REAL TIME SOFT SHADOW GENERATION FOR ARBITRARY PLANAR LIGHTS

BY

MUSHFIQUR ROUF

A thesis submitted to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

DHAKA, BANGLADESH

AUGUST 2007

CERTIFICATE OF APPROVAL

The thesis titled “Real Time Soft Shadow Generation for Arbitrary Planar Lights” submitted by Mushfiqur Rouf, Roll No: 100505056P (Session: October 2005) to the Department of Computer Science and Engineering of Bangladesh University of Engineering and Technology has been accepted as satisfactory in partial fulfillment of the requirement for the degree of Master of Science in Computer Science and Engineering on August 19, 2007.

Board of Examiners

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
August, 2007

CANDIDATE'S DECLARATION

It is hereby declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.

[illegible]

Mushfiqur Rouf

To the Almighty

To my family

ACKNOWLEDGEMENT

All praise to Allah, the most benevolent and the Almighty, for His boundless grace in successful completion of this thesis.

I would like to express my sincere respect and gratitude to my thesis supervisor, Dr. Md. Mostofa Akbar, Associate Professor of the Department of Computer Science and Engineering (CSE), Bangladesh University of Engineering and Technology (BUET), Dhaka, for his thoughtful suggestions, constant guidance and encouragement throughout the progress of this research work. If he did not make my life miserable through asking to solve the unsolvable, I would not be able to come up with this novel algorithm.

This research heavily involves computational geometry. I had to try different approaches before finding the final solution. Dr. Masud Hasan, Assistant Professor, Department of Computer Science and Engineering, BUET, Dhaka, always helped me with his knowledge and experience. I would like to thank Dr. M. Kaykobad for his continuous encouragement and Dr. Muhammad Masroor Ali for his continuous support to place quick orders for official processing of my Masters degree.

My younger brother Mr. Shahriar Rouf, student of the Department of Computer Science and Engineering, BUET, Dhaka, helped me on several occasions by writing code segments and carrying out mathematical derivations.

I also express my profound thanks to Mr. Jamiur Rahman, one of my past undergraduate thesis students at BRAC University. His tremendous interest in the field of computer graphics led me to this problem.

I am very much thankful to my colleagues and my students at the Department of Computer Science and Engineering, BRAC University, Dhaka, for continuous support through sharing work load and other responsibilities during my busy days.

Finally, it was the inspiration from my father, Dr. Md. Abdur Rouf, Professor, Dept of Civil Engg and Dean, Faculty of Civil Engg., BUET, Dhaka, which encouraged me to start pursuing Masters Degree at BUET and start my research in the first place. I would like to extend my sincere thanks to my parents and my beloved wife whose continuous inspiration, sacrifice and support encouraged me to complete the thesis successfully.

ABSTRACT

Shadows are crucial for the human perception of the 3D world. Hard shadows are produced when a typical 0/1 definition of shadows is applied: if the light source can be seen from a pixel, it is lit; otherwise, it is in shadow. However, in nature, most light sources have a measurable size and hence they can be partially occluded. This fuzzy definition of shadows introduces a new genre of shadow generation problem: soft shadow generation. Soft shadows are obviously very important for realistic image rendering. But true soft shadow calculation is a computation intensive problem which requires considering complicated receiver-occluder-light source relation in three dimensions. So, to give real time graphics a realistic touch, we must come up with approximation algorithms that produce almost physically correct shadow.

Of the main two trends of shadow algorithms, shadow map approach suffers from sampling artifacts due to image space computation. Hence, to produce convincing shadows in real time, a shadow volume algorithm is needed that can generate soft shadows. We have developed an algorithm, under the shadow volume theme, to compute soft shadows for triangulated simple three dimensional objects. We have introduced a double silhouette concept; we are considering two silhouettes – umbra silhouette, responsible for the outline of the umbra region; and the penumbra silhouette, responsible for the outline of the penumbra region of the generated soft shadow. Our algorithm basically interpolates between these two extremes of the penumbra region. We have devised a way to break up the occluders into silhouette rings, and compute shadow contributions at any pixel based on only these structures. Our goal was to produce smooth shadow, so we devised an interestingly innovative way of combining contributions from overlapping projections of silhouette edges: 'edge shadows'. We have compared the quality of the shadow generated from our algorithm with that of a few well known recent algorithms. Our algorithm does not suffer from the artifacts previous algorithms had. On top of that, since we compute shadows only depending on the silhouette edges, special cases like the idea of overlapping shadow and self shadow degenerated into the common case of generating soft shadow from silhouette edges.

Table of Contents

List of Figures

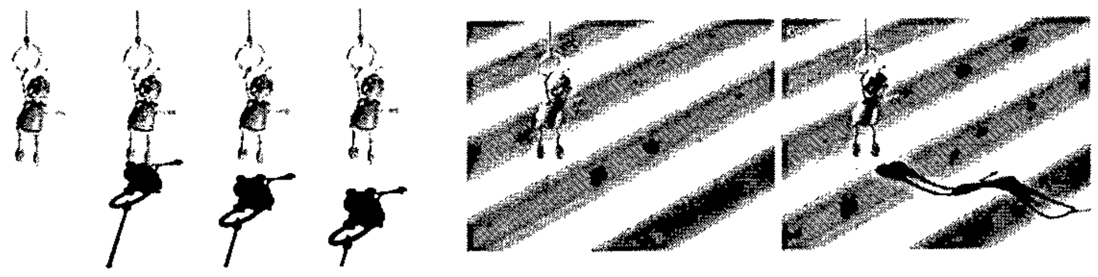
Chapter 1. Introduction

1.1. Motivation

Shadows are crucial for the human perception of the 3D world.

With the emergence of computer graphics technology, researchers have developed experiments to understand the impact of shadows on our perception of a scene. Through different psychophysical experiments they established the importance of the shadows in understanding:

- the position and size of the occluder [1], [2], [3], [4], [5] (see Figure 1-1 below);
- the geometry of the occluder [2] (see Figure 1-2 below);
- the geometry of the receiver [2] (see Figure 1-1 below).

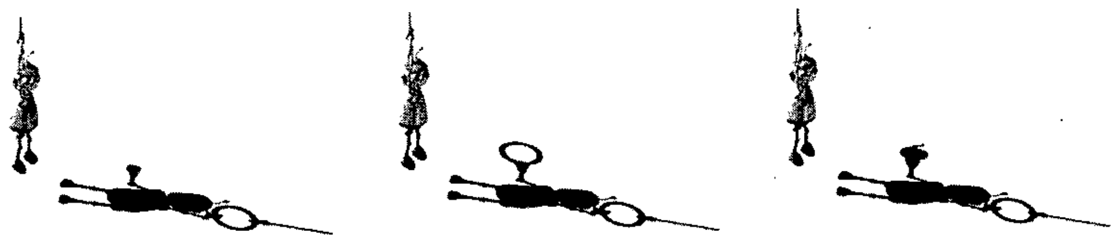


(a) Shadows provide information about the relative positions of objects.
On the left-hand image, we cannot determine the position of the robot, whereas on the other three images we understand that it is more and more distant from the ground.

(b) Shadows provide information about the geometry of the receiver.
Left: not enough cues about the ground.
Right: shadow reveals ground geometry.

Figure 1-1. Shadows provide information about the geometry of the occluder. This is extracted from [9].

Wanger [1] studied the effect of shadow quality on the perception of object relationships, basing his experiments on shadow sharpness. Hubona *et al.* [3] discuss the general role and effectiveness of object shadows in 3D visualization. In their experiments, they put in competition shadows, viewing mode (mono/stereo), number of lights (one/two), and background type (flat plane, "stair-step" plane, room) to measure the impact of shadows.



(a) Robot holds nothing in its left hand

(b) Robot holding a ring in its left hand

(c) Robot holding a teapot in its left hand

Figure 1-2. Shadows provide information about the geometry of the occluder. This is extracted from [9].

Kersten *et al* [4], [5] and Mamassian *et al* [2] study the relationship between object motion and the perception of relative depth. In fact, they demonstrate that simply adjusting the motion of a shadow is sufficient to induce dramatically different apparent trajectories of the shadow-casting object.

These psychophysical experiments convincingly establish that it is important to take shadows into account to produce images in computer graphics applications. Shadows in general help in our understanding of 3D environments and blurred *soft shadows* take part in realism of the images.

1.2. Problem Definition

To generate real shadows in real time, instead of generating the computationally correct true shadow, the shadow algorithms try to approximate the effects. The different approaches explored by researchers are discussed in brief in Chapter 2.

To produce an almost perfect shadow in real time, the shadow volume approach must be used as it does not suffer from any sampling artifacts. The best soft shadow algorithm of this type published to date is the *Penumbra Wedges* algorithm (see Section 2.4.1). It renders almost realistic soft shadows. But this algorithm has two serious limitations:

- It uses the basic hard shadow algorithm to produce the soft shadow volumes. This approximation results in an error called *single silhouette artifact* (see Section 2.5).

- There is no mechanism as to how to combine overlapping shadow from two separate objects.

To solve these limitations, we need to go back to the root of the problem and identify the differences between hard shadows and soft shadows. In this thesis, we have tried to produce an algorithm that explores this idea and tries to overcome the limitations stated above.

1.3. Objective and Scope of the Work

We are designing an algorithm that

- is a soft shadow volume algorithm; and
- can render soft shadows in real time.

This algorithm fixes three important features not handled well in previous algorithms:

- it does not produce Single Silhouette Artifacts (see Section 2.5);
- it can combine shadows of multiple objects; and
- it can compute self shadows.

Previous shadow volume algorithms support only circular or square light sources. Our algorithm can handle arbitrary shaped convex polygonal planar light sources. In case of concave planar light sources, we suggest breaking up the light source into a minimal number of convex polygons.

Although our algorithm can perform best if implemented in programmable graphics hardware, for the purpose of this thesis, software simulation is used. We are comparing visual output with soft shadow algorithms; only real time algorithms are compared.

This algorithm is intended for convex planar light sources and triangulated, simple, non-self-intersecting and hole-free objects (or *occluders*). For parametric smooth curves and splines, a preprocessing step will be necessary. This preprocessing is required to construct a polyhedron composed of triangular facets that can approximate the smooth object.

1.4. Outline

The remaining part of the thesis is organized as follows.

Chapter 2 discusses the existing algorithms, from general to specific to our problem. The strengths and weaknesses of these algorithms are stated.

Chapter 3 contains the detailed description of how our algorithm generates soft shadow.

Chapter 4 contains the outcome of our work. Discussions on several sets of snapshots of rendered shadow explain different aspects of the algorithm. The artifacts are presented and discussed. Our algorithm is compared to the existing algorithms in terms of the quality of the shadow rendered.

Chapter 5 concludes the thesis and presents possible future trends of research in the same field.

Chapter 2. Literature Review

Probably the first thorough analysis of shadows was Leonardo Da Vinci's [6], focusing on paintings and static images. Also of note is the work of Lambert [7] who described the geometry underlying cast shadows.

Progress in computer graphics technology and the development of consumer-grade graphics accelerators have made real-time 3D graphics a reality [8]. However, incorporating shadows, and especially realistic soft shadows, in a real-time application, has remained a difficult task (and is generating a great research field).

2.1. Basic Concepts

2.1.1. What is a Shadow

Consider a light source *L* illuminating a scene: *receivers* are objects of the scene that are potentially illuminated by *L*. A point *P* of the scene is considered to be in the *umbra* if it cannot see any part of *L*, *i.e.* it does not receive any light directly from the light source.

If *P* can see a part of the light source, it is in the *penumbra*. The union of the umbra and the penumbra is the shadow, the region of space for which at least one point of the light source is occluded. Objects that hide a point from the light source are called *occluders*.

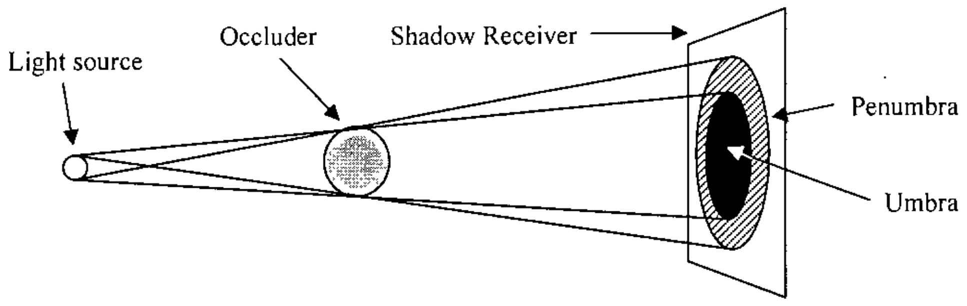


Figure 2-1. Basics of shadows.

2.1.2. Hard Shadows vs. Soft Shadows

The common-sense notion of shadow is a binary status, i.e. a point is either "in shadow" or not. This corresponds to *hard shadows*, as produced by point light sources: indeed, a point light source is either visible or occluded from any receiving point.

However, point light sources do not exist in practice and hard shadows give a rather unrealistic feeling to images. Note that even the sun, probably the most common shadow-creating light source in our daily life, has a significant angular extent and does not create hard shadows. Still, point light sources are easy to model in computer graphics and several algorithms let us compute hard shadows in real time.

On the other hand, the determination of the umbra and penumbra is a difficult task in general, as it amounts to solving visibility relationships in 3D, a very hard problem.

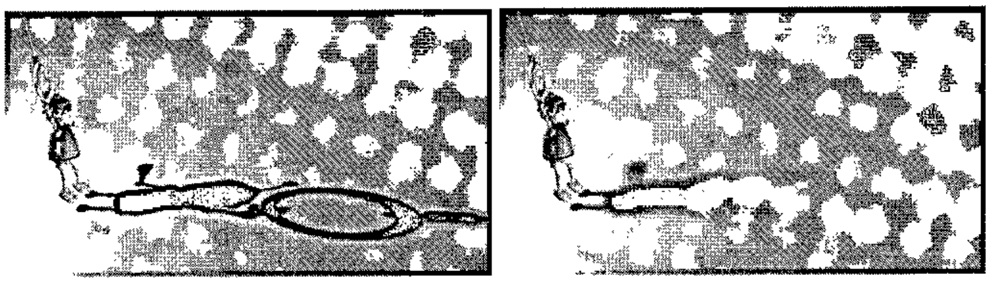


Figure 2-2. ***Left:*** **Illustration of hard shadow;** ***Right:*** **Illustration of soft shadow. This is extracted from [9].**

Soft shadows are obviously much more realistic than hard shadows; in particular the degree of softness (blur) in the shadow varies dramatically with the distances involved between the source, occluder, and receiver (see Figure 2-2 above). Note also that a hard shadow, with its crisp boundary, could be mistakenly perceived as an object in the scene, while this would hardly happen with a soft shadow.

In computer graphics, we can approximate small or distant light source as point sources in a very small number of cases. When the distance from the light to the occluder is much larger than the distance from the occluder to the receiver, the penumbra region almost vanishes, so the hard shadow looks quite real. Also, if the resolution of the final image does not allow proper rendering of the penumbra, there will be no point to render soft shadow. In all other cases great benefits can be expected from properly representing soft shadows.

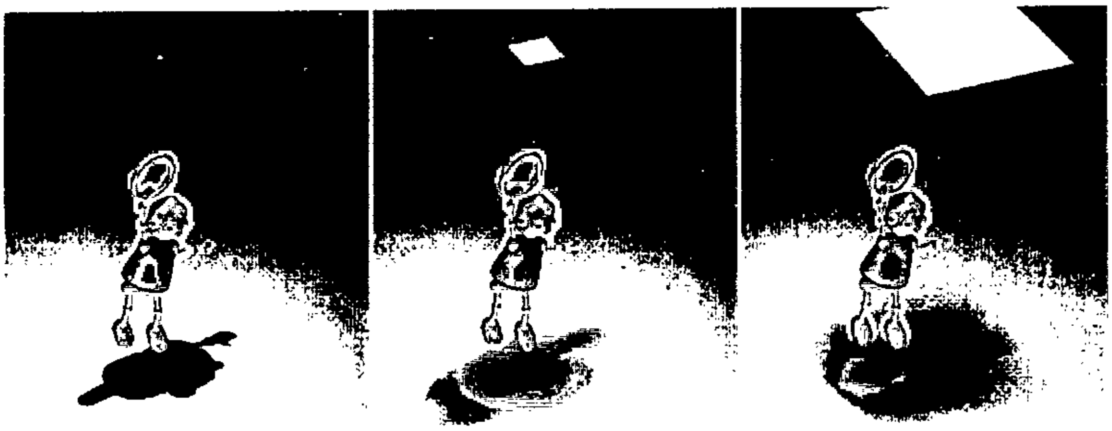


Figure 2-3. When the light source is significantly larger than the occluder, the shape of the shadow is very different from the shape computed using a single sample; the sides of the object are playing a part in the shadowing. This is extracted from [9].

2.2. Shadow Computation

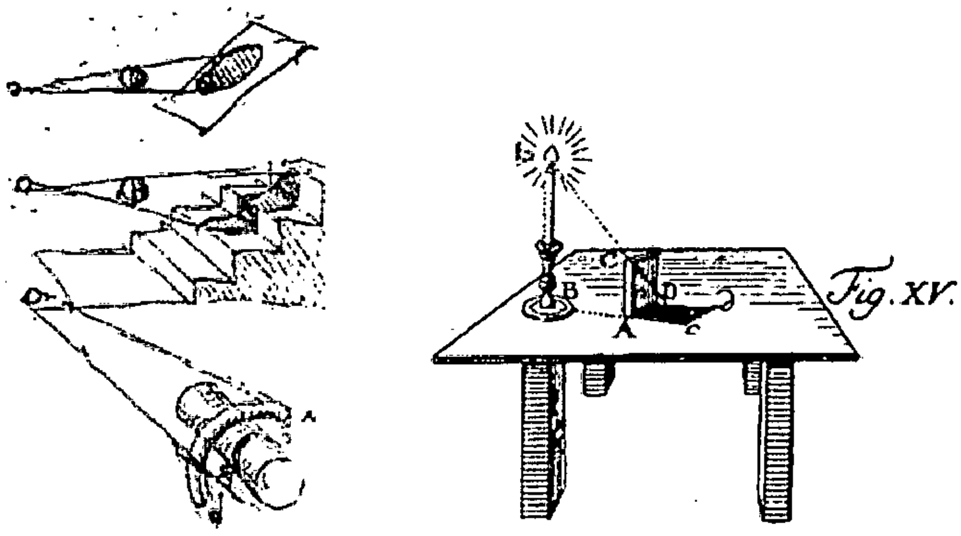


Figure 2-4. *Left:* Study of shadows by Leonardo da Vinci [6]; *Right:* Shadow construction by Lambert [7]. This is extracted from [9].

2.2.1. Geometry of Soft Shadows

Soft shadows can be considered as a combination of shadows, generated for *all* points of an object. Since in computer graphics, all objects are polygonal, we can take advantage by doing much less computation; by projecting the light source onto the receiver through the vertices on the object.

