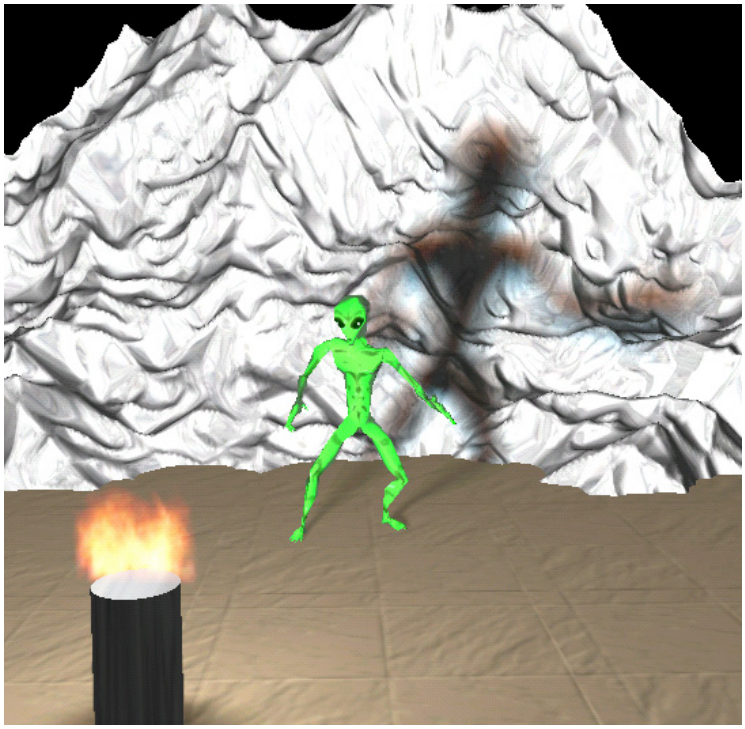
CHALMERS



A Real-Time Soft Shadow Volume Algorithm

ULF ASSARSSON

Department of Computer Engineering
School of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2003

Thesis for the degree of Doctor of Philosophy

A Real-Time Soft Shadow Volume Algorithm

Ulf Assarsson

Department of Computer Engineering
School of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden 2003

A Real-Time Soft Shadow Volume Algorithm
Ulf Assarsson
ISBN 91-7291-333-9

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie 2015
ISSN 0346-718X

School of Computer Science and Engineering
Chalmers University of Technology
Technical Report No. 18D

Department of Computer Engineering
School of Computer Science and Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
Phone: +46 (0)31–772 1000
www.ce.chalmers.se

Author email address: uffe@ce.chalmers.se

Printed by Vasastadens Bokbinderi AB
Göteborg, Sweden 2003

A Real-Time Soft Shadow Volume Algorithm

Ulf Assarsson
Department of Computer Engineering, Chalmers University of Technology

Abstract

Rendering of shadows is a very important ingredient in three-dimensional graphics since they increase the level of realism and provide cues to spatial relationships. Area or volumetric light sources give rise to so called soft shadows, i.e., there is a smooth transition from no shadow to full shadow. For hard shadows, which are generated by point light sources, the transition is abrupt. Since all real light sources occupy an area or volume, soft shadows are more realistic than hard shadows. Fast rendering of soft shadows, preferably in real time, has been a subject for research for decades, but so far this has mostly been an unsolved problem.

Therefore, this thesis, which is based on five papers, focuses on how to achieve real-time rendering of soft shadows. The first four papers constitute the foundation and evolution of a new algorithm, called the *soft shadow volume algorithm*, and the fifth paper provides an essential proof for correctness and generality of this and some previous shadow algorithms.

The algorithm augments and extends the well-known shadow volume algorithm for hard shadows. Two passes are used, where the first pass consist of the classic shadow volume algorithm to generate the hard shadows (umbra). The second pass compensates to provide the softness (penumbra). This is done by generating penumbra wedges and rasterizing them using a custom pixel shader that for each rasterized pixel projects the hard shadow quadrilaterals onto the light source and computes the covered area.

A result of the thesis is an algorithm capable of real-time soft shadows that utilizes programmable graphics hardware. The algorithm produce high-quality shadows for area light sources and volumetric light sources. It also handles textured light sources, which currently is a very rare capability among real-time soft shadow algorithms. Even video textures are allowed as light sources.

Keywords: Computer Graphics, Three-Dimensional Graphics and Realism, Shading, Shadowing, Soft Shadows, Graphics Hardware, Pixel Shaders.

List of Papers

This thesis is based on the following papers. References to the papers will be made using the Roman numbers associated with the papers.

I. Tomas Akenine-Möller and Ulf Assarsson, "Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges," *13h Eurographics Workshop on Rendering*, Eurographics, Pages 309–318, June 2002.

II. Ulf Assarsson and Tomas Akenine-Möller, "Interactive Rendering of Soft Shadow using an Optimized and Generalized Penumbra Wedge Algorithm", *conditionally accepted by the Visual Computer*, submitted May 2002.

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

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

V. Tomas Akenine-Möller and Ulf Assarsson, "On Shadow Volume Silhouettes", Submitted to *Journal of Graphics Tools*, April 4, 2003.

Papers by the author of this thesis that are not included:

- Ulf Assarsson and Tomas Möller, *"Optimized View Frustum Culling Algorithms,"* Technical Report 99-3, Department of Computer Engineering, Chalmers University of Technology, http://www.ce.chalmers.se/staff/uffe/, March 1999.
- Ulf Assarsson and Tomas Möller, "Optimized View Frustum Culling Algorithms for Bounding Boxes," *Journal of Graphics Tools*, 5(1), Pages 9–22, 2000.

- Ulf Assarsson and Per Stenström, "A Case Study of Load Distribution in Parallel View Frustum Culling and Collision Detection," *Euro-Par 2001 Parallel Processing Proceedings*, Pages 663–673, 2001.

- Jonas Lext, Ulf Assarsson, and Tomas Möller, "BART: A Benchmark for Animated Ray Tracing," *IEEE Computer Graphics and Applications*, Pages 22–31, March/April 2001.

Table of Contents

Preface

"If something is hard, then it is not worth doing."
–Homer J. Simpson, words of wisdom to his son Bart.

Comment: This statement may sound a bit controversial to many, but if the word *hard* is interpreted as *complicated*, what Homer says is often true. If a solution is complex, then it is probably not a very good one. We strive to derive simple algorithms, because they are often faster and less error-prone. Furthermore, the real-time demands of computer graphics often prohibits complicated computations. The beauty lies within simple and fast solutions.
Ulf Assarsson

When reading really old articles, written decades or even centuries ago, one thing that have struck me is that it often is extremely hard to judge the validity of the statements or results in the document. Any problem or solution comes from a context. A clue to such a context could be a time stamp, since it is then possible to investigate the relevant circumstances at that time. Preferably the environment for the problem and solution should be clearly given in the text.

In my thesis I want to clarify the context for which it is written so that the validity of the results can be estimated and understood in the future.

The latest available commodity graphics hardware is the Geforce FX 5900 and the ATI 9800. The Geforce FX consists of 125 million transistors clocked at 500 MHz, typically has 128 or 256 MB texture memory, 20GB/s internal bandwidth, and according to the specifications processes 330M triangle vertices/s and 3GPixels/s. In theory, this correspond to about

100M triangles per second. In reality, this probably means that about 1M triangles can be drawn to the screen at 50 fps, i.e., in real time.

A typical example of the latest commodity CPU is the Intel Pentium 4, clocked at 3GHz, and a common memory size in a desktop computer is 1GB.

Which computer algorithms that are the most efficient will always depend on the hardware. But the available hardware will also depend on what algorithms are intended to run on it. For the soft shadow algorithms in this thesis, I have not restricted the research to algorithms for a specific hardware. If existing hardware is not suitable, it has been valid to suggest changes. However, the suggestions should be realistic at the present time. In the end, the algorithm was adapted to suit the new programmable hardware that just have entered the market.

Since the conditions for the algorithms change as hardware change, a question that arise is how long the results of this thesis will be of interest. How long will it take before it is more efficient to compute real-time soft shadows in a very different way than described in this thesis? No one, of course, knows, but it is reasonable to assume that the fundamentals of hardware design still applies during the foreseeable future of perhaps 5 to 10 years. Someone once said that a good algorithm in computer graphics should at least have a life time of 5 years. The successfulness of graphics hardware of today relies on the streaming architecture. Triangle rendering allows for efficient parallelism, pipelining, and tolerance of relatively high memory latency, because the processing of elements can be done independently of each other. That is, the result of processing vertices are independent of the results from other vertices, and the same applies between pixels.

Due to the above, it seems reasonable that the streaming architecture will stay around for a while for graphics hardware, instead of turning into one or several general purpose processors. A very important property of the proposed soft shadow algorithm is that it can efficiently utilize graphics hardware built on the streaming architecture, and therefore the algorithm will likely still be efficient a long time from now.

The soft shadow volume algorithm is based on the shadow volume algorithm [14] for hard shadows, also called the stencil buffer method [20], that originates from 1977. It has taken about 20 years for that algorithm to mature, with improved speed and robustness [16], and I hope that the soft shadow volume algorithm has a similar fate. In other words, I do not believe that what is presented here is the final version of the algorithm.

One thing I really want to have in my thesis is some nice color plates, so here they are. These are relevant, since these plates demonstrate the quality of the soft shadows produced by the algorithm.

Plate 1: Screen shot of a lizard model, using the implementation described in **IV**.

Plate 2: Screen shots from implementation in **IV**.

1 Introduction

Computer graphics is the science of how to generate (render) images with the help of computers. In three-dimensional computer graphics, a scene is modelled geometrically, typically using triangles, and the computer is then used to calculate what the scene looks like from a specific view point at a particular instant.

Since the dawn of computer graphics, a major goal has been to create photo-realistic images in real time. Since this has not been and still is not possible, there, generally, are two approaches. One is to compute photo-realistic images using more time than real-time performance permits. The other is to compute as realistic images as possible and still meet the real-time requirement.

In computer graphics, the meaning of *real-time* is practically never used in the strict sense. Real-time typically refers to soft real-time, as opposed to hard real-time. That is, it is acceptable not to meet the timing requirements all the time. On top of this, real-time graphics generally means more than one frame per second (fps) and typically within an interval of 20-80 fps. If a television or monitor is the target of presentation, the refresh rate may control the amount of time and computations that can be spent on a frame.

Shadows are very important ingredients to increase the realism of a scene, and they also provide important spatial cues [30]. For instance, without shadows it can be impossible to determine whether an object is located on the floor or some distance above the floor (see Figure 1).

In computer graphics, it is common to differentiate between hard

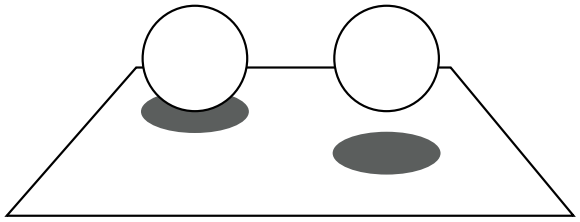


Figure 1: Example of the importance of shadows as spatial cues.

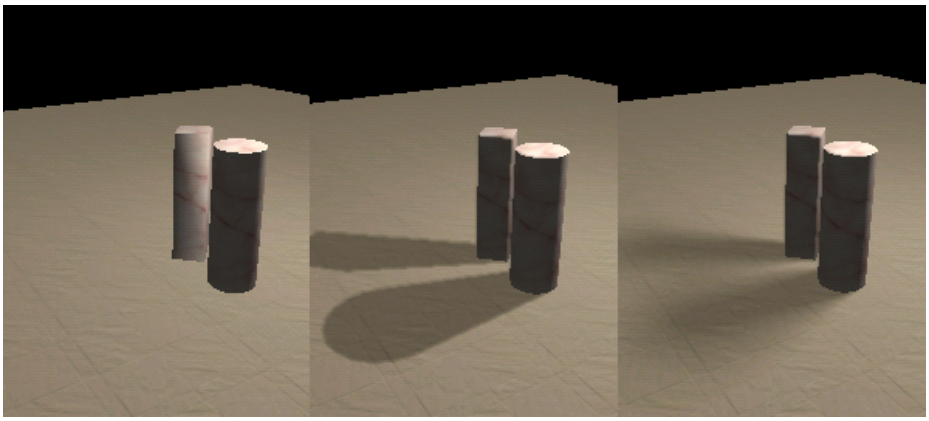


Figure 2: a) no shadows, b) hard shadows, c) soft shadows.

shadows and soft shadows. For a hard shadow, the transition from no shadow to full shadow is instant at the border of the shadowed region. For soft shadows, the transition is smooth. Hard shadows can only be generated by infinitely small point light sources, and thus do not occur in reality. All real light sources have an area or volume, which gives rise to a soft transition, called the penumbra. It should be noted, though, that the soft region can be very small, and therefore a hard shadow sometimes is a reasonable approximation. The fully shadowed region is called the umbra. Soft shadows typically increase the perceived realism significantly over hard shadows [27]. The hardness also makes it possible to misinterpret the shadows as geometric features [3]. Figure 2 illustrates a scene with no shadows, hard shadows, and soft shadows.

Simple, non-accurate shadows have been used for long, such as a dark ellipse or rectangle below an object, since some representation of shadows is better than no shadows at all [31]. Accurate dynamic hard shadows on arbitrary geometry has been possible to compute in real time since 1991, when Heidmann [20] presented his implementation, using the stencil buffer, of the shadow volume algorithm [14]. At that time, it only worked on fairly expensive systems. This kind of shadows started to become commonly used around 1999-2000 as graphics hardware evolved, and due to that an important generalization of the algorithm was presented by Bilodeau & Songy and Carmack [16].

The next natural step for games and virtual reality applications is likely to be accurate dynamic soft shadows, but so far no algorithm has been capable of producing this with commodity hardware, unless expensive clusters are used [21]. This is due to that soft shadow generation is an inherently difficult problem. There has been research on the sub-

ject for decades [5, 11, 23, 24, 25], but so far it has been an unsolved issue. Many algorithms on shadows and soft shadows have been suggested over the years and been presented at the most prominent conferences [1, 6, 14, 15, 29]. Recently, a new survey on soft shadow algorithms has been published [18].

The problem of generating soft shadows is related to cell-based occlusion culling, which also is a very difficult problem, generating lots of research [3, 13].

1.1 Overall Objective and Research Question

One of the major remaining unsolved problems in real-time three-dimensional graphics is how to generate soft shadows. So far, all algorithms have either produced disturbingly inaccurate shadow quality, or the algorithms have been too computationally intensive for real-time purposes. Solving this problem is the overall objective of this thesis. It is assumed that the target system is a PC or similar, equipped with commodity hardware acceleration for graphics.

The shadows do not necessarily have to be physically accurate down to the level of available precision for the hardware, but they should look and behave realistically and preferably be free from obvious artifacts. These requirements are sufficient for many applications, such as games and virtual reality systems. Thus, the main research question is: *Can these soft shadows be achieved in real time without waiting for hardware to be much faster, and if so, how would such an algorithm work?*

A common method to achieve speed for graphics algorithms is to utilize graphics hardware. This hardware is targeted towards rendering of triangles that are sent to the card sequentially. Triangle rendering can be implemented in hardware very efficiently through pipelining and due to the inherent amount of parallelism.

One way to obtain a fast soft shadow algorithm can therefore be to utilize graphics hardware. The speed of graphics hardware has so far more than doubled each 12 months [2], compared to CPU:s that double approximately each 18 months[1] [3]. For cases where real-time performance cannot be achieved today, an algorithm that utilize graphics hardware will therefore probably reach real-time performance before an algorithm that is CPU-based. A more specific research question is then: *Is it possible to create a soft shadow algorithm that efficiently utilizes graphics hardware, and how?*

[1] Moore's law says that the number of transistors per square inch on integrated circuits double approximately each 18 months, which is the major reason why CPU's approximately double in speed at the same pace.

The short answer is that high quality real-time soft shadows can be achieved today by using graphics hardware. How, is explained in the remainder of this thesis.

1.2 Why Previous Methods Do Not Suffice

According to a recently published State-of-the-Art report [18], the soft shadow volume algorithm presented in this thesis is the only currently existing soft shadow algorithm that provides both high quality and good real-time performance for dynamic scenes. In that report, the algorithm is called *Penumbra Wedges*.

1.2.1 High Quality but Insufficient Speed

Worth noting is that one way to achieve soft shadows using graphics hardware is to compute the superposition of hard shadows from several point lights [9]. Point light samples are then used to approximate the area or volumetric light source. The hard shadow contribution from each point light can be computed using graphics hardware and for instance the stencil buffer method [20]. The running time is proportional to the number of point light samples and the scene complexity (actually it is only dependent on the complexity of the shadow casters, not shadow receivers). Several hundreds of point light samples are required to avoid obvious sampling artifacts. Although this method uses graphics hardware, it is currently two orders of magnitude too slow even for fairly simple scenes ($\sim$ 10.000 triangles) to be useful for real-time purposes such as in games. This approach also uses two orders of magnitude more bandwidth than the soft shadow volume algorithm, which is an undesirable property since high bandwidth usage can be a performance bottleneck [6].

If current acceleration rate of graphics hardware performance holds, it is reasonable to believe that approximately seven years of evolution is required before using superpositions of hard shadows to approximate soft shadows can be used in real time even for simple scenes. At that time, it is likely that the soft shadow volume algorithm presented in this thesis has experienced a similar performance increase, and therefore also will be two orders of magnitude faster. This means that the algorithm can handle two orders of magnitude more complex scenes in real time than now, which clearly is desirable.

It should be possible to achieve real-time performance for point light samples using a cluster of PC's to distribute the computations of hard shadows using the shadow volume algorithm, in a similar fashion as Isard et al. distribute hard shadows using the shadow map algorithm [21].

However, such an expensive system falls outside the target platform of this thesis.

1.2.2 High Speed but Insufficient Quality

Hasenfratz et al. [18] reports that existing soft shadow algorithms with real-time performance but with low quality are; 1) Distributed Multi-samples [21], i.e., using a PC-cluster for distributing computations for several point light samples, 2) Single Sample Soft Shadows [26, 8], and 3) Smoothies [10]. Both Single Samples and Smoothies produce usually very noticeable incorrect umbra and penumbra regions, when compared side by side with the correct result. Distributed Multi-samples is based on the superposition of hard shadows from point light samples created using shadow maps [32], and the reportedly [18] low quality likely stems from the fact that Isard et al. only use 32 point light samples in order to achieve real-time performance, and also that shadow maps suffers from aliasing artifacts.

Penumbra Maps [33] is a recently published method that reaches real-time performance, but has similar quality as Smoothies and Plateaus [17]. These methods approximate the umbra region with that of a hard shadow, which gives a too large umbra region. The difference is pronounced as the size of the light source and corresponding penumbra regions increase.

1.3 Methodology

The method used to measure the quality of the produced soft shadows, was side-by-side visual comparisons with reference examples. In **I** and **II**, Heckbert/Herf shadows [19] and Soler/Sillion shadows [29] were used as references. **III** and **IV** instead used the superposition of hard shadows, created using the shadow volume algorithm and 1024 point light samples, to create accurate reference soft shadows for more general situations. When the number of point light samples reaches infinity, the corresponding soft shadows approach a correct result, and 1024 samples was shown to be sufficient.

Speed comparisons was performed by observing the frame rate and also using the reported results of other algorithms. See Hasenfratz et al. [18] for a more thorough examination. The running time of the algorithms depends on factors such as for instance the screen resolution, the number of shadow casters, number and size of the light sources, desired quality, graphics hardware and CPU speed. No detailed comparison using various parameter configurations has been performed and is left for future work.

The test scenes are depicted in several of the images in the papers **I–IV**. The Quake2-models were downloaded from NVIDIA and are used in several other shadow papers [10, 16, 28]. They are good examples of complex and arbitrarily shaped objects. In addition, cylinders, cubes, spheres and other objects at hand were used.

The soft shadow volume algorithm and the reference algorithm using point light samples were implemented and tested on PC:s with CPU:s from 1.3 GHz to 2.4 GHz and the Geforce FX 5800 Ultra, ATI 9700 and ATI 9800. The test system configuration were determined by what was available at the time being.

1.4 Main Contribution

The main contribution of this thesis is a soft shadow algorithm that achieves both real-time performance and high quality soft shadows for fairly complex scenes with existing graphics hardware, and as hardware evolves, will be able to handle more and more complex scenes.

The algorithm augments the classic shadow volume algorithm for hard shadows [14], to capture the umbra, and is then extended with a second pass that compensates for the first pass in order to produce the penumbra. This second pass uses a new primitive, called the *penumbra wedge*, that each contains a portion of the penumbra volume, and together includes all penumbra regions. The penumbra wedges are rasterized using a custom *pixel shader* [22] that can be implemented on programmable graphics hardware.

The algorithm is suitable for applications where speed is essential and where it is more important that the shadows look and behave realistic than that they are physically accurate. Examples of such applications are games and virtual reality systems. However, since the produced soft shadows often are of very high quality, they could likely also be used in effect tools for movie production. The soft shadow algorithm is possible to use together with ray tracing. The only requirements are that the shadow receiving geometry is represented in a z-buffer or an equivalent, and that the occluders can be described by planar polygons and are closed (two-manifold). Transparent occluders have been ignored and constitute a potential problem. The latter applies to most real-time shadow algorithms, such as for instance the shadow volume algorithm [14] and the shadow map algorithm [32].

The computation time of the algorithm is proportional to the number of pixels rasterized by the wedges, which typically is proportional to the number and complexity of the shadow casters, number and size of the light sources and also the screen size if the algorithm is fill rate limited.

It also depends on the viewing position relatively to the soft shadows.

Most algorithms for soft shadows use trade-offs between quality and speed. No detailed performance comparison has been done between the soft shadow volume algorithm and others in terms of speed and quality, but a firm estimation is that the proposed soft shadow volume algorithm is between one and two orders of magnitude faster than using point light samples, for the same visual quality. Compared to other algorithms, it is often both significantly faster and produces higher quality [18, 7].

1.5 Thesis Structure

This thesis is based on five papers. Each of the first four paper incrementally refines the soft shadow volume algorithm by identifying limitations or unsolved problem areas and investigating different solutions. The fifth paper provides an important proof of generality and correctness of this and other shadow volume related algorithms.

The remainder of this thesis is organized into three sections and then the five papers are appended.

- **Section 2** is divided into a subsection for each paper, which describes the problems, methodology and the contribution for each paper.
- **Section 3** discusses the results and also contains miscellaneous comments about the work in this thesis.
- **Section 4** concludes and suggest how to follow up on the work in this thesis.

2 A Real-Time Soft Shadow Volume Algorithm

This section starts with some preliminaries and then continues with a subsection for each appended paper containing the presentation of the problem, methodology and contribution of the corresponding paper.

