Efficient Generation of Soft Shadow Textures

Michael Herf

16 May 1997

CMU-CS-97-138

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

email: herf@cmu.edu
World Wide Web: http://www.cs.cmu.edu/~ph/shadow.html

This paper extends work described in CMU-CS-97-104.

Abstract

Soft shadows give important cues of realism to 3D scenes. Existing methods can produce hard shadows in real time and soft shadows in minutes to hours per frame. Here a technique for producing accurate soft shadows quickly in software is presented, allowing soft shadow texture maps to be computed for simple scenes on commodity hardware in real time.

Contents

Chapter 1

Shadows in Computer Graphics

Computer graphics has long been split in personality, with one side seeking photorealistic simulation of visual phenomena, and the other trying to generate pictures quickly enough that interaction is possible. Improved algorithms and faster hardware are allowing these two personalities to converge. One of the techniques typically relegated to offline "batch" renderers is that of realistic soft shadow generation from area light sources. Proposed is a technique that allows real-time generation of realistic soft shadows.

1.1 The Anatomy of a Shadow

Accurate shading and visibility computations are important to making realistic 3D imagery. For example, lighting designers often use the position or orientation of shadows to create a specific mood. In complex 3D scenes, shadows also give important distance cues. For shadows produced from an area light source, objects nearly touching a *receiver*, or polygon to have shadows drawn on it, will cast relatively hard-edged shadows, while objects far away will produce softer diffuse edges.

1.1.1 Definitions

A soft shadow is produced by a single light source, and a surface may have several overlapping shadows from many light sources. Soft shadows have both a *penumbra* and an *umbra*. The umbra is the completely-occluded region of a shadow, where all illumination is from other light sources. Surrounding the umbra is the penumbra, the partially shaded region, which gives rise to the term "soft" shadow, since the existence of a penumbra gives a gentle edge to a shadow. Shadows with penumbra are produced by area or linear light sources, which subtend a region of the receiver polygon's view of the scene.

1.2 Applications

Soft shadows can be used to make more realistic images nearly everywhere that computer graphics is used. For example, interactive lighting design for computer-based animation is now quite a laborious process, because a full computation of the shadows in a scene can take a very long time. For interaction with a 3D environments, for instance in Virtual Reality systems, soft shadows can help increase the illusion of reality. In a similar vein, computer games could also benefit from more realistic shadows. All of these methods require real-time rendering of shadows to provide interaction with dynamic geometry.

1.3 Producing Soft Shadows

When considering only direct illumination, shadows and visibility in general are some of the most computationally difficult effects to simulate. At a single point, a shadow depends on all scene geometry, which may be very complex. Most other aspects of direct illumination can be evaluated by considering only local parameters. In the realm of global illumination, this complexity persists, with a large portion of form-factor computation for radiosity algorithms being devoted to visibility computations.

Here the focus is in producing soft shadows for direct illumination, where visibility determines which parts of each area light source are visible from a specific point.

1.4 Overview

This section describes some of the assumptions made for the algorithm described in this paper. These assumptions need not apply to all shadow algorithms (in fact, they do not), but they are useful for real-time shadows.

A good representation for soft-shadow information on a planar surface is a texture map. This representation is sometimes used to display radiosity solutions [23], and lends itself to quick display and simple re-use. On an increasing number of machines, texture maps can be displayed entirely in hardware. Texture mapping requires computational resources for display roughly bounded by the number of pixels on a screen times depth complexity (which in practice is small), whereas analytic or geometric shadow methods can require resources that scale with apparent geometric complexity in a scene, which is potentially unbounded.

As an assumption, we restrict our computation to diffuse scenes, since diffuse reflectors are view-independent. This allows a texture map to be computed once and to be re-used from multiple viewpoints, requiring updates only when the scene geometry changes. Some specular information can be optionally added as an additional pass for each frame.

Light sources are sampled discretely, each point sample resulting in a hard shadow texture. The contributions of all such samples are added together, to form a soft shadow texture.

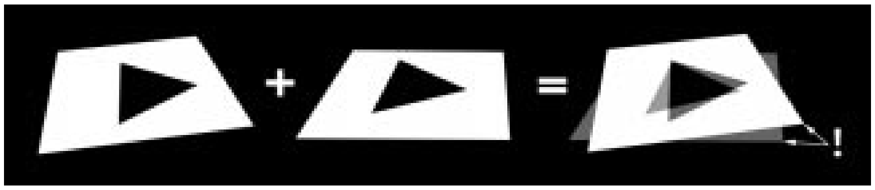

Figure 1.1: *Unregistered projections. If shadow projections are done with a standard perspective camera then views from different light samples do not line up.*

Each hard shadow texture can be computed by viewing a *receiver polygon* (the polygon which will have the texture map applied to it) from the point of view of a light sample point. The receiver polygon is drawn shaded as it would be in world space, and intervening objects are drawn in the scene's ambient color. This computation accurately models the radiance contribution of each point light source across the surface of each receiver polygon.

Combining these hard shadows proves to be difficult (See figure 1.1), since a normal camera projection results in a number of hard shadow images that do not register with one another. In earlier work with Paul Heckbert [18] a solution to this registration problem using a perspective projection is presented.

The remainder of this report begins with an overview of previous work in simulating shadows and a summary of an earlier technical report that gives an overview of the algorithm. Then there is a detailed discussion of how certain assumptions about area light sources in a scene can lead to a fast software implementation of the algorithm. The discussion concerns how to reduce memory bandwidth requirements of the algorithm by several orders of magnitude while maintaining the correctness of the approximation.

This soft shadow effort began as a research project for a graduate course in Rendering taught by Professor Paul Heckbert in Fall, 1995. High-end graphics hardware (a RealityEngine from Silicon Graphics) was used in the initial project. Shadows were produced using a method described in [18], but most scenes required seconds to update their shadow textures, rather than the fraction of a second desirable for real-time interaction. Unforunately, this performance characterization is somewhat misleading, since the implementation does not take full advantage of the speed of the graphics hardware. Some efficiency problems were due to non-optimal use of OpenGL, and others were due to limitations in current implementations of OpenGL.

During the Fall of 1996, the author began investigating ways to enhance this method using algorithms that could be implemented in software on PCs, hoping to discover a simple method that would be widely usable. This report is the result of that effort.

For many scenes, the optimizations presented here enable a software implementation of this algorithm on typical commodity hardware to exceed the speeds achievable using the former implementation on the fastest high-end workstations we have tested, among them a two-pipe RealityEngine^2. Of course, these workstations would perform equally well with a similar software implementation, and could take advantage of hardware acceleration for scene display as well.

Chapter 2

Previous Work

Shadow algorithms until 1990 were surveyed by Woo [30] in much more detail than is given here. For reasons of practicality, this discussion splits existing algorithms based on how they are commonly used. Since most of these techniques have been known for several years, this distinction allows us to highlight several hidden factors, including ease of implementation and generality.

We make some additional distinctions for clarity. *Real-time* shadows should be understood to be those which can be generated for common dynamic scenes at interactive rates, where a "dynamic scene" is one in which objects move with respect to one another. As was mentioned previously, radiosity