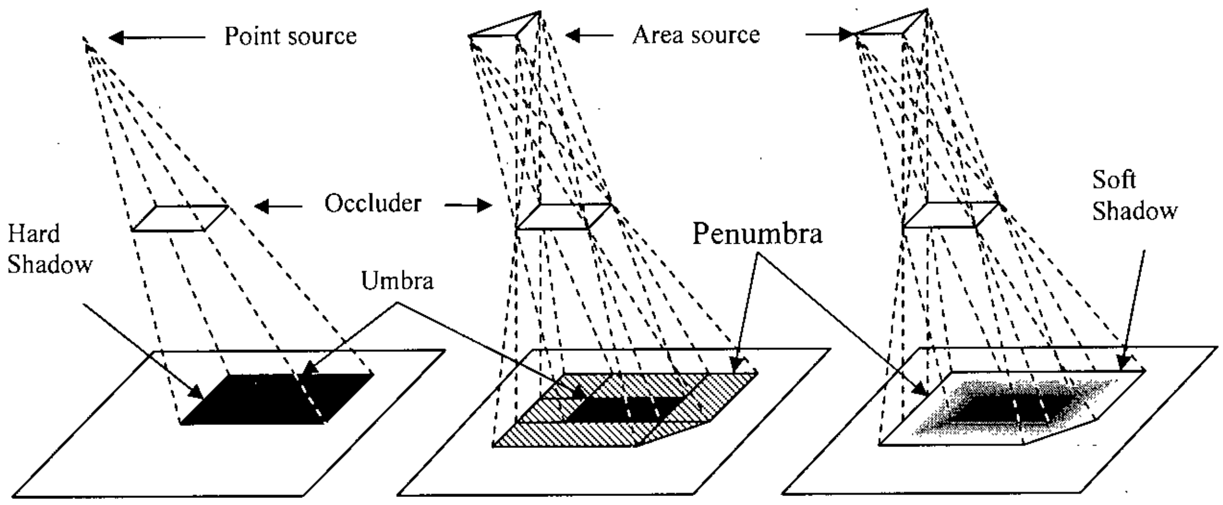


Figure 2-5. Demonstration of umbra and penumbra regions for light source.

2.3. Shadow Algorithms

In this section, we are describing the main two trends of (hard) shadow algorithms, namely, shadow volume and shadow map. Both of these algorithms have been extended in a number of ways [9] to produce soft shadow. In 2.4, we are going to discuss a few such soft shadow generation algorithms developed using the shadow volume approach. We will be presenting the algorithms, along with their improvements and fundamental drawbacks inherent to the basic idea of the same.

Now, we will be trying to justify why we preferred the shadow volume approach.

2.3.1. Ray Tracing

Soft shadow generation is a difficult problem. Ray tracing can solve it perfectly, but it cannot be used for real time rendering. This is because ray tracing traces back from eye through each pixel to be rendered. The number of rays to be traced back can be exponentially large, since at each incidence of the light source onto a surface, one of three things can happen: transmission, absorption and reflection.

2.3.2. Shadow Map

2.3.2.1. Method

The basic operation for computing shadows is identifying the parts of the scene that are hidden from the light source. Intrinsically, it is equivalent to visible surface determination, from the point-of-view of the light source.

Computing shadows [10], [11], [12] starts by computing a view of the scene, from the point-of-view of the light source. When all objects are rasterized, the per pixel depth information is stored for the purpose of executing the Z-buffer algorithm. We store the z values of this image. This Z-buffer is the shadow map (see Figure 2-6 below).

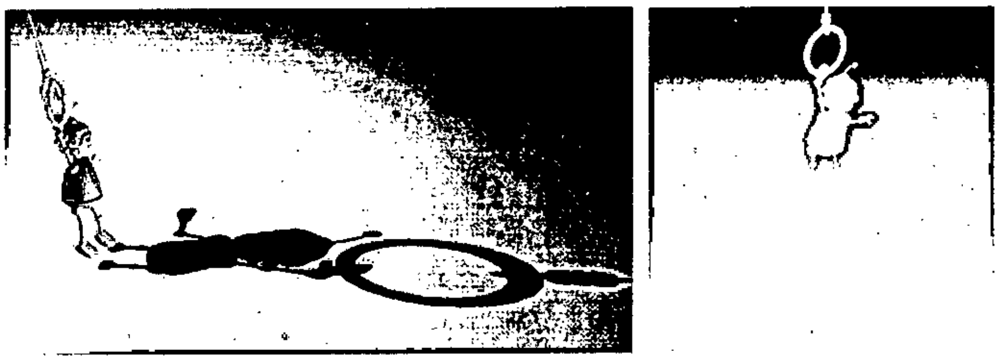


Figure 2-6. Shadow map for a point light source. *Left:* view from the camera. *Right:* depth buffer computed from the light source. This is extracted from [9].

The shadow map is then used to render the scene (from the normal point-of-view) in a two-pass rendering process:

- First, a standard Z-buffer technique, for hidden-surface removal.
- For each pixel of the scene, we now have the geometrical position of the object seen in this pixel. If the distance between this object and the light is greater than the distance stored in the shadow map, (the object must have been overwritten by another object nearer to the light source, and hence,) the object is in shadow. Otherwise, it is illuminated.

- The color of the objects is modulated depending on whether they are in shadow or not.

Shadow mapping has been implemented in most of the currently available graphics hardware. It uses an OpenGL library extension for the comparison between Z values, GL_ARB_SHADOW. The OpenGL Architecture Review Board (ARB) is the governing body of OpenGL. New extensions proposed by hardware vendors are tested for conformance and then accepted as a GL_ARB_* extension. The fact that GL_ARB_SHADOW is an ARB extension implies this capability is not vendor specific any more, any OpenGL 2.0 compatible hardware will accommodate this extension.

2.3.2.2. Benefits and Limitations

Shadow mapping has many advantages:

i) it can be implemented entirely using graphics hardware;
ii) creating the shadow map is relatively fast, although it still depends on the number and complexity of the occluders;
iii) it handles self-shadowing.

It also has several drawbacks:

i) it is subject to many sampling and aliasing problems;
ii) it cannot handle omni-directional light sources;
iii) at least two rendering passes are required (one from the light source and one from the viewpoint);

2.3.3. Shadow Volume

Another way to think about shadow generation is purely geometrical. This method was first described by Crow [13], and first implemented using graphics hardware by Heidmann [14].

2.3.3.1. Method

The algorithm consists in finding the silhouette of occluders along the light direction. The silhouette is the outline of the object when seen from the point of view of the light source. Only the silhouette determines the shape of the shadow. Hence computing the silhouette is the first step for computing the shadow.

Then this silhouette is extruded along the light direction, thus forming a shadow volume. Objects that are inside the shadow volume are in shadow, and objects that are outside are illuminated. The shadow volume is calculated in two steps:

- The first step consists in finding the silhouette of the occluder as viewed from the light source. The simplest method is to keep edges that are shared by a triangle facing the light and another in the opposite direction. This actually gives a superset of the true silhouette, but it is sufficient for the algorithm.
- Then we construct the shadow volume by extruding these edges along the direction of the point light source. For each edge of the silhouette, we build the half-plane subtended by the plane defined by the edge and the light source. All these half-planes define the shadow volume.

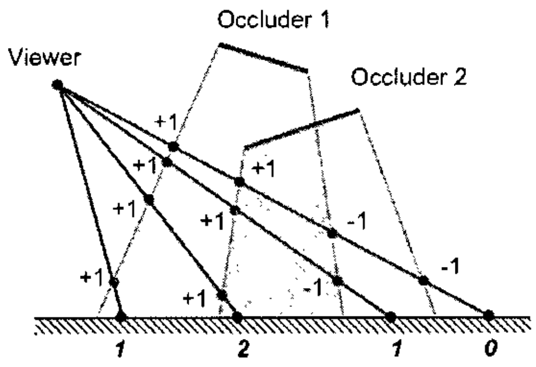


Figure 2-7. Determining whether a point is inside a shadow volume. This is extracted from [9].

- For each pixel in the image rendered, we count the number of faces of the shadow volume that we are crossing between the view point and the object rendered. Front facing faces of the shadow volume (with respect to the view point) increment the count; back-facing faces decrement the count (see Figure

2-7 above). If the total number of faces is positive, then we are inside the shadow volume, and the pixel is rendered using only ambient lighting.

The rendering pass is easily done in hardware using a stencil buffer [13], [15], [16]; faces of the shadow volume are rendered in the stencil buffer with depth test enabled this way: in a first pass, front faces of the shadow volumes are rendered incrementing the stencil buffer; in a second pass, back faces are rendered, decrementing it. Pixels that are in shadow are "captured" between front and back faces of the shadow volume, and have a positive value in the stencil buffer. This way to render volumes is called *zpass*.

Therefore the complete algorithm to obtain a picture using the Shadow Volume method is:

- Render the scene with only ambient/emissive lighting;
- Calculate and render shadow volumes in the stencil buffer;
- Render the scene illuminated with stencil test enabled: only pixels which stencil value is 0 are rendered, others are not updated, keeping their ambient color.

2.3.3.2. Advantages and Disadvantages

The shadow volume algorithm has many advantages:

- It works for omni directional light sources;
- It renders eye-view pixel precision shadows;
- It handles self-shadowing.

It also has several drawbacks:

- The computation time depends on the complexity of the occluders;
- It requires the computation of the silhouette of the occluders as a preliminary step;
- At least two rendering passes are required;
- Rendering the shadow volume consumes *fillrate* of the graphics card.

2.4. Soft Shadows

The simplest way to produce soft shadows with the shadow volume algorithm is to take several samples on the light source, compute a hard shadow for each sample and average the pictures produced. It simulates an area light source, and gives us the soft shadow effect. However, the main problem with this method is the number of samples it requires to produce a good-looking soft shadow, which precludes any real-time application. Also, it requires the use of an accumulation buffer, which is currently not supported on standard graphics hardware [9].

Soft shadows come from spatially extended light sources. To model properly the shadow cast by such light sources, we must take into account all the parts of the occluder that block light coming from the light source. This requires identifying all parts of the object casting shadow that are visible from at least one point of the extended light source, which is algorithmically much more complicated than identifying parts of the occluder that are visible from a single point. Because this visibility information is much more difficult to compute with extended light sources than with point light sources, most real-time soft shadow algorithms compute visibility information from just one point (usually the center of the light source) and then simulate the behavior of the extended light source using this visibility information (computed for a point).

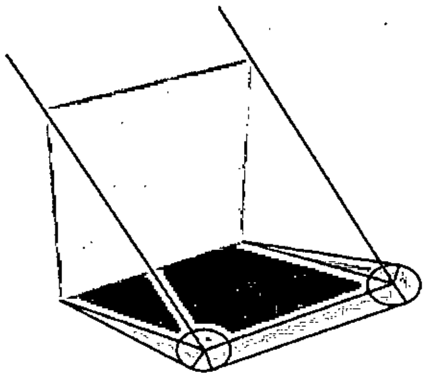


Figure 2-8. ***Plateaus*** **algorithm extends the shadow volume of an occluder with cones and planes. This is extracted from [17].**

The first geometric approach to generate soft shadows ("Soft Planar Shadows Using Plateaus") has been implemented by Haines [17]. It assumes a planar receiver, and generates an attenuation map that represents the soft shadow. The attenuation map is created by converting the edges of the occluders into volumes, and is then applied to the receiver as a modulating texture (see Figure 2-8).

For extended light sources, it can be assumed that the influence of the shape of the light source on a soft shadow is not directly perceptible. Most real-time soft shadow methods use this assumption by restricting themselves to simple light source shapes, such as spheres or rectangles:

- Single-sample soft shadows [18], [19], plateaus [17] and smoothies [20] assume a spherical light source. Soft shadow volumes [21] also work with a spherical light source.
- Visibility channel [22] was originally restricted to linear light sources.
- Subsequent implementation of the visibility channel works with polygonal light sources [18].
- Other methods place less restriction on the light source. Multi-sample methods [23], [24] can work with any kind of light source. Convolution [25] is also not restricted. However, in both cases, the error in the algorithm is smaller for planar light sources.
- Convolution [25] and soft shadow volumes [26], [21] work with textured rectangles, thus allowing any kind of planar light source. The texture can even be animated [26], [21].

2.4.1. Penumbra Wedges

Akenine-Möller and Assarsson [27], Assarsson and Akenine-Möller [26] and Assarsson et al. [21] have developed the *penumbra wedges* algorithm to compute soft shadows that builds on the shadow volume method and uses the programmable capability of modern graphics hardware [28], [29], [30] to produce real-time soft shadows. The algorithm starts by computing the silhouette of the object, as seen from a single sample on the light source. For each silhouette edge, it builds a silhouette wedge that encloses the penumbra caused by this edge. Then in three passes, the final image is rendered.

2.4.1.1. Method

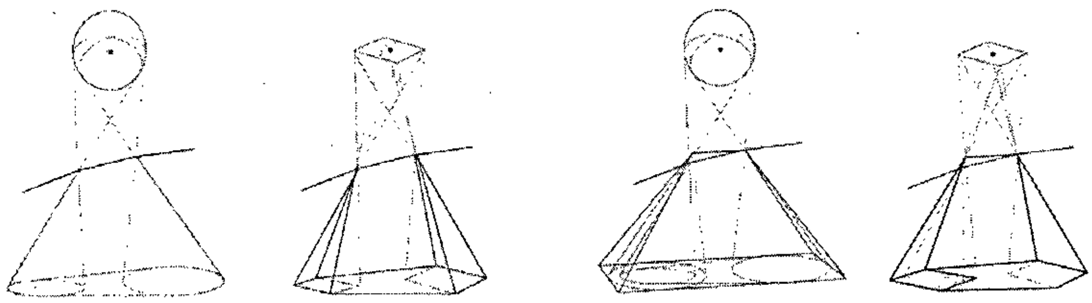


Figure 2-9. *Left:* Penumbra volumes. *Right:* Respective silhouette wedges. This is extracted from [21].

The algorithm starts by computing the silhouette of the object, as seen from a single sample on the light source. For each silhouette edge, a silhouette wedge is built that encloses the penumbra caused by this edge (see Figure 2-9 above). The wedge can be larger than the penumbra. Then, the shadow volume is rendered, using the standard method (see 2.3.3) in a visibility buffer. After this first pass, the visibility buffer contains the hard shadow.

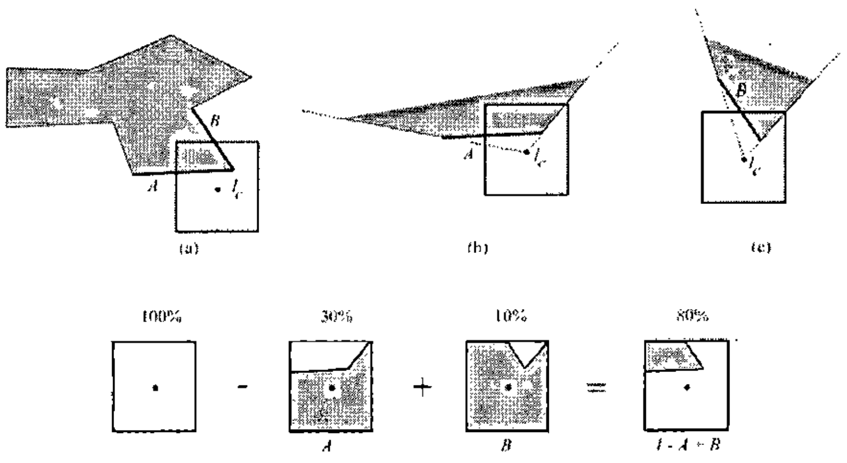


Figure 2-10. Splitting of shadow contribution for each wedge for a point p. A and B are two silhouette edges of a shadow casting object. This is extracted from [26].

In a subsequent pass, this visibility buffer is updated so that it contains the soft shadow values. This is done by rendering the front-facing triangles of each wedge. For each pixel covered by these triangles, the percentage of the light source that is occluded is computed, using fragment programs [29]. For pixels that are covered by the wedge but in the hard shadow (as computed by the previous pass), the percentage of the light source that is visible is computed and added to the visibility buffer. For pixels covered by the wedge but in the illuminated part of the scene, the percentage of the light source that is occluded is computed and subtracted from the visibility buffer (see Figure 2-10). After this second pass, the visibility buffer contains the percentage of visibility for all pixels in the picture.

In a third pass, the visibility buffer is combined with the illumination computed using the standard OpenGL lighting model, giving the soft shadowed picture of the scene.

2.4.1.2. Advantages and Disadvantages

The complexity of the algorithm depends on the number of edges in the silhouette of the object, and on the number of pixels covered by each penumbra wedge. As a consequence, the easiest optimization of the algorithm is to compute tighter penumbra wedges [21].

The main advantage of this algorithm is its speed. It is reported in [9] that using programmable graphics hardware for all complex computations, and tabulating complex functions into pre-computed textures, frame rates of 150 frames per second are obtained on simple scenes, 50 frames per second on moderately complex scenes (1,000 shadow-casting polygons, with a large light source), with very convincing shadows. Performance depends mostly on the number of pixels covered by the penumbra wedges, so smaller light sources will result in faster rendering.

The weak point of the algorithm is that it computes the silhouette of the object using only a single sample. It would fail on scenes where the actual silhouette of the object, as seen from the area light source, is very different from the silhouette computed using the single sample. Such scenes include scenes where a large area light source is close to the object (see Figure 2-3) and scenes where the shadows of several objects are combined together (as in Figure 2-11).

In those circumstances, it is possible to compute a more accurate shadow by splitting the light source into smaller light sources. The authors report that splitting large light sources into 2×2 or 3×3 smaller light sources is usually enough to remove visible artifacts. It should be noted that splitting the light source into n light sources does not cut the speed of the algorithm by n, since the rendering time depends on the number of pixels covered by the penumbra wedges, and smaller light sources have smaller penumbra wedges. One key to the efficiency of the algorithm is its use of fragment programs [29].

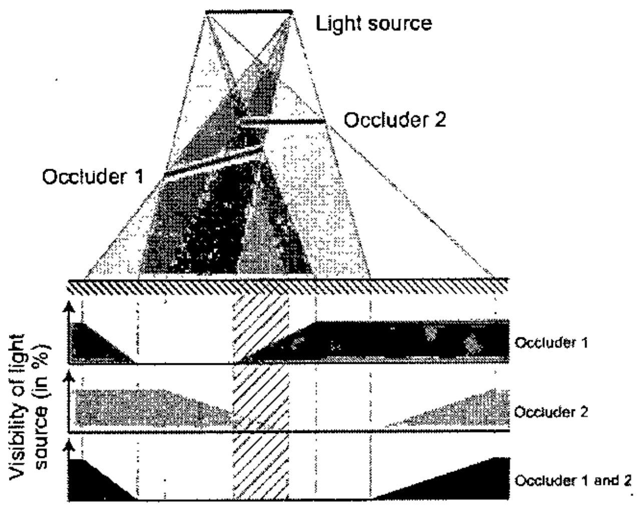
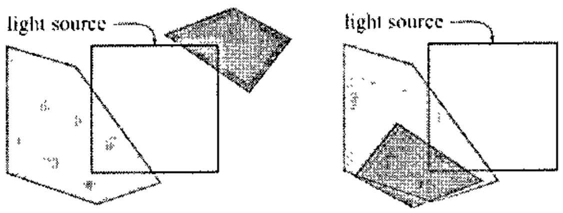


Figure 2-11. The shadow of two occluders is not a simple combination of the two individual shadows. Note in particular the highlighted central region which lies in complete shadow (umbra) although the light source is never blocked by a single occluder. This is extracted from [9].

The fragment programs take as input the projections of the extremities of the edge onto the plane of the light source, and give as output the percentage of the light source that is occluded by the edge. If several edges are projecting onto the light source, their contributions are simply added (see Figure 2-10) — this addition is done in the frame buffer. However, this simple addition can result in an object overlap approximation error (see Figure 2-12 below)



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 over 4 percent. To the left, the shadow casters together cover 20 percent, while to the right they cover 16 percent.

Figure 2-12. Overlap approximation: Penumbra Wedges compute equal light source visibility proportion for both cases above. This is extracted from [21].

2.4.2. Smoothies

Chan and Durand [18] present a variation of the shadow volume method that uses only graphics hardware for shadow generation.

2.4.2.1. Method

We start by computing the silhouette of the object. This silhouette is then extended using "smoothies", that are planar surfaces connected to the edges of the occluder and perpendicular to the surface of the occluder.

We also compute a shadow map, which will be used for depth queries. The smoothies are then textured taking into account the distance of each silhouette vertex to the light source, and the distance between the light source and the receiver.