2.1 Preliminaries

In soft shadow rendering, a common approximation is to separate lighting from visibility [1]. The desired irradiance computation for an area light source with homogeneous radiation over the surface is shown below [12]:

$$E = \int_A \frac{L \cos \phi_i \cos \phi_l}{\pi r^2} v dA. \quad (1)$$

Here, E is the incoming irradiance at point $\mathbf{p}$, A is the area of the light source, L is the radiance from the light source, v is the binary visibility term, r is the distance from the point to be shaded to the point on the light source, ϕ_i is the incident angle, and ϕ_l is the angle between the normal of the light source and the vector from the light to the point to be shaded (see Figure 3).

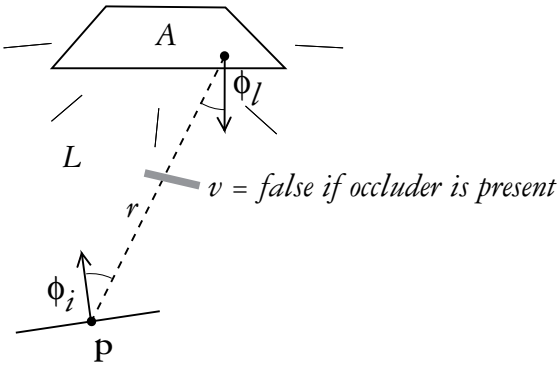


Figure 3: Lighting model

By separating lighting and visibility, as shown in equation 2:

$$E \approx l\overline{v} \tag{2}$$

we can compute the average visibility term as shown below [1]:

$$\overline{v} = \frac{1}{A}\int_A v \; dA. \tag{3}$$

Lighting can be computed as if the area light source was a point light source, or more accurate lighting computation can be used if necessary.

It should be noted that for a volumetric light source, the radiance L could be a more complex function of surface location and direction. For all papers in this thesis, spherical light sources are treated as they are spherical area light sources with homogenous radiation for all locations of the surface. The implication is that the amount of shadow at a location is directly proportional to the visibility of the light source from that location.

An area light source with non-homogenous radiation over the surface can be approximated into several piecewise constant sections. This is essentially how soft shadows from textured area light sources are handled in **III**.

For real-time purposes, the above approximations are reasonable, and the presented algorithms in this thesis therefore focus on computing $\overline{v}$, which is the major contributor to the appearance of soft shadows.

In the test applications and screen shots of **I**–**IV**, the lighting is computed using a point light source, typically at the center of the area or volumetric light source. The consequence of this approximation is that diffuse shading and specular effects will correspond to a point light source and shadows will correspond to an area light source. The diffuse and specular deviations are often much more subtle than for shadows, which makes this approximation very suitable for real-time purposes.

A common method to render soft shadows is to approximate an area or volumetric light source with several point light sources, and use the average (superposition) of the hard shadows from each sample point. This is of course slow if many sample points are used, and the computation time typically scales linearly with the number of sample points. To avoid obvious sample artifacts, a great number of point light samples is necessary. For 256 levels of softness, at least 256 point light sources are required in the general case, but often more have to be used. For some of the test scenes in this thesis, sampling artifacts have been observed for 512 samples, although such situations seem to be fairly uncommon. No artifacts have been detected when using 1024 samples.

2.2 Approximate Soft Shadows using Penumbra Wedges

2.2.1 Problem

No previous algorithms exist that are capable of rendering high quality soft shadows that are cast onto arbitrary, animated objects in real time [18, 4].

Pursuing the goal of creating such an algorithm, **I** suggests and investigates a new technique using *penumbra wedges*. The approach includes identifying and solving problems such as creating penumbra wedges from shadow casters and developing an efficient rasterization method for the wedges.

2.2.2 Methodology

The penumbra wedge algorithm was implemented in software on a PC. The quality and speed of the soft shadows was then compared to Heckbert/Herf shadows [19] and Soler/Sillion shadows [29]. The Heckbert/Herf shadows averaged the hard shadows from 128 point light samples which gives fairly accurate shadows on planar surfaces. The soft shadow method by Soler and Sillion was chosen because it is one of the few methods that give real-time performance, but it has problems for general scenes. The speed were compared by observing the frame rates. The quality of the soft shadows was compared visually.

2.2.3 Contribution

The contribution of **I** is the first generation of a new soft shadow algorithm, which is capable of rendering soft shadows on arbitrary geometry at interactive frame rates (1–10 fps). It only handles spherical light sources, so light sources with other shapes are approximated by a surrounding bounding sphere. The algorithm extends Crow's hard shadow volume algorithm [14], by using 256 levels of shadow intensity instead of just two (no shadow and full shadow). This is done by replacing the shadow quadrilaterals in the shadow volumes with penumbra wedges that models the penumbra regions (see figure 4).

The behavior of the shadows correspond to the following goals: 1) the softness of the penumbra increases linearly with the distance from the occluder, and 2) the umbra region disappears when the light source is large enough. A strong advantage is that typical sampling artifacts are avoided, such as from superpositioned hard shadows.

The algorithm handles the same shadow casting objects as the commonly used shadow volume algorithm for hard shadows by Crow [14],

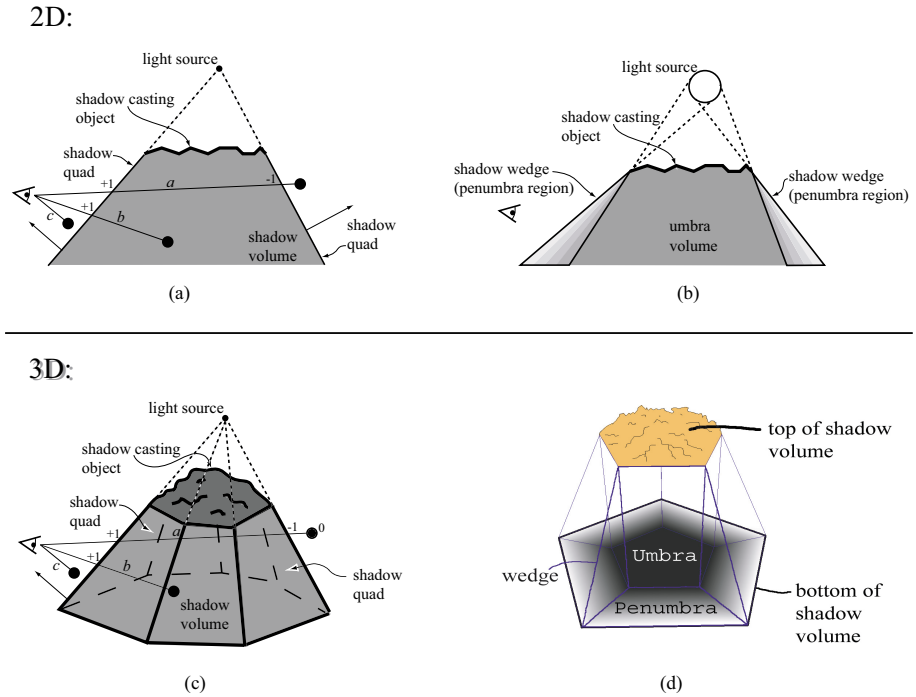


Figure 4: The quads in the shadow volumes for hard shadows, are replaced with penumbra wedges to achieve soft shadows. a) and c) show the shadow quads for a point light source. b) and d) shows the penumbra wedges for a volumetric light source.

with some limitations on the quality for complex scenes. The former constraints are that the occluders must be polygonal, closed (2-manifold) and the polygons must be planar. The shadow receivers should be represented in a z-buffer. Since real-time computer graphics is based on triangle rasterization, these limitations are of very little importance, as all objects typically consist of triangles, rendered into a z-buffer used for the depth sorting. Non-closed objects are often easy to convert to closed objects, and it can often be done automatically. Multiple light sources can be handled.

The main problem with the approach presented in this paper is that it is hard to compute reasonable wedges that approximate the penumbra for all silhouette edge situations. The wedges should approximate the inner and outer surface of the penumbra volume, which can be a fairly complicated geometrical problem for complex occluders. However, when such wedges are at hand, soft shadows can be generated at interactive frame rates.

An approximate version of the algorithm that utilizes graphics hardware was also implemented. For real-time performance, much fewer levels of shadow intensity were used - typically 6-16 instead of 256. The quality and smoothness was judged visually. This algorithm is very similar to using several shadow volumes from point light samples, but can give slightly smoother results. To get really smooth shadows, however, the computation cost becomes similarly high, and therefore this approach was discarded from further investigations in this thesis.

2.3 An Optimized and Generalized Penumbra Wedge Algorithm

2.3.1 Problem

The algorithm in **I** has a few shortcomings of which the most important are: 1) it does not run in real time since the wedge rasterization has no efficient hardware accelerated implementation, 2) it has robustness problems and visible artifacts, and 3) it does not correctly treat the case when the viewer is inside a shadow volume, since the usual solution of using the *z-fail* algorithm [16] cannot easily be incorporated.

Therefore, **II** investigates how the number of computations of the wedge rendering can be reduced and how the well-known problem with the viewer inside a shadow volume can be solved [16]. The resulting algorithm is also evaluated from a future hardware implementation perspective in terms of memory accesses of a single-pass version versus a three-pass version.

2.3.2 Methodology

The algorithms and modifications were implemented in software and the speedups were compared against the original algorithm in **I**. Except for that the problem when the viewer is inside the shadow volume is removed, the quality of the soft shadows is unaffected.

2.3.3 Contribution

The algorithm is restructured so that the umbra and penumbra rendering is split into different parts which enables 1) the z-fail algorithm for correctly managing the case when the viewer is in shadow, 2) using commodity graphics hardware rasterization for the umbra, 3) efficient culling with speedups of 3-4 times, and 4) significantly reducing the number of computations for wedge rasterization.

A single pass version of the algorithm is then compared to a three-pass version in terms of memory reads and writes. The single pass version

requires fewer memory accesses and is targeted for a future hard-wired implementation added into graphics hardware, whereas the three-pass version can utilize non-modified existing graphics hardware for the umbra contribution.

2.4 A Geometry-Based Soft Shadow Volume Algorithm

2.4.1 Problem

Two shortcomings of the previous algorithms are that 1) they still do not run in real time, since there is no fully hardware accelerated implementation of the rasterization, and 2) there are robustness issues during wedge construction (see Figure 5b and 6). To add a new hardware mechanism that would provide support for rendering of penumbra wedges, graphics cards vendors would have to be convinced to add it into their designs. At the time for this paper, massively[2] programmable graphics hardware has very recently appeared on the market. Although the number of instructions, memory lookups and length of programs are still very restricted, it is highly probable that better programmability will come shortly. Preferably, the soft shadow volume algorithm should be implementable using programmable graphics hardware instead of requiring a new hardware mechanism.

A major problem with the previous algorithms of **I** and **II** is that they need an explicit computation of the umbra volume. That is, the polygons that constitute the border between the umbra and the penumbra should preferably be computed exactly. If they are not computed exactly, artifacts may appear, such as visible cracks in the umbra. These polygons are generated from the back plane polygons of the wedges, but they must be clipped against each other. This is a geometric problem that can be difficult and inconvenient to solve in real time for a large number of polygons, especially since a back plane polygon may have to be clipped against more than just its two neighboring wedges' back plane polygons (see Figure 5a). For a large light source and a small shadow generator, the umbra will disappear when the shadow receiver is far enough away, which further complicates the matter, since it then is unclear what to use as back planes. Furthermore, it is very difficult to generate sensible wedges for silhouette edges that makes sharp angles to each other. Another particularly difficult case is when a silhouette edge is nearly parallel to its direction to the light source, and also when silhouette edges forms a zigzag pattern along the direction to the light source (see Figure 6 and Figure 5b).

[2]Programs longer than 100 pixel shader instructions.

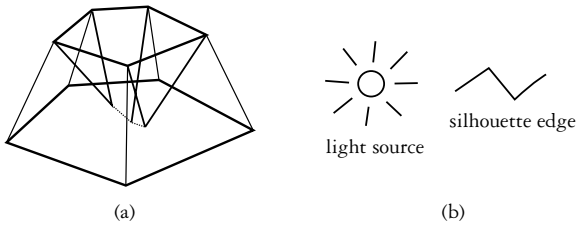


Figure 5: a) Example of a soft shadow volume with the back planes clipped against each other. b) Zigzag patterns along the direction to the light source cannot be handled with the wedge generation method in **I** and **II**. This situation is also illustrated in the right of figure 6.

The robustness issues were ignored in the previous approaches. For simplicity, when computing the umbra volume, clipping of back plane polygons is done only against the two neighboring wedges' back planes.

Another undesirable property of the algorithms is that the shadow from a polygon or any other object differs from that when the polygon or object is split in two halves and shadows are generated from each part.

To overcome the problems mentioned above, an algorithm is developed that does not need explicit computation of the umbra volume. It uses the ordinary hard shadow volumes, which can be computed both easily and robustly. Furthermore, the shape of a wedge is decoupled from the shape of the neighboring wedges, which avoids problems with difficult silhouette edge situations. The algorithm uses a more physically accurate penumbra computation, which requires more calculations than the previous algorithms, but can be implemented using programmable hardware and therefore reaches real-time performance.

2.4.2 Methodology

The algorithms were implemented in software on a commodity PC, and the wedge rasterization was also implemented targeting a software emulator of upcoming programmable graphics hardware. The soft shadow quality was compared against the superposition of 1024 hard shadows from point light samples, since that produces very accurate results. New test scenes were added, with animated and more complex shaped objects.

2.4.3 Contribution

This paper presents a method targeted for real-time soft shadows that is implementable using programmable graphics hardware, handles spherical light sources or arbitrarily shaped planar light sources that can be

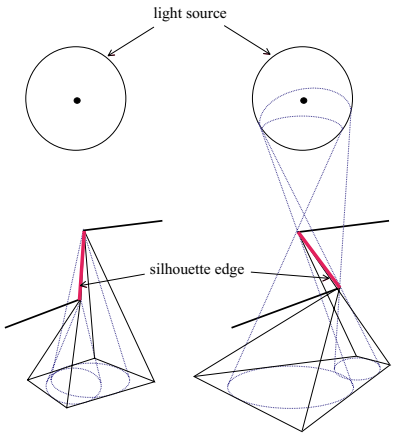


Figure 6: Left: The edge nearly points towards the light center, resulting in a non-convex wedge. Right: The edge is shadowed by one adjacent edge that is closer to the light source, making the left and right planes intersect inside the wedge. Unfortunately, the two tops of the cones cannot be swapped to make a better-behaved wedge, because that results in a discontinuity of the penumbra wedges between this wedge and the adjacent wedges.

textured. The algorithm has no known robustness issues, for instance concerning the appearance of the shadow casters, as long as they are polygonal and two-manifold, or shadow receivers. It handles all scenes that the hard shadow volume algorithm handles. The shadows are usually of very high quality, although two approximations are known to sometimes cause artifacts.

An important feature of the algorithm is that the resulting soft shadows always are smooth, i.e., there are no artifacts that cause erroneous sharp steps in the penumbra, which can be the case for other algorithms [8]. Undesirable steps in the penumbra are particularly disturbing, since the human visual perception system is very good at detecting edges and may try to interpret them as geometric features.

2.5 Soft Shadows with Real-Time Performance

2.5.1 Problem

At the time of writing and developing the algorithm in **III**, there was no programmable hardware at hand on which to do test runs of the implementation. Later, when the soft shadow algorithm was tested on such hardware (the Geforce FX) unforseen bottlenecks appeared which required major changes in the algorithm. Also, it stood clear that the

pixel shader was too computationally intensive to provide good real-time performance.

IV investigates how the computational time can be further reduced and also how the quality of the soft shadows can be increased.

2.5.2 Methodology

The algorithm of **III** was tested on a Geforce FX 5800 Ultra, since this was the only available graphics hardware that manages pixel shaders with the required length of about 250 instructions. The implementation in **IV** were targeted towards the ATI 9700, since two of the co-authors had these cards. The ATI 9700 is only capable of running pixel shaders with length of at most 64 instructions, and this affected the solutions.

The speedups of the optimizations were measured by frame rate, using the implementation in **III** running on the Geforce FX and the implementation in **IV** running on the ATI 9800. Due to practical reasons, the two implementations currently do not run on the same graphics accelerators. However, this probably only has fairly subtle impact on the results, since the official reported speed differences between the cards are at a small percentage level.[3] The test scenes from **III** were reused.

2.5.3 Contribution

The algorithm in **III** can render real-time soft shadows, but only for small resolutions and fairly small scenes, since it is heavily fill-rate limited due to a long pixel shader. In paper **IV**, the pixel shader is brought down from ~ 250 instructions to about 60 instructions. This is done by, among other things, using hand optimized assembler and more lookup tables, stored in textures. It should be emphasized that these lookup tables only depend on the shape of the light source, which must be planar, and not on the occluders or shadow receivers. Furthermore, the lookup tables are independent of any light source transformation, such as for instance rotation, scaling or shearing.

A large part of the shader consisting of coordinate transformations is lifted out and now done by ordinary triangle rasterization, which proved to be much faster.

A culling technique is added to avoid running the pixel shader for pixels with points outside the wedges and thus will get zero contribution. Additionally, a new wedge creation method is developed that is not as conservative as the previous method. Thus, tighter wedges around the

[3] See for instance Tom's Hardware: http://www.tomshardware.com.

penumbra regions can be created, which lowers the number of pixels that will be processed by the pixel shader.

A previous problem of either having to use several passes for the wedge rendering or using few levels of penumbra for the shadows, is solved by utilizing the 8 bit per component rgba-buffer to get 12 bits of precision for the intermediate soft shadow results.

Together, all the presented optimizations provide 15-20 times speedup, which results in real-time performance for fairly complex scenes and large resolutions.

Due to the fundamental approximations of the soft shadow algorithms, two major artifacts can still appear; 1) the single silhouette artifact, and 2) the object overlap artifact [6]. Therefore, two methods for increasing the quality of the shadows are presented. The first method takes a heuristical approach and requires an extra rendering pass for each wedge. The second approach is capable of producing the correct result by splitting the light source into smaller pieces, and is a classical trade-off between speed and accuracy.

2.6 On Shadow Volume Silhouettes

This paper presents a proof essential for the correctness of Crow's hard shadow volume algorithm [14], and also for the soft shadow volume algorithm.

2.6.1 Problem

The shadow volume algorithm and soft shadow volume algorithms in **I**–**IV** use the silhouette edges as seen from a point light source to create wedges. The true silhouettes are hard to compute and they will generally not form closed loops, i.e., there can be discontinuities between the edges of a silhouette. A simple example is the silhouette for two overlapping polygons, as seen from the light source. Usually, *possible silhouettes* [16] are computed instead, which consist of the true silhouette edges and edges that would have been silhouette edges if they were not shadowed by an object closer to the light source. The definition of a possible silhouette edge is an edge between two triangles of which one is front facing and the other is back facing as seen from the point light source. These possible silhouette edges are very easy to compute and form closed loops for two-manifold objects.

For correctness of the shadow volume and soft shadow volume algorithms, it is important that the generated set of all possible silhouette edges can always be split into one or more separate single loops. These loops must not have to share the same generated edge instances. Then,

the shadow volumes for any silhouette edge configuration can be split into independent volumes that do not share the same surface instance, i.e., it is true that each silhouette edge correspond to one and only one hard shadow quad in a shadow volume, and one and only one wedge in a soft shadow volume.

If a shadow quad or wedge would have to be shared, this quad or wedge would need to be duplicated with one instance for each loop that contains the shared edge instance, which would complicate the algorithms, since a detection and copying mechanism would have to be added. It would also be bad for the soft shadow algorithm, since a duplicated wedge would duplicate the corresponding amount of shadow contribution as well.

2.6.2 Methodology

A proof that the silhouette edge connectivity must be even is presented. It is then shown with a simple example that the connectivity can be larger than two. Therefore, if a number of silhouette edge loops meet at a single vertex, each loop can be assigned two unique silhouette edges (one incoming and one outgoing) at the vertex. This implies that no edge instances need to be shared between the loops.

2.6.3 Contribution

The proof shows that *possible silhouettes*, computed as described above, always can be split into separate closed loops, which means that the creation and rendering of a hard shadow quad and wedge can be done only dependent on the corresponding silhouette edge and the light source, and totally independently of other possible silhouette edges. It should be noted that it is not necessary to actually compute the splitting.

The algorithm in **III** and **IV** computes the correct soft shadows for an object whose silhouette remains constant as seen from all locations within the light source. Since this proof states that all possible silhouette edges can be grouped into closed loops, this fact might be used to create correct soft shadows for more general objects as well. This is briefly discussed in paper **IV**, in section 5.1, on how to reduce the overlap artifact.

3 Discussion

The first four papers in this thesis presented the evolution of the algorithm. In **III**, the algorithm significantly matured through major improvements, and I refer to the algorithm in **III** and **IV** as the soft shadow volume algorithm.

The algorithm in **III** and **IV** can handle any planar light source in two slightly different ways; it can use a bounding rectangle and precompute the coverage texture from 1) either a discretized two-dimensional image of the light source, or 2) the exact representation of the planar light source. The planar light source could also be translated, sheared or rotated without requiring recomputation of the coverage textures.

Spheres have been used as simple examples of volumetric light sources. Volumetric lights could also be created from combining several area light sources. For instance, a cubical light can be represented using 6 rectangular area light sources.

It should be mentioned that the single silhouette artifact [6] can be accentuated if the light source is moving. On the other hand, splitting the light source into several smaller area light sources, covering the same area as the larger, can remove the artifacts, at an additional cost.

4 Future Work

I will continue the research on real-time soft shadows, since paper **IV** has given rise to several new questions. The last algorithm of this thesis is based on two major approximations that leave room for improvements. My experience is that the overlap artifact typically is more apparent than the single silhouette artifact – at least for complex objects. To solve the overlap artifact, information about which parts of the light source that are covered has to be stored. Regarding the single silhouette artifact, it should be possible to remove by adding wedges from silhouettes created from more than one sample point on the light source. By selectively choosing the added sample points, it is likely that this latter solution can be done without extensively increasing the computational load. Thus, the next goal is real-time soft shadows that are visually indistinguishable from correct ones.

Other interesting subjects for investigation are if the soft shadow volume algorithm can be used for rendering of soft shafts of lights or as a hardware accelerated solution to the closely related problem of cell-based occlusion culling [3, 13].

5 Acknowledgements

My first thanks goes to Tomas Akenine-Möller, my supervisor. Without his help my life as a Ph.D. student would have been much more complicated, if not almost impossible. He is a great person, both as a friend and as a supervisor. The second thanks goes to my professor Per Stenström who has forced me to learn how to perform and write about academic research - not without me silently swearing from time to time, just to realize in the end that he is right. The third thanks, and a major one, goes to my grandmother for following my research, keeping track of my publishing results and also forcing me to explain my algorithms and work to someone who has never used a computer.

There are several good fellows at the department I like to thank for various reasons, such as Björn, Thomas, Jim, Biff, Fredrik W, Cecilia and Jonas J. I want to thank Jonas Lext as a computer graphics colleague and all the other boys and girl at Illuminate Labs. I also thank Henrik Holmdahl and Erich Björnekull for lots of beer talk and the former also for keeping track of the games world for me.

Other thanks, more closely related to soft shadows, goes to Eric Haines, Michael Dougherty and Michael Mounier, Mark Segal and Michael Doggett at ATI, and Greg James, Gary King, Randy Fernando, Mark Kilgard and Chris Seitz at NVIDIA.

