum of the red components that the corresponding projected quad covers is computed and stored in the red component of that texel. The other components are computed analogously.

Since we store each color component in 8 bits in a texel, a coverage texture for color-textured light sources requires 3 MB[1] of storage when $n = 32$. For some applications, it may be reasonable to download a 3MB texture to the graphics card per frame. To decrease bandwidth usage to texture memory, blending between two coverage textures is possible to allow longer time between texture downloads. However, for short video textures, all coverage textures can fit in texture memory.

[1]For some hardware, 24 bit textures are stored in 32 bits, so for these cases, the texture usage becomes 4 MB.

4 Implementation

We have implemented the algorithm purely in software with exact clipping as described in Section 3.2, and also with coverage textures. The implementation with clipping avoids all sampling artifacts. However, our goal has been to implement the algorithm using programmable graphics hardware as well. Therefore this section describes two such implementations.

The pixel shader implementations were done using NVIDIA's Cg shading language and the GeForce FX emulator. Here follow descriptions of implementations using both 32 and 8 bits for the V-buffer. For both versions, the pixel shader code is about 250 instructions.

4.1 32-bit version

For the V-buffer, we used the 32-bit floating point texture capability with one float per r, g, and b. This allows for managing textured light sources and colored soft shadows. If a 16-bit floating point texture capability is available, it is likely that those would suffice for most scenes.

The GeForce FX does not allow reading from and writing to the same texture in the same pass. This complicates the implementation. Neither does it allow blending to a floating point texture. Therefore, since each wedge is rendered one by one into the V-buffer in order to add its shadow contribution, a temporary rendering buffer must be used. For each rasterized pixel, the existing shadow value in the V-buffer is read as a texture-value and is then added to the new computed shadow contribution value and written to the temporary buffer. The region of the temporary buffer corresponding to the rasterized wedge pixels are then copied back to the V-buffer.

We chose to implement the umbra- and penumbra contribution in two different rendering passes (see Section 3.2) using pixel shaders. These passes add values to the V-buffer and thus require the use of a temporary rendering buffer and a succeeding copy-back. In total, this means that 4 rendering passes (the umbra and penumbra passes and two copy-back passes) are required for each wedge.

4.2 8-bit version

We have also evaluated an approach using an accuracy of only eight bits for the visibility buffer. Here, only one component (i.e., intensity) could be used in the coverage texture. One advantage is that no copy-back passes are required. Six bits are used to get 64 levels in the penumbra, and two bits are used to manage overflow that may arise when several penumbra regions overlap. The penumbra contribution is rendered in a separate pass into the frame buffer. All additive contribution is rendered to the red channel and all subtractive contribution is rendered as positive values to the green channel using ordinary additive blending in OpenGL. Then, the frame buffer is read back and the two channels are subtracted by the CPU to create a visibility mask, as shown in Figure 3. In the future, we plan to let the hardware do this subtraction for us without read-back to the CPU. The umbra contribution is rendered using the stencil buffer and the result is merged into the visibility mask. Finally, the visibility mask is used to modulate the diffuse and specular contribution in the final image.

5 Results and Discussion

In this section, we first present visual and performance results. Then follows a discussion of, among other things, possible artifacts that can appear.

5.1 Visual Results

To verify our visual results, we often compare against an algorithm that places a number, e.g., 1024, of point light samples on an area light source, and renders a hard shadow image for each sample. The hard shadow volume algorithm is used for this. The average of all these images produces a high-quality soft shadow image. To shorten the text, we refer to this as, e.g., "1024-sample shadow."

Figure 11 compares the result of the previously proposed penumbra wedge algorithm [Akenine-Möller and Assarsson 2002] that this work is based upon, our algorithm, and a 1024-sample shadow. As can be seen, our algorithm provides a dramatic increase in soft shadow quality over the previous soft shadow volume algorithm, and our results are also extremely similar to the 1024-sample shadow image.

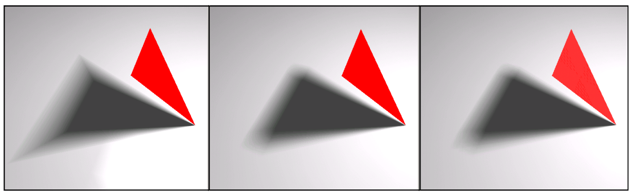


Figure 11: Comparison of the previous penumbra wedge algorithm, our algorithm, and using 1024 point light samples.