In the rendering step, first we compute the hard shadow using the shadow map, and then the texture from the smoothies is projected onto the objects of the scene to create the penumbra.

2.4.2.2. Advantages and Disadvantages

This algorithm only computes the outer penumbra. As a consequence, occluders will always project an umbra, even if the light source is very large with respect to the occluders. As mentioned earlier, this makes the scene appear much darker than anticipated, an effect that is clearly noticeable except for very small light sources (see Figure 2-13 below).

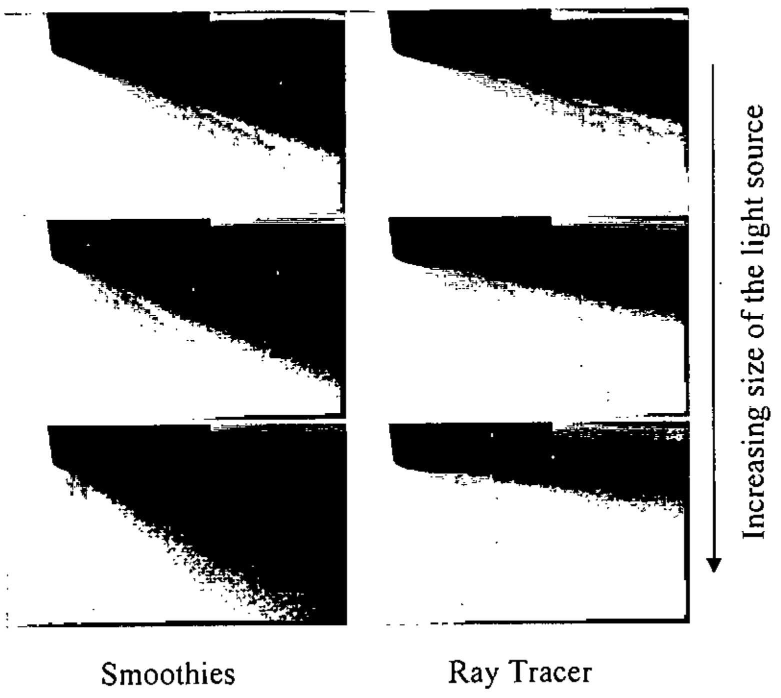


Figure 2-13. Smoothies algorithm always produces an umbra. This is extracted from [18].

The size of the penumbra depends on the ratio of the distances between the occluder and the light source, and between receiver and light source, which is perceptually correct.

Connection between adjacent edges is still a problem with this algorithm, and artifacts appear clearly except for small light sources.

The shadow region is produced using the shadow map method, which removes the problem with the fill rate bottleneck experienced with all other methods based on the shadow volume algorithm.

The weak point of this algorithm is that it produces fake shadows.

2.5. Single Silhouette Artifact

The hard shadow algorithms can act upon only one silhouette of a polyhedron since the silhouette is merely the outline of the planar projection of the polyhedron when viewed from the point light source.

On the other hand, the soft shadow algorithms cannot use only one silhouette since the same object looks different from different points of the lights source (see Figure 2-14 below).

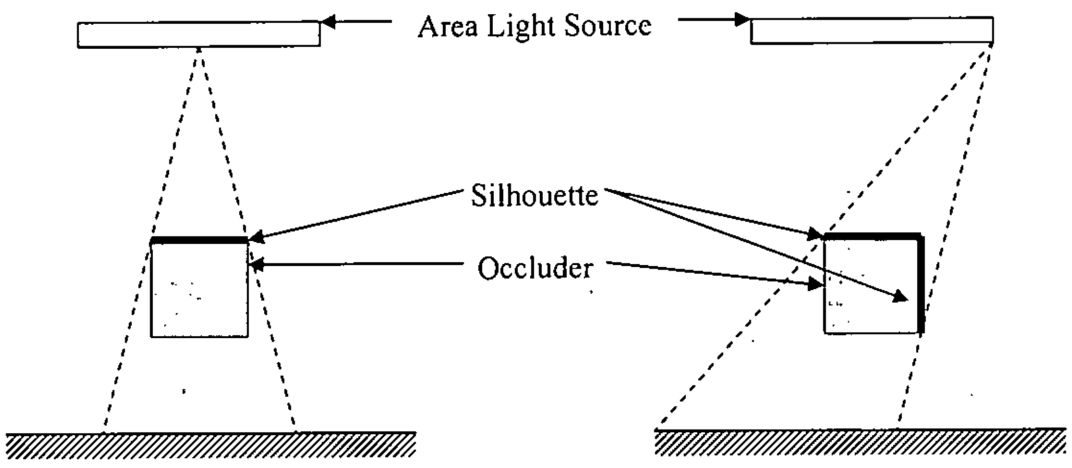
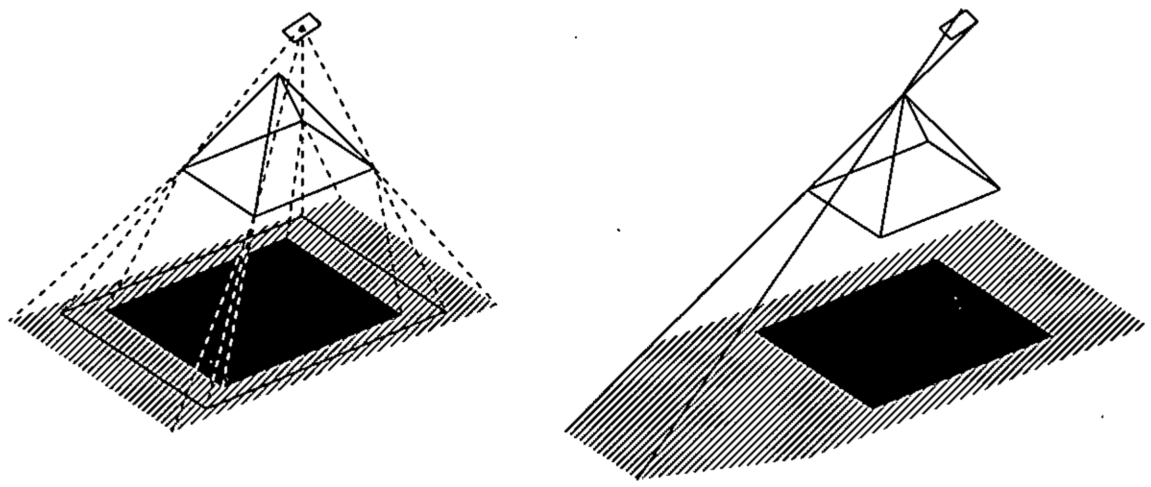


Figure 2-14. Different points on the area light source sees different silhouette.

Current soft shadows algorithms follow typical hard shadow approaches, so they rely on only one silhouette. But this may not lead to correct shadow computation at all times (see Figure 2-15 below).



A single-silhouette artifact produced by *Penumbra Wedges*

Our proposed algorithm makes better approximation

Figure 2-15. Single Silhouette Artifact.

Chapter 3. Soft Shadow Volume Algorithm

3.1. Introduction

The previous chapter described current well known algorithms along with their strengths and weaknesses. We now present the central part of this thesis – the algorithm we have developed to compute soft shadows.

3.2. The Algorithm

3.2.1. Basic Principle of the Algorithm

The proposed shadow algorithm uses two sets of silhouette edges:

1. *Penumbra Silhouettes*, responsible for the outer boundary of the penumbra;
2. *Umbra Silhouettes*, responsible for the outer boundary of the umbra (see Figure 3-1 below).

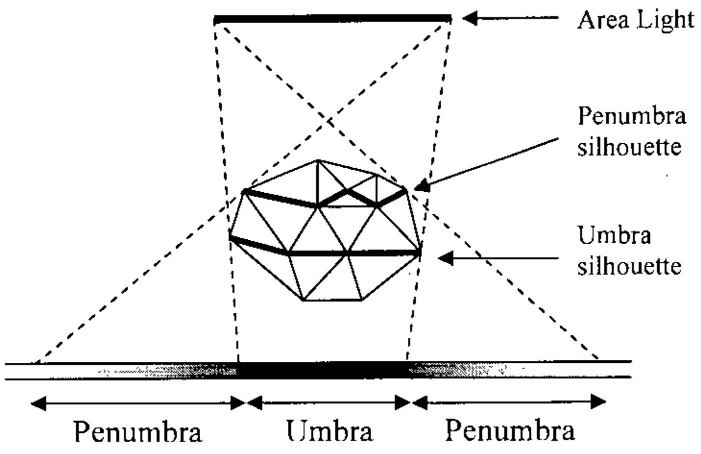


Figure 3-1. Double Silhouette.

Computing a perfect soft shadow requires three dimensional volume intersections. This is highly computation intensive and not applicable for soft shadow generation in real time. The proposed algorithm produces a shadow with the same shape as the true shadow and interpolates between the two extremes of the soft shadow area.

The proposed algorithm has two phases. In Phase 1, the silhouette edges are identified and the soft shadow volumes, *edge shadows*, are computed. In Phase 2, each pixel's illumination level is determined.

3.2.2. Computing the Silhouette Edges

It is obvious that computing shadows using the whole object is costly. To improve time performance, we use the silhouette edges to complete the shadow.

In traditional shadow algorithms, due to point light sources, the facets of an object are either *lit* or *in shadow*. However, since we are considering area light sources, some of the facets now can be *partially lit*, that is, from any point on any of these facets, only a fraction of the light source can be seen.

For any convex object, the facets facing the light source are *fully lit*, the back facing facets are *in shadow* and the facets in between are *partially lit.*

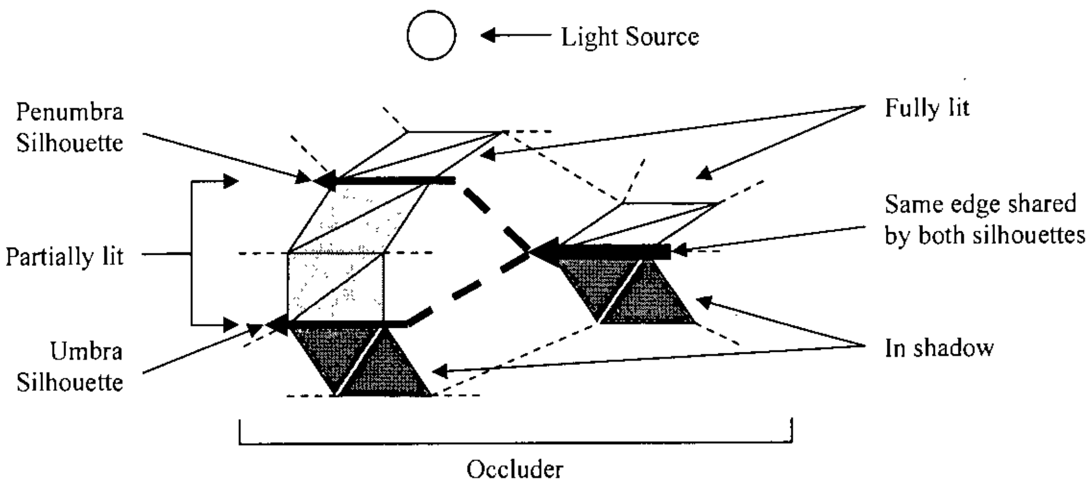


Figure 3-2. Penumbra and Umbra Silhouettes.

The boundary of the fully lit area is the *penumbra silhouette*. The boundary of the shadowed area is the *umbra silhouette*. A single edge can be a part of both silhouettes (see Figure 3-2 above).

It should be noted that a *fully lit* facet may not be truly fully lit – the light source may be obscured by another part of the same object or by another object. Hence by the term fully lit, we mean the facet is *potentially* fully lit *provided* no other facet is obscuring the light source.

Computing the silhouette requires traversing through all the facets and checking with the adjacent facets.

The proposed algorithm assumes:

1. All *facets* are triangular. Facets are the smallest planer polygons that make up the surface of the objects to be rendered. Any object input to the algorithm must be broken into triangular facets. Parametric smooth surfaces or curves must be approximated by triangulated polyhedra. There cannot be any edge that has a facet on one side only;
2. The occluders are simple polyhedra; they do not have holes or do not self-intersect. If any complex object does have holes, it has to be broken into two pieces.

3.2.3. Computing the 'Edge Shadows'

Each silhouette edge is then projected backwards from the light to produce a semi-infinite volume. It is semi-infinite since it is extended up to infinity away from the light source but the volume can be bounded by a front plane, a back plane and two corner planes. This volume contains all points from which the edge is blocking a part of the light source.

Without loss of generality, we can assume the planar light is placed at $z = L_z$ and minimum z coordinate of all of the occluders is 0. Then each edge shadow can be safely cut by the $z = 0$ plane to make it finite along the direction away from the light source.

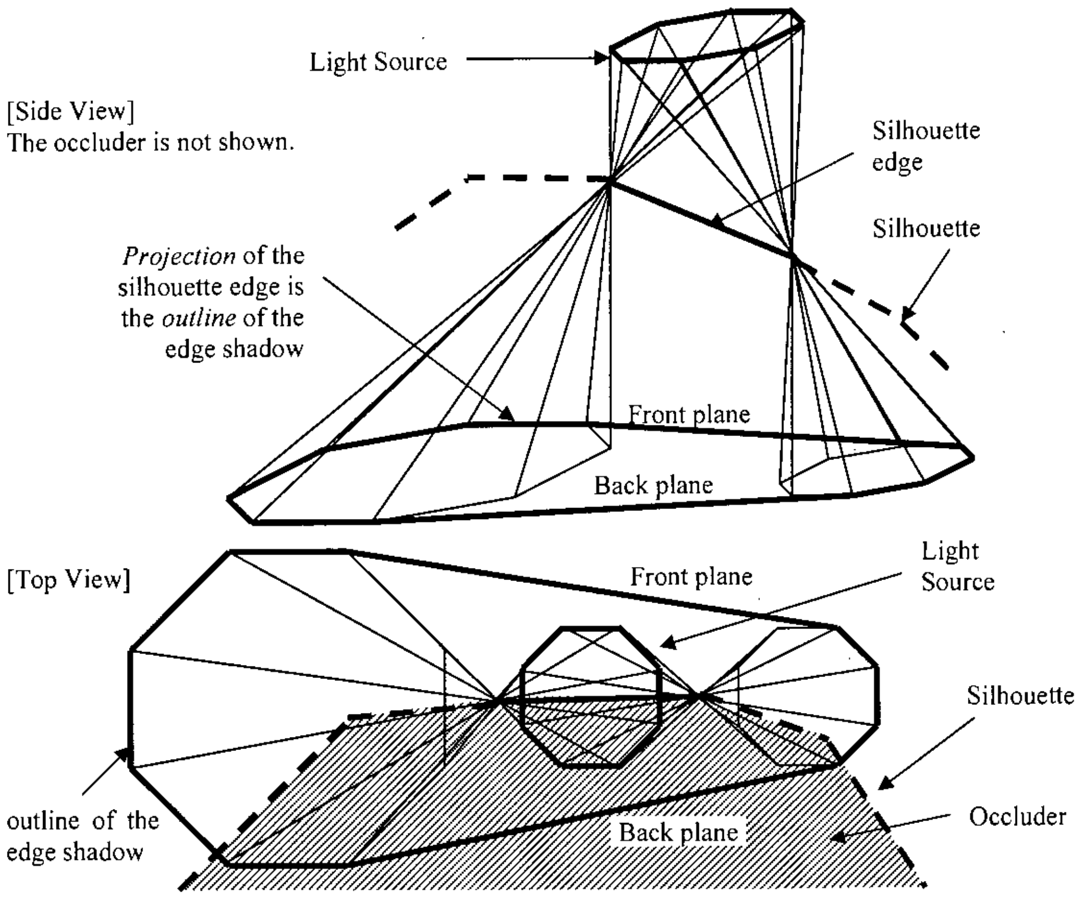


Figure 3-3. 'Edge Shadow': A Penumbra Volume.

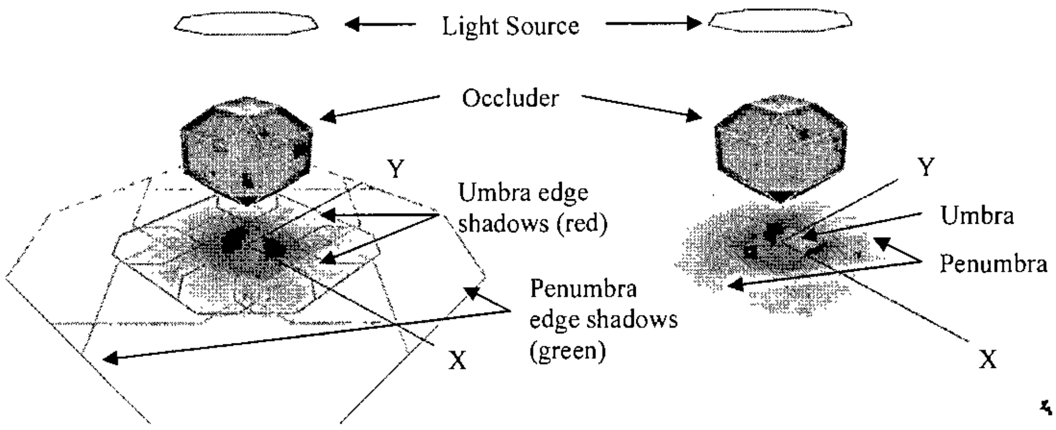


Figure 3-4. Edge Shadows and the generated shadow.

Each edge shadow has a mid quadrilateral area and two corner areas. Clearly, these corners areas are due to the two end points of the silhouette edge (see Figure 3-3 above). These two endpoints are shared by two other neighboring silhouette edges (see Figure 3-5 below). Clearly, the two corner areas are shared by the neighboring edge shadows.

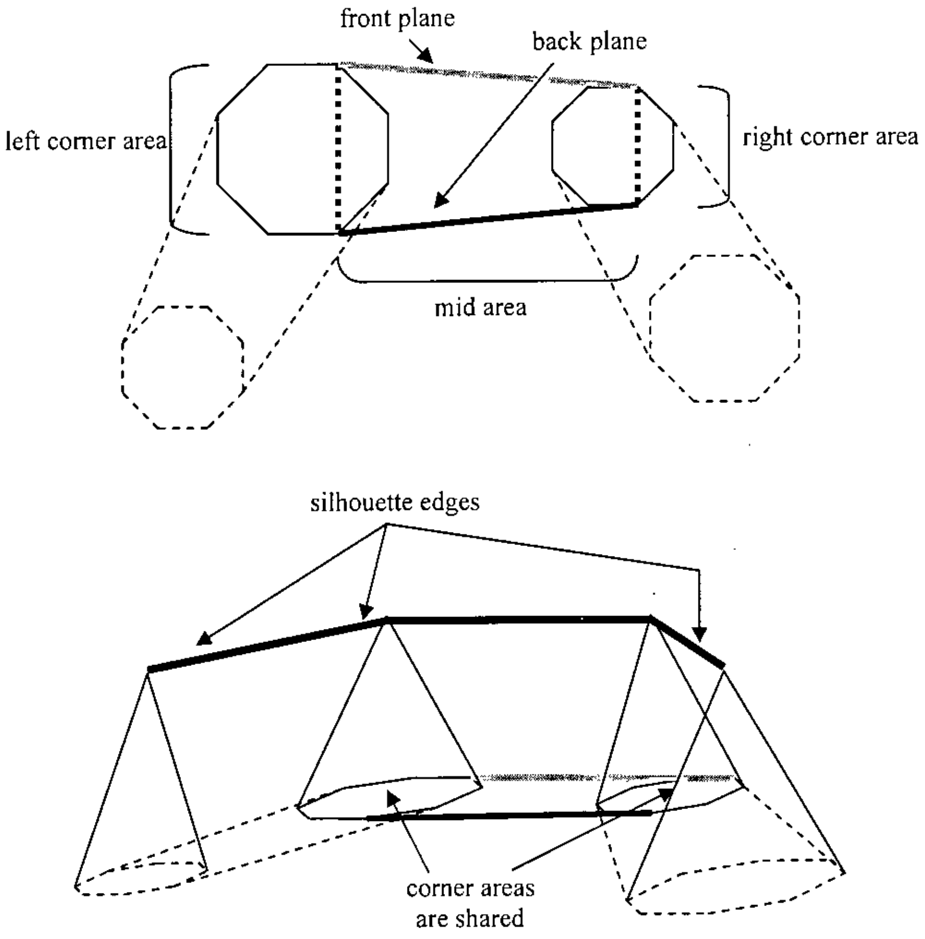


Figure 3-5. An edge shadow has mid and corner areas.