The final thanks goes to my whole family, and this includes a special thanks to my brother Tor for, among other things, keeping the engine on the motor cycle alive, and to my beloved Ewelina for making my heart burning!

References

[1] Agrawala, M., R. Ramamoorthi, A. Heirich, and L. Moll, "Efficient Image-Based Methods for Rendering Soft Shadows," *Proceedings of ACM SIGGRAPH 2000*, ACM Press/ACM/Siggraph, New York. K. Akeley, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 375–384. 3, 9, 10

[2] Akeley, Kurt, and Pat Hanrahan, ´´Real-Time Graphics Architectures," Course CS448A Notes, Fall 2001. `http://graphics.stanford.edu/courses/cs448a-01-fall/lectures/lecture1/walk015.html`. 3

[3] Akenine-Möller, Tomas, and Eric Haines, *Real-Time Rendering*, AK Peters, Ltd., 2nd edition, June 2002. 2, 3, 23

[4] Akenine-Möller, Tomas, and Ulf Assarsson, "Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges," *Eurographics Workshop on Rendering 2002*, pp. 309–318, June 2002. 11

[5] Amanatides, J., "Realism in Computer Graphics: A Survey", *IEEE Computer Graphics and Applications*, Vol 7(1), pp 44-56, January 1987. 3

[6] Assarsson, Ulf, and Tomas Akenine-Möller, "A Geometry-Based Soft Shadow Volume Algorithm Using Graphics Hardware," *Proceedings of ACM SIGGRAPH 2003*, Pages 511–520, 2003. 3, 4, 18, 21

[7] Assarsson, Ulf, Michael Dougherty, Michael Mounier, and Tomas Akenine-Möller, "An Optimized Soft Shadow Volume Algorithm with Real-Time Performance," *Graphics Hardware 2003*, ACM SIGGRAPH / Eurographics Workshop Proceedings, Pages 33–40, 2003. 7

[8] Brabec, Stefan, and Hans-Peter Seidel, "Single Sample Soft Shadows using Depth Maps," *Graphics Interface 2002*, pp. 219–228, 2002. 5, 16

[9] Brotman, Lynne Shapiro, and Norman I. Badler, "Generating Soft Shadows with a Depth Buffer Algorithm," *IEEE Computer Graphics and Applications 4*, 10, pp. 5–12, October, 1984. 4

[10] Chan, Eric, and Fredo Durand, "Rendering Fake Soft Shadows with Smoothies," *Rendering Techniques 2003 (14 Eurographics Symposium on Rendering)*, Springer-Verlag, 2003. 5, 6

[11] Cohen, M.F. and D.P. Greenberg, "The Hemi-Cube: A Radiosity Solution for Complex Environments," *Computer Graphics*, 19(3), pp. 31-40, July 1985. 3

[12] Cohen, M. F., and J. R. Wallace, *Radiosity and Realistic image Synthesis*, Academic Press Professional, 1993. 9

[13] Cohen-Or, D., Y Chrysanthou, F. Durand, N. Greene, V. Koltun, and C. Silva, "Visibility: Problems, Techniques, and Applications," *Course #30 at SIGGRAPH 2001*, August, 2001. 3, 23

[14] Crow, Frank, "Shadow Algorithms for Computer Graphics," *Computer Graphics (Proceedings of ACM SIGGRAPH 77)*, pp. 242–248, July 1977. viii, 2, 3, 6, 11, 18

[15] Drettakis, George, and Eugene Fiume, "A Fast Shadow Algorithm for Area Light Sources Using Backprojection," *Computer Graphics (SIGGRAPH 1994)*, Annual Conference Series, pp 223–230, ACM SIGGRAPH, 1994. 3

[16] Everitt, Cass, and Mark Kilgard, "Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering," `http://developer.nvidia.com/`, 2002. viii, 2, 6, 13, 18

[17] Haines, Eric, "Soft Planar Shadows Using Plateaus," *Journal of Graphics Tools*, **6**(1):19–27, 2001. 5

[18] Hasenfratz, J.-M., M. Lapierre, N. Holzschuch, and F.X. Sillion, "A Survey of Real-Time Soft Shadows Algorithms," Eurographics State-of-the-Art Report, *Eurographics*, 2003. 3, 4, 5, 7, 11

[19] Heckbert, Paul, and Michael Herf, *Simulating Soft Shadows with Graphics Hardware*, Carnegie Mellon University, Technical Report CMU-CS-97-104, January, 1997. 5, 11

[20] Heidmann, Tim, "Real shadows, real time," *Iris Universe*, no. 18, pp. 23–31, November 1991. viii, 2, 4

[21] Isard, M., M. Shand, and A. Heirich, "Distributed rendering of interactive soft shadows," *4th Eurographics Workshop on Parallel Graphics and Visualization*, pp. 71–76, Eurographics Association, 2002. 2, 4, 5

[22] Mark, William R., R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard, "Cg: A system for programming graphics hardware in a C-like language," *Proceedings of ACM SIGGRAPH 2003*, pp 896–907, 2003. 6

[23] Nishita, T. and E. Nakamae, "Half-Tone Representation of 3-D Objects Illuminated by Area or Polyhedron Sources," *Proc. of IEEE Computer Societys Seventh International Computer Software and Applications Conference (COMPSAC83)*, pp. 237-242, Nov 7-11, 1983. 3

[24] Nishita, T., I. Okamura, and E. Nakamae, "Shading Models for Point and Linear Sources," *ACM Trans. on Graphics*, 4(2), pp. 124-146, April 1985. 3

[25] Nishita, T. and E. Nakamae, "Continuous Tone Representation of Three-Dimensional Objects Taking Account of Shadows and Interreflection," *Computer Graphics*, 19(3), pp. 23-30, July 1985. 3

[26] Parker, S., P. Shirley, and B. Smits, *Single Sample Soft Shadows*, University of Utah, Technical Report UUCS-98-019, October 1998. 5

[27] Rademacher, Paul, Jed Lengyel, Eward Cutrell, and Turner Whitted, "Measuring the Perception of Visual Realism in Images," *Eurographics Workshop on Rendering 2001*, 2001. 2

[28] Sen, Pradeep, Michael Cammarano, and Pat Hanrahan, "Shadow Silhouette Maps," *Proceedings of ACM SIGGRAPH 2003*, Pages 521–526, 2003. 6

[29] Soler, Cyril, and F. X. Sillion, "Fast Calculation of Soft Shadow Textures Using Convolution," *Computer Graphics (SIGGRAPH 1998)*, Anual Conference Series, pp 321–332, ACM SIGGRAPH, 1998. 3, 5, 11

[30] Wanger, Leonard R., James A. Ferwerda, and Donald P. Greenberg, "Perceiving Spatial Relationships in Computer-Generated Images," *IEEE Computer Graphics & Applications*, pp. 44–58, 1992. 1

[31] Wanger, Leonard, "The effect of shadow quality on the perception of spatial relationships in computer generated imagery," *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, vol 25, no. 2, pp 39–42, 1992. 2

[32] Williams, Lance, "Casting Curved Shadows on Curved Surfaces," *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, pp. 270–274, August 1978. 5, 6

[33] Wyman, Chris, and Charles Hansen, "Penumbra Maps: Approximate Soft Shadows in Real-Time," *Proceedings of the 2003 Eurographics Symposium on Rendering*, pp 202–207, 2003. 5

The numbers after an entry list the pages that have a reference to that entry.

Paper I

Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges

Published at

13h Eurographics Workshop on Rendering,
Eurographics, Pages 309–318, June 2002.

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

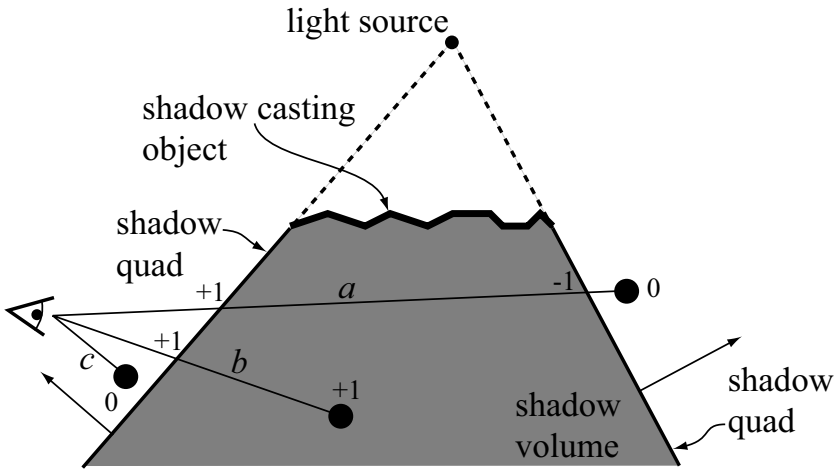


Figure 1: The standard shadow volume algorithm. Ray *b* is in shadow, since the stencil buffer has been incremented once, and the stencil buffer values thus is $+1$. Rays *a* and *b* are not in shadow, because their stencil buffer values are zero.

SV algorithm is a multipass algorithm. 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. Back-facing 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.

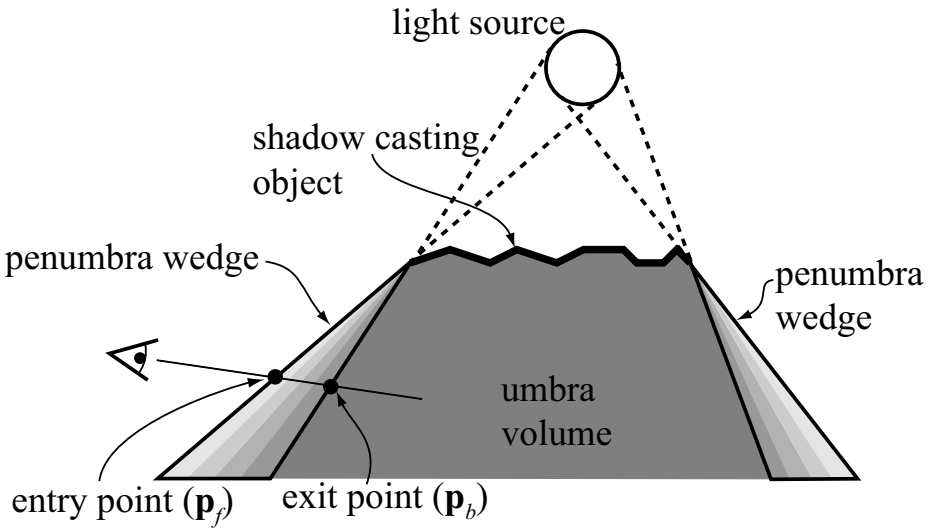


Figure 2: The new algorithm uses penumbra wedges to capture the soft region in the shadow.