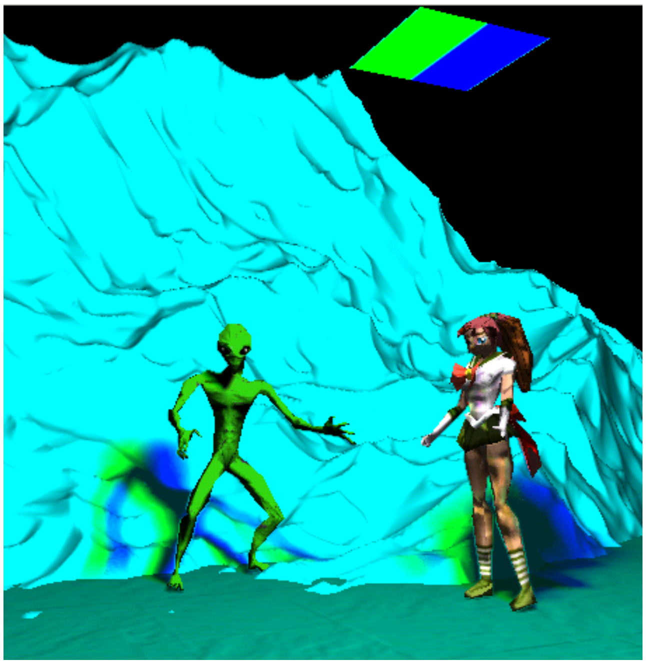


Figure 12: Example of a simple textured light source with two colors, demonstrating that the expected result is obtained.

In Figure 1, an image of fire is used as a light source. It might be hard to judge the quality of this image, and therefore Figure 12 uses

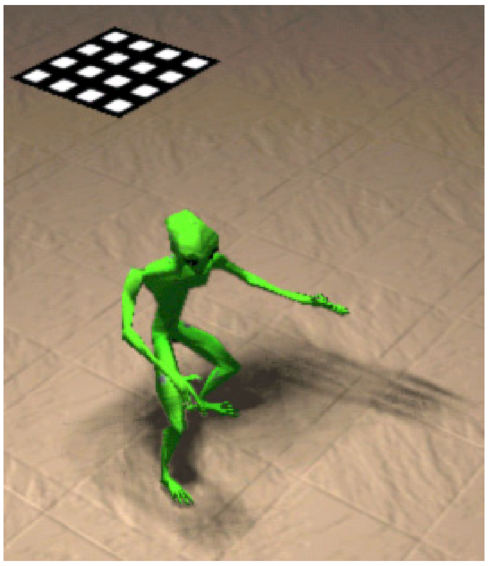


Figure 13: Here, a single rectangular light replaces 16 small rectangular light sources. Sampling artifacts can be seen in the shadow. This can be solved by increasing the resolution of the coverage texture.

a colored light source as well. However, the light source here only consists of two colors. As can be seen, the shadows are colored as one might expect. A related experiment is shown in Figure 13, where a single texture is used to simulate the effect of 16 small area light sources. This is one of the rare cases where we actually get sampling artifacts.

In Figure 14, we compare our algorithm to 256-sample shadows and 1024-sample shadows. In these examples, a large square light source has been used. As can be seen, no sampling artifacts can be seen for our algorithm, while they are clearly visible using 256 samples. We believe that our algorithm behaves so well because we discretize in a place where the function varies very slowly. This can also be seen in Figure 10. Sampling artifacts can probably occur using our algorithm as well, especially when the light source is extremely large. However, we have not experienced many problems with that.

In Figure 15, a set of overlapping objects in a more complex situation are shown. Finally, we have been able to render a single image using actual hardware.[2] See Figure 16. The point here is to verify that the hardware can render soft shadows with similar quality as our software implementation.

5.2 Performance Results

At this point, we have not optimized the code for our Cg implementations at all, since we have not been able to locate bottlenecks due to lack of actual hardware. Therefore, we only present performance results for our software implementation, followed by a discussion of bandwidth usage.

The scene in Figure 14 was rendered at 100×100, 256×256, and 512×512 resolutions. The actual image in the figure was rendered with the latter resolution. The frame rates were: 3, 0.51, and 0.14 frames per second. Similarly, the scene in Figure 13 was rendered at 256×256, and 512×512 resolution. The frame rates were: 0.8, and 0.4 frames per second. When halving the side of the square light source, the frame rate more than doubled for both scenes.