Since the corner areas are shared by two edge shadows, light source visibility at different point inside the corners become difficult to compute. Since we are trying to *interpolate* light visibility proportion, we have to come up with a mechanism to compute the light visibility at these shared areas in such a way as the shadow may not be perfect, but it must be pleasing to the eye.

3.2.3.1. Direction of Edge Shadows

We define the direction of an edge shadow as an imaginary vector representing the projection of the silhouette edge from a *center point* of the light. As the silhouette is connected, these projections also form a loop. Furthermore, the directions of edge shadows are assigned in a counter clockwise orientation.

This edge shadow direction information help interpolate the visibility of the light source. The direction dictates which part of the light source is obstructed by the respective silhouette edge.

The interpolated value merely says *how much* of the light source is visible. But to determine *which part* is visible, we actually need to project the silhouette edge onto the light source. Instead, we do a rough estimate using the direction (see Figure 3-6 below).

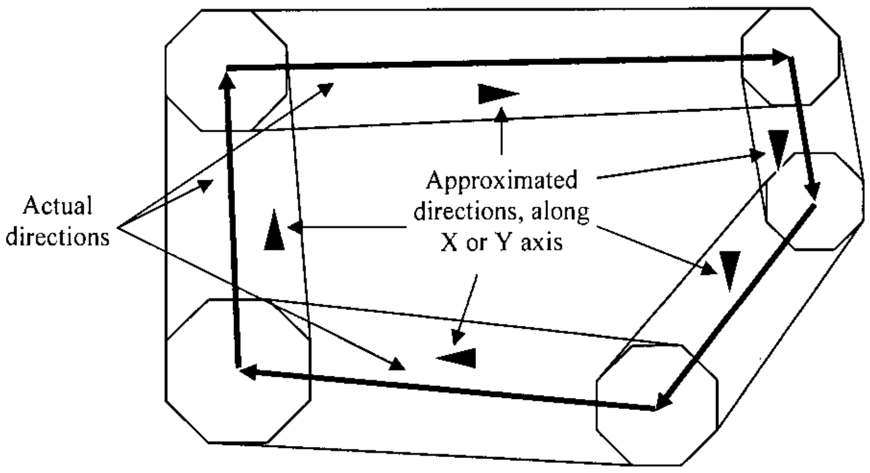


Figure 3-6. Edge shadow directions: Actual and Approximated.

At the corners areas, the visibility determination is tricky. Two silhouette edges give two proportion values. We need to take *union* of these to proportions, we cannot simply *add* them. For example, consider two neighboring silhouettes each allowing 50% visibility from some point. If these silhouettes are at 90° with one another, we should be able to see 75% of the light source, not 100% (see Figure 3-7 below).

The direction is approximated to nearest multiple of 90° (see Figure 3-6 above). This way the light visibility problem degenerates from complicated area intersection problem into rectangular shape intersection problem.

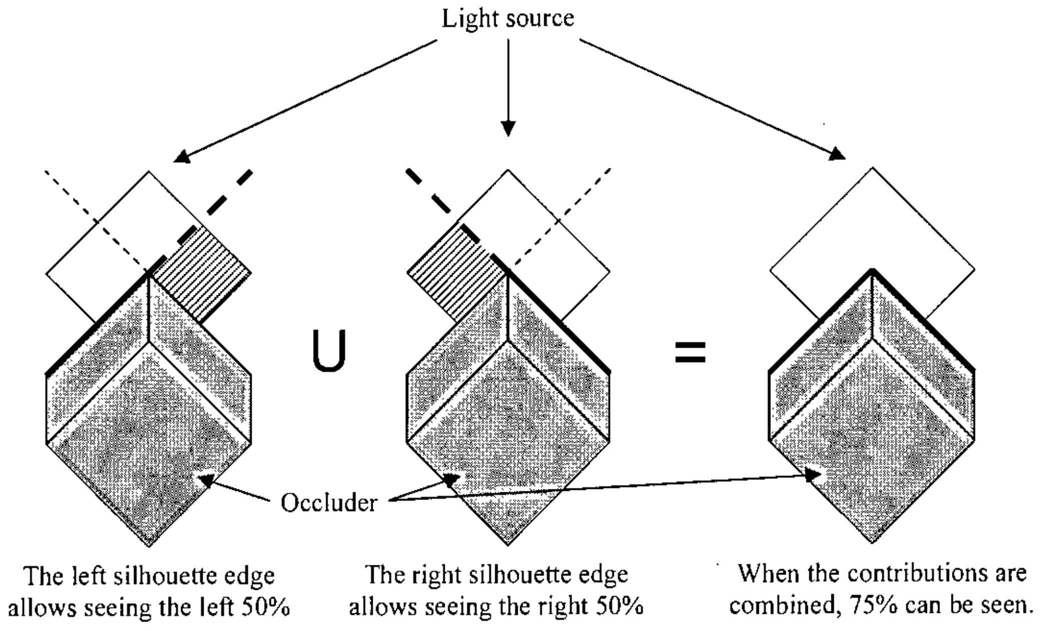

Light source
U
=
Occluder
The left silhouette edge allows seeing the left 50%
The right silhouette edge allows seeing the right 50%
When the contributions are combined, 75% can be seen.

Figure 3-7. Light source visibility combination using directions.

3.2.3.2. The Edge Shadows Are Organized in Rings

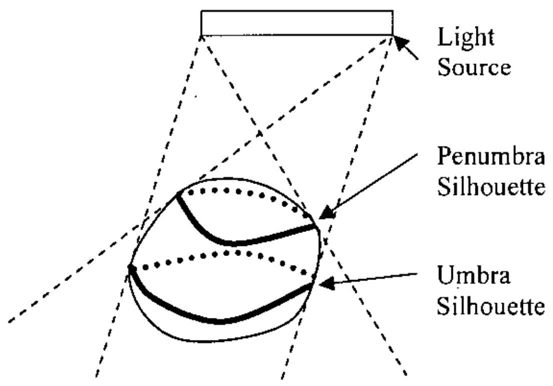


Figure 3-8. Silhouettes form rings.

The silhouette edges always *surround* the object. Since the light source is assumed omni directional, the points on the occluder, which constitute the periphery of the fully lit region, must be connected. These points constitute the penumbra silhouette. Same goes for the umbra silhouette.

A convex object will have exactly one umbra silhouette and exactly one penumbra silhouette. Both of these silhouettes will be of the form of a ring; surrounding the whole object.

For more complex objects, there can be more than just the two rings. Now the problem becomes more complicated as the rings can intersect one another. It is worth mentioning that a vertex of a silhouette edge always is connected to an even number of silhouette edges [31]. In our case, there are two types of silhouettes – the umbra silhouettes and the penumbra silhouettes. At some vertex, the umbra silhouette edges are due to one vertex on the light source and the penumbra silhouette edges are due to another vertex of the light source. Clearly, applying the previous observation, any vertex is connected to even number of umbra silhouettes and even number of penumbra silhouettes. It can be further proved that equal number of them will be *incoming* and equal number will be *outgoing*. Since indegree and outdegree at each vertex are equal, we can decompose the whole silhouette graph into simple cycles or in other words, *silhouette rings*.

However, the edge shadows corresponding to the *concave* silhouette edges are left out from the per pixel umbra *and* penumbra calculation step.

3.2.3.3. Dividing the Edge Shadows

Each shadow has to be divided into simpler 3D geometry, such as pyramids and prisms. This way computing a point's shade should become easy.

Determining the light source visibility is easy at the front plane and the back plane of an edge shadow. From the front plane, the whole light source is visible (see Figure 3-3). From the back plane, the whole light source is invisible. We can now linearly interpolate the area between the front and back faces to guess the light source visibility at different points in the central area (see 0 below).

The rest of the outline edges around the corner areas are colored in a way to assist combining the shadow contributions from each *part* (described in the following sections). The shade at any point inside the edge shadow is then linearly interpolated using the edge colors. The silhouette edge colors on the corner depends on the corner type – whether it is convex, concave or straight.

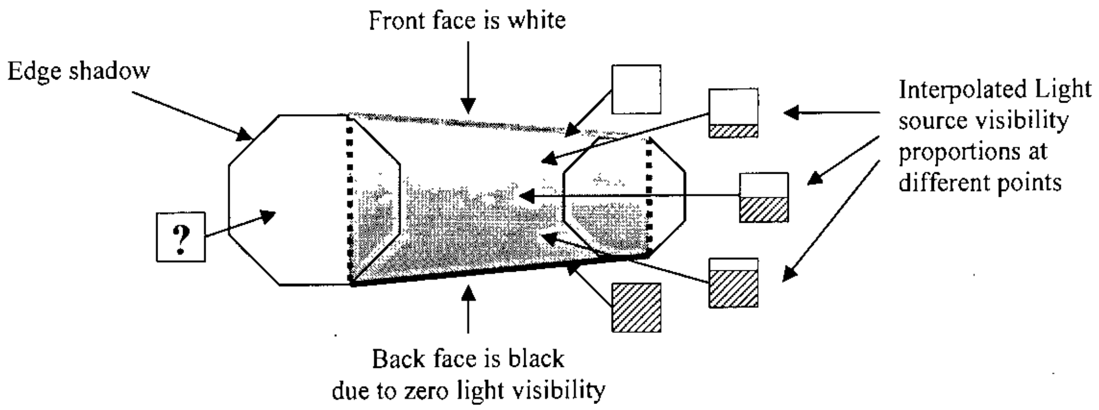

Front face is white
Edge shadow
Interpolated Light source visibility proportions at different points
?
Back face is black due to zero light visibility

Figure 3-9. Light source visibility at the mid area of an edge shadow.

To assist the interpolation operation at the corner areas, we divide the corner areas into simpler 3D geometry.

The mid section is easily colored. The back face is black (i.e., light source is totally invisible from any point) and the front face is white (i.e., light source is fully visible from all points).

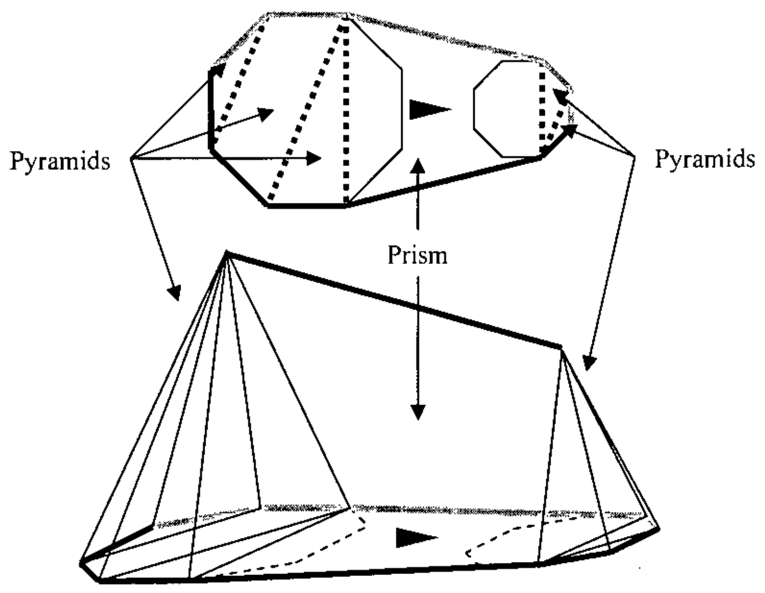


Figure 3-10. Light Corner areas are divided for easy interpolation.

Given an edge shadow, we need to find the proportion of light visible from a point inside it. Rather than producing a prefect shadow, we can simply interpolate the visibility factor.

3.2.3.4. Convex Corners

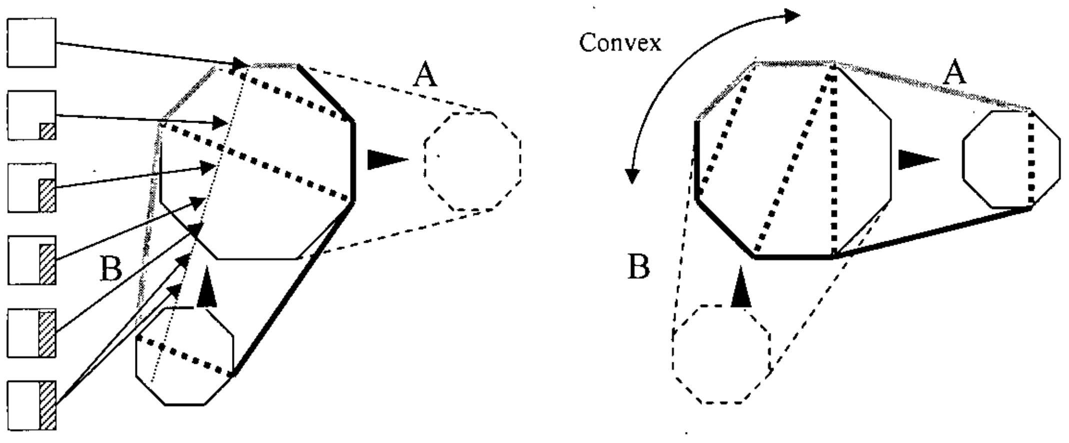


Figure 3-11. Convex corner division and Light source visibility.

Since the direction of shading with the neighboring edge is perpendicular at the corner area, to make smooth transition from one edge shadow to the next one the outline edges that fall *within* the neighboring edge shadow need to be black. The outer outline edges need to be white, as before, due to full light source visibility.

Consider the combination in Figure 3-11 above. Edge shadow B, having an *upward* direction contributes a light visibility proportion like ▨, without loss of generality. Now, as we move upward and enter into edge shadow A, this new interaction should not produce any drastic change in visibility proportion computation so that the shadow remains smooth enough. If edge shadow A contributes starting from zero gradually up to 1, the visibility proportion will gradually increase from ▨, through ▨, ▨, ▨, ..., ▨, to □. Hence we can generate a combined smooth shade as we enter into a neighboring edge shadow.

A similar transition can be found if we move from A to B. Now we have found how to shade the convex corner areas smoothly.

After the outline is colored properly, the corner area is divided such that opposite edges/points in each simpler shape have different colors. Otherwise the interpolation will break down.

3.2.3.5. Concave Corners

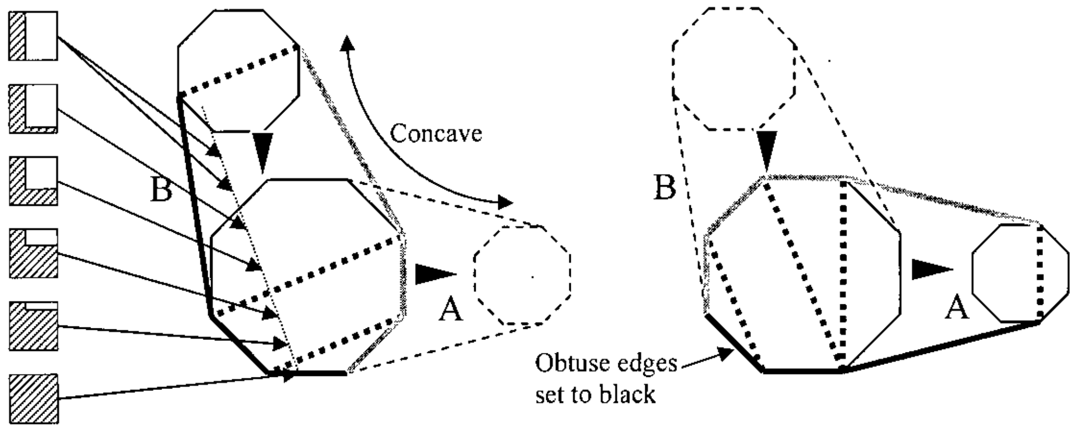


Figure 3-12. Concave corner division and Light source visibility.

Since the direction of shading with the neighboring edge is perpendicular, to make smooth transition from one edge shadow to the next one, the outline edges that fall *within* the neighboring edge shadow need to be white. The *obtuse* edges are automatically set to black, since they represent the back plane.

Consider the combination above. Say, edge shadow B contributes a ◧ light source visibility, without loss of generality. Now, as one moves downward and enters into the edge shadow A, this new interaction should not produce any *drastic* change in light source visibility proportion value. If the edge shadow A contributes starting from full visibility and gradually down to zero visibility, the visibility proportion will gradually decrease from ◧, through ▨, ▨, ▨, …, ▨, to ▨.

Clearly, this is what happens, since on the edge shadow A, the more downward one goes the lesser proportion of the light should be visible, as is evident from the fact that one is approaching the back plane.

Again, at the intersection of the two white *outer* outline edges, the visibility proportion is approaches 1. Hence this division of the edge shadow produces smooth shading.

3.2.3.6. Straight Corners

Straight corners are actually a joining area of two equi-directional edge shadows. Since we are approximating the directions, it may be the case that actually the corner is convex or concave, but for the sake of shadow calculations, assuming the joint is straight is enough.

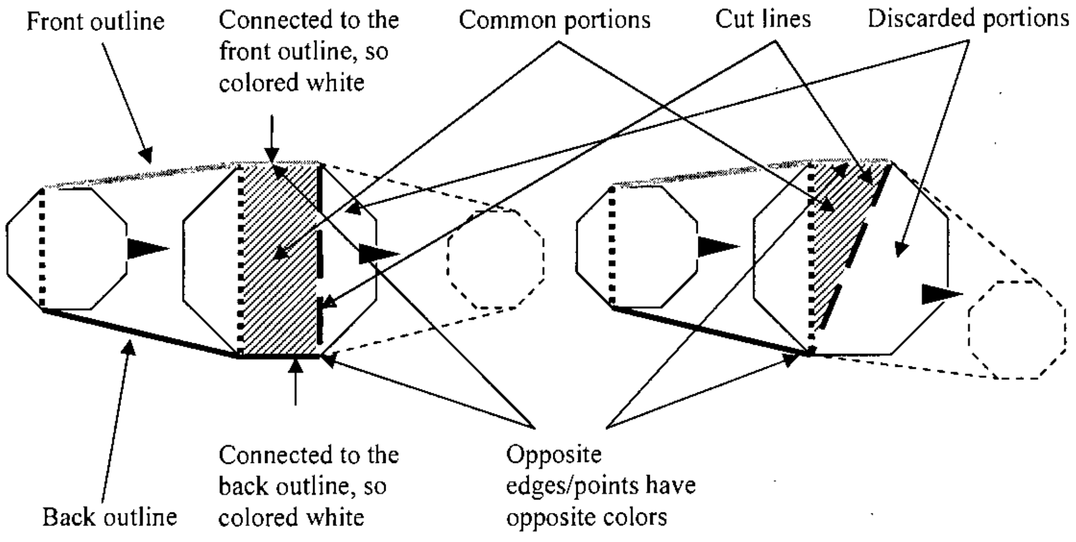

Front outline
Connected to the front outline, so colored white
Common portions
Cut lines
Discarded portions
Back outline
Connected to the back outline, so colored white
Opposite edges/points have opposite colors

Figure 3-13. Straight corner with common vertices.