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 over-

lapping (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:   p_f = computeEntryPointOnWedge(x,y);
 5:   p_b = computeExitPointOnWedge(x,y);
 6:   p = point(x,y,z); – z is the Z-buffer value at (x,y)
 7:   p_i = choosePointClosestToEye(p, p_b);
 8:   s_f = computeLightIntensity(p_f);
 9:   s_i = computeLightIntensity(p_i);
10:   addToLIBuffer(round(255 * (s_i – s_f)));
11: end;
```

Lines 4 and 5 compute the points on the wedge where 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 con-

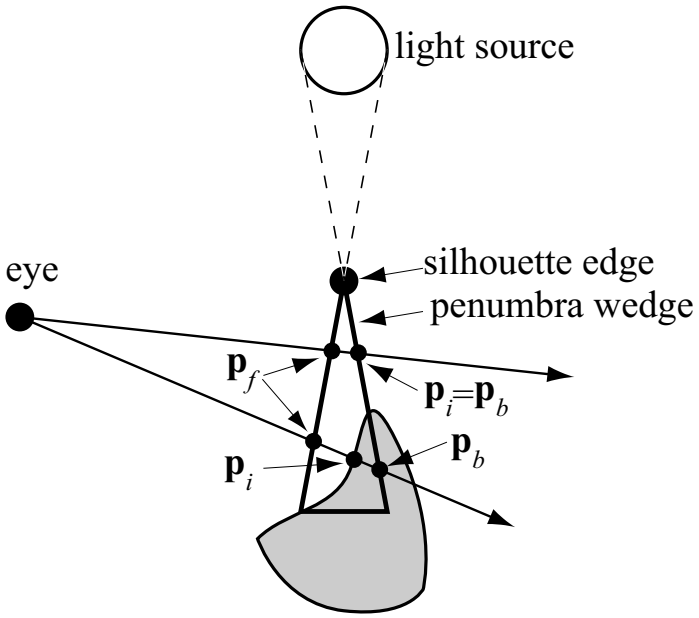


Figure 3: Illustration of the $\mathbf{p}_f$, $\mathbf{p}_b$, and $\mathbf{p}_i$ values for two rays.

structed, 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.

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.

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

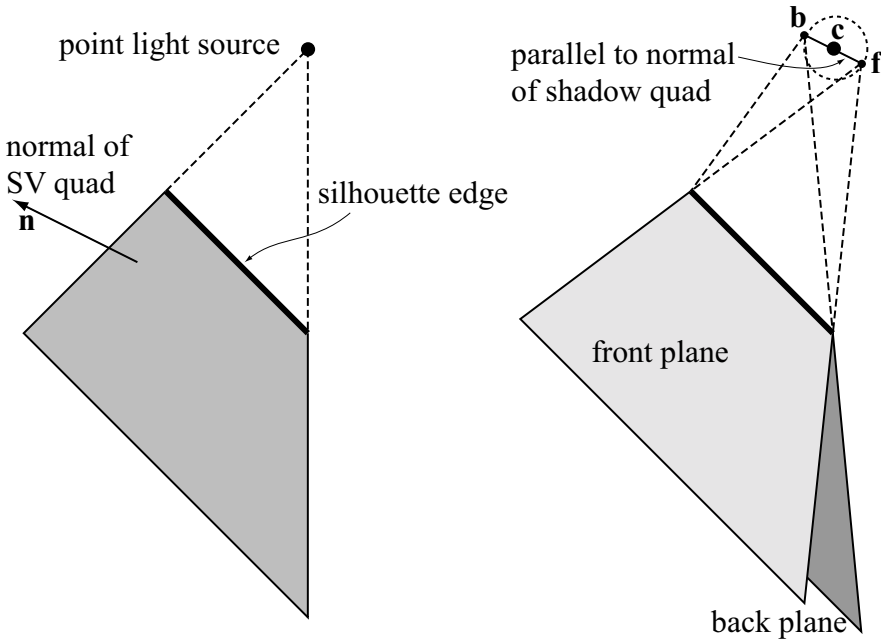


Figure 4: Left: shadow volume quad. Right: front and back planes of a wedge.

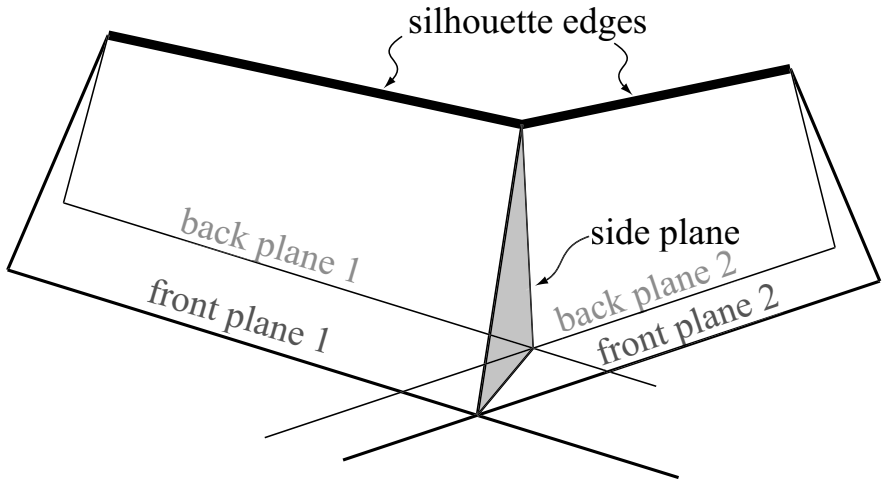


Figure 5: Two adjacent wedges in general configuration. Their front and back planes define their shared side plane.

defined as shown in Figure 6.

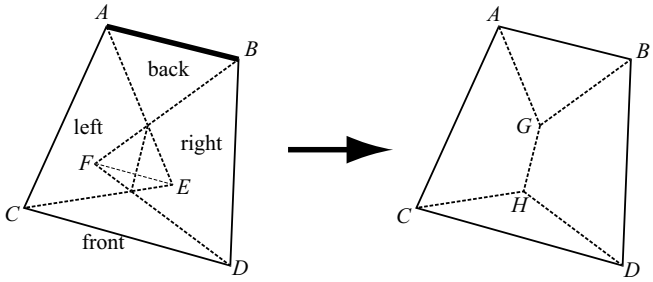


Figure 6: Left: *ABDC* define the front plane's quadrilateral, and *ABFE* the back plane's quadrilateral, *ACE* the left side plane, and *BFD* the right side plane. The wedge on the right is used when rendering soft shadow, in cases where the side planes overlap.

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

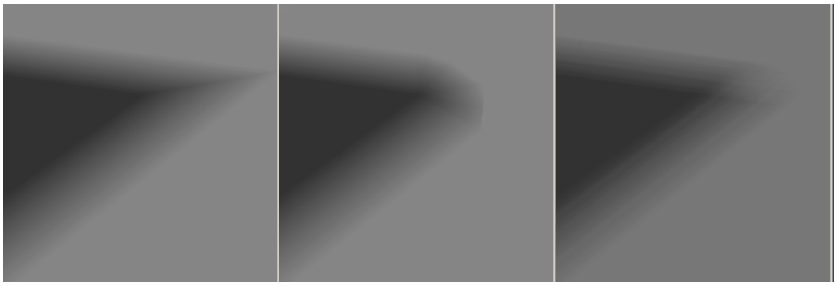


Figure 7: A (partial) soft shadow of a triangle with an acute angle. Left: one wedge per silhouette edge. Middle: one wedge per silhouette edge plus 6 extra wedges around each vertex. Right: Heckbert/Herf shadows. Also, when comparing images on screen, a stepping effect of Heckbert/Herf shadows is apparent,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

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.

4.2 Light Intensity Interpolation

In this section, we describe how the light intensity, s, for a point, **p**, inside a penumbra wedge is computed. Recall that **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.

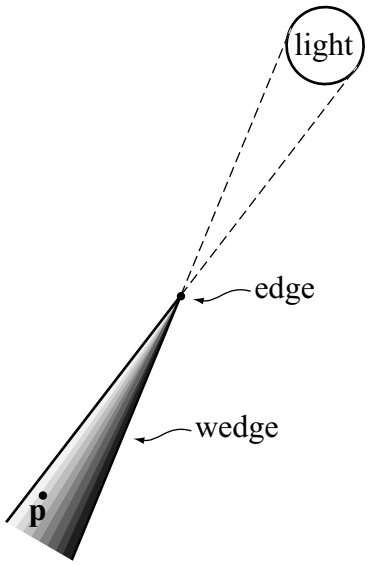


Figure 8: The point **p** is in the penumbra wedge volume. The rationale for our interpolation scheme is that s should approximate how much the point **p** "sees" of the light source.

Clearly, the minimal level of continuity of s between two adjacent wedges should be C^0. Our first attempt created a ray from **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 com-

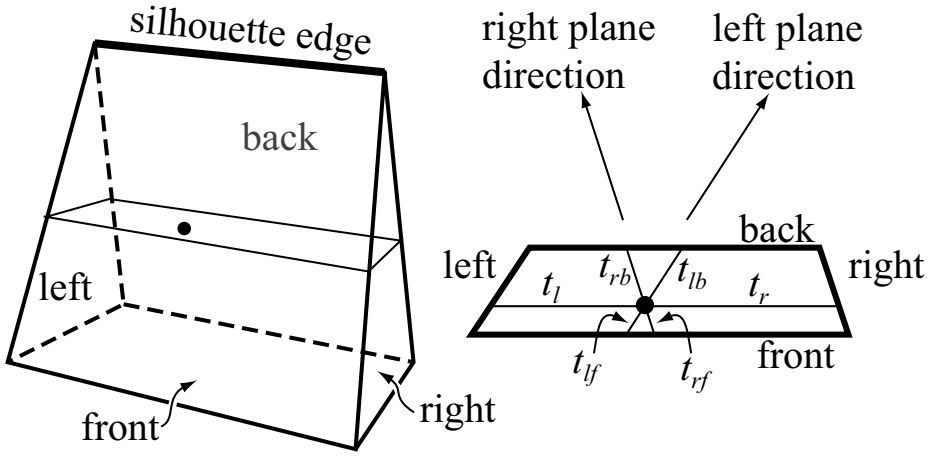


Figure 9: Light intensity interpolation inside a penumbra wedge. Left: penumbra wedge. Right: cross-section of the wedge, where the positive intersection distances, t's, from the point (black dot) to the planes are shown.

putations are:

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

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

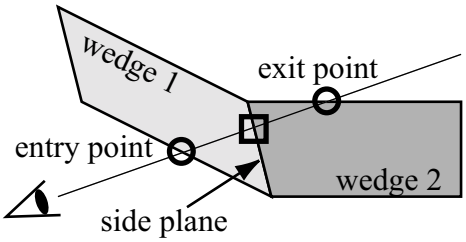


Figure 10: A cross-section view through two adjacent wedges. The square shows where the ray intersects the shared side plane of the wedges. The LI values for wedge 1 and 2 in the shared side plane cancel each other.

entering 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.

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

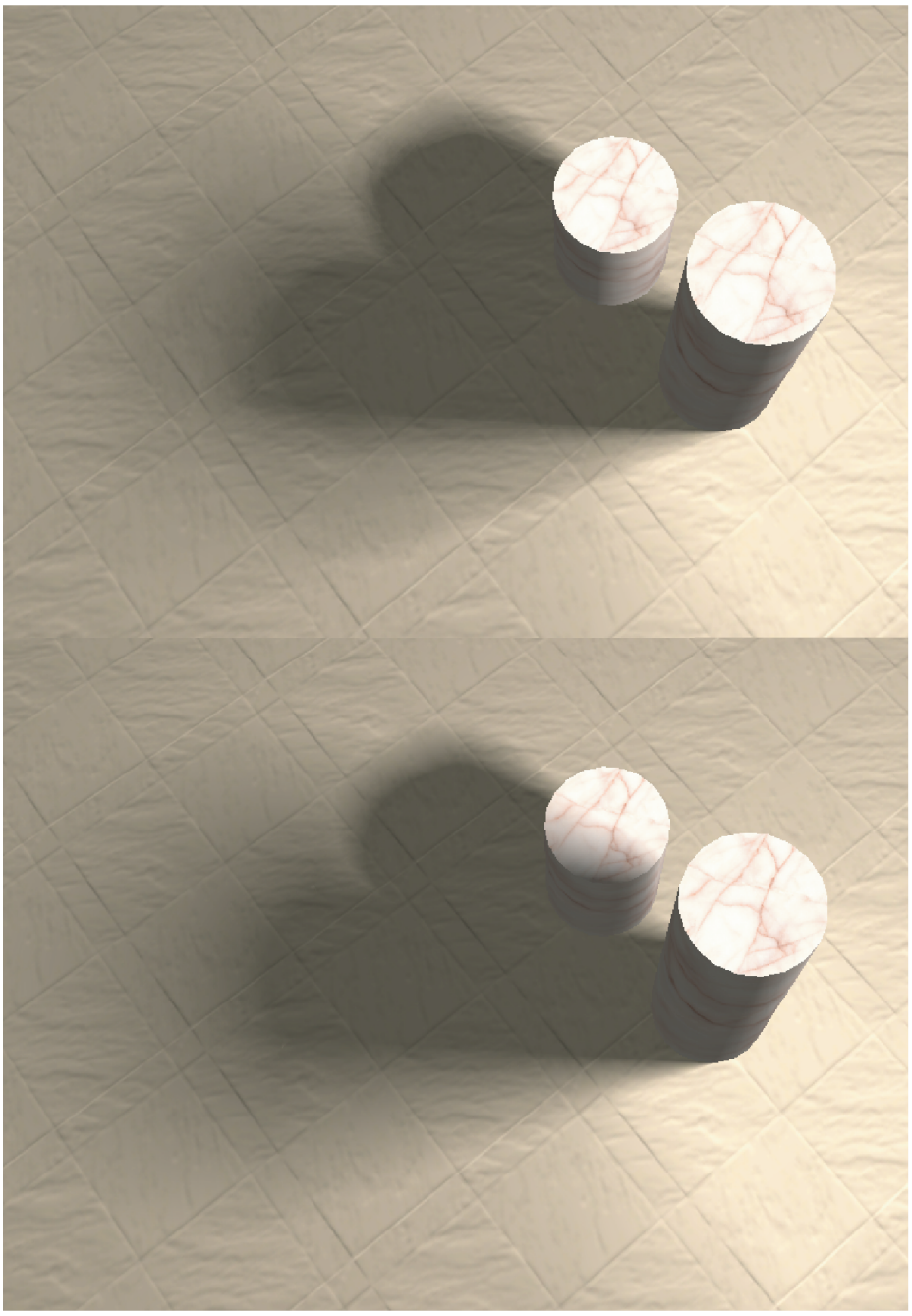


Figure 11: Overlapping soft shadows. Top: rendered with Heckbert/Herf's algorithm with 128 samples. Bottom: result produced with our algorithm.

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. 34, 36

[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. 36

[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` 37, 47, 48, 49

[4] Fernando, R., S. Fernandez, L. Bala, and D. P. Greenberg, "Adaptive Shadow Maps," *SIGGRAPH 2001 Proceedings*, pp. 387–390, August 2001. 36

[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. 35

[6] Haines, Eric, and Tomas Möller, "Real-Time Shadows," *Game Developers Conference*, March 2001. 35

[7] Haines, Eric, "Soft Planar Shadows Using Plateaus," *Journal of Graphics Tools*, vol. 6, no. 1, pp. 19–27, 2001. 35, 40, 48

[8] Hart, David, Philip Dutre, and Donald P. Greenberg, "Direct Illumination with Lazy Visbility Evaluation," *SIGGRAPH 99 Proceedings*, pp. 147–154, August 1999. 36

[9] Heckbert, P., and M. Herf, *Simulating Soft Shadows with Graphics Hardware*, Technical Report CMU-CS-97-104, Carnegie Mellon University, January 1997. 34, 35, 46, 49

[10] Heidmann, Tim, "Real shadows, real time," *Iris Universe*, No. 18, pp. 23–31, Silicon Graphics Inc., November 1991. 36

[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. 35, 36

[12] Morein, Steve, "ATI Radeon—HyperZ Technology," *SIGGRAPH/Eurographics Graphics Hardware Workshop 2000*, Hot3D session, 2000. 45

[13] Parker, S., Shirley, P., and Smits, B., *Single Sample Soft Shadows*, TR UUCS-98-019, Computer Science Department, University of Utah, October 1998. 34, 36, 44

[14] Pineda, Juan, "A Parallel Algorithm for Polygon Rasterization," *SIGGRAPH 88 Proceedings*, pp. 17–20, August 1988. 46

[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. 36

[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. 36

[17] Soler, Cyril, and François X. Sillion, "Fast Calculation of Soft Shadow Textures Using Convolution," *SIGGRAPH 98 Proceedings*, pp. 321–332, July 1998. 34, 36, 46

[18] Stark, Michael M., and Richard F. Riesenfeld, "Exact Illumination in Polygonal Environments using Vertex Tracing," *Rendering Techniques 2000*, pp. 149–160, June 2000. 36

[19] Williams, Lance, "Casting Curved Shadows on Curved Surfaces," *SIGGRAPH 78 Proceedings*, pp. 270–274, August 1978. 35

[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. 35

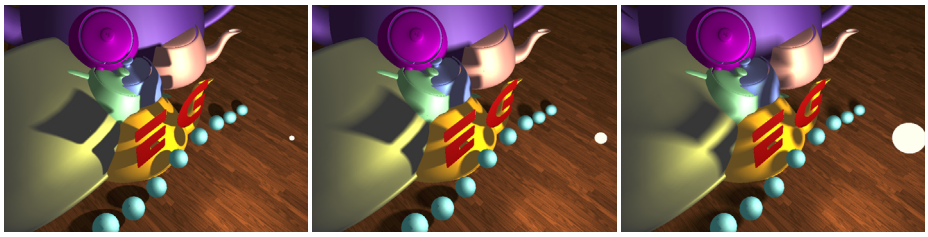


Figure 12: Increasing light source size from left to right. Only the EG logo, and the spheres are casting shadows. Notice that the umbra region correctly gets smaller and smaller with increasing light source.

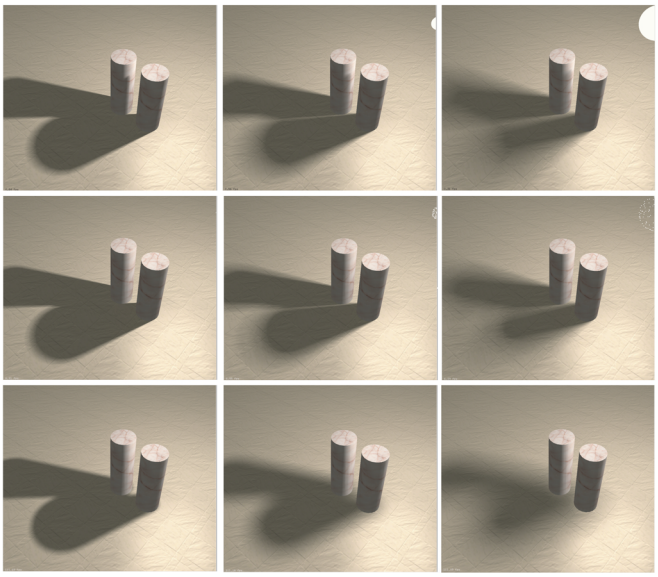


Figure 13: Comparison of our algorithm (top), Heckbert/Herf (middle), and Soler/Sillion (bottom). Our algorithm provides the accuracy of the much more expensive Heckbert/Herf algorithm. In addition, our algorithm handles all surfaces, and so casts a shadow from the right cylinder onto the left, which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

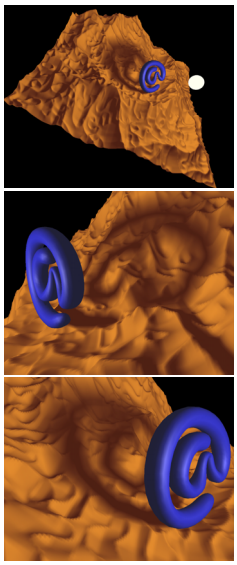


Figure 14: A fractal landscape with 100k triangles is used as a complex shadow receiver from different viewpoints.

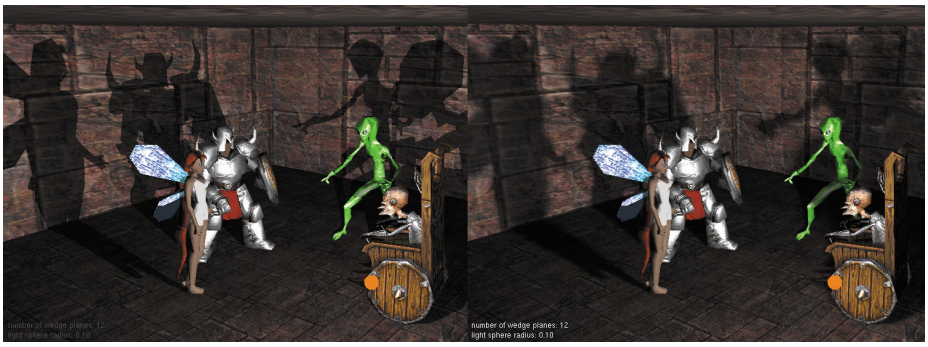


Figure 15: Rendered at 5 fps on a 1.5 GHz PC with a Geforce3. We modified nVidia's shadow volume demo (left) to render soft shadows (right).

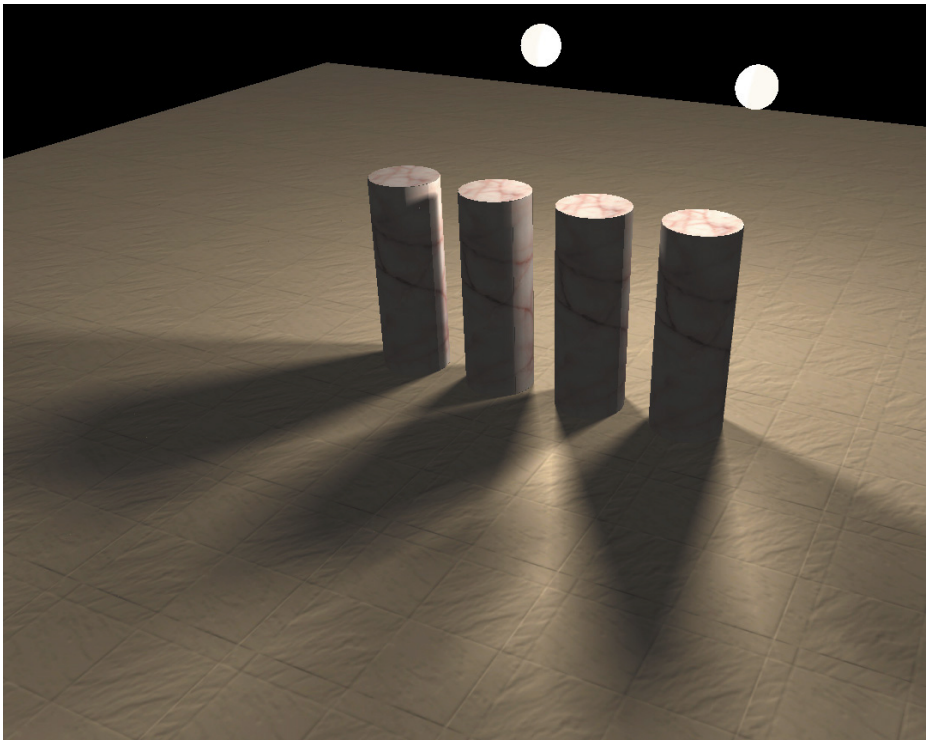


Figure 16: Two light sources are used in this simple test scene.

Paper II

Interactive Rendering of Soft Shadow using an Optimized and Generalized Penumbra Wedge Algorithm

Conditionally accepted for publication by

the Visual Computer,
submitted May 2002.

Interactive Rendering of Soft Shadows using an Optimized and Generalized Penumbra Wedge Algorithm

Ulf Assarsson and Tomas Akenine-Möller

Chalmers University of Technology
Hörsalsvägen 11
412 63 Gothenburg
Sweden

Abstract

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

CR Categories: I.3.7 [Computer Graphics]Three-Dimensional Graphics and Realism.
Keywords: soft shadows, graphics hardware, shadow volumes.

1 Introduction

Rendering realistic shadows in real time is highly desirable, both for increasing the level of realism, and because shadows give important spatial clues. For real-time purposes, it is common to approximate all light sources as point lights, i.e., with an infinitely small extension. This gives rise to, so called hard shadows, where the transition from no shadow to full shadow is instant. However, in reality, all light sources have some extension (area or volume), which gives a smooth transition, called the penumbra region, from no shadow to full shadow,

called the umbra region. Several soft shadow algorithms exist, but most of them suffer from 1) either not being suitable for real-time rendering, or 2) only being able to handle planar shadow receivers, or 3) suffering from sampling artifacts. Our recently presented penumbra wedge algorithm [1] can handle all of the following goals:

I. The softness of the penumbra should increase linearly with distance from the occluder, starting at zero at the occluder [16].

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

III. Typical sampling artifacts should be avoided. Often a number of superpositioned hard shadows can be discerned [18]. The result should be visually smooth [16].

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

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

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

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

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

The contributions of this paper are as follows; 1) We present a restructured version of our original algorithm that significantly reduces the number of calculations to rasterize a wedge. 2) A method for very efficient occlusion culling, made possible by the restructuring, is presented, and it gives general speedups of 3-4 times for our test scenes. 3) We show how the restructuring also enables the use of the *z-fail* algorithm [2] to correctly handle the case when the eye is inside a shadow region. Neither occlusion culling nor the z-fail algorithm can be

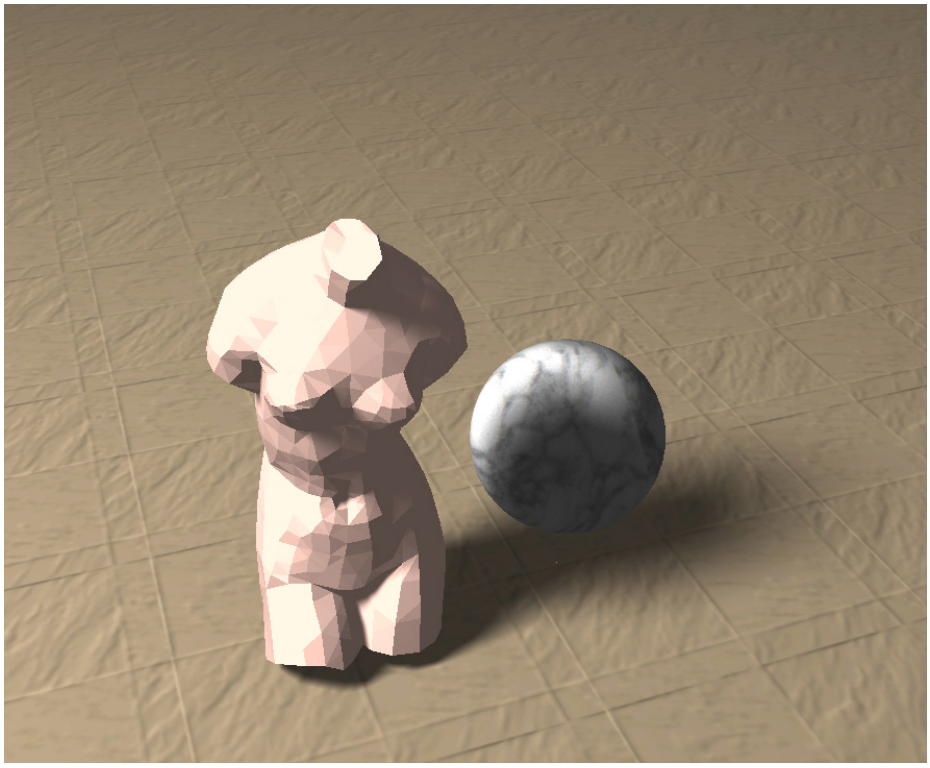


Figure 1: This image of Venus was rendered using the three pass algorithm (see Section 6.1) in 1 fps using a P4 1700MHz and a GeForce3 graphics accelerator. The Venus model casts soft shadows onto itself, the sphere and the floor. The image size is 640×427 pixels.

incorporated in any obvious way in the originally proposed algorithm without the restructuring. 4) We suggest two different implementations of the restructured algorithm; a single pass algorithm for a possible hardware implementation of the wedge rasterization, and a three pass algorithm when no special hardware support of rasterizing the wedges is available. Furthermore, we evaluate software implementations of the two algorithms and present figures for the number of memory access used and frame rates.

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

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

2 Review of the Soft Shadow Algorithm using Penumbra Wedges

In 1977, Crow presented his shadow volume (SV) algorithm for hard shadows [3]. Tim Heidmann extended the algorithm, in 1991, with hardware acceleration using the stencil buffer [12]. For each shadow casting object, its shadow volume is created. The shadow volume is created in the following manner. Each silhouette edge, as seen from the light position, and rays from the edge's two vertices in the direction from the light source forms a quadrilateral (quad). Together, all quads represent the shadow volume (see Figure 2). First the scene is rendered from the eye with only ambient light enabled. Secondly, all front facing quads, as seen from the eye, of the shadow volumes are rendered to the stencil buffer, incrementing each rasterized pixel that passes the depth test. Each pixel in the stencil buffer has now recorded the number of times a virtual ray from the eye through the pixel to the point represented by its z-value, enters a shadow region. Then, all the back facing quads are rendered, counting the number of times the virtual rays exits the shadow regions. Afterwards, if the stencil value for a pixel is larger than zero, the point is in shadow. That is, the

virtual ray from the eye to the point enters shadow regions more times than it exits shadow regions on its way from the eye to the point. This algorithm has to be modified if the eye is inside a shadow volume (see Section 5). Finally, the stencil buffer is used as a mask when rendering the specular and diffuse contribution. For the stencil passes, the depth test is set to accept objects closer than the stored value, as usual, but no new depth values are written to the depth buffer and no color values are written to the frame buffer.

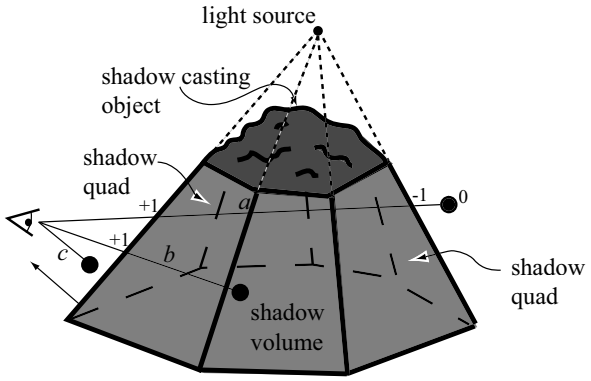


Figure 2: The standard shadow volume algorithm. The shadow volume here consists of seven quads. Ray *b* is inside shadow with a stencil value of 1. Ray *a* and *c* are outside shadow with stencil values of 0.

Our soft shadow algorithm [1] is based on Crows's SV algorithm in the sense that it also uses shadow volumes and a stencil buffer. However, instead of an instant transition from no shadow to full shadow, given by the quads, those are replaced by penumbra wedges, where the light intensity (LI) varies linearly inside the wedge (see Figure 3 and Figure 4). The penumbra wedges can be thought of as a new volumetric primitive, conceptually rasterized as outlined below:

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

The wedge is rasterized into a 16-bit stencil buffer, which we further on refer to as the light intensity (LI) buffer.

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

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

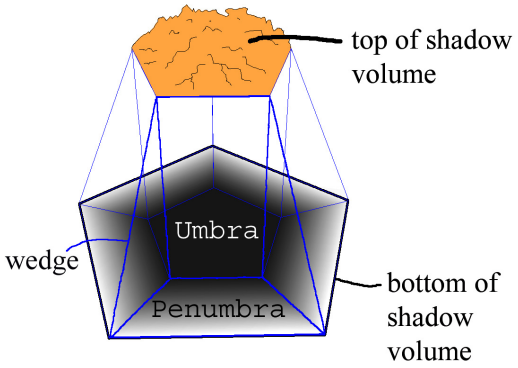


Figure 3: Example of a shadow volume for five silhouette edges. Each wedge is outlined.

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

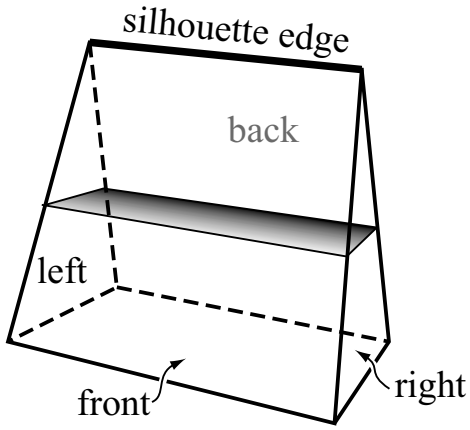


Figure 4: A penumbra wedge with its light intensity interpolation inside.

3 A Restructured Soft Shadow Algorithm

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

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

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

Using the fact that the rasterization of the back planes can be skipped, the

algorithm can be restructured as follows:

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

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

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

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

4 Occlusion Culling

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

Normally, hardware occlusion culling avoids rendering for pixel tiles where the object to be rendered is behind everything that is stored in the z-buffer positions for the tile. This can be done in hardware by storing the maximum z-value, z_{max}, for each tile [15]. A common tile size is 8×8 pixels.

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

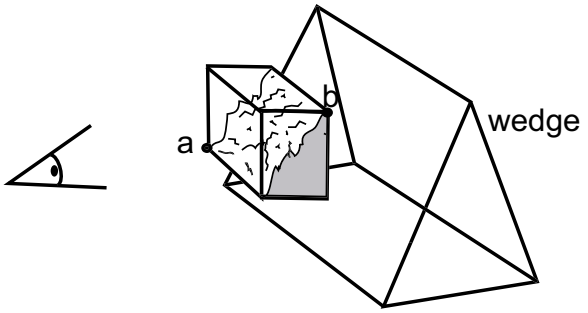