Another interesting fact about our algorithm is that it uses little bandwidth. We compared the bandwidth usage for the shadow

[2]Our program was sent to NVIDIA, and they rendered this image.

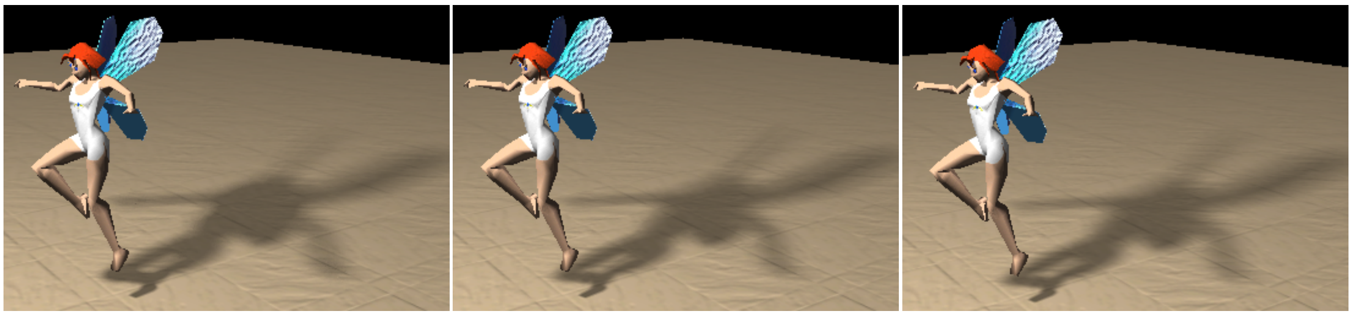


Figure 14: Soft shadow rendering of a fairy using (left to right) our algorithm, 256 samples on the area light source, and 1024 samples. Notice sampling artifacts on the middle image for the shadow of the left wing.

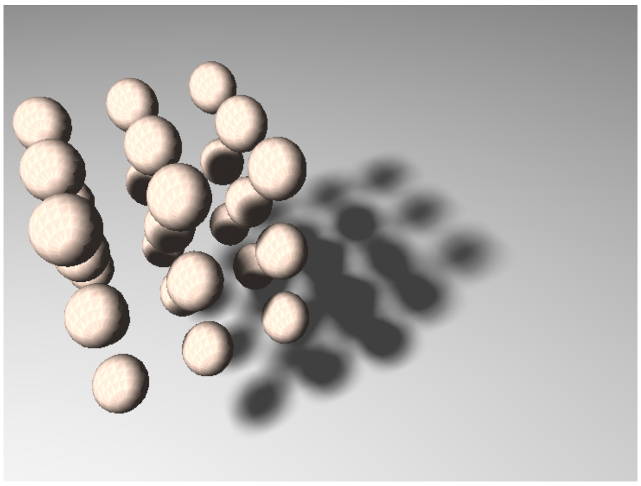


Figure 15: A grid of $3 \times 3 \times 3$ spheres is used as a shadow casting object.

pass for the software implementation of our algorithm and for a 1024-sample shadow image. In this study, we only counted depth buffer accesses and V-buffer/stencil buffer accesses. The latter used 585 MB per frame, while our algorithm used only 6.0 MB. Thus, the 1024-sample shadow version uses almost two orders of magnitude more bandwidth. We believe that this comparison is fair, since fewer samples most often are not sufficient to render high-quality images. Furthermore, we have not found any algorithm with reasonable performance that can render images with comparable soft shadow quality, so our choice of algorithm is also believed to be fair. Even if 256 samples are used, about 146 MB was used, which still is much more than 6 MB.

Our algorithm's performance is linear in the number of silhouette edges and in the number of pixels that are inside the wedges. Furthermore, the performance is linear in the number of light sources, and in the number of shadow casting objects.

5.3 Discussion

Due to approximations in the presented algorithm, artifacts can occur. We classify the artifacts as follows:

1. single silhouette artifact, and
2. object overlap artifact.