In the figure above, note the shaded area in both cases. This area is a part of corners of both the edge shadows. The common portion of the corners will agree at how to shade a point since the bordering edges will have identical colors on both instances of the common portion.

Hence, the edge shadows are effectively cut off at straight corners. The common portions are kept. The dangling portion protruding into the other edge shadows mid area is cut off.

Note that there may not be any common portion at all. If one edge shadows corner area completely protrudes into the others mid area, the whole corner area will be cut off. This means both of them will be sharing a *cut line* only. However, both the edge shadows must agree at this cut line too, since the cut line has identical colors at both edge shadows.

The corner areas usually help us *smooth out* the transition from one edge shadow to the next. However, in this case the transition is automatically smooth (since both edge shadows agree at the cut line or the common portion).

The remaining outline edges of the corner section are colored white if connected to the front outline, and colored black, if connected to the back outline.

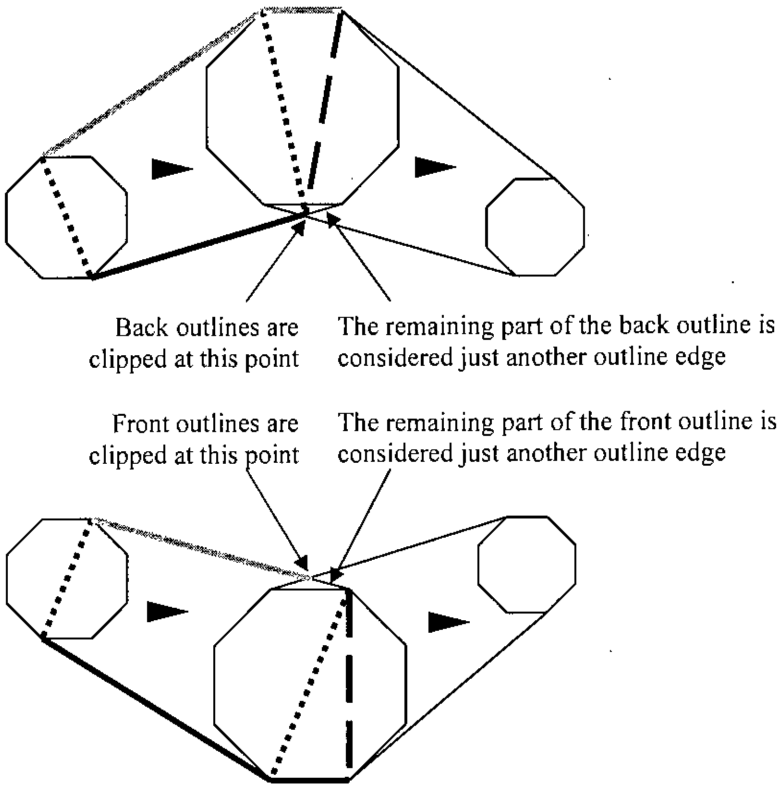

Back outlines are clipped at this point
The remaining part of the back outline is considered just another outline edge
Front outlines are clipped at this point
The remaining part of the front outline is considered just another outline edge

Figure 3-14. Straight corner without common vertices.

However, if actually the corner is convex, there may not be any common vertex. In these cases, the black back outline edge intrudes into the neighboring edge shadow, disrupting the smooth shading. To prevent this, the back edges are clipped with respect to one another and broken into two parts. The *left* part continue to be the back plane, but the other part is treated as just another outline edge. See Figure 3-14 above, now we do have a common vertex so that we can proceed as discussed before.

Similar scenario occurs if the edge shadows are slightly concave. As before, the front edges are clipped with respect to one another.

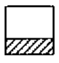
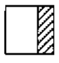
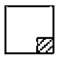
3.2.4. Combining the Shadow Contributions

If a point falls within a single edge shadow, the shade of that point can be uniquely determined. However, if more than one shadow edges are involved, these contributions need to be combined.

In the following sections, we first describe how to combine contributions from two adjacent edge shadows. After that, we will describe more complex situations

3.2.4.1. Collaborative Edge Shadow Contribution

The adjacent edges are collaborative. We call them *collaborative* because each of the participating collaborative edges gives parts of the same truth.

Consider two neighboring edges with a common convex corner. If a point within the corner needs to be shaded, both of these two edge shadows have to be interpolated. However, one of these replies with a  visibility proportion and the other replies with a  visibility proportion. Clearly, the total visibility proportion needs to be computed by *OR*-ing the visibility proportions  (see Figure 3-7).

Obviously this definition seems to be not working for concave contribution combinations. We will be presenting an even more elaborate definition of the term *collaborative* in upcoming sections.

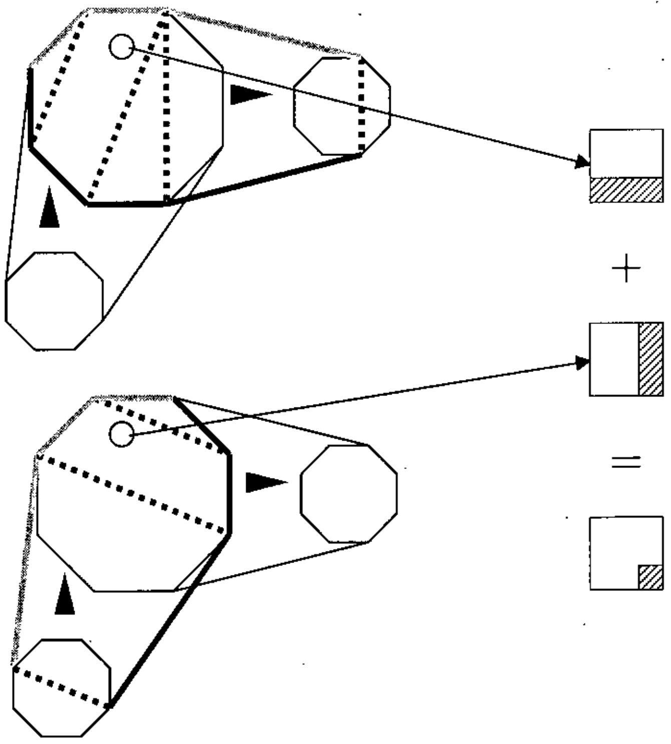


Figure 3-15. Adding up contributions from collaborative edge shadows.

3.2.4.2. Competing Edge Shadow Contribution

Edge shadows from different rings compete. They give different statements about the visibility of the light source; all of them cannot be simultaneously true. First consider two competing edge shadows.

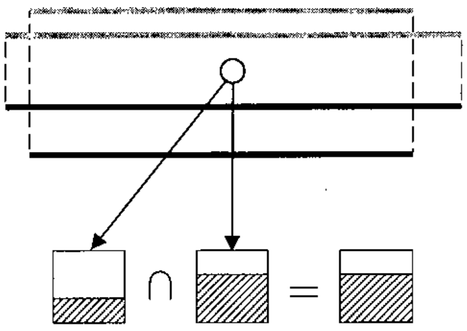


Figure 3-16. Adding up contributions from competitive edge shadows.

Clearly, taking the minimum smoothly mixes up the two contributions. To prove this approach is correct, we need to simulate how the light source disappears.

Note that, the edge shadow with the larger width must belong to a penumbra edge shadow. This is because the penumbra silhouette must be closer to the light source, so it should have a larger shadow. The other one is an umbra edge shadow.

As we move from outside to *inside* the penumbra region, the light source begins to disappear. In the beginning, we can see both silhouette edges. Clearly, at that point the penumbra silhouette edge is responsible for the disappearance of the light source (see Figure 3-18 below). At some point the penumbra silhouette edge is no longer visible. From that point the umbra silhouette takes over.

Intuitively, the light source visibility portions need to be *AND*-ed to get the resultant value. Consider a case when two competing contributions are claiming to have a light visibility of 50% of the light from the left, i.e., ▨□, and a light visibility of 50% of the light from the right, i.e., □▨, respectively. Now, since these two contributions do not belong to the same *ring*, one must be covering another's view. In the end, the resulting visibility is zero.

If the competing rings are of same type, they belong to different objects or different convex parts of the object – which never collaborate, since the idea of collaboration is that they both are telling parts of the same truth. In case of different objects, only one can be right at a time.

Similar phenomenon occurs at corners from different rings. Some examples of combining the competing contributions are given below.

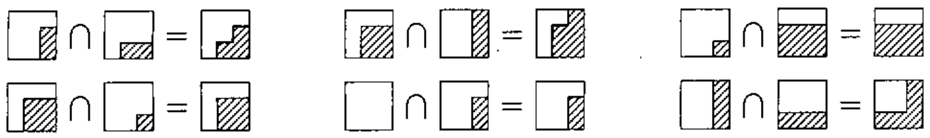


Figure 3-17. Competing Interactions at the corners.

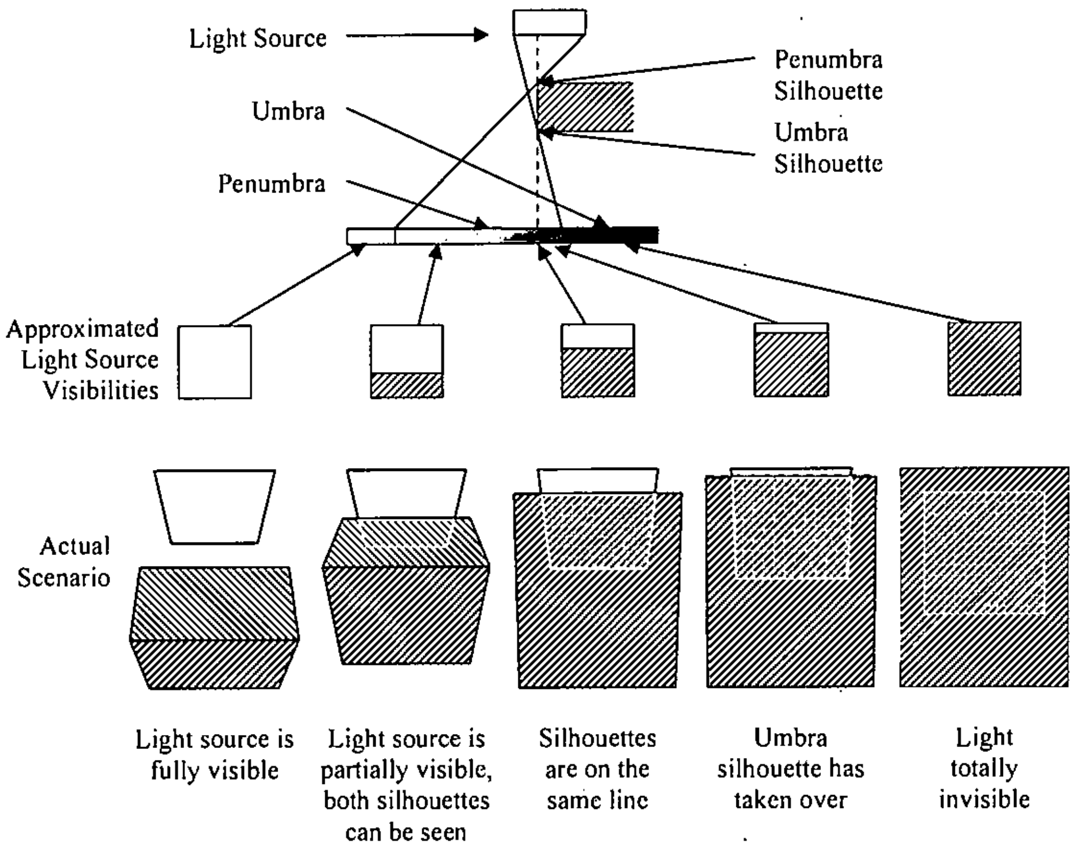

Light Source
Umbra
Penumbra
Penumbra Silhouette
Umbra Silhouette
Approximated Light Source Visibilities
Actual Scenario
Light source is fully visible
Light source is partially visible, both silhouettes can be seen
Silhouettes are on the same line
Umbra silhouette has taken over
Light totally invisible

Figure 3-18. As we move *inside* the shadow, the light gradually disappears.

3.2.4.3. Combining More than Two Contributions

More than two edge shadows can participate too. In concave objects and in small convex objects, edge shadows not adjacent to one another can overlap. There may be cases when three, four or more edge shadows of the same edge shadow ring overlaps.

The proposed algorithm assumes the silhouettes form rings. We first describe how the edge shadows from one ring combine their contributions. In the next section we are going to present how contributions from multiple rings can be combined.

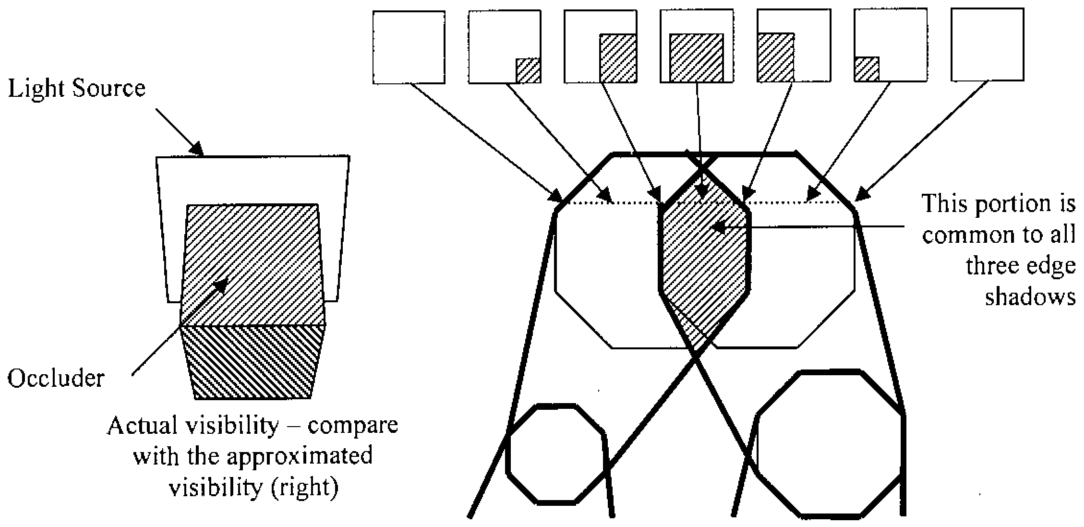

Light Source
Occluder
Actual visibility – compare with the approximated visibility (right)
This portion is common to all three edge shadows

Figure 3-19. More than two edge shadows may overlap.

Not all edge shadows of one ring contribute to the same point. For the ones that contribute, we draw connected visibility boundaries. No portion of the light source that falls on the right side of this *visibility line* can be seen from that point.

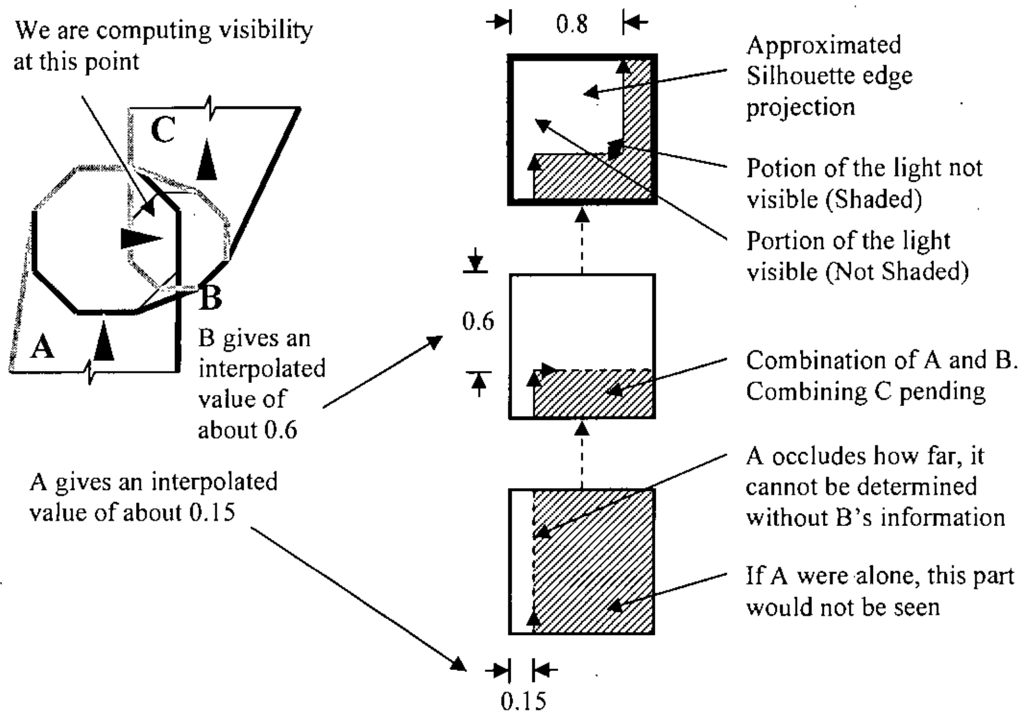

We are computing visibility at this point
C
B
A
B gives an interpolated value of about 0.6
A gives an interpolated value of about 0.15
0.8
0.6
0.15
Approximated Silhouette edge projection
Potion of the light not visible (Shaded)
Portion of the light visible (Not Shaded)
Combination of A and B. Combining C pending
A occludes how far, it cannot be determined without B's information
If A were alone, this part would not be seen

Figure 3-20. Computing light source visibility proportion using visibility lines.

We start with a full visibility: □. As we move along the edge shadows, the visibility line is being drawn depending on the visibility proportion computed from each edge shadow.

In Figure 3-20, consider the first *visibility curve*. This curve is an approximation of light source visibility at the point on the picture on the left where three edge shadows are interacting. To find the actual light source visibility, we need to project the corresponding silhouette edges onto the light source to determine the occluded portion. Instead, in our algorithm, we are approximating this by using the *direction* and *visibility value* parameters. The visibility value is the interpolated shading value at any point. There might be four possible directions of a shadow after 90° approximation. Using this information from all of the interacting edge shadows, we can approximate the projection of the silhouette edges.

Note that, C appears to be *not* collaborating with A and B. However, A and B did not commit about any visibility proportion. Instead, A and B produced a visibility curve that would be true if only two of them interacted.

An ever more complicated example is given below.

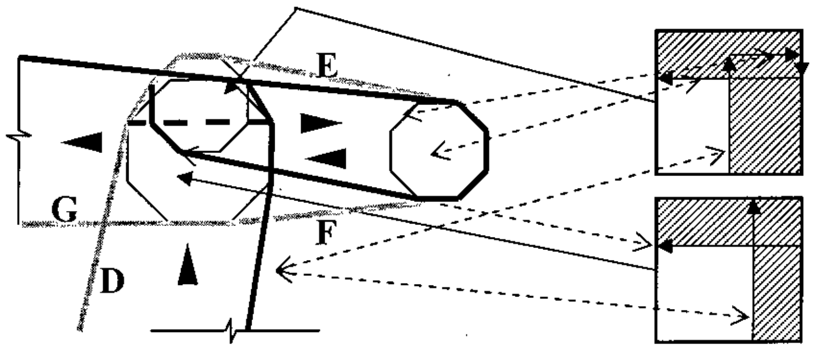


Figure 3-21. Computing light source visibility proportion using visibility lines.

3.2.4.4. Combining Contributions from Different Rings

Different rings are competitive. So the visibility portions need to AND-ed, as discussed in 3.2.4.2. However, due to the complex shape of the light source visibility, finding the intersection analytically is very tricky. Instead, we have used a numerical solution.

We have a benefit however; all the edges are parallel to X- or Y-axis. This enables us to utilize *slicing*. Slicing is quantization with unequal steps. We quantize at x and y values available from the visibility curves, then populate a 2D buffer using visibility information and finally compute total visibility.