Figure 5: The screen space bounding box of a tile is tested for intersection with the wedge. Only if they intersect, the tile is rasterized for the penumbra contribution. Here, the box contains a piece of a fractal mountain (see Figure 10). Since the box does not intersect the wedge, in this example, the tile will be culled from penumbra rasterization.

Before rasterizing a wedge, its screen space plane equations are precomputed in a setup routine. Upon rasterizing the wedge triangles, the screen space axis aligned bounding box of a tile that is about to be rasterized is tested for intersection with the wedge (see Figure 5). If the bounding box is outside the wedge, the whole tile is culled. The Separating Axis Theorem can be used to determine whether they overlap [6]. The theorem states that for two convex, disjoint polyhedra, A and B, there exists a separating axis where the projections of the polyhedra also are disjoint. Furthermore, it states that it is sufficient to test only the axes that are orthogonal (i.e., the planes with its normal orthogonal) to a face of A or B, or an edge from each polyhedron. If such an axis cannot be found, we know the box and the wedge are overlapping. Testing all the axes

is often unnecessarily time consuming. It is usually better to only do the tests corresponding to the faces of A or B, and ignore the tests corresponding to the edges of the polyhedra. This may sometimes give incorrect indication of overlap, but that will only force the tile to be rasterized with occlusion tests for each pixel, and causes no visual error. The advantage is that the tile overlap test will be significantly faster.

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

When testing the box vertices against a wedge plane, it is sufficient to test only the closest and the furthest of the vertices instead of all eight, which saves a lot of computations [7, 8]. The two vertices are easily recognized by the signs of the *x, y* and *z* components of the normal of the wedge plane (see Figure 6).

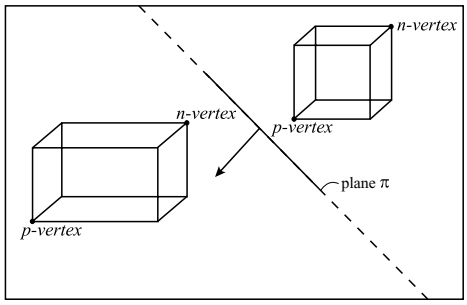


Figure 6: This figure illustrates the *n* and *p* vertices of two boxes with respect to a plane.

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

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

5 Eye Inside Shadow Regions

The original SV algorithm [3] , does not properly handle the case when the eye is inside a shadow volume, and the same applies to our original algorithm. An elegant solution for the SV algorithm was documented by for instance Everitt and Kilgard [2], although the algorithm had been known to the gaming industry since 2000 through John Carmack. Instead of determining if a point is in shadow by testing intersections with the shadow volumes of a virtual ray from the eye to the point, a virtual ray from the point to the infinity could be tested. In this way, the testing will be independent of the eye position. In practice, this is achieved by modifying the stencil buffer passes. The first stencil buffer pass becomes: render all back facing shadow volume polygons and increment the stencil value when the polygon is equal to or farther than the stored z-depth. In the second stencil pass, all front facing shadow volume polygons are rendered, decrementing the stencil value when the polygon is equal to or farther than the stored z-depth. This algorithm is often called the z-fail algorithm [2].

Since the depth test has been altered, it is now also necessary to render the *top* of the shadow volume. The top consists of all polygons of the shadow generator that are front facing with respect to the light source center. If the shadow volume is of finite length, the volume must be closed at the *bottom*, by for instance adding a far capping polygon. The top and the bottom polygons should be rendered in exactly the same way as the shadow quads. Problems can still occur if the polygons are clipped by the far plane of the view frustum. This could, however, be solved by extensions in the rasterizing hardware. Such an extension is included in NVIDIA's GeForce3 [2].

The solution for the SV algorithm described above can be applied to our soft shadow algorithm as well. If the wedge planes are of finite length, a capping *bottom plane* of the wedge must be added too, and rasterized by **rasterizePenumbra**() when it is back facing. In **rasterizeUmbra**(), the front planes of all wedges plus the top and bottom polygons of the shadow volume are ras-

terized. The pseudo code for rasterizing a soft shadow volume now becomes:

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

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

6 Implementation

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

6.1 Three Pass Algorithm

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

Current graphics hardware normally do not have a 16-bit stencil buffer, but an 8-bit stencil buffer usually suffices for the rendering of the front planes. Ini-

tially, the 8-bit stencil buffer is cleared with a value of 0. In the first pass all front facing front planes of the wedges are rasterized, and for each time a pixel passes the depth-test, the corresponding stencil value is decremented by one. In the second pass all back facing front faces are rasterized similarly, but incrementing the stencil values by one instead. The buffer is then added to the 16-bit software light intensity (LI) buffer, which has been initialized with a value of 255, pre-multiplying each 8-bit stencil value with 255, or -255 if the z-fail algorithm is used. In the third pass, software rendering of the penumbra regions is done as outlined in the pseudo code for **rasterizePenumbra**() in section 3.

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

6.2 Single Pass Algorithm

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

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

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

7 Simulation Results

In this section, experimental results for the three pass algorithm and the single pass algorithm are presented. We did not implement the single pass algorithm in custom hardware. Instead, all tests were done using software implementations. The only hardware acceleration used was for rendering the front planes

for the umbra contribution in **rasterizeUmbra**() in the three pass algorithm. For this, we rendered quads into an 8-bit stencil buffer using a GeForce3 graphics accelerator.

The test scenes used are shown in Figure 9, 10, and 11. All the test scenes were rendered with an image size of 640×427 pixels. The software rasterization is highly fill-rate limited. A hardware implementation would presumably suffer significantly less from this.

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

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

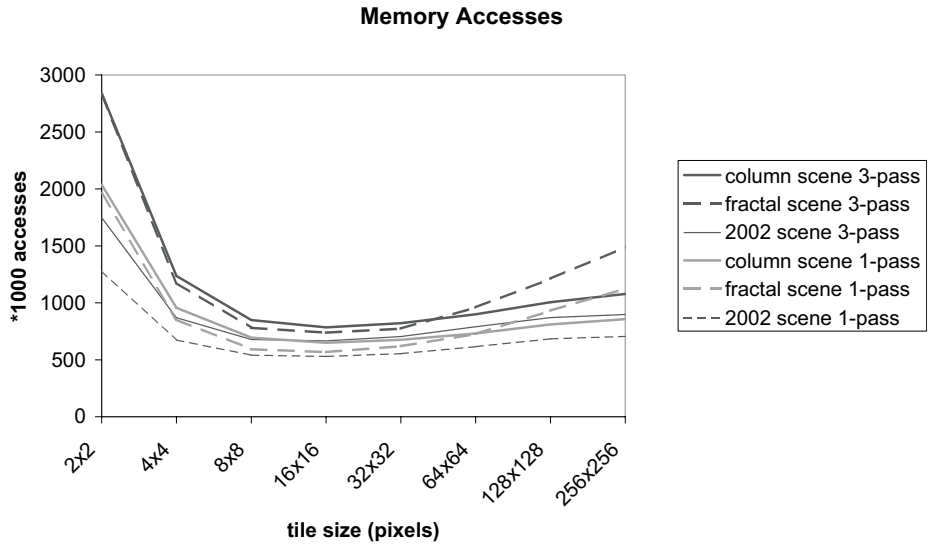


Figure 7: This graph shows the total number of z-buffer reads and stencil buffer writes for different tile sizes, when rendering one frame of the column scene, fractal scene, and 2002-scene. The results from using the three pass algorithm and the single pass algorithm are shown. The images can be seen in Figure 9, 10, and 11.

Figure 7 shows the total number of stencil buffer writes and z-buffer reads, needed to render one frame of each test scene. As can be seen, the optimal tile size for minimizing the number of memory accesses, is from 8×8 to 32×32 pixels for these scenes. This correlates very well with the graph for the frame rates, in Figure 8. It means that the optimal tile size for minimizing memory

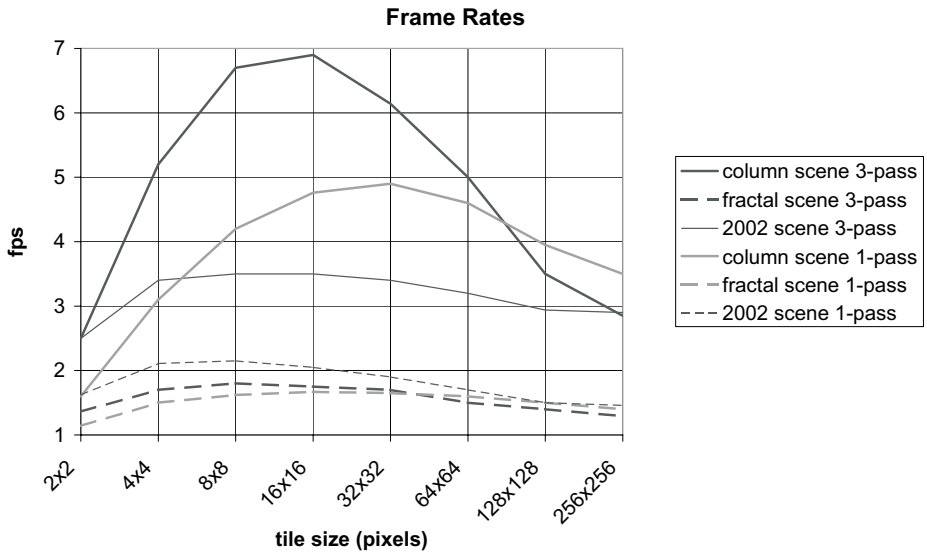


Figure 8: The graph shows the frame rates, using different tile sizes for rendering the column scene, fractal scene, and 2002-scene. Result from the single pass algorithm and the three pass algorithm are shown.

accesses matches the optimal tile size for minimizing the rasterization work, which seems natural. As expected, the frame rates are higher for all scenes using the three pass algorithm, than the single pass algorithm, since we do not have any hardware implementation of the latter.

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

The occlusion culling described in section 4 generally gave a speedup of 3-4 times for the penumbra rasterization, resulting in an up to three-fold overall frame rate improvement compared to not using occlusion culling. If only z_{max} is used, which is common in current hardware, and not z_{min}, then practically no speedup is obtained in any of the test scenes, since almost nothing in the scenes occludes the shadows. This is a strong argument for accepting the extra com-

plexity of storing z_{min} in hardware as well. As an example of this, only using z_{max} lowers the frame rate from 7.0 frames per second (fps), to 2.6 fps, and increases the number of depth buffer read accesses with 50% for the three pass algorithm with hardware rendering of the umbra contribution, when rendering the column scene in Figure 9. The number of stencil buffer write accesses is unaffected by occlusion culling, since the occlusion culling only avoids unnecessary rasterization. If a tile is rasterized although it correctly could be culled, the corresponding points in the depth buffer are tested whether they are inside the wedge. Since all points will be found outside, no stencil buffer values will be written for this tile.

In total, the three pass algorithm with occlusion culling, is up to almost six times faster for the column scene, in Figure 9, than the original algorithm. Our conclusion for the single pass algorithm is that the added complexity of rendering the umbra and penumbra contributions in the same pass definitely could pay off, for a hardware implementation. To get efficient occlusion culling both z_{min} and z_{max} should be used, and the optimal tile size is from 8×8 to 32×32 pixels.

All test results were done using a standard PC with an Intel P4 1.7 GHz, and a GeForce3 graphics card. For the Venus scene in Figure 1, we used a different interpolation technique that we have developed recently. This modification computes the light intensity more exactly than the previously presented linear interpolation [1], and thus avoids several of the artifacts with the old method. Currently, it uses much more calculations though, giving lower frame rates. We are currently refining and optimizing the modification and will present those details in a future paper.

8 Related Work

In this section, we will briefly mention the most relevant related algorithms for generating hard or soft shadows. For a more thorough presentation, consult Woo et al [20] or Haines and Möller [9].

There are two dominating algorithms for generating hard shadows in real-time. One is the shadow volume algorithm (see Section 2) presented by Crow in 1977 [3], and the other one is shadow mapping, which was first introduced by Williams in 1978 [19].

Shadow mapping, also called the shadow z-buffer algorithm, first renders a z-buffer image from the view of the light source. This shadow z-buffer image is then used to determine if an object point visible from the eye is also visible from the light source, and thus lit by it. The main problems with shadow mapping are 1) biasing is needed due to numerical imprecisions in the z-buffer, and 2) choosing a reasonable size of the shadow map to avoid aliasing. In 1987, Reeves et al. [17] introduced *percentage closer filtering* to reduce aliasing along shadow

edges. Shadow mapping with percentage closer filtering is now implemented in commodity hardware, such as the the GeForce3. Adaptive shadow maps [4], which iteratively refines the shadow maps where needed, could also be used to avoid aliasing. In 2000, Lokovic and Veach present an extension called *deep shadow maps* [14] that can render shadows from objects like hair, fur and smoke. It stores fractional visibility for each pixel at different depths.

A method for generating soft shadows for linear light sources was presented in 2000 by Heidrich et al [13]. Two shadow maps are created; one for each end point of the light source. The visibility is then interpolated across the linear light into a visibility map used at rendering.

In 1997, Heckbert and Herf presented their algorithm for generating hardware accelerated soft shadows [11]. Their method handles arbitrary shapes of the light source and arbitrary shadow generators. An extended light source is approximated by several point lights. For each point light, the corresponding hard shadows are generated by projecting the scene, as seen from the light source, onto a receiving plane. The major disadvantages are that many point light samples are needed (64-256) to give the impression of a soft shadow instead of several superimposed hard shadows, and the shadow receivers must be planar. It should be noted that non-planar shadow receivers could be split up in several individual planar objects and soft shadows could then be generated for each of them. However, for real-time purposes of complex receivers, this is mostly suitable for pre-generation of textures containing static soft shadows. Gooch et al. presented an alternative method for generating soft shadows in 1998 [5]. It produces more concentric hard shadows, which in general looks better, and thus requires fewer sample points. The method also project hard shadows onto planes and compute the average of these, but takes the sample points from a line parallel to the normal of the receiver that passes through the light center.

The SV algorithm could be used to create soft shadows, by simply averaging a number of images generated for different point light positions on the area or volume light source [2]. This method can be used to approximate any shape of the light source. However, n samples only allows for $n+1$ different shadow intensity values [13], and also requires a proportionally amount of computations. Our technique avoids this.

In 1998, Soler and Sillion presented an algorithm based on the insight that for parallel configurations a soft shadow image can be generated by convolving the hard shadow image with an image of the light source [18]. This approach was then extended in the same paper with a hierarchical error-driven algorithm to compute soft shadow textures with a given precision.

A technique for generating approximate planar soft shadows was presented 2001 by Haines [10]. It uses hardware and is thus very fast. The algorithm renders the soft shadow to a ground plane texture. It begins with a hard shadow from a point light, then uses the object's silhouette edges to generate penum-

brae in a single pass. Our original penumbra wedge algorithm [1], which is the base of this paper, can be seen as an extension of Haines' method and the SV algorithm. The umbra regions in Haines' method is the umbra region of a hard shadow from a point light source. Thus the region is overestimated. Our penumbra wedge algorithm overcomes this limitation and also allows soft shadows to be cast on any shadow receiver that can be rendered into a z-buffer.