Artifact 1 occurs because we are only using a single silhouette as seen from the center of the area or volume light source. This is

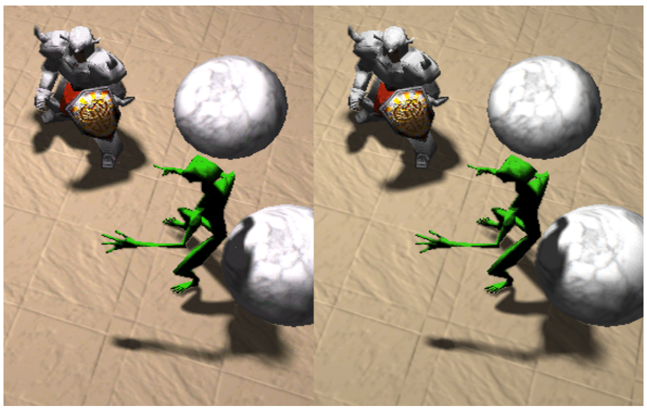


Figure 16: Left: image rendered using our software implementation. Right: rendered using GeForce FX hardware.

obviously not always the case; the silhouette may differ on different points on the light source. Artifact 2 occurs since two objects may overlap as seen from the light source, and our algorithm treats these two objects independently and therefore combines their shadowing effects incorrectly. For shadow casting objects such as an arbitrary planar polygon, that do not generate artifact 1 and 2, our algorithm computes physically correct visibility.

Figure 17 shows a scene with the objective to maximize the single silhouette artifact. Figure 18 shows an example with object overlap artifacts. For both figures, the left images show our algorithm. The right images were rendered using 1024-sample shadows. Thus, these images are considered to provide very accurate soft shadows. Since we only use the silhouette as seen from the center point of the light source, artifact 1 occurs in Figure 17. As can be seen, the umbra disappears using our algorithm, while there is a clear umbra region for the other. Furthermore, the shadows on the sides of the box are clearly different. This is also due to the single silhouette approximation. In the two bottom images of Figure 18, it can be noticed that in this example the correct penumbra region is smoother, while ours becomes too dark in the overlapping section. This occurs since we, incorrectly, treat the two objects independently of each other.

As has been shown here, those artifacts can be pronounced in some cases, and therefore our algorithm cannot be used when an exact result is desired. However, for many applications, such as games, we believe that those artifacts can be accepted, especially, since the errors are are hard to detect for, e.g., animated characters. Other applications may be able to use the presented algorithm as well.

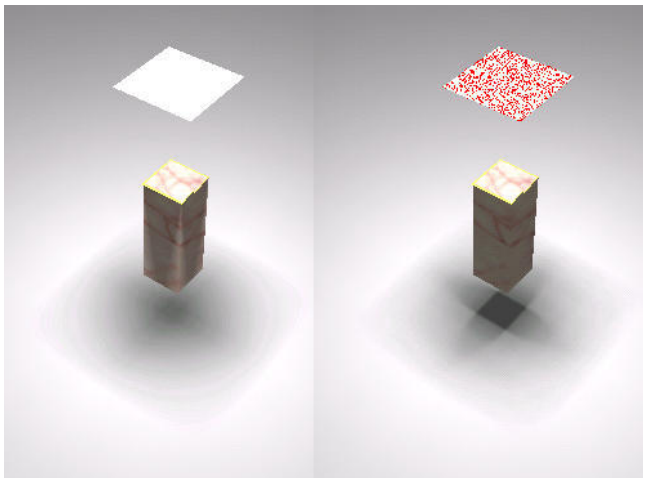


Figure 17: Single silhouette error: the left image shows results from our algorithm, and the right from rendering using 1024 point samples on the area light source. Notice differences in the shadow under the box, and on the sides on the box. The noise on the right light source are the 1024 sample locations.

In general, for any shadow volume algorithm the following restrictions regarding geometry apply: the shadow casting objects must be polygonal and closed (two-manifold) [Bergeron 1986]. The z-fail algorithm [Everitt and Kilgard 2002] can easily be incorporated into our algorithm to make the algorithm independent of whether the eye is in shadow or not [Assarsson and Akenine-Möller 2003]. For the penumbra pass, this basically involves adding a bottom plane to the wedge to close it and rasterize the back-facing wedge-triangles instead of the front-facing. The solution of the robustness issues with the near- and far clipping planes [Everitt and Kilgard 2002] could easily be included as well.