The difficulty with this approach is that the number of different x and y values can be very high for complicated occluders with many rings and many silhouettes with a high degree of overlapping. The possible way out is: we can set the highest number of quantization levels, say 8, in both X and Y dimensions. That way, we will be populating an 8x8 grid, and we need to test for 64 points only.

The problem with this approach is that we need to be satisfied with only 64 different shades. In addition to that, shadow areas with no overlap will have only 8 different shades.

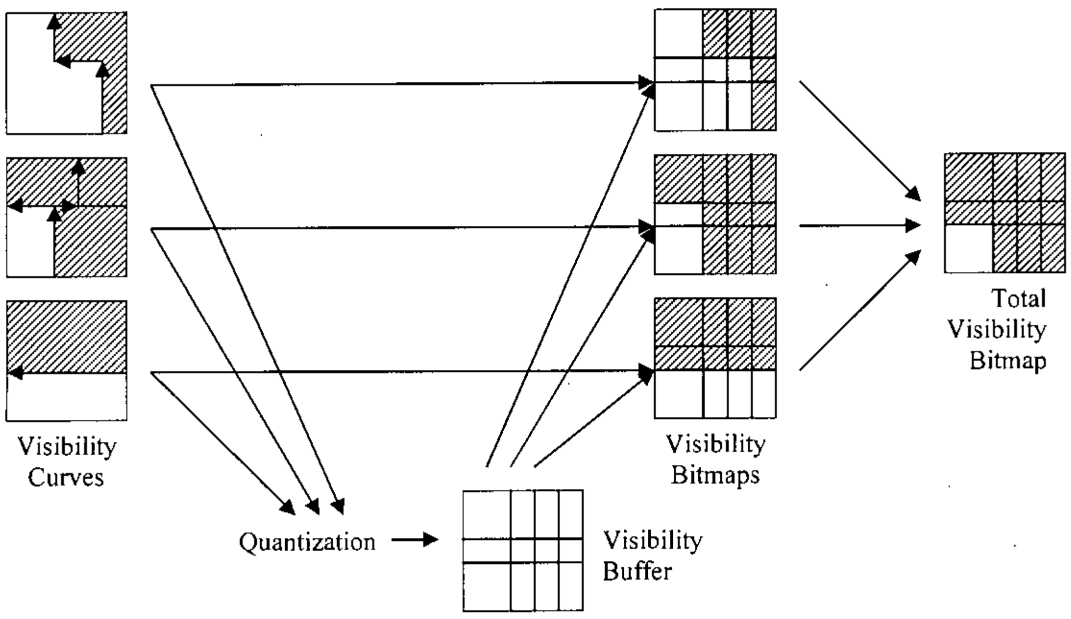


Figure 3-22. Slicing to combine visibility curves.

Instead of doing a variable quantization, we could do a fixed quantization; but then we would lose two benefits:

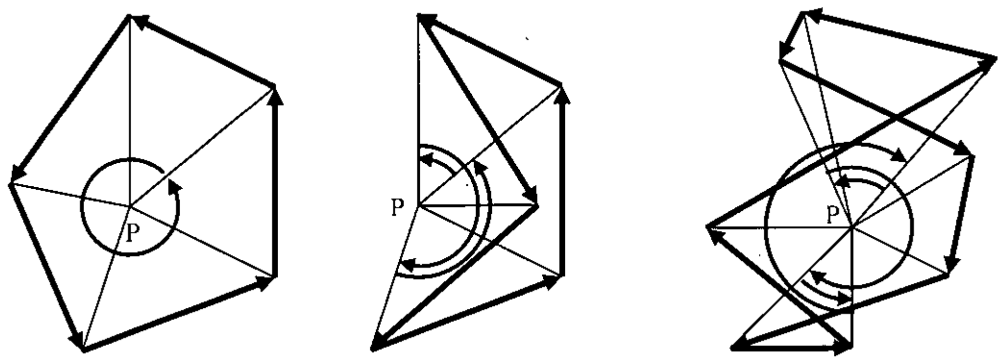
1. We would have to work 64 times as much even if no overlap occurs at a point.
2. In case all different x and y values are different but falls in only a few quantization steps, we lose accuracy.

On the other hand, fixed level quantization is fast – since we do not need to search for all different values and group them up for optimality.

3.2.5. Computing the Hard Shadow

If a point falls behind *all* back planes of *any* umbra ring, the point is in hard shadow. However, computing this in 3D requires solving a difficult plane intersection problem in three dimensions. To complicate this even more, a shadow may not have *any* hard shadow portion (i.e., from all points on the receiver a fraction of the light source is visible.

This problem can be easily solved in 2D by angle sweeping. Taking the intersections of the backplanes with $z = z_p$ plane, the problem degenerates into a 2D problem. This approach even works with self intersecting polygons, as shown in Figure 3-23 below.



Total Angle = 2π
P lies *inside* the polygon.

Total Angle = 0
P lies *outside* the polygon.

Total Angle = -2π
P lies *outside* the polygon (and *inside* the *inverted* portion).

Figure 3-23. Point-is-inside check in 2D, for different shapes.

Consider the case when there is no hard shadow. In that case, all the edge shadows must overlap each other, making the whole backplane curve oriented clockwise. No point will have an angular sum of 2π now.

3.3. Approximations in the Algorithm

As discussed before, the true shadow generation algorithm is a difficult problem. The proposed algorithm approximates the true solution at a number of points. These approximations are detailed below.

However, interestingly, the benefit with soft shadows is: we do not always have a precise idea how the soft part of the shadows should look. We need not incorporate edge-like sharp discontinuities in the shadow. Throughout the algorithm, all we are doing is producing and combining interpolated visibility proportion in such a way as two neighboring pixels do not disagree at a large extent. Some observations in this regard are presented in Chapter 4.

Firstly, the shading is assumed linear. However, this is only true with linear lights and also with rectangular lights when the visibility is considered along a direction parallel to one of its edges.

In most other cases, the visibility does not drop linearly.

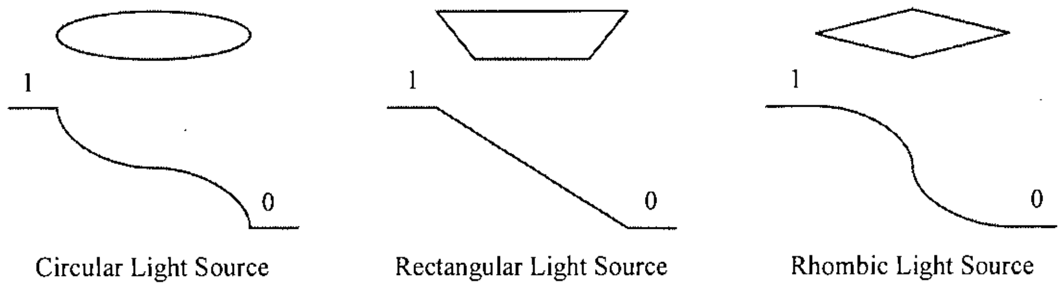


Figure 3-24. Light visibility function is not linear.

Secondly, true light-visibility proportion is not used; instead we are interpolating the value. True edge shadow directions are not used either; we are using one of four possible direction values parallel to the X- or Y-axis.

Thirdly, we are computing with silhouettes only. That means we are discarding most of the information on the objects. The intermediate edges between the two silhouette edges are ignored, essentially making the object look like a prism. However, this modified object cannot produce exactly the same shadow.

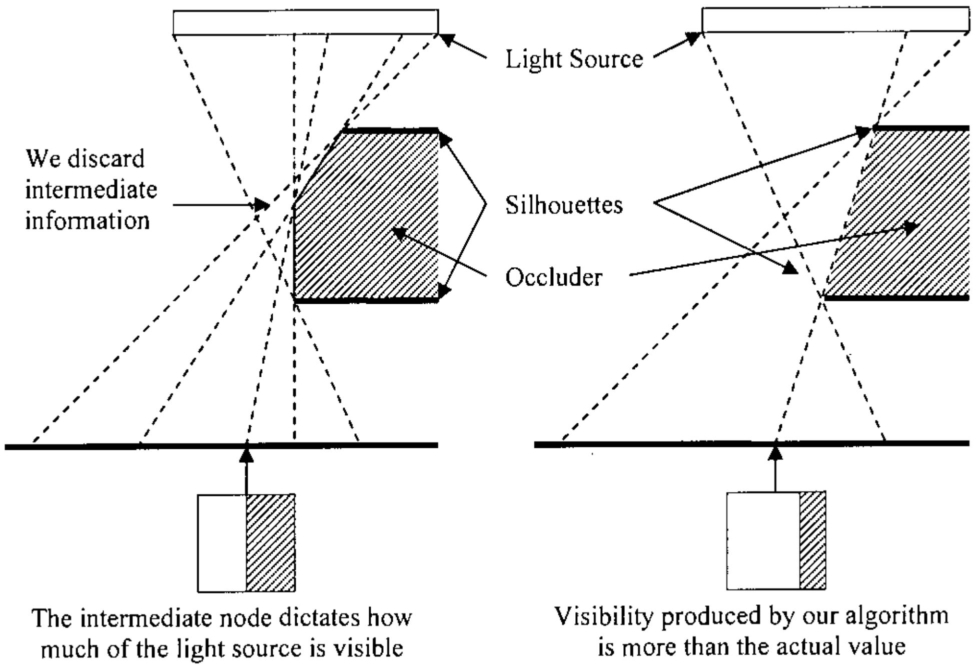

Light Source
We discard intermediate information
Silhouettes
Occluder
The intermediate node dictates how much of the light source is visible
Visibility produced by our algorithm is more than the actual value

Figure 3-25. Light visibility deviation.

More specifically, an intermediate protruding edge between the umbra and penumbra silhouette edges can *take over* before the umbra silhouette can take over finally. Obviously, both shadows will not be the same; even with linear lights.

3.4. Complexity Analysis

3.4.1. Generating the Edge Shadows

We consider the following to do complexity analysis of our algorithm:

f = number of facets;

n = number of vertices $\approx f$; and

L = number of vertices on the shape of the light source.

We assume the adjacency between neighboring facets will be the inputs to the system. With any arbitrary list of facets, it would require us $O(n^2)$ time complexity to compute the adjacency. This costly operation can be easily resolved at scenario building time.

At first, our algorithm will compute the silhouettes. If temporal coherence is not utilized, we need to traverse through all the facets. For each facet, for each neighbor, if the neighboring facet's lighting status (fully lit, partially lit, in shadow) is different from this facet's lighting status, clearly the shared edge is a silhouette edge. For the *partially lit* test, we need to consider all vertices on the light source. Thus, the time complexity for computing the silhouettes: $O(nL)$.

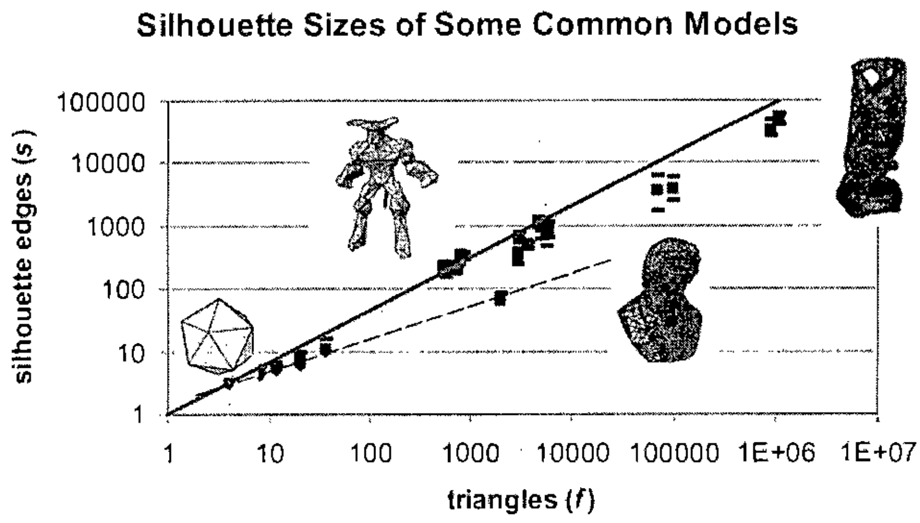


Figure 3-26. Log-log graph of f vs s. The dashed line is the $f = s^{1/2}$ trend for convex models and the thick line is the $f = 0.7s^{0.8}$ trend observed over all data. This is extracted from [34].

Although the maximum number of silhouette edge can be as large as $O(n)$, according to a study [34] over currently available models, the average number of silhouette edges varies as approximately $O(n^{0.8})$ for complex models and $O(\sqrt{n})$ for convex objects (see Figure 3-26). It is reported in [35] that $O(\sqrt{n})$ is quite often the case. However, for our complexity analysis, we are assuming the number of silhouette edges varies as $O(n^{0.8})$.

We will be doing our search through the facets such that we use the adjacency information to keep connecting the silhouette edges. When all the silhouette edges are found, there will be rings and even graphs. The even graphs can be split into rings in linear time with respect to number of silhouette edges.

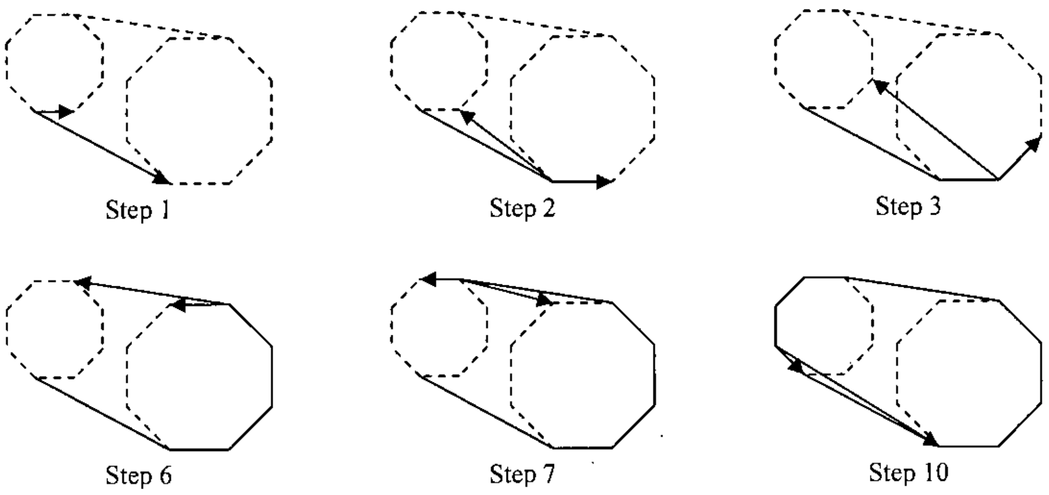


Figure 3-27. Computing the edge shadow with *L* = 8.

Now, for each silhouette edge $E = \langle a, b \rangle$, we need to produce an edge shadow. We start with projecting *a*. Finding the vertex on the light source that projects *a* to produce the *left* end of the back outline edge requires $O(L)$ time. Then at each step, either we advance to the next light vertex, or to *b*, always maintaining a convex outline. At each step, we have two options to choose from. Either the current light vertex projects the *other* silhouette edge node (*b*, if now we are at *a*, vice versa); or the next light vertex projects the current silhouette edge node. A few steps are shown in Figure 3-27.

After finding the outline, the edge shadow must be divided. To assist fast division, the edge shadows have to be sorted for fast searching. The division procedure will run linear number of iterations ($O(L)$). At each iteration it will check whether an outline edge is shared by a neighboring edge shadow's outline ($O(\lg L)$). Thus the following complexities could be easily derived:

Computing an edge shadow: $O(2L+2) \equiv O(\mathrm{L})$
Sorting outline for fast searching: $O(L \lg L)$
Dividing the edge shadow: $O(L \lg L)$
Average Total Complexity = Time complexity of Silhouette Computation + Time complexity of $O(n^{0.8})$ edge shadow computation = $O(nL + n^{0.8} L \lg L)$

Since usually n is in thousands and $L < 100$, Average Total Complexity $\approx O(nL)$

3.4.2. Computing Per Pixel Shadow

To determine a point's illumination, we need to combine the shadow contributions from edge shadows of all rings. In the worst case, all of the edge shadows will be searched, and the match will be found in the last division searched. Worst case complexity of determining contribution from each edge shadow: $O(n^{0.8} L)$. After the visibility curve is prepared, a bounded slicing algorithm is run, with a complexity approximately at $O(n^{0.8} L)$ in the worst case. Hence the total complexity in this step is $O(n^{0.8} L)$.

This per pixel shadow determination step is the bottleneck of this algorithm. In particular, we do need to process all of the pixels to be drawn. But, the total processing time will increase if the edge shadows are too large. If edge shadows are large, more edge shadows will interact at each pixel, resulting in a more complicated visibility curve and higher time requirement for visibility proportion determination.

Chapter 4. Implementation and Observations

We have implemented the algorithm described in the previous chapter. The implementation details are given here.

We will be briefly describing different aspects of our implementation in the next section. Then several observations of generated shadow will be presented followed by a discussion on the artifacts produced. Our algorithm will be compared with two well known real time soft shadow algorithms, namely, Penumbra Wedges and Smoothies. We will be comparing the outputs in terms of quality.

4.1. Implementation

The algorithm is implemented in Visual Studio 6.0. The written program is a console application. We worked on an Intel Centrino 1.5GHz machine with a graphics card with no advanced capabilities. The operating system is Windows XP SP2.

4.1.1. API Related to This Implementation

OpenGL and GLUT was used for Graphics manipulation.

Standard Template Library (STL) is used for flexible data container support. The algorithm maintains several lists of structures, which were held using vectors when appropriate. To minimize *thrashing* like behavior, a minimum size of 100 is set to these containers during initialization.

4.1.2. Data Structures Used

The occluders are represented by polygonal meshes. A simple, efficient data structure is used that can answer to the following queries in constant time:

1. Given a triangular facet, find the neighbors adjacent to a given edge.
2. Given two triangular facets, find the shared edge, if any.

4.1.3. Hardware Implementation

This algorithm is perfect for hardware implementation. This is because all shadow decisions are made locally, on a per pixel basis. Today's GPU pipeline supports vertex and fragment shading.

This algorithm can be implemented using Cg on NVIDIA GeForce FX series or compatible GPU based hardware.

Hardware acceleration can improve performance in several folds:

1. The algorithm can be run for only the visible pixels/vertices. The first pass of rendering can be run only for populating the depth buffer, simply by disabling the pixel/vertex shader. In the next pass, the shadow algorithm is run, but only for the pixels which have equal depth value as stored in the depth buffer.
2. Costly functions, such as sin and cos, can be approximated using the texturing capabilities of the GPU instructions. Furthermore, the built in hardware interpolation capabilities can be easily utilized to interpolate the function to arbitrary parameter values with acceptable precision.
3. The basic data type of GPUs is float4. Point or color can be manipulated in a single instruction, all dimensions in parallel.
4. Several graphics computation functionalities are built in the hardware of GPUs. For example, the linear interpolation instruction `lerp(a,b,t)`, equivalent to $a + (b - a) \times t$, can perform the operation in a single cycle. Linear interpolation is heavily used throughout the algorithm.

We did not do a hardware implementation because such programmable graphics hardware was not available at our laboratories.