9 Discussion and Future Work

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

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

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

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

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

10 Conclusion

We have presented a restructured version of our original penumbra wedge algorithm, which significantly reduces the number of calculations to rasterize a wedge. We have also shown how to incorporate occlusion culling using both z_{min} and z_{max}, to get a substantial speedup of the rasterization and reach near real-time performance on a standard PC. Empirical results show that the optimal tile size is between 8×8 and 32×32 pixels for our three test scenes. Furthermore, the restructured algorithm allows the use of the z-fail algorithm to correctly handle the case when the eye is inside a shadowed region. We want to

emphasize that neither occlusion culling nor the z-fail algorithm can be incorporated in any natural way with the original penumbra wedge algorithm, presented in [1]. Thus, the restructuring presented in this paper significantly enhances the efficiency and usability of penumbra wedges for simulating soft shadows.

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

References

[1] T. Akenine-Möller, U. Assarsson (2002) Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges. 13th Eurographics Workshop on Rendering 2002, 309–318. 58, 61, 63, 64, 72, 74, 75

[2] C. Everitt, M. J. Kilgard (2002) Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering. Published on-line at http://developer.nvidia.com/. 58, 67, 73

[3] F. C. Crow (1977) Shadow Algorithms for Computer Graphics. SIGGRAPH '77 Proceedings, 242–248. 60, 67, 72

[4] R. Fernando, S. Fernandez, L. Bala, and D. P. Greenberg (2001) Adaptive Shadow Maps. SIGGRAPH 2001 Proceedings, 387–390. 73

[5] B. Gooch, P. J. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld (1999) Interactive Technical Illustration. Symposium on Interactive 3D Graphics, Proceedings 1999, 31–38. 73

[6] S. Gottschalk,M.C. Lin, and D. Manocha (1996) OBBTree: A Hierarchical Structure for Rapid Interference Detection. Computer Graphics (SIGGRAPH Proceedings '96), 171–180. 65

[7] N. Greene (1994) Detecting Intersection of a Rectangular Solid and a Convex Polyhedron. In: P. S. Heckbert (1994) Graphics Gems IV, pp. 74–82. 66

[8] E. A. Haines, and J. R. Wallace (1994) Shaft Culling for Efficient Ray-Traced Radiosity. Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering), Springer-

Verlag, New York, 122–138, also in SIGGRAPH ’91 Frontiers in Rendering course notes. 66

[9] E. Haines, and T. Möller (2001) Real-Time Shadows. Game Developers Conference. 72

[10] E. Haines (2001) Soft Planar Shadows Using Plateaus. journal of graphics tools, 6(1), 19–27. 73

[11] P. Heckbert, and M. Herf, (1997) Simulating Soft Shadows with Graphics Hardware. Technical Report CMU-CS-97-104, Carnegie Mellon University. 73

[12] T. Heidmann, (1991) Real shadows, real time. Iris Universe, Silicon Graphics Inc., No. 18, 23–31. 60

[13] W. Heidrich, S. Brabec, and H-P. Seidel (2000) Soft Shadow Maps for Linear Lights. 11th Eurographics Workshop on Rendering, 269–280. 73

[14] T. Lokovic, and E. Veach (2000) Deep Shadow Maps. SIGGRAPH 2000 Proceedings, 385-392. 73

[15] S. Morein (2000) ATI Radeon—HyperZ Technology. SIGGRAPH/Eurographics Graphics Hardware Workshop 2000, Hot3D session. 65

[16] S. Parker, P. Shirley, and B. Smits (1999) Single Sample Soft Shadows. TR UUCS-98-019, Computer Science Department, University of Utah. 58

[17] W. T. Reeves, D. H. Salesin, and R. L. Cook (1987) Rendering Antialiased Shadows with Depth Maps. SIGGRAPH ’87 Proceedings, 283–291. 72

[18] C. Soler, and F. X. Sillion (1998) Fast Calculation of Soft Shadow Textures Using Convolution. SIGGRAPH ’98 Proceedings, 321–332. 58, 73

[19] L. Williams, (1978) Casting Curved Shadows on Curved Surfaces. SIGGRAPH ’78 Proceedings, 270–274. 72

[20] A. Woo, P. Poulin, and A. Fournier (1990) A Survey of Shadow Algorithms. IEEE Computer Graphics and Applications, 10(6), 13–32. 72

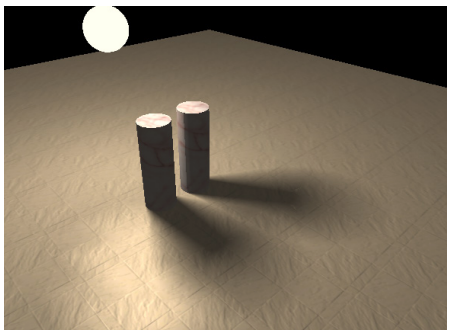


Figure 9: This image was rendered with the three pass algorithm in 7 fps on a Pentium4 1700 MHz using software and a GeForce3 graphics accelerator for the penumbra wedge rendering. The image size is 640×427 pixels.

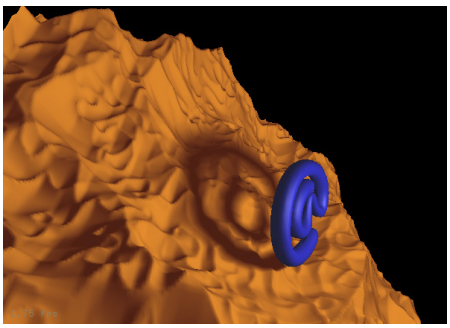


Figure 10: These images show a fractal landscape with 100k triangles used as a complex shadow receiver. The scene was rendered in 2.3 fps without soft shadows, and in 1.75 fps with soft shadows using the three pass algorithm.

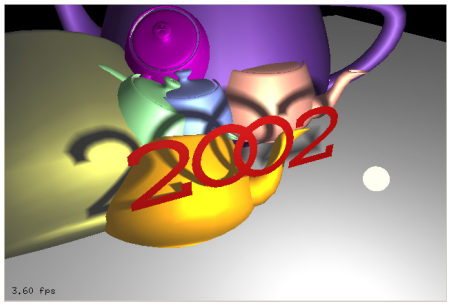


Figure 11: In this scene, the 2002 text casts its soft shadow onto a number of teapots and a floor. The image was rendered in 3.6 fps with the three pass algorithm.

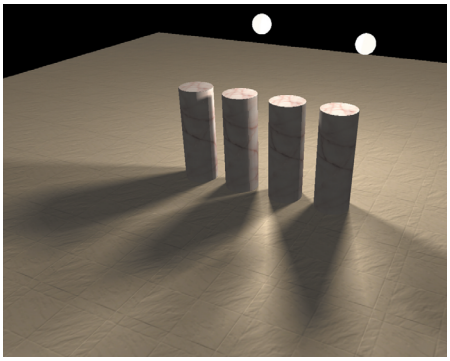


Figure 12: This is an example of using multiple light sources.

Author Biography

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

Tomas Akenine-Möller is an assistant professor at the Department of Computer Engineering at Chalmers University of Technology, Sweden. He has received an MSc in Computer Science and Engineering from Lund University of Technology, and a PhD in Computer Graphics at Chalmers University. He is the coauthor with Eric Haines of the book "Real-Time Rendering", and his

main research interests are rapid and realistic real-time rendering, interactive ray tracing, spatial data structures, and algorithms for future graphics hardware.

Paper III

A Geometry-Based Soft Shadow Volume Algorithm Using Graphics Hardware

Published in

Proceedings of ACM SIGGRAPH 2003,
Pages 511–520, 2003.

A Geometry-based Soft Shadow Volume Algorithm using Graphics Hardware

Ulf Assarsson and 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

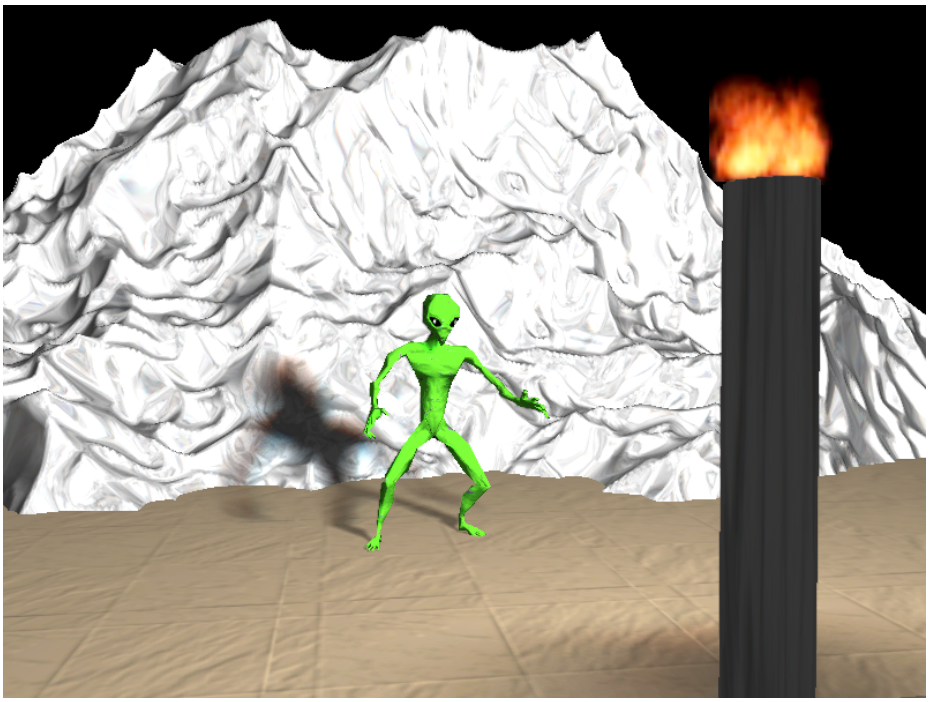


Figure 1: An image texture of fire is used as a light source, making the alien cast a soft shadow onto the fractal landscape. The soft shadow pass of this scene was rendered at 0.8 frames per second.

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.

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 [2]. 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 [28]. A more recent presentation covering real-time algorithms is also available [12].

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 [15]. 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 [13]. 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 [27] and shadow volumes [8] 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 [22]. A hardware implementation of SM has been presented [23], 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 [11] and perspective shadow maps have been proposed [26]. By using more than one shadow map, and interpolating visibility, soft shadows can be generated as well [17]. 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 [5], which adapts Parker et al's algorithm [21] 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 [1]. 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 [8], which can be implemented using the stencil buffer on commodity graphics hardware [16]. 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 [10].

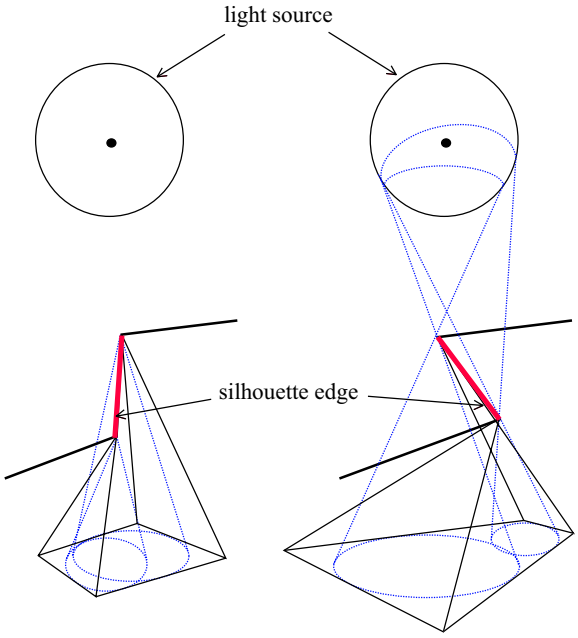


Figure 2: Difficult situations for the previous wedge generation algorithm. Left: The edge nearly points towards the light center, resulting in a non-convex wedge. Right: The edge is shadowed by one adjacent edge that is closer to the light source, making the left and right planes intersect inside the 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 [2]. 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 [6], 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 [9] 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) [25]. 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 [14], 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 [21]. Sloan et al. precompute radiance transfer and then renders several difficult light transport situations in low-frequency environments [24]. 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.

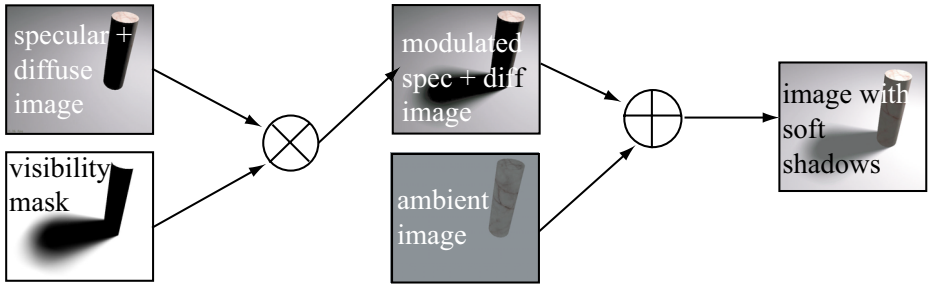


Figure 3: Overview of how the soft shadow algorithm works. Our work focuses on rapidly computing a visibility mask using a V-buffer, as seen from the eye point.

The V-buffer stores a *visibility factor*, $\overline{v}$, 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 $\overline{v} = 1$. This is in contrast to a point that is fully occluded, and thus has $\overline{v} = 0$. A point that can see x percent of a light source has $\overline{v} = x/100 \in [0,1]$. Thus, if a point has $0 < \overline{v} < 1$, then that point is in the penumbra region.

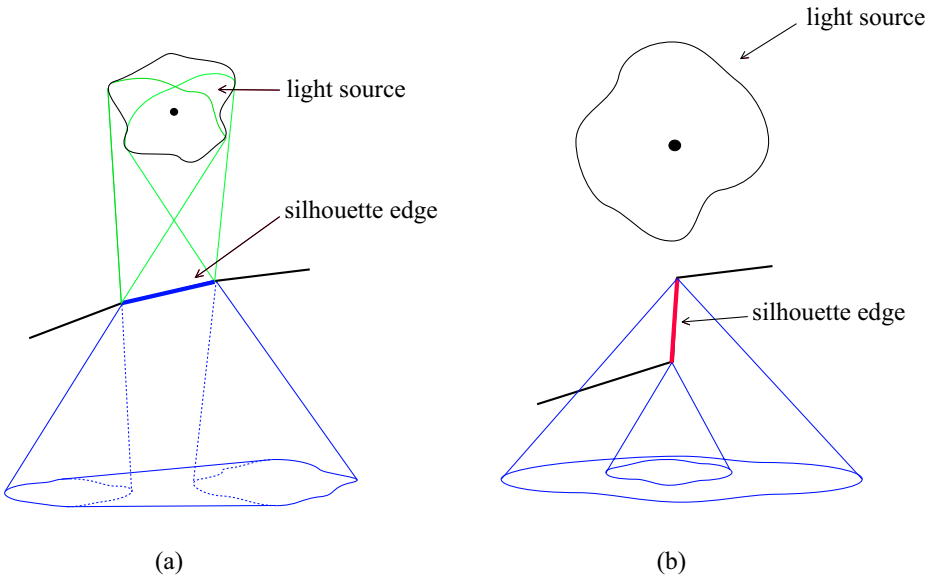


Figure 4: a) The penumbra volume generated by an edge. b) The penumbra volume can degenerate to a single cone, i.e., one end cone completely encloses the other end cone.

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 [2], 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. [19]. 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 envi-

ronments. 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 [2] 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 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.

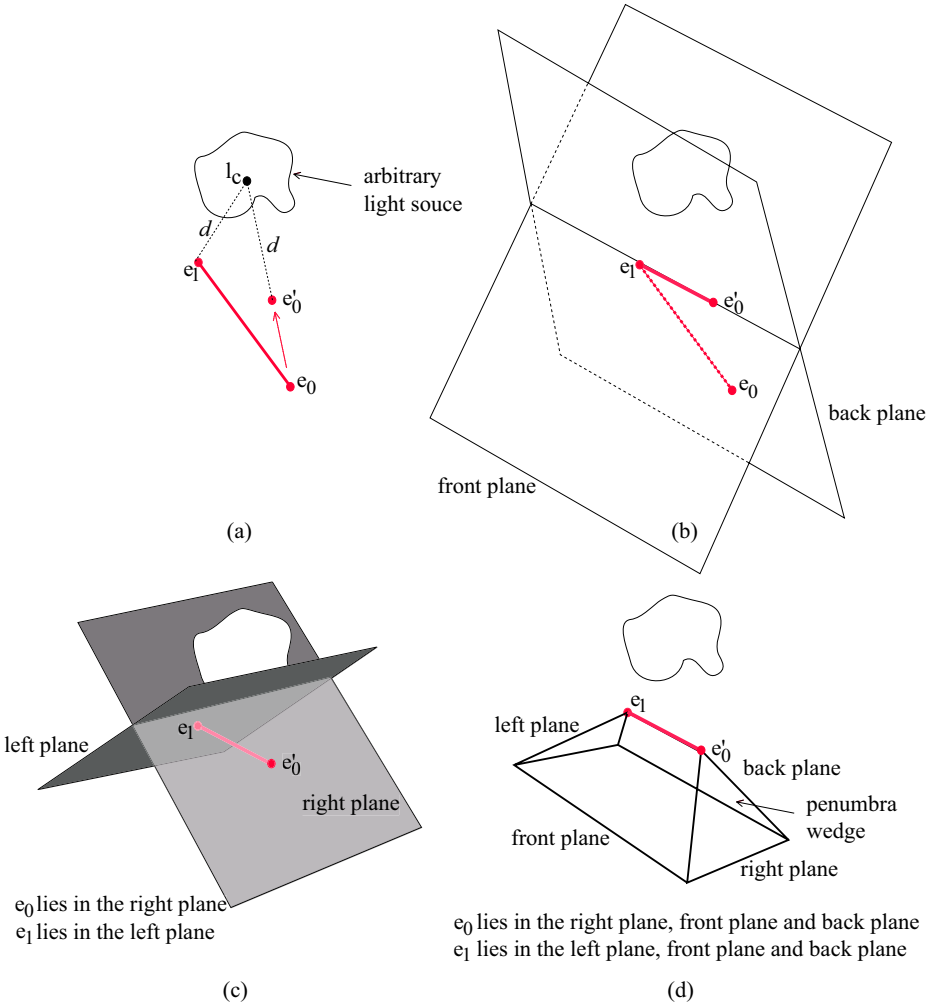


Figure 5: Wedge construction steps. a) Move the vertex furthest from the light center l_c towards l_c to the same distance as the other vertex. b) Create the front and back planes. c) Create the left and right planes. d) The final wedge is the volume inside the front, back, left, and right planes.

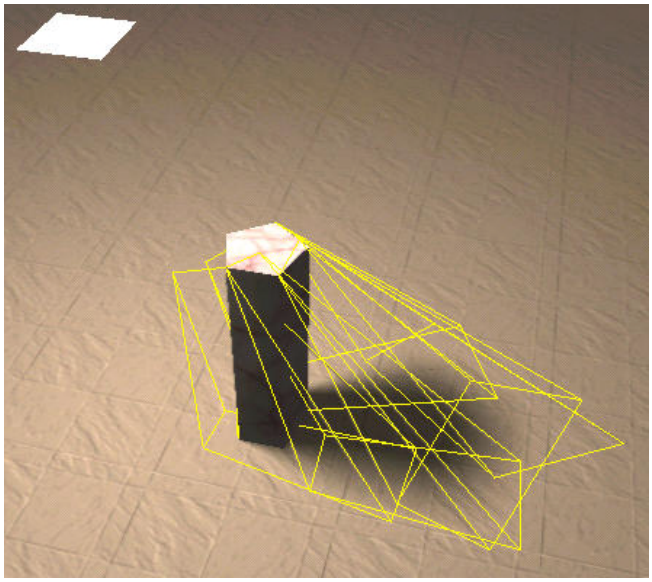


Figure 6: The wedges for a simple scene. At 512×512 resolution, this image was rendered at 5 frames per second using our software implementation. Note that there are wedges whose adjusted top edges differ a lot from the original silhouette edge. This can especially be seen to the left of the cylinder.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

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 [8], 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

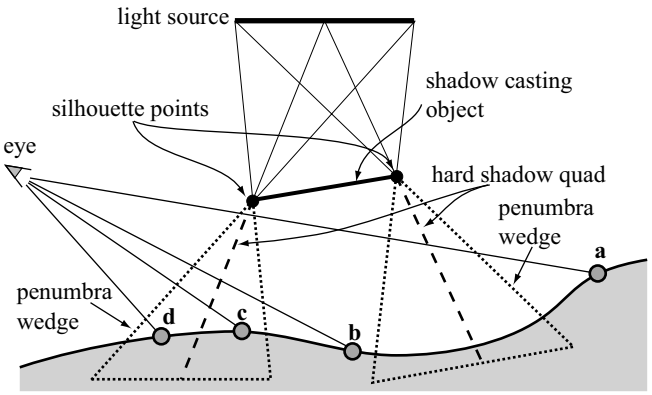


Figure 7: A two-dimensional example of how the two passes in computing the visibility cooperate. The two penumbra wedges, generated by the two silhouette points of the shadow casting object, are outlined with dots.

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 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 [2] used an overly simplified model of the penumbra/umbra surface, 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 [9].

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.

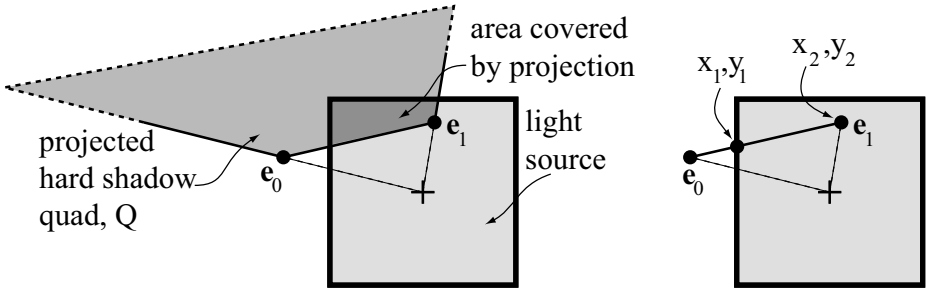


Figure 8: Left: Computation of coverage (dark gray area divided by total area) of a point **p** with respect to the edge $\mathbf{e}_0\mathbf{e}_1$. A three-dimensional view is shown, where **p** looks at the light source center. It can also be thought of as the projection of the hard shadow quad, Q, onto the light source as seen from **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 :   p = point(x,y,z);   // z is depth buffer value
4 :   if p is inside the wedge
5 :     v_p = projectQuadAndComputeCoverage(W,p,Q);
6 :     if p is in positive half space of Q
7 :       v̄(x,y) = v̄(x,y) − v_p;   // update V–buffer
8 :     else
9 :       v̄(x,y) = v̄(x,y) + v_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 **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 **p** looking towards the light source. The gray area is an occluder, i.e., a shadow casting

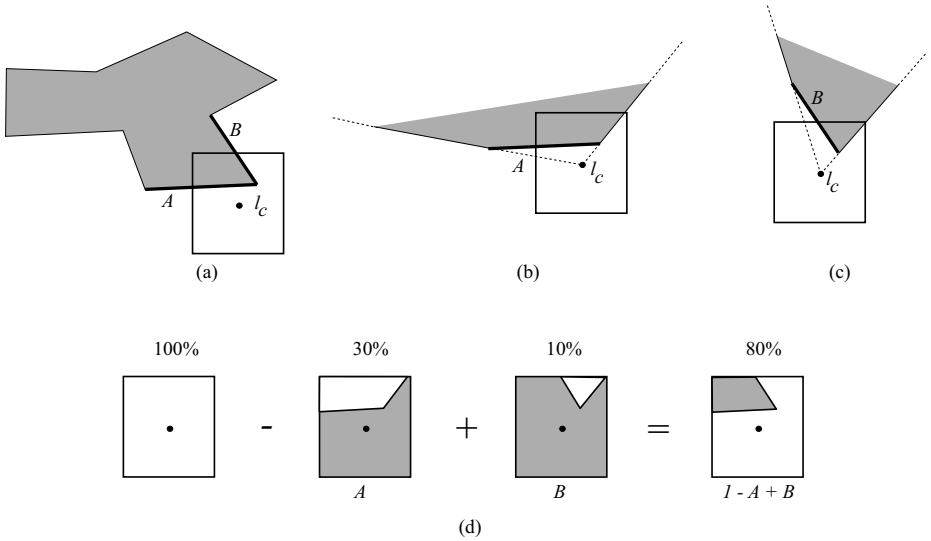


Figure 9: Splitting of shadow contribution for each wedge for a point **p**. *A* and *B* are two silhouette edges of a shadow casting object.

object, as seen from **p**. Both edge *A* and *B* contribute to the visibility of **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 **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

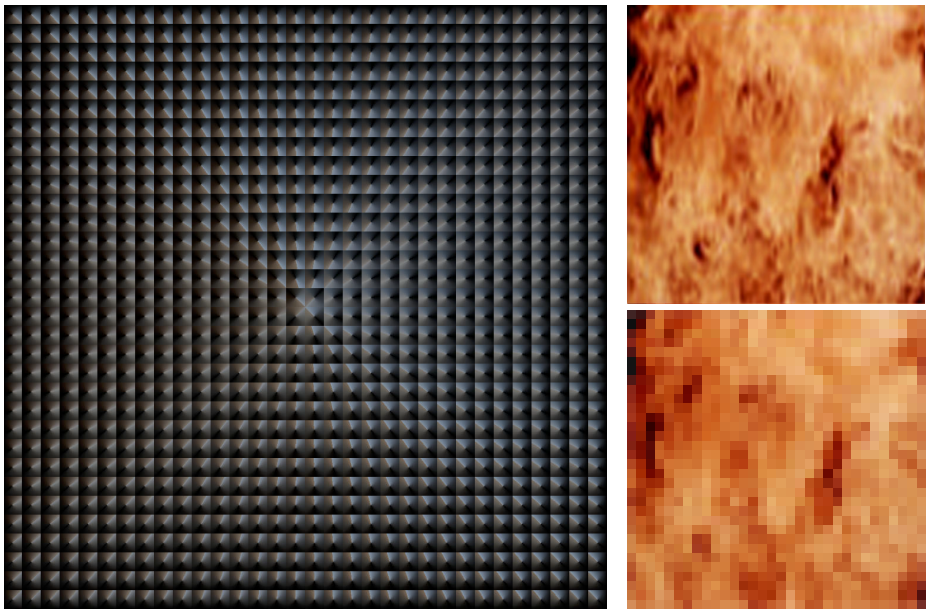


Figure 10: Left: example of precomputed coverage texture with $n = 32$. Top right: the original fire image. Bottom right: an undersampled 32×32 texel version of the original fire texture, used as light texture when computing the coverage texture. Each of the small 32×32 squares in the coverage texture 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).

texel centers, which 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.

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.

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

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 [2] 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.

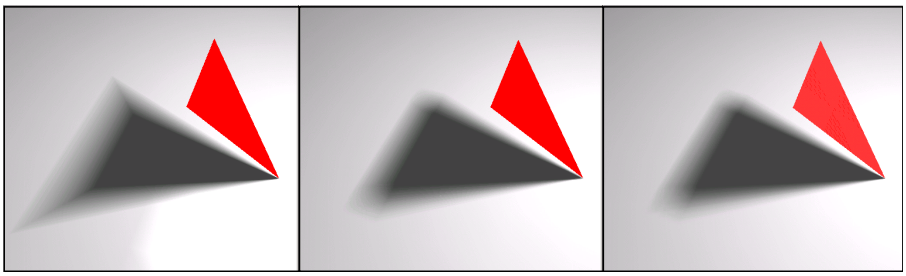


Figure 11: Comparison of the previous penumbra wedge algorithm, our algorithm, and using 1024 point light samples.

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 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

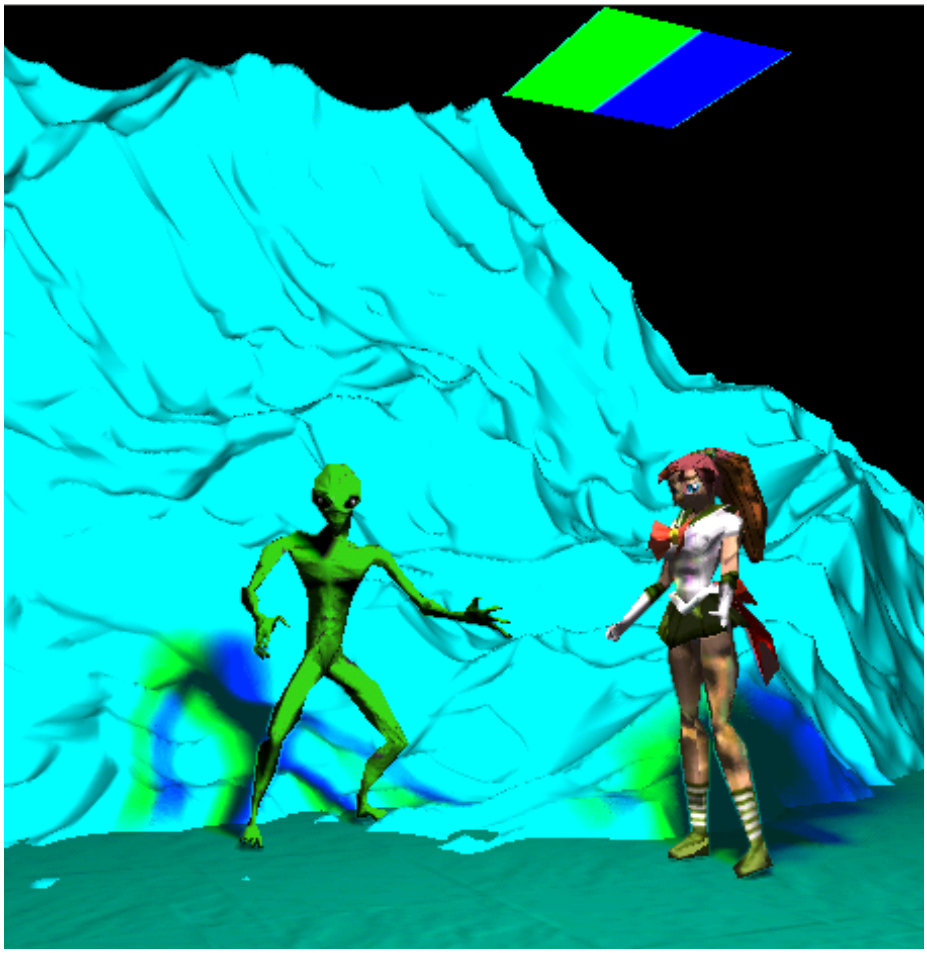


Figure 12: Example of a simple textured light source with two colors, demonstrating that the expected result is obtained.

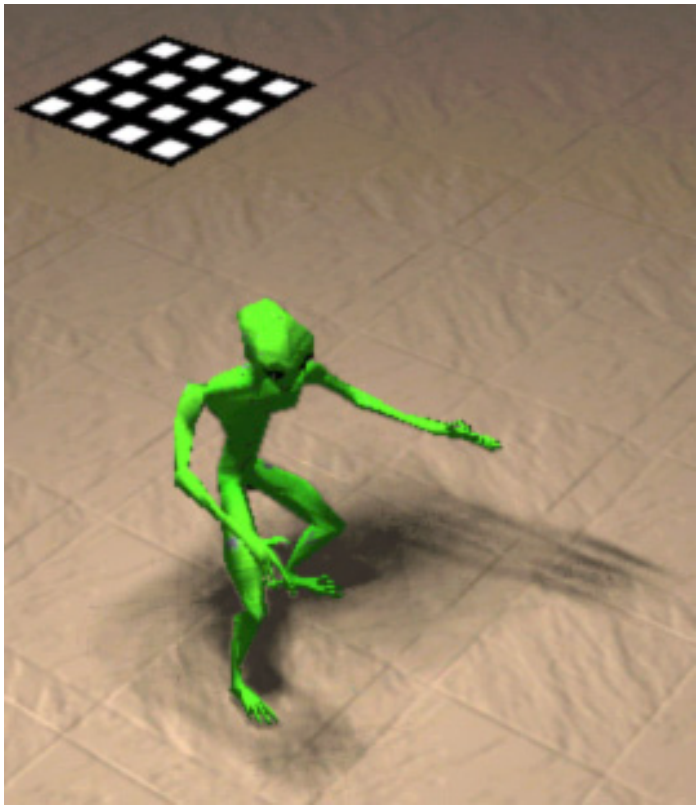


Figure 13: Here, a single rectangular light replaces 16 small rectangular light sources. Sampling artifacts can be seen in the shadow. This can be solved by increasing the resolution of the coverage texture.

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.

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 hard-

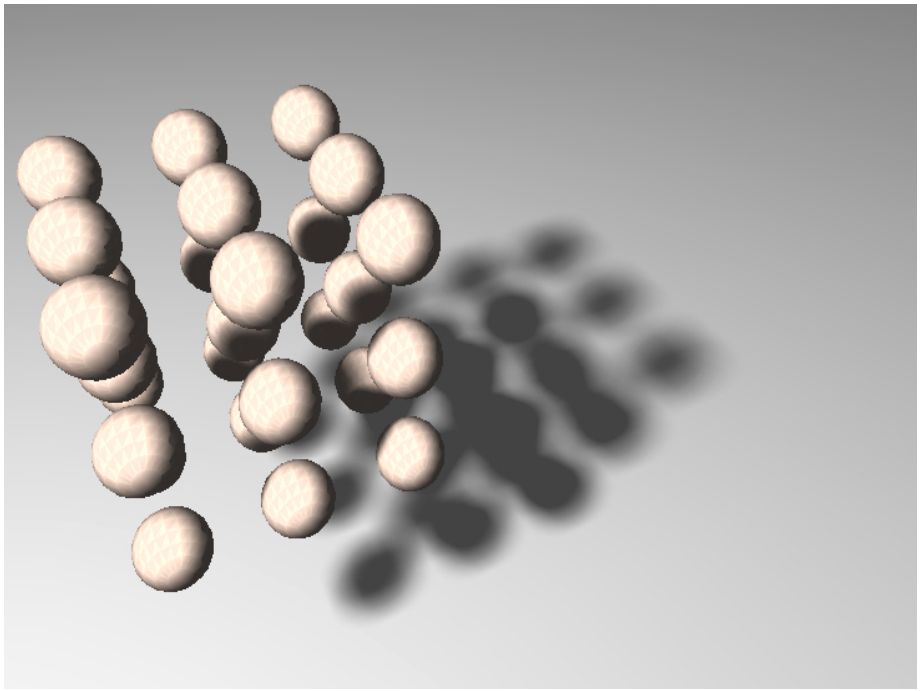


Figure 15: A grid of $3 \times 3 \times 3$ spheres is used as a shadow casting object.

ware.[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

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

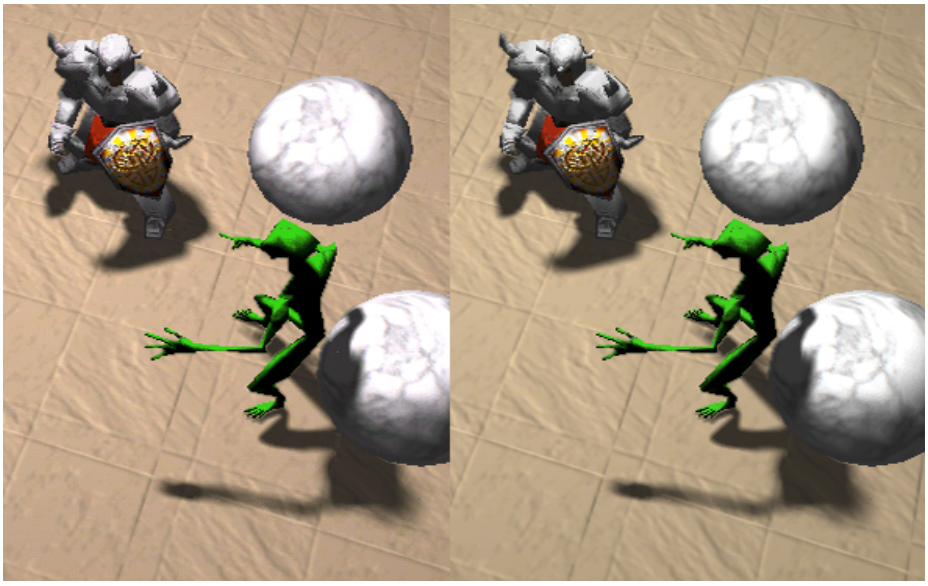


Figure 16: Left: image rendered using our software implementation. Right: rendered using GeForce FX hardware.

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 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 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

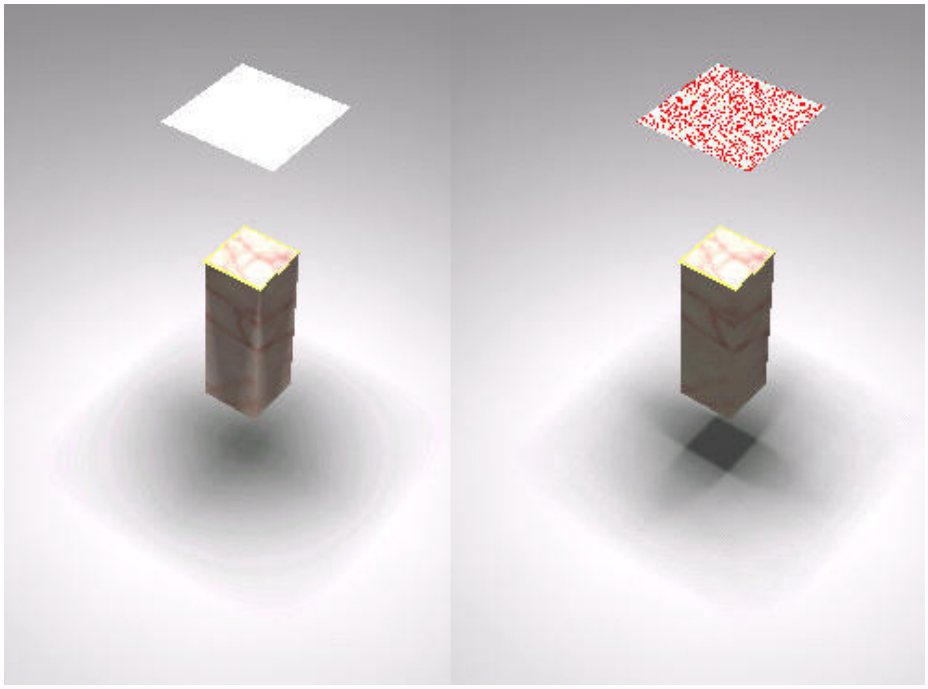


Figure 17: Single silhouette error: the left image shows results from our algorithm, and the right from rendering using 1024 point samples on the area light source. Notice differences in the shadow under the box, and on the sides on the box. The noise on the right light source are the 1024 sample locations.

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.