Regarding penumbra wedge construction, we have never experienced any robustness issues. Also, it is possible to use any kind of area/volumetric light source, but for fast rendering we have restricted our work to rectangular and spherical light sources. It is trivial to modify visibility pass 2 (see Section 3.2.2) to handle a spherical light source shape instead of a rectangular. In this case, we do not use a precomputed coverage texture, since the computations become much simpler, and therefore, all computations can be done in the pixel shader. We have not yet experimented with spherical textured light sources.

6 Conclusion

We have presented a robust soft shadow volume algorithm that can render images that often are indistinguishable from images rendered using the average of 1024 hard shadow images. The visibility computation pass of our algorithm was inspired by the physics of the geometrical situation, which is key to the relatively high quality. Another result is that we can use arbitrary shadow casting and shadow receiving objects. Our algorithm can also handle light sources with small textures, and even video textures on them. This allows for spectacular effects such as animated fire used as a light source. We have implemented our algorithm both in software and using the GeForce FX emulator. With actual hardware, we expect that our algorithm will render soft shadows in real time. Our most important task for the future is to run our algorithm using real hardware, and to optimize our code for the hardware. We would also like to do a more accurate comparison in terms of quality with other algorithms. Furthermore, it would be interesting to use Kautz and McCool's [1999] work on factoring low frequency BRDF's into sums of products for our four-dimensional coverage textures. It might be possible to greatly reduce memory usage for coverage textures this way. We also plan to investigate when quadrilinear filtering is needed for the coverage textures.

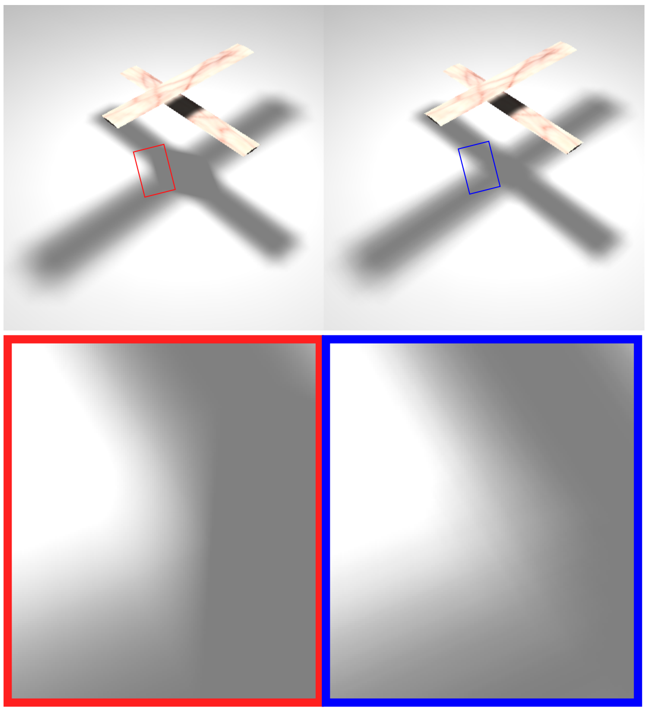


Figure 18: Object overlap error: the left images show results from our algorithm, and the right from rendering using 1024-sample shadows. The right images are correct, with their curved boundaries to the umbra. The left images contain straight boundaries to the umbra.

Acknowledgements: Thanks to Randy Fernando, Eric Haines, Mark Kilgard, and Chris Seitz.

References

AGRAWALA, M., RAMAMOORTHI, R., HEIRICH, A., AND MOLL, L. 2000. Efficient Image-Based Methods for Rendering Soft Shadows. In *Proceedings of ACM SIGGRAPH 2000*, ACM Press/ACM SIGGRAPH, New York. K. Akeley, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 375–384.

AKENINE-MÖLLER, T., AND ASSARSSON, U. 2002. Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges. In *13th Eurographics Workshop on Rendering*, Eurographics, 309–318.

ASSARSSON, U., AND AKENINE-MÖLLER, T. 2003. Interactive Rendering of Soft Shadows using an Optimized and Generalized Penumbra Wedge Algorithm. *submitted to the Visual Computer*.

BERGERON, P. 1986. A General Version of Crow's Shadow Volumes. *IEEE Computer Graphics and Applications 6*, 9 (September), 17–28.

BRABEC, S., AND SEIDEL, H.-P. 2002. Single Sample Soft Shadows using Depth Maps. In *Graphics Interface 2002*, 219–228.