4.2. Observation of Generated Shadow

4.2.1. Snapshot Set 1

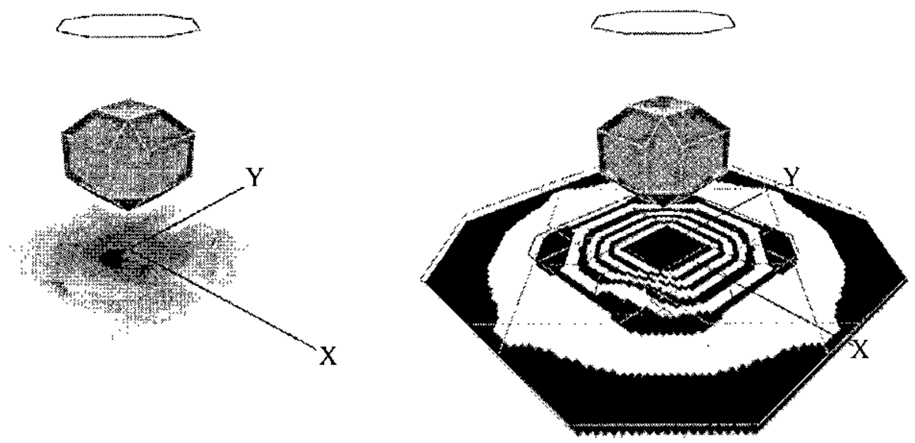


Figure 4-1. The general output.

In the left, the generated shadow is presented. The image on the right shows the contours with almost same intensity value.

The outermost contour contains pixels with intensity>0.95. The central contour contains pixels with intensity <=0.05. All remaining contours contain pixels with intensity value lying within 0.1 boundaries (0.05, 0.15], (0.15, 0.25], …, (0.85, 0.95].

The contours show whether the generated shadow is *good enough*. Since visual acceptability is the ultimate factor, distorted shadow contours may still produce quite pleasing shadows.

In the shadows produced here, an ambient intensity of 0.3 is assumed if not stated otherwise.

4.2.2. Snapshot Set 2

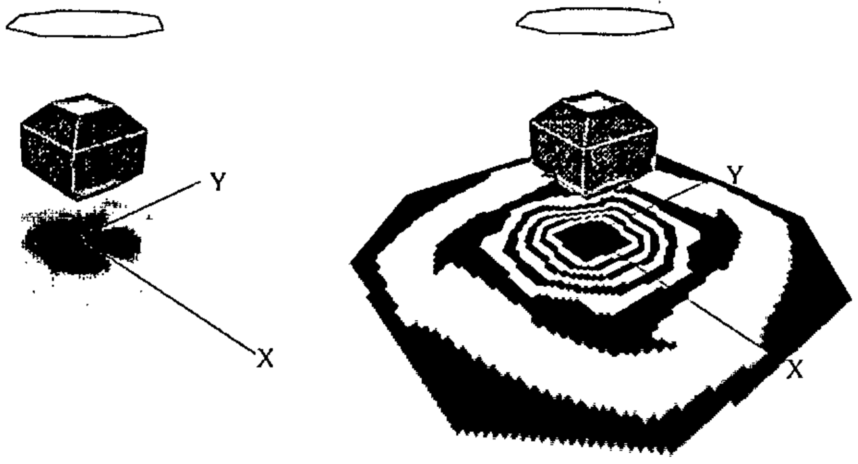


Figure 4-2. Sharp angle in the contours.

This snapshot has the same object-light pair, but the light source slightly rotated. The image on the right shows the sharp angles in the contour. Fortunately this discrepancy is not visible to naked eye (left).

The sequence of shadow contours generated for light rotation is provided below:

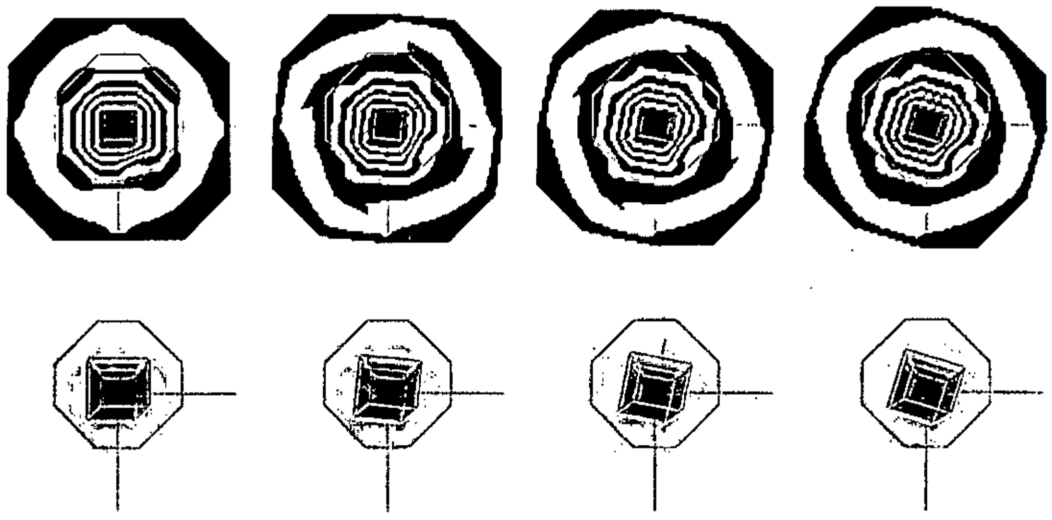


Figure 4-3. Appearance and disappearance of sharp angle in the contours.

4.2.3. Snapshot Set 3

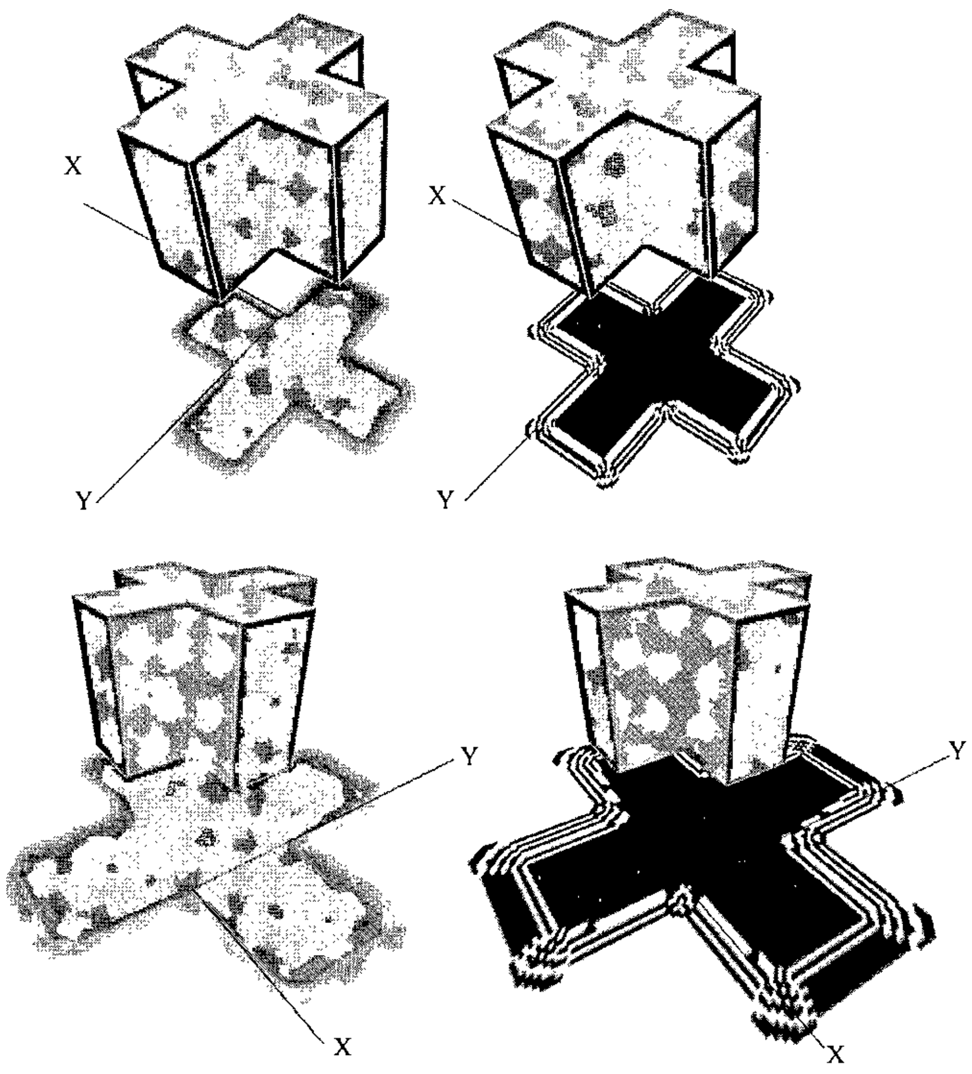


Figure 4-4. A simple Concave Object *Up:* The light source is far from the occluder. *Down:* The light source is near to the occluder.

This snapshot shows a simple 3D concave object. The light source is right above the object. Note the curvature on the inside of the concave points.

4.2.4. Snapshot Set 4

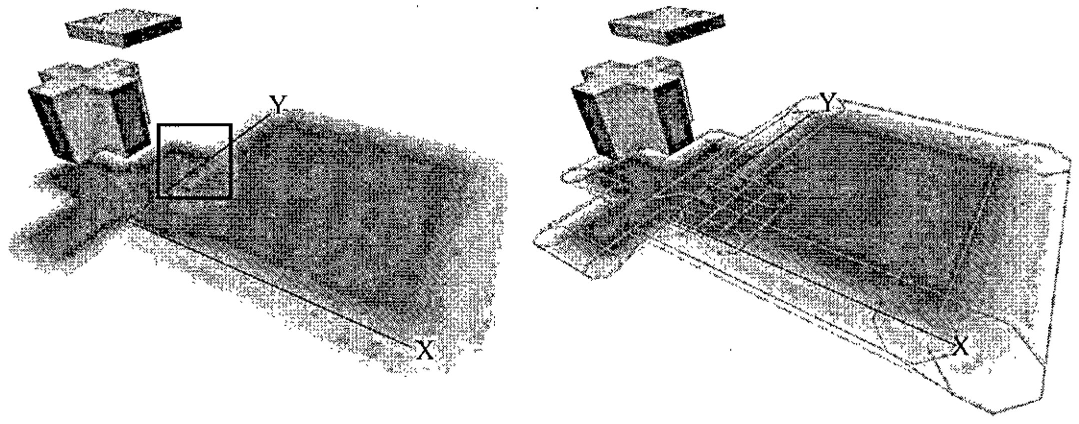


Figure 4-5. A combination of two objects.

In the images above, shadows from two different objects are rendered.

On the right snapshot, the complex interplay between the silhouette edges is shown.

Note the grayish slit in the above image. This appears because a thin part of the light source is visible from those points (see the figure below)

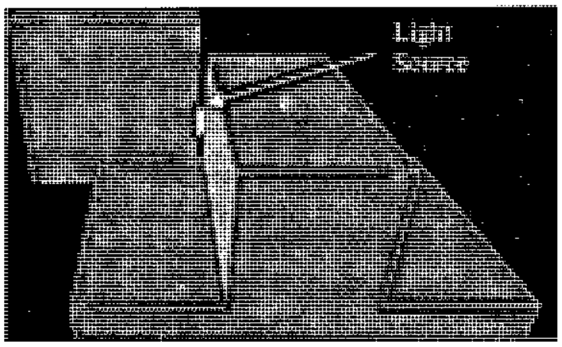
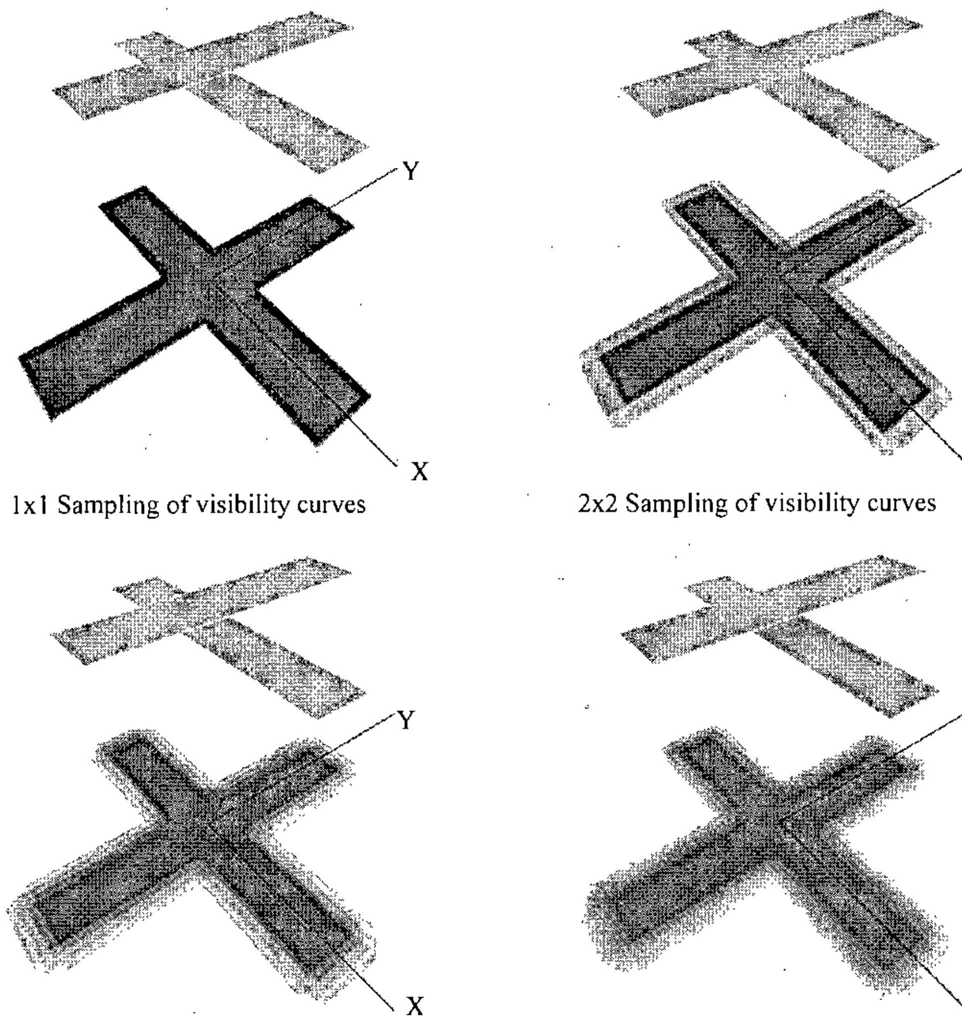


Figure 4-6. A thin portion of the light source can be seen through a slit.

4.2.5. Snapshot Set 5



1x1 Sampling of visibility curves

2x2 Sampling of visibility curves

4x4 Sampling of visibility curves

8x8 Sampling of visibility curves

Figure 4-7. Numerical computation of visibility proportion from the visibility curves produces sampling artifacts.

Numerical computation of the visibility function provides an upper limit to the computation time. However, as the number of sample points decreases, the smooth penumbra appears stepped.

For only 1x1 sample point, the shadow degenerates into a hard shadow, as expected.

Numerical computation also puts unnecessary burden on points where a small number of edge shadows interact and so solving analytically is easy. This observation is illustrated in the next set of snapshots.

4.2.6. Snapshot Set 6

The following image pair show fixed sampling vs. variable sampling (slicing) in solving the problem of visibility proportion determination from visibility curves.

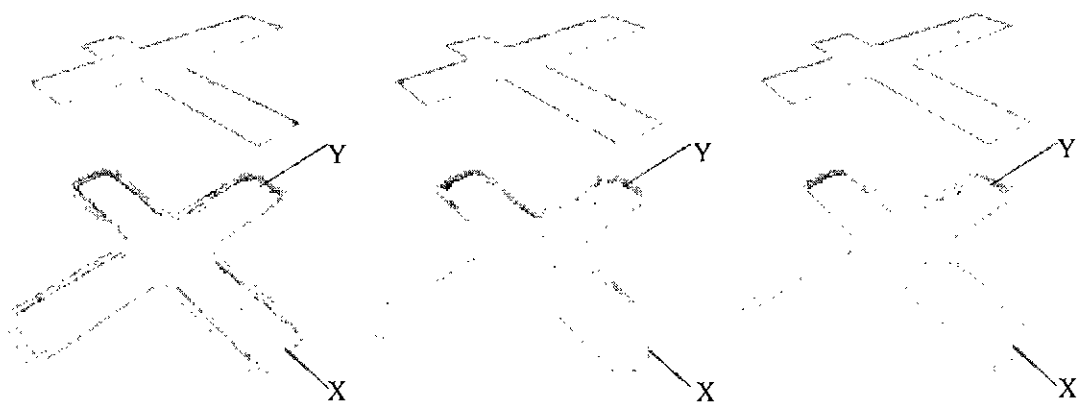


Figure 4-8. ***Left:*** **8x8 Sampling;** ***Middle:*** **16x16 Sampling;** ***Right:*** **Slicing.**

As can be seen in Figure 4-8, slicing produces much better result than sampling strategies. When the occluder geometry is not very complex and the light size is small, hence the edge shadows do not overlap much and slicing works better than sampling – since slicing will determine visibility as fewer number of sample points.

Sampling has its own benefits over slicing: as number of overlapped edge shadows increases at a pixel, the number and complexity of different visibility curves increases. Clearly, in the worst case, all different *x* and *y* coordinates of the edge curve lines will be different, resulting in a quadratic number of slices.

The result is obvious: combine the power of slicing and that of sampling. In our implementation, we considered two coordinate values *equal* is they differ by less than 0.01. Increasing this threshold introduces sampling artifacts within the shadow.

4.3. Artifacts

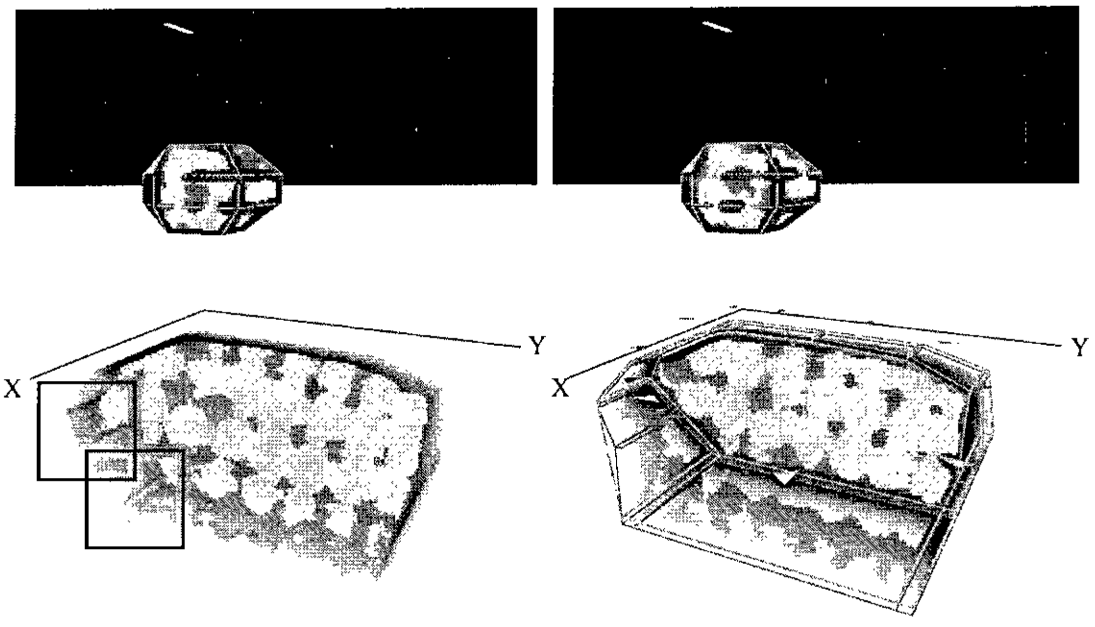


Figure 4-9. Artifact due to a thin light source.

The shadow is generated for a thin light source. The light source is shown as a white line in the sky right above the occluder.

Note the thin edge-like white artifact at the penumbra region. The edge shadows on either side of the white patch have convex perpendicular *directions*. These two portions collaborated at the common corner, as if they can see orthogonal portions of the light source.