In general, for any shadow volume algorithm the following restrictions regarding geometry apply: the shadow casting objects must be polygonal and closed (two-manifold) [4]. The z-fail algorithm [10] can easily be incorporated into our algorithm to make the algorithm independent of whether the eye is in shadow or not [3]. 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 [10] 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

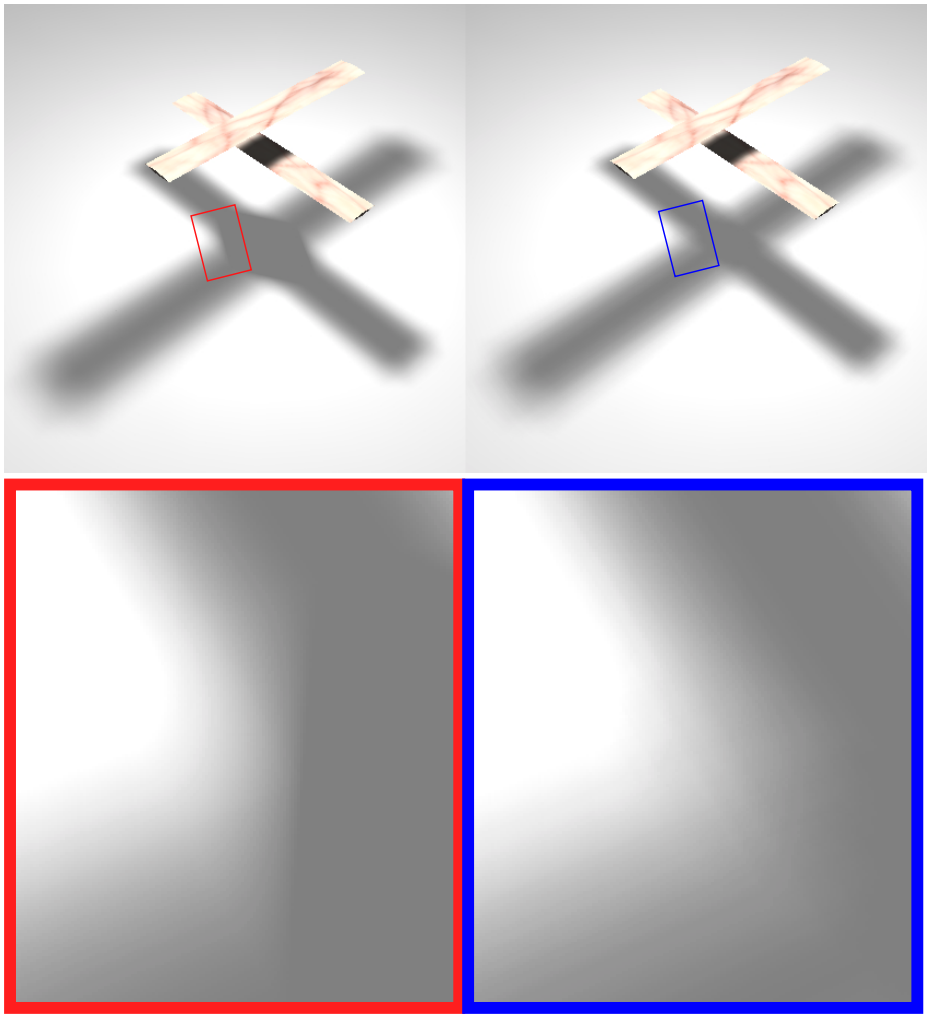


Figure 18: Object overlap error: the left images show results from our algorithm, and the right from rendering using 1024-sample shadows. The right images are correct, with their curved boundaries to the umbra. The left images contain straight boundaries to the umbra.

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 [18] 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.

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

References

[1] Agrawala, M., R. Ramamoorthi, A. Heirich, and L. Moll, "Efficient Image-Based Methods for Rendering Soft Shadows," *Proceedings of ACM SIGGRAPH 2000,* ACM Press/ACM/Siggraph, New York. K. Akeley, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 375–384. 86

[2] Akenine-Möller, Tomas, and Ulf Assarsson, "Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges," *13th Eurographics Workshop on Rendering 2002*, pp. 309–318, June 2002. 85, 87, 89, 90, 94, 100

[3] Assarsson, Ulf, and Tomas Akenine-Möller, "Interactive Rendering of Soft Shadows using an Optimized and Generalized Penumbra Wedge Algorithm," submitted to the *Visual Computer*, 2002. 106

[4] Bergeron, P., "A General Version of Crow's Shadow Volumes," *IEEE Computer Graphics and Applications*, **6**(9):17–28, September 1986. 106

[5] Brabec, Stefan, and Hans-Peter Seidel, "Single Sample Soft Shadows using Depth Maps," *Graphics Interface 2002*, pp. 219–228, 2002. 86

[6] Brotman, Lynne Shapiro, and Norman I. Badler, "Generating Soft Shadows with a Depth Buffer Algorithm," *IEEE Computer Graphics and Applications 4*, 10, pp. 5–12, October, 1984. 88

[7] Cohen, M. F., and J. R. Wallace, *Radiosity and Realistic image Synthesis*, Academic Press Professional, 1993.

[8] Crow, Frank, "Shadow Algorithms for Computer Graphics," *Computer Graphics (Proceedings of ACM SIGGRAPH 77)*, pp. 242–248, July 1977. 86, 93

[9] Drettakis, George, and Eugene Fiume, "A Fast Shadow Algorithm for Area Light Sources Using Backprojection," *Computer Graphics (SIGGRAPH 1994)*, Annual Conference Series, pp 223–230, ACM SIGGRAPH, 1994. 88, 94

[10] Everitt, Cass, and Mark Kilgard, "Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering," `http://developer.nvidia.com/`, 2002. 86, 106

[11] Fernando, R., S. Fernandez, K. Bala, and D. P. Greenberg, "Adaptive Shadow Maps," *Proceedings of ACM SIGGRAPH 2001*, pp. 387–390, August 2001. 86

[12] Haines, Eric, and Tomas Möller, "Real-Time Shadows," *Game Developers Conference*, pp. 335–352, March, 2001. 85

[13] Haines, Eric, "Soft Planar Shadows Using Plateaus," *Journal of Graphics Tools*, **6**(1):19–27, 2001. 86

[14] Hart, David, and Philip Dutré, and Donald P. Greenberg, "Direct Illumination with Lazy Visbility Evaluation," *Proceedings of ACM SIGGRAPH 99*, pp. 147–154, August, 1999. 88

[15] Heckbert, Paul, and Michael Herf, *Simulating Soft Shadows with Graphics Hardware*, Carnegie Mellon University, Technical Report CMU-CS-97-104, January, 1997. 85

[16] Heidmann, Tim, "Real shadows, real time," *Iris Universe*, no. 18, pp. 23–31, November 1991. 86

[17] Heidrich, W., S. Brabec, and H-P. Seidel, "Soft Shadow Maps for Linear Lights," *11th Eurographics Workshop on Rendering*, pp. 269–280, 2000. 86

[18] Kautz, J., and M. D. McCool, "Interactive Rendering with Arbitrary BRDFs using Separable Approximations," *10th Eurographics Workshop on Rendering*, pp. 281–292, 1999. 108

[19] Markosian, Lee, and Michael A. Kowalski, and Samuel J. Trychin, and Lubomir D. Bourdev, and Daniel Goldstein, and John F. Hughes, "Real-Time Nonphotorealistic Rendering," *Proceedings of ACM SIGGRAPH 97*, ACM Press/ACM SIGGRAPH, New York. T. Whitted, Ed., Computer

Graphics Proceedings, Annual Conference Series, ACM, pp. 415–420, August, 1997. 89

[20] Nishita, T. and E. Nakamae, "Half-Tone Representation of 3-D Objects Illuminated by Area or Polyhedron Sources," *Proc. of IEEE Computer SocietyŠs Seventh International Computer Software and Applications Conference (COMPSAC83)*, pp. 237-242, Nov 7-11, 1983.

[21] Parker, S., P. Shirley, and B. Smits, *Single Sample Soft Shadows*, University of Utah, Technical Report UUCS-98-019, October 1998. 86, 88

[22] Reeves, William T., and David H. Salesin, and Robert L. Cook, "Rendering Antialiased Shadows with Depth Maps," *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, pp 283–291, July, 1987. 86

[23] Segal, M., and C. Korobkin, and R. van Widenfelt, and J. Foran, and P. Haeberli "Fast Shadows and Lighting Effects Using Texture Mapping," *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, ACM, pp. 249–252, July, 1992. 86

[24] Sloan, Peter-Pike, Jan Kautz, and John Snyder, "Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments," *ACM Transactions on Graphics*, **(**21)(3):527–536, July 2002. 88

[25] Soler, Cyril, and F. X. Sillion, "Fast Calculation of Soft Shadow Textures Using Convolution," *Computer Graphics (SIGGRAPH 1998)*, Anual Conference Series, pp 321–332, ACM SIGGRAPH, 1998. 88

[26] Stamminger, Marc, and George Drettakis, "Perspective Shadow Maps," *ACM Transactions on Graphics*, **21**(3):557–562, July 2002. 86

[27] Williams, Lance, "Casting Curved Shadows on Curved Surfaces," *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, pp. 270–274, August 1978. 86

[28] Woo, A., P. Poulin, and A. Fournier, "A Survey of Shadow Algorithms," *IEEE Computer Graphics and Applications*, **10**(6):13–32, November 1990. 85

Paper IV

An Optimized Soft Shadow Volume Algorithm with Real-Time Performance

Published in

Graphics Hardware 2003,
ACM SIGGRAPH/Eurographics Workshop Proceedings,
Pages 33–40, 2003.

An Optimized Soft Shadow Volume Algorithm with Real-Time Performance

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

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

Abstract

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

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

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

shadows. Therefore, we conclude that dynamic soft shadows are very important for real-time computer graphics.

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

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

2 Previous Work

Shadow generation has become a well-documented topic within computer graphics, and the amount of literature is vast. Therefore, we will only cover the papers that are most relevant to our work. For an overview, consult Woo's et al's survey [20], and for real-time algorithms, consult Akenine-Möller and Haines [1].

The two most widely used real-time shadow algorithms are shadow mapping and shadow volumes. The shadow mapping algorithm by Williams [19], renders a depth image, called the shadow map, as seen from the light source. To create shadows, the scene is rendered from the eye, and for each pixel, its corresponding depth with respect to the light source is compared to the shadow map depth value. This determines whether the point is in shadow. To alleviate resolution problems in the shadow maps, Fernanado et al. [11] presented an algorithm that increased the shadow map resolution where it was needed the most, and Stamminger and Drettakis presented perspective shadow maps for the same reason [18]. Heidrich et al. [15] presented a soft version of the shadow map algorithm. It could handle linear light sources by interpolating visibility using more than one shadow map. Recently, Brabec and Seidel presented a more general soft shadow map algorithm [7]. Their work was inspired by Parker et al's [16] soft shadow generation technique for ray tracing. In that work, "soft-edged" objects were ray traced at only one sample per pixel using a parallel ray tracer. Thus, Brabec and Seidel presented a hardware-accelerated version of

Parker et al's algorithm.

The shadow volume algorithm by Crow [8] is often implemented using a stencil buffer on commodity graphics hardware [14]. The algorithm first renders the scene using ambient lighting. In a second pass, each silhouette edge as seen from the light source creates a shadow volume quadrilateral which is rendered from the eye. Note that all that is required for these silhouette edges is that two polygons share that edge, and one of the polygons is frontfacing, and the other is backfacing as seen from the light source. The generated quads are rendered as seen from the eye. Frontfacing quads that pass the depth test add one to the stencil buffer, and backfacing quads subtract one. Therefore, at the end of this pass, the stencil buffer contains a mask where a zero indicate no shadow, and anything else indicates that the pixel is in shadow. The third pass is rendered with full lighting where the stencil buffer is zero. Everitt and Kilgard have presented techniques to make the shadow volume robust, especially for cases when the eye is inside shadow [10].

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

There are also several algorithms that only can handle planar soft shadows. For example, Haines presents shadow plateaus, where a hard shadow, which is used to model the umbra, is drawn from the center of the light source [12]. The penumbra is rendered by letting each silhouette vertex, as seen from the light source, generate a cone, which is drawn into the Z-buffer. The light intensity in a cone varies from 1.0, in the center, to 0.0, at the border. A Coons patch is drawn between two neighboring cones, and similar light intensities are used. Heckbert and Herf use the average of 64–256 hard shadows into an accumulation buffer [13]. These images can then be used as textures on the planar surfaces. Radiance transfer can be precomputed, as proposed by Sloan et al. [17], and then used to render several difficult light transport situations in low-frequency environments. Real-time soft shadows are included there.

3 Soft Shadow Volume Algorithm

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

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

4 Optimizations

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

4.1 Tighter Wedges for Rectangular Light Sources

Our previously presented method created wedges from bounding spheres surrounding the light source [5]. The penumbra volume corresponding to an edge and a spherical light source is the swept volume of the circular cone created by reflecting the light source through the sweeping point that moves from one edge end point to the other (see Figure 1a). To handle robustness issues and avoid the hyperbolic [12] front and back wedge surfaces along the edge, the edge vertex

furthest from the light center is temporarily moved straight towards the light center to a new location at the same distance from the light center as the other edge vertex. Then, this new edge is used for sweeping the wedge volume. This will make the front and back wedge surfaces planar, and the resulting wedge will completely enclose the true penumbra volume (Figure 1c). This process also simplifies the computation of the wedge polygons.

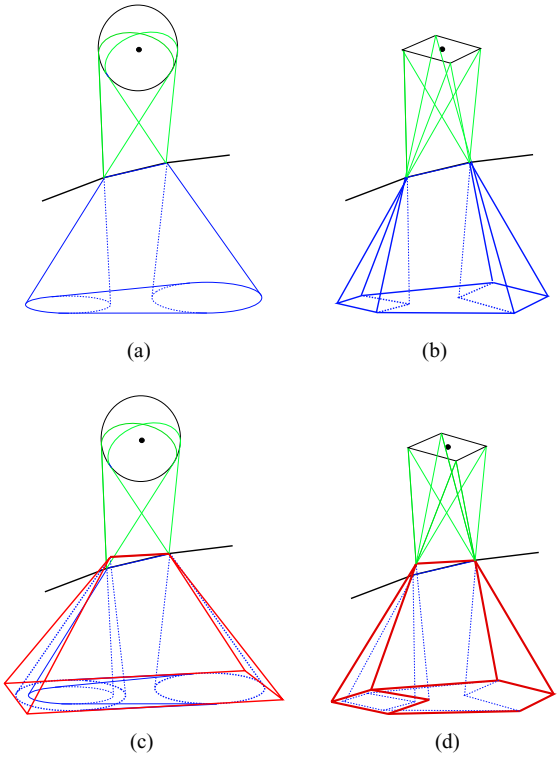


Figure 1: a) and b) show the penumbra volume for an edge for a spherical and a rectangular light source respectively. c) and d) show the corresponding wedges, which enclose the penumbra volumes. In c) and d) the inner left cone is the reflected cone through the left silhouette vertex. The outer left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

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

For a spherical light source, a left and a right plane was used to close the wedge on the sides. Now instead, we use the two leftmost and two rightmost triangles respectively of the two end pyramids that are formed by the reflection of the light source through the edge end points. This will result in a more tight fitting wedge at the sides. Along the edge, this wedge will typically be thinner

than one created from a spherical light source (see Figure 1d). A bottom plane may be added for the z-fail algorithm and culling (see section 4.4).

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

4.2 Optimized Pixel Shaders

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

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

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

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

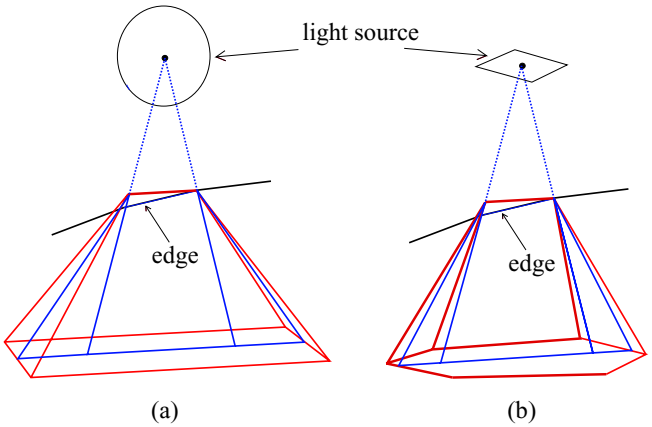


Figure 2: The wedge is split by the hard shadow quad in an outer and inner half. Since the hard shadow quad does not extend all the way out to the left and right wedge sides, two triangles are used on either side to achieve the split. Together, these three polygons constitute the wedge center polygons.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

4.2.1 Spherical Light Source Shader

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

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

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

final coverage (shaded in gray on Figure 5a).

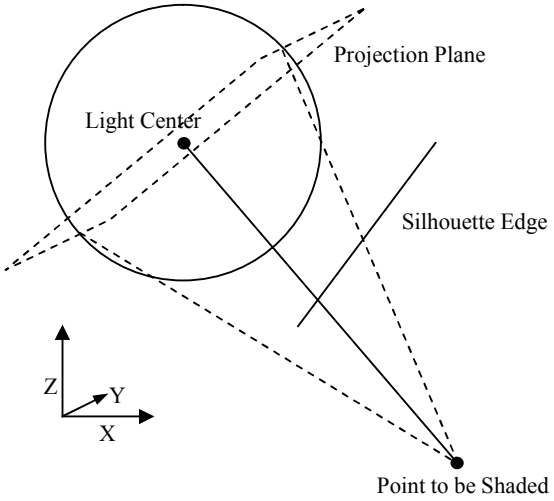


Figure 3: The spherical light source clipping cone and projection plane shown in world space.

4.2.2 Rectangular Light Source Shader

To compute coverage it is necessary to project the edge onto the light source and clip it to the border of the light source. For robustness it is better to clip the edge first and then project it; otherwise points behind the origin of the projection will be inverted. Both clipping and projection can be done efficiently using homogenous coordinates. The endpoints of the edge are initially transformed so that the point to be shaded is at the origin and the normal to the light source plane is parallel to the *z*-axis. The matrix for the projection is computed with the point to be shaded as the origin of the projection and the near plane as the rectangular light source. The endpoints of the edge are then transformed into clip space using the projection transform, followed by homogenous clipping to each side of the rectangular light source. After clipping, the endpoints are projected by dividing by the homogenous *w*-coordinate. The endpoints can then be used to look up the area in a 4D coverage texture or be used for analytic computation of the area.

For non-textured rectangular light sources, the coverage of the projected silhouette edge quadrilateral can be computed analytically instead of being computed using a 4D coverage texture. The advantage of computing the coverage analytically is that higher accuracy is possible using less texture space. The area covered by the projected edge quadrilateral is equal to the light source area between the two vectors of infinite length from the center of the light source