BROTMAN, L. S., AND BADLER, N. I. 1984. Generating Soft Shadows with a Depth Buffer Algorithm. *IEEE Computer Graphics and Applications 4*, 10 (October), 5–12.

COHEN, M. F., AND WALLACE, J. R. 1993. *Radiosity and Realistic Image Synthesis*. Academic Press Professional.

CROW, F. 1977. Shadow Algorithms for Computer Graphics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 77)*, ACM, 242–248.

DRETTAKIS, G., AND FIUME, E. 1994. A Fast Shadow Algorithm for Area Light Sources Using Back Projection. In *Proceedings of ACM SIGGRAPH 94*, ACM Press/ACM SIGGRAPH, New York. A. Glassner, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 223–230.

EVERITT, C., AND KILGARD, M. 2002. Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering. *http://developer.nvidia.com/*.

FERNANDO, R., FERNANDEZ, S., BALA, K., AND GREENBERG, D. P. 2001. Adaptive Shadow Maps. In *Proceedings of ACM SIGGRAPH 2001*, ACM Press/ACM SIGGRAPH, New York. E. Fiume, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 387–390.

HAINES, E., AND MÖLLER, T. 2001. Real-Time Shadows. In *Game Developers Conference*, CMP, 335–352.

HAINES, E. 2001. Soft Planar Shadows Using Plateaus. *Journal of Graphics Tools 6*, 1, 19–27.

HART, D., DUTRÉ, P., AND GREENBERG, D. P. 1999. Direct Illumination with Lazy Visbility Evaluation. In *Proceedings of ACM SIGGRAPH 99*, ACM Press/ACM SIGGRAPH, New York. A. Rockwood, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 147–154.

HECKBERT, P., AND HERF, M. 1997. Simulating Soft Shadows with Graphics Hardware. Tech. rep., Carnegie Mellon University, CMU-CS-97-104, January.

HEIDMANN, T. 1991. Real Shadows, Real Time. *Iris Universe*, 18 (November), 23–31.

HEIDRICH, W., BRABEC, S., AND SEIDEL, H.-P. 2000. Soft Shadow Maps for Linear Lights. In *11th Eurographics Workshop on Rendering*, Eurographics, 269–280.

KAUTZ, J., AND MCCOOL, M. D. 1999. Interactive Rendering with Arbitrary BRDFs using Separable Approximations. In *10th Eurographics Workshop on Rendering*, Eurographics, 281–292.

MARKOSIAN, L., KOWALSKI, M. A., TRYCHIN, S. J., BOURDEV, L. D., GOLDSTEIN, D., AND HUGHES, J. F. 1997. Real-Time Nonphotorealistic Rendering. In *Proceedings of ACM SIGGRAPH 97*, ACM Press/ACM SIGGRAPH, New York. T. Whitted, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 415–420.

PARKER, S., SHIRLEY, P., AND SMITS, B. 1998. Single Sample Soft Shadows. Tech. rep., University of Utah, UUCS-98-019, October.

REEVES, W. T., SALESIN, D. H., AND COOK, R. L. 1987. Rendering Antialiased Shadows with Depth Maps. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, ACM, 283–291.

SEGAL, M., KOROBKIN, C., VAN WIDENFELT, R., FORAN, J., AND HAEBERLI, P. 1992. Fast Shadows and Lighting Effects Using Texture Mapping. In *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, ACM, 249–252.

SLOAN, P.-P., KAUTZ, J., AND SNYDER, J. 2002. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. *ACM Transactions on Graphics 21*, 3 (July), 527–536.

SOLER, C., AND SILLION, F. X. 1998. Fast Calculation of Soft Shadow Textures Using Convolution. In *Proceedings of ACM SIGGRAPH 98*, ACM Press/ACM SIGGRAPH, New York. M. Cohen, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 321–332.

STAMMINGER, M., AND DRETTAKIS, G. 2002. Perspective Shadow Maps. *ACM Transactions on Graphics 21*, 3 (July), 557–562.

WILLIAMS, L. 1978. Casting Curved Shadows on Curved Surfaces. In *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, ACM, 270–274.

WOO, A., POULIN, P., AND FOURNIER, A. 1990. A Survey of Shadow Algorithms. *IEEE Computer Graphics and Applications 10*, 6 (November), 13–32.