The dark patch just above the thin white artifact has similar explanation. In that region, two edge shadows have concave orthogonal directions, so they are acting as if they are creating a concave corner, resulting in a much less total visibility proportion that it really is. The neighboring areas have equidirectional edge shadow contributions so these areas are behaving as expected.

The algorithm's inherent property is to avoid these edges. However, due to extreme thin shape of the light source, the corner areas have become very thin, leaving little or no space for a smooth change to occur, resulting in a visible artifact.

4.4. Comparison with Well-known Algorithms

In this section, we are going to compare the quality of shadow produced by our algorithm with a few well known soft shadow volume algorithms.

4.4.1. The algorithms

4.4.1.1. Penumbra Wedges Algorithm

Penumbra Wedges algorithm attempted to produce soft shadows by extending the hard shadow algorithm only (see 2.4.1).

Penumbra Wedges shows single silhouette artifact. The proposed algorithm does not show this.

4.4.1.2. Smoothies

Smoothies algorithm produces fake shadows. This algorithm simply adds a smooth area around the hard shadow (see 2.4.2). Shadows always contain a hard shadow part, which may not be the case always.

The proposed algorithm obviously produces better shadows.

4.4.1.3. Ray Tracing

Ray Tracing produces excellent shadow but it is not a real time algorithm. That is why we are not comparing the outputs here.

4.4.2. Testing Code

We have implemented a rudimentary version of all three algorithms for the comparison purpose. However, the discrepancies stated are fundamental to the algorithms and hence can be observed in this implementation as well.

Supporting images extracted from original publications are presented as appropriate.

4.4.3. Comparison

In the first comparison snapshots, the same object rendered using three different algorithms are given below. The snapshots are taken from light's point of view.

Our algorithm produces the right shadow. The other two suffer from single silhouette artifact shown below. Because of using only one silhouette, the outline of the shadow appears to be concave. But a convex object must have convex shadow.

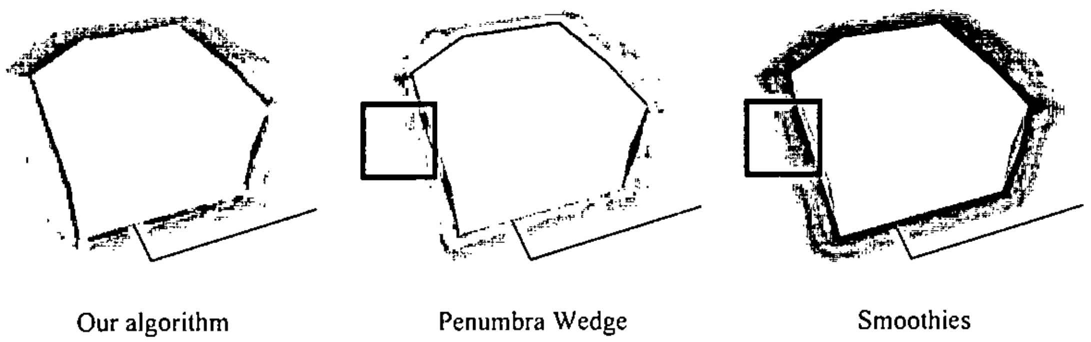


Figure 4-10. Single Silhouette Artifact: The concave portion of the outline of the shadow is indicated by a rectangular box.

The reason behind this is depicted in the following set of snapshots. Among the three points on the silhouette, A, B and C, B is below the imaginary line AC. That is why B has a smaller projection than both A and B, resulting in a concave corner. The same reason applies for Smoothies also.

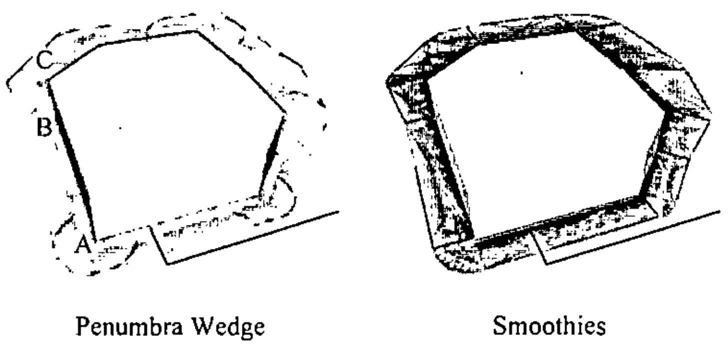
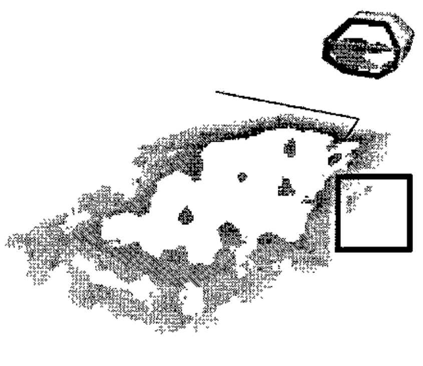
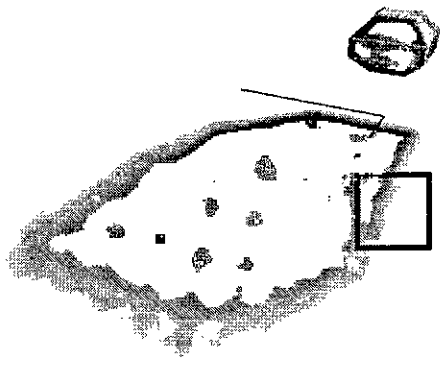


Figure 4-11. The wedges due to a single silhouette.

Figure 4-12 shows this same image, rendered from a different point of view.


Penumbra Wedge


Smoothies

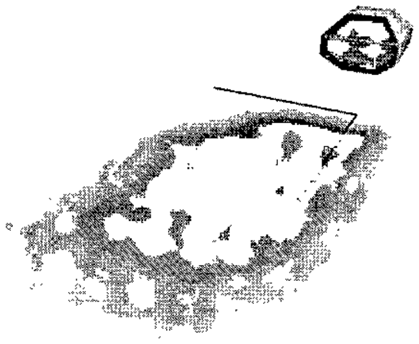

Our Algorithm

Figure 4-12. Smoothies produce a large umbra.

The above set of images produce another comparison with Smoothies. Smoothies always produce an umbra. That is why the umbra portion of the shadow produced will always be a lot bigger than that produced by Penumbra Wedges algorithm or our algorithm.

In the following images, we are demonstrating that our algorithm does not show the overlap approximation artifact, as shown by the Penumbra Wedges algorithm.

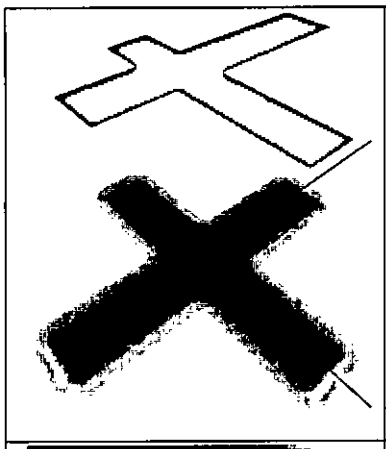
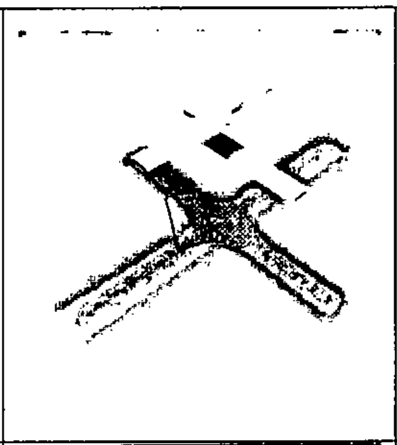
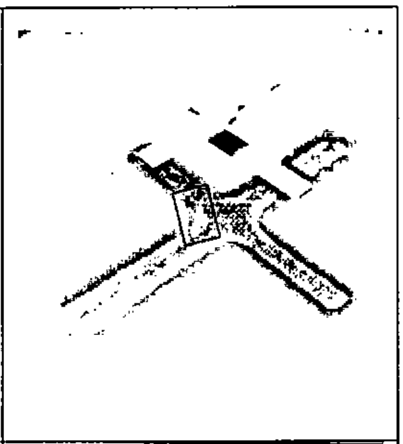
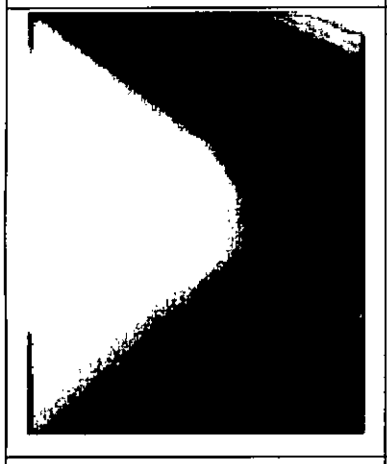
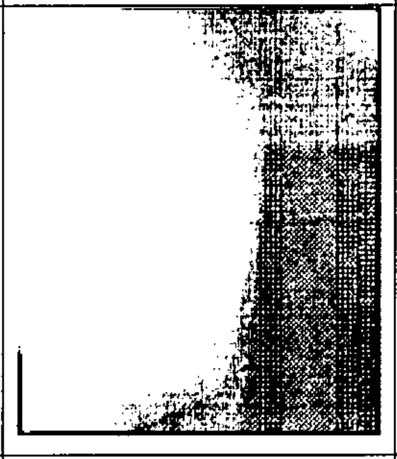
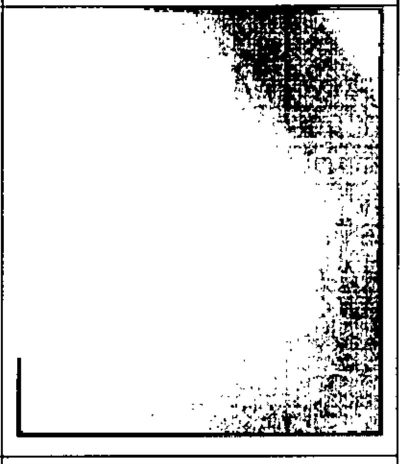
		
		
Our Algorithm	Penumbra Wedge (Extracted from [26])	1024 Sample (Extracted from [26])

Figure 4-13. Overlap Approximation.

Due to overlap approximation, a visible edge-like discontinuity is seen at the corner of the shadow generated by Penumbra Wedges. Our algorithm is free from this approximation and the output produced is quite similar to the output from a 1024 sample shadow.

Chapter 5. Conclusion

5.1. Conclusion

In this thesis, we have virtually reinvented the soft shadow volume algorithms. Instead of modifying an existing hard shadow algorithm, we built a totally new algorithm under the basic theme of shadow volume approach. In particular, our work has following original contributions:

- Double silhouette approach. We have shown that using only penumbra and umbra silhouettes is enough to produce excellent shadows.
- We have utilized linear interpolation to interpolate visibility of the light source. We have come up with a mechanism to smoothly combine interpolated visibility information from different silhouette edge; with little artifact.
- Relying only on silhouettes has made combining shadows of multiple objects trivial. Also from this perspective, self shadowing is no more a special case.

5.2. Future Works

It has been almost three decades since Crow published the first paper on shadow generation in computer graphics [13]. Both the algorithm and the hardware took over two decades to evolve. It was not until very recently that hard shadows have become an integral part of graphics consumer products, including computer games and CAD software.

5.2.1. Reducing the Complexity of Occluder Geometry

Occluder geometry is very important in shadow generation. But, when the occluder to receiver distance is large compared to the light to occluder distance, the shadow becomes so blurred out that the geometry of the occluder becomes almost irrelevant. A mechanism to compute hierarchical occluder geometry (as in [35]) can be devised to produce almost the same shadow. It should be noted that with a lower geometry complexity, the number of silhouette edges will drop dramatically, resulting in an increase in performance of our algorithm.

5.2.2. Supporting More General Objects

We believe we can incorporate support for more general objects, for example, objects with holes or self intersections.

To hide a silhouette ring's penumbra contribution to a point on the receiver when the light source is completely occluded by another object, the umbra regions must be identified correctly. In our algorithm, each point is first tested whether it lies within any umbra region produced by the penumbra rings. In case of objects with holes, a ring representing the silhouette edges inside the hole will fall completely within the umbra region for an outside ring. A kind of inclusion-exclusion principle may be utilized to solve this.

5.2.3. Shadow Map Version

Each variation of shadow volume algorithm has a similar shadow map version. The difficulty of our algorithm to readily come up with a shadow map version is that a single point is not being used to produce the shadow which could be used to populate the Z-buffer used in shadow map algorithms.

However, for regular shaped polygonal light sources, a central bending point can be computed through which all light rays pass while producing the penumbra outline.

5.2.4. GPU Implementation

Many object-based algorithms suffer from the fact that they need to compute the silhouette of the occluders, a costly step that can only be done on the CPU. Wyman and Hansen [32] report that computing the silhouette of a moderately complex occluder (5000 polygons) uses 10 ms in their implementation. If the next generation of graphics hardware would include the possibility to compute this silhouette entirely on the graphics card [33], object-based algorithms [20], [21], [26], [27], [32] and our algorithm would greatly benefit from the speedup.

5.2.5. Concave Light Sources

This thesis gives no idea on how to handle the concave light sources without breaking them up. We believe the concave light source can be accommodated in the algorithm through a small modification in edge shadow generation.

It should be noted that the shadow produced by concave light sources are almost like convex sources, except at the corners of the penumbra outline, where the light source's shape can be perceived. Things get even more complicated when concave occluders are considered.

References

[1] Wanger, L., “The effect of shadow quality on the perception of spatial relationships in computer generated imagery”, Computer Graphics (Interactive 3D Graphics 1992), Vol-25, No. 2, pp 39–42, 1992.

[2] Mamassian, P., Knill, D. C. and Kersten, D., “The perception of cast shadows”, Trends in Cognitive Sciences, Vol-2, No. 8, pp 288–295, 1998.

[3] Hubona, G. S., Wheeler, P. N., Shirah, G. W. and Brandt, M., “The role of object shadows in promoting 3D visualization”, ACM Transactions on Computer-Human Interaction, Vol-6, No. 3, pp 214–242, 1999.

[4] Kersten, D., Mamassian, P. and. Knill, D. C., “Moving cast shadows and the perception of relative depth”, Technical Report No. 6, Max-Planck-Institut fuer biologische Kybernetik, 1994.

[5] Kersten, D., Mamassian, P. and. Knill, D. C., “Moving cast shadows and the perception of relative depth”, Perception, Vol-26, No. 2, pp 171–192, 1997.

[6] Vinci, L. D., “Codex Urbinas”, 1490.

[7] Lambert, J. H., “Die freye Perspektive”, 1759.

[8] Akenine-Möller, T. and Haines, E., “Real-Time Rendering”. A K Peters Ltd, 2nd edition, 2002.

[9] Hasenfratz, J. M., Lapierre, M., Holzschuch, N. and Sillion, F., “A survey of real-time soft shadows algorithms”, Computer Graphics Forum, Vol-22, No. 4, pp 753–774, 2003.

[10] Everitt, C., Rege, A., and Cebenoyan, C., “Hardware shadow mapping”, http://developer.nvidia.com/object/hwshadowmap_paper.html.

[11] Segal, M., Korobkin, C., Widenfelt, R. V., Foran, J. and Haeberli, P., “Fast shadows and lighting effects using texture mapping”, Computer Graphics (SIGGRAPH 1992), Vol-26, No. 2, pp 249–252, 1992.

[12] Williams, L., “Casting curved shadows on curved surfaces”, Computer Graphics (SIGGRAPH 1978), Vol-12, No. 3, pp 270–274, 1978.

[13] Crow, F. C., “Shadow algorithms for computer graphics”, Proceedings of the International Conference on Computer Graphics and Interactive Techniques (ACM SIGGRAPH 1977), Vol-11, No. 3, pp 242–248, 1977.

[14] Heidmann, T., “Real shadows, real time”, Iris Universe, Vol-18, pp 23–31. Silicon Graphics Inc., 1991.

[15] Kilgard, M. J., “Improving shadows and reflections via the stencil buffer”, http://developer.nvidia.com/object/Stencil_Buffer_Tutorial.html, 1999.

[16] Everitt, C. and Kilgard, M. J., “Practical and robust stenciled shadow volumes for hardware-accelerated rendering”, Technical report March 2002, NVIDIA Cooperation, 2002.

104355

[17] Haines, E., "Soft planar shadows using plateaus", Journal of Graphics Tools, Vol-6, No. 1, pp 19–27, 2001.

[18] Brabec, S., and Seidel, H. P., "Single sample soft shadows using depth maps", Graphics Interface, 2002.

[19] Kirsch, F. and Doellner, J., "Real-time soft shadows using a single light sample", Journal of WSCG (Winter School on Computer Graphics 2003), Vol-11, No. 1, 2003.

[20] Chan, E. and Durand, F., "Rendering fake soft shadows with smoothies", Proceedings of the 14th Eurographics workshop on Rendering, pp 208–218, 2003.

[21] Assarsson, U., Dougherty, M., Mounier, M. and Akenine-Möller, T., "An optimized soft shadow volume algorithm with real-time performance", Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware, pp 33–40, 2003.

[22] Heidrich, W., Brabec, S. and Seidel, H. P., "Soft shadow maps for linear lights high-quality", Proceedings of Rendering Techniques 2000 (11th Eurographics Workshop on Rendering), pp 269–280, 2000.

[23] Herf, M., "Efficient generation of soft shadow textures", Technical Report CMU-CS-97-138, Carnegie Mellon University, 1997.

[24] Agrawala, M., Ramamoorthi, R., Heirich, A. and Moll, L., "Efficient image-based methods for rendering soft shadows", Computer Graphics (SIGGRAPH 2000), Annual Conference Series, pp 375–384. ACM SIGGRAPH, 2000.

[25] Soler, C. and Sillion, F. X., "Fast calculation of soft shadow textures using convolution", Computer Graphics (SIGGRAPH 1998), Annual Conference Series, pp 321–332. ACM SIGGRAPH, 1998.

[26] Assarsson, U. and Akenine-Möller, T., "A geometry-based soft shadow volume algorithm using graphics hardware", Proceedings of the International Conference on Computer Graphics and Interactive Techniques (ACM SIGGRAPH 2003), pp 511–520, 2003.

[27] Akenine-Möller, T. and Assarsson, U., "Approximate soft shadows on arbitrary surfaces using penumbra wedges", Proceedings of the 13th Eurographics workshop on Rendering, pp 297–306, 2002.

[28] ATI. Smartshader^{TM} technology white paper. http://www.ati.com/products/pdf/smartshader.pdf, 2001.

[29] Hart, E., "ARB Fragment Program: Fragment level programmability in OpenGL", http://www.ati.com/developer/gdc/GDC2003_OGL_ARBFragmentProgram.pdf, 2003.

[30] Everitt, C., "OpenGL ARB vertex program", http://developer.nvidia.com/docs/IO/8230/GDC2003_OGL_ARBVertexProgram.pdf, 2003.

[31] Akenine-Möller, T. and Assarsson, U., “On Shadow Volume Silhouettes”, Journal of Graphics Tools, 2003.

[32] Wyman, C. and Hansen, C., “Penumbra maps: Approximate soft shadows in real-time”, Rendering Techniques 2003 (14th Eurographics Symposium on Rendering). 2003.

[33] Brabec, S. and Seidel, H. P., “Shadow volumes on programmable graphics hardware”, Computer Graphics Forum (Eurographics 2003), Vol-25, No. 3, 2003.

[34] McGuire, M., “Observations on Silhouette Sizes”, Journal of Graphics Tools, Vol-9, No. 1, 2003.

[35] Sander, P. V., Gu, X., Gortler, S. J., Hoppe, H. and Snyder, J., ”Silhouette Clipping”, Proceedings of the 27th annual conference on Computer graphics and interactive techniques, pp 327—334, 2000.