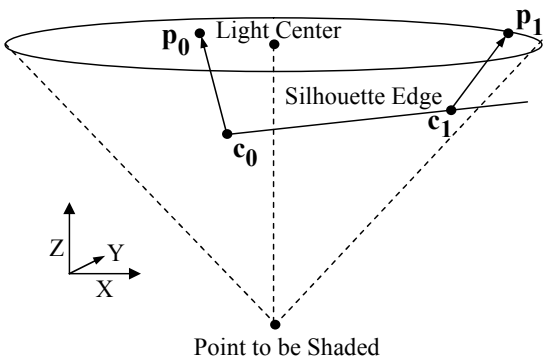


Figure 4: The clipping of the silhouette edge with the light cone in light space. The clipped edge end points $\mathbf{c}_0$ and $\mathbf{c}_1$ are then projected onto the plane $z = 1$, which gives $\mathbf{p}_0$ and $\mathbf{p}_1$.

through the projected clipped endpoints of the edge minus the area of the triangle defined by the center of the light source and the projected clipped endpoints of the edge (Figure 6a). The area of the triangle defined by the center of the light source and the projected clipped endpoints is computed using a 2D cross product. The light source area between the two vectors of infinite length is looked up in a 2D texture (Figure 6b) based on the angles of elevation for the two vectors (θ_0 and θ_1). A cube-map that implements **atan2(**y,x**)** is used to look up the angles to the two vectors.

4.3 Frame Buffer Blending

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

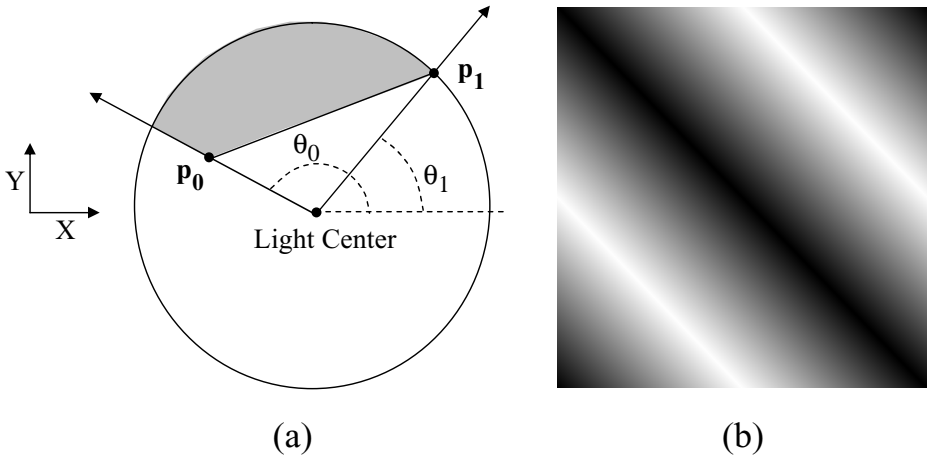


Figure 5: a) The coverage is computed from the projected 2D points $\mathbf{p}_0$ and $\mathbf{p}_1$ in projected space. b) Area lookup texture.

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

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

4.4 Culling

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

Therefore, ideally, the pixel shader should only be executed for points in-

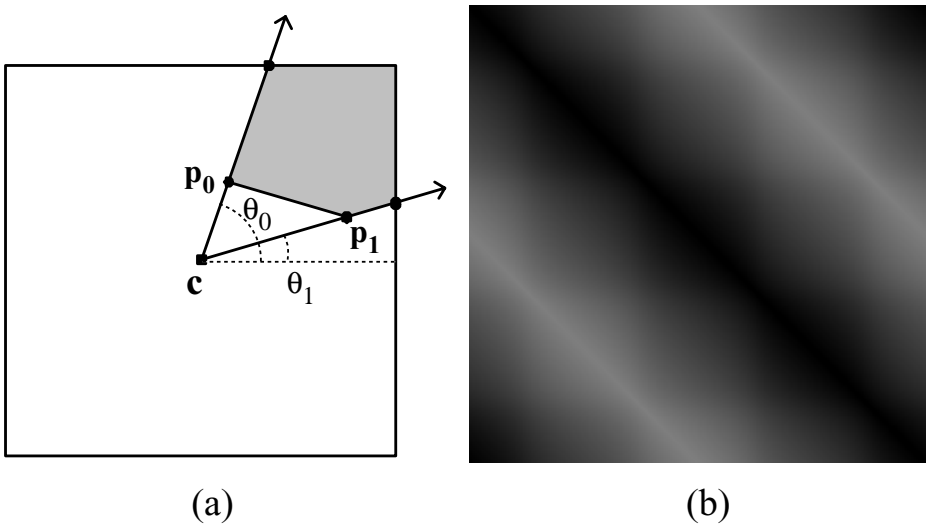


Figure 6: a) The area covered by the silhouette edge ($\mathbf{p}_1\mathbf{p}_0$) projected onto the light source is equal to the area of the light source covered by the triangle defined by the light center $\mathbf{c}$ and the two vectors ($\mathbf{c}, \mathbf{p}_0$) and ($\mathbf{c}, \mathbf{p}_1$)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

side the wedge, and culling should reject all other pixels whose corresponding points are outside the wedges. This culling can be done using two passes and combining the depth-test and the stencil test.

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

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

is located outside the wedge half, and the fragment is culled from rendering. Early rejection in the hardware can then avoid executing the shader for culled fragments.

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

5 Improving the Visual Results

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

5.1 Overlap Approximation

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

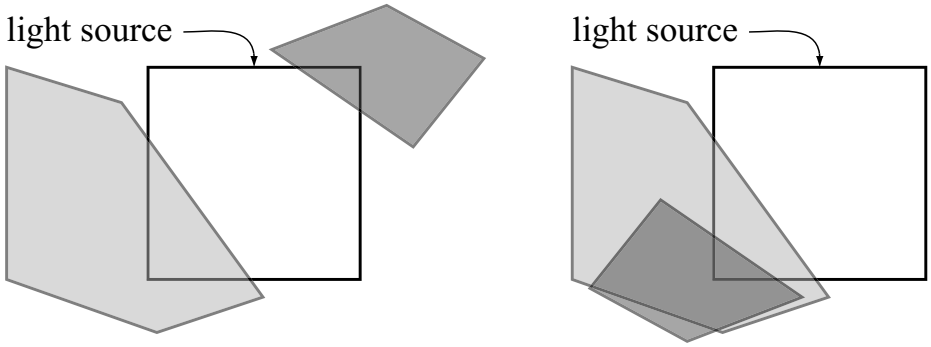


Figure 7: Overlap problems due incorrect combination of coverage. The light gray shadow caster covers 16 percent of the light source, while the darker gray shadow caster cover 4 percent. To the left, the shadow casters together covers 20 percent, while to the right they cover 16 percent.

The left part shows two pieces of gray-shaded geometry projected onto a square light source. The geometry does not overlap on the light source. Our

algorithm handles this case correctly. However, it always assumes that the geometry is non-overlapping, so for the situation in the right part of Figure 7, the same geometry with different positions should create another coverage. Unfortunately, our algorithm treats the situation to the right in the same way as to the left, i.e., the result is the same.

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

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

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

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

5.2 Single Silhouette Approximation

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

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

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

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

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

6 Results

The pixel shader for spherical light sources requires 59 arithmetical and 4 texture load instructions. The optimized pixel shader program for a textured light source, using the 4D coverage texture lookup, consists of 61 arithmetic instructions and 2 texture load instructions. The optimized shader for a non-textured light, using the analytic coverage texture, consists of 60 arithmetic instructions and 5 texture load instructions. Code for all the three shaders are available online at:

- *http://www.ce.chalmers.se/staff/uffe/NonTexturedRect.txt*
- *http://www.ce.chalmers.se/staff/uffe/TexturedRect.txt*
- *http://www.ce.chalmers.se/staff/uffe/Sphere.txt*.

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

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

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

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

7 Conclusions

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

Furthermore, a method is presented for creating tighter wedges, which typically improves the overall frame rate with 1.2 to 2 times. A culling technique is also described that can increase performance a bit. Finally, we show how to improve the visual quality by reducing the effects of the overlap and single silhouette approximations. With the improvements presented in this paper,

real-time soft shadows with very good quality can now be used in, for instance, games and virtual reality applications.

8 Future Work

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

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

Acknowledgements

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

References

[1] T. Akenine-Möller and E Haines, *Real-Time Rendering*, 2nd edition, June 2002. 114

[2] T. Akenine-Möller and U. Assarsson, "Rapid Soft Shadows on Arbitrary Surfaces using Penumbra Wedges," *Eurographics Workshop on Rendering 2002*, pp. 309–318, June 2002. 113, 114, 115

[3] T. Akenine-Möller and U. Assarsson, "On Shadow Volume Silhouettes," submitted to *Journal of Graphics Tools*, 2003. 125

[4] U. Assarsson and T. Akenine-Möller, "Interactive Rendering of Soft Shadows using an Optimized and Generalized Penumbra Wedge Algorithm," submitted to the *Visual Computer*, 2002. 113, 114, 115

[5] U. Assarsson and T. Akenine-Möller, "A Geometry-Based Soft Shadow Volume Algorithm using Graphics Hardware," to appear in *SIGGRAPH 2003*, July 2003. 113, 114, 115, 116, 124, 126, 127

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

[7] S. Brabec, and H-P. Seidel, “Single Sample Soft Shadows using Depth Maps,” *Graphics Interface 2002*, pp. 219–228, 2002. 114

[8] F. Crow, “Shadow Algorithms for Computer Graphics,” *Computer Graphics (Proceedings of ACM SIGGRAPH 77)*, pp. 242–248, July 1977. 115

[9] D. Eberly, “Intersection of a Line and a Cone,” *http://www.magic-software.com*, Magic Software Inc., October 2000.

[10] C. Everitt and M. Kilgard, “Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering,” *http://developer.nvidia.com/*, 2002. 115

[11] R. Fernando, S. Fernandez, K. Bala, and D. P. Greenberg, “Adaptive Shadow Maps,” *Proceedings of ACM SIGGRAPH 2001*, pp. 387–390, August 2001. 114

[12] E. Haines, “Soft Planar Shadows Using Plateaus,” *Journal of Graphics Tools*, **6**(1):19–27, 2001. 115, 116

[13] P. Heckbert and M. Herf, *Simulating Soft Shadows with Graphics Hardware*, Carnegie Mellon University, Technical Report CMU-CS-97-104, January, 1997. 115

[14] T. Heidmann, “Real shadows, real time,” *Iris Universe*, no. 18, pp. 23–31, November 1991. 115

[15] W. Heidrich, S. Brabec, and H-P. Seidel, “Soft Shadow Maps for Linear Lights,” *11th Eurographics Workshop on Rendering*, pp. 269–280, 2000. 114

[16] S. Parker, P. Shirley, and B. Smits, *Single Sample Soft Shadows*, University of Utah, Technical Report UUCS-98-019, October 1998. 114

[17] P-P. Sloan, J. Kautz, and J. Snyder, “Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments,” *ACM Transactions on Graphics*, **(**21)(3):527–536, July 2002. 115

[18] M. Stamminger, and G. Drettakis, “Perspective Shadow Maps,” *ACM Transactions on Graphics*, **21**(3):557–562, July 2002. 114

[19] L. Williams, “Casting Curved Shadows on Curved Surfaces,” *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, pp. 270–274, August 1978. 114

[20] A. Woo, P. Poulin, and A. Fournier, “A Survey of Shadow Algorithms,” *IEEE Computer Graphics and Applications*, **10**(6):13–32, November 1990. 114

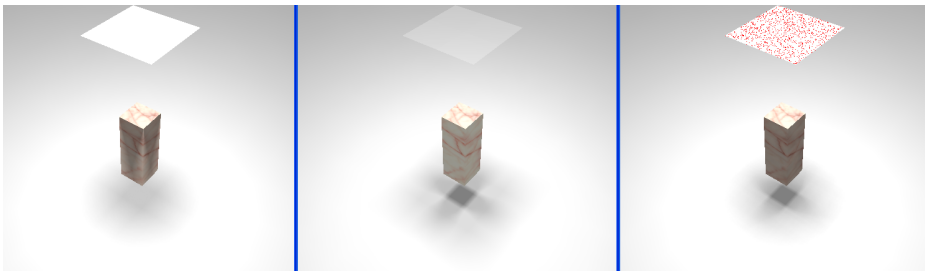


Figure 8: One area light source, 2×2 area light sources, 1024 point light sources.

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

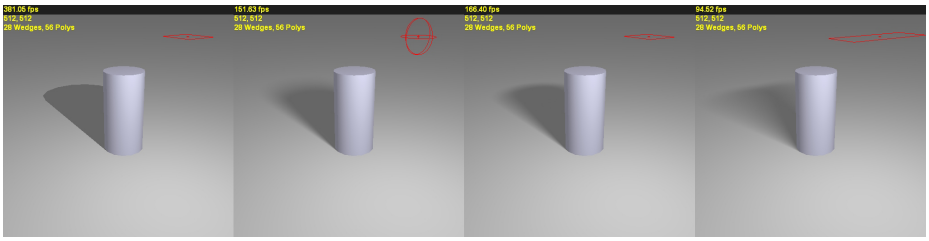


Figure 10: Comparison of appearance and frame rate for a cylinder with hard shadow, soft shadow from a spherical light source, soft shadow from a square light source, and soft shadow from a wide rectangular light source .

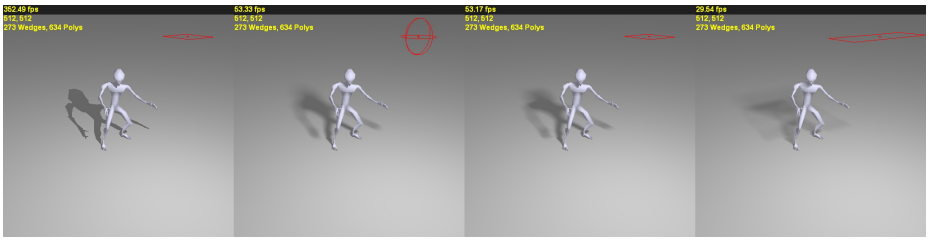


Figure 11: Same situations as for Figure 10, but for a more complex shadow caster.

Paper V

On Shadow Volume Silhouettes

Submitted to

Journal of Graphics Tools, April 4, 2003.

On Shadow Volume Silhouettes

Tomas Akenine-Möller and Ulf Assarsson
Chalmers University of Technology

Abstract

In shadow volume rendering, the shadow volume silhouette *edges are used to create primitives that model the shadow volume. A common misconception is that the vertices on such silhouettes can only be connected to two silhouette edges, i.e., have degree two. Furthermore, some believe that the degree of such a vertex can have any degree. In this short note, we present a geometric proof that shows that the degree of a silhouette vertex must be even, and not necessarily two.*

1 Introduction

The shadow volume (SV) algorithm [4] has become a very popular algorithm for real-time rendering [5] of hard shadows. Recently, the SV algorithm has been extended to handle soft shadow as well [1, 2]. In all these algorithms the shadow volume silhouette (SVS) of the shadow casting objects are found. An edge of such an SVS is connected to two polygons, where one is frontfacing and the other is backfacing as seen from the light source. The *degree* of an SVS vertex is the number of SVS edges connected to it. Note that silhouettes with edges as defined here may not necessarily be true silhouettes, and should therefore rather be referred to as possible silhouettes. For example, an SVS edge may very well be in shadow of another geometric object. In order for SV algorithms to work properly, the shadow casting objects must be polygonal and closed (two-manifold) [3].

During our research on soft shadows, we realized (to our surprise) that a vertex of an SVS can be connected to more than two SVS edges. This might be obvious to some, but we have realized that many others also believed (or believe) that an SVS vertex must have degree two. This is not so, as we will show.

2 The Degree of SVS Vertices

Theorem The degree of a shadow volume silhouette vertex is always even.

In the following, when we say vertex or edge, we refer to a SVS vertex or edge.

Proof: A geometric proof by contradiction follows here. Assume that we have a vertex with degree two, which is the smallest degree of a vertex. The definition of an edge says that the geometry connected to the edge on one side must be frontfacing (FF) and on the other side it must be backfacing (BF), or vice versa. This is illustrated in Figure 1a. Now, assume that we desire to augment the vertex so that it has degree three. This is done by inserting an edge, c, that connects to the vertex. See Figure 1b. Note that the same must hold for this new edge: FF on one side and BF on the other. However, as can be seen to the right in the figure, the geometry between edge b and c now contains both FF and BF geometry, which implies that a new edge must be located there. When the new edge c was inserted, one can also swap places between the FF and BF for c, but this leads to the same inconsistency. The only difference is that another new edge appears between a and c. Induction gives that each newly inserted edge that makes the degree odd generates a new edge to maintain consistency.

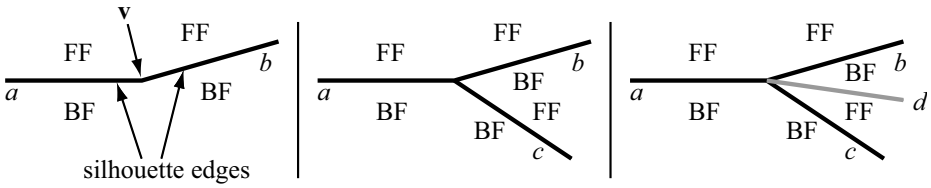


Figure 1: Left: two silhouette edges a and b meet at a vertex $\mathbf{v}$. Assume that a light source is located above the edges, and that the geometry above the edges are front facing (FF) as seen from the light source, and that the geometry below is backfacing (BF). Middle: add a new silhouette edge, c. Right: To cure the FF/BF inconsistency, a new edge d must be inserted.

At this point we have shown that the degree of an SVS vertex must be even. To prove that it can be larger than two, we give a simple example of a very simple geometrical model with vertices with degree four. See Figure 2.

Finally, we end this short note with an intuitive example that explains

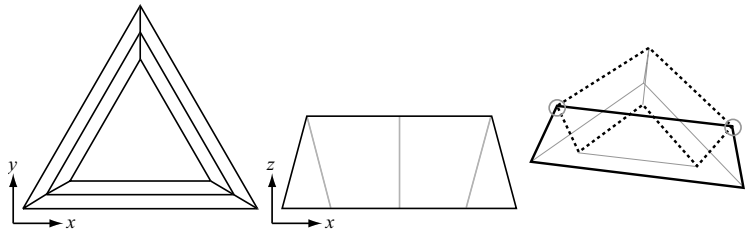


Figure 2: The left and middle parts illustrate a simple object from two orthographic views. To the right, a projection of the three-dimensional object is shown. Here, we asssume that a light source is located where the viewer is. This implies that the black continuous loop is an SVS, and the black dashed loop is another SVS. The circled vertices have degree four.

why SVSs with vertices with odd degrees would not work for shadow volume based algorithms. As can be seen in Figure 3, such scenes would easily create inconsistent results as well. In our silhouette generation software, we have seen vertices with degrees 2,4,6, and 8.

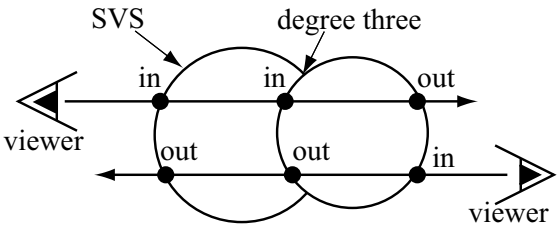


Figure 3: An example of incorrect shadow volumes. To understand the purpose of this illustration, imagine that the two view rays coincide. Normally, one would expect that the reciprocity law holds here, i.e., the left viewer experiences exactly what the right viewer experiences (only difference is direction).

References

[1] Akenine-Möller, Tomas, and Ulf Assarsson, "Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges," *13th Eurographics Workshop on Rendering*, pp. 309–318, June 2002. 133

[2] Assarsson, Ulf, and Tomas Akenine-Möller, "A Geometry-based Soft Shadow Volume Algorithm using Graphics Hardware," to appear in *SIGGRAPH 2003*, 2003. 133

[3] Bergeron, P., "A General Version of Crow's Shadow Volumes," *IEEE Computer Graphics and Applications*, vol. 6, no. 9., pp. 17–28, September 1986. 133

[4] Crow, Frank, "Shadow Algorithms for Computer Graphics," *Computer Graphics (Proceedings of ACM SIGGRAPH 77)*, pp. 242–248, July 1977. 133

[5] Heidmann, Tim, "Real shadows, real time," *Iris Universe*, no. 18, pp. 23–31, November 1991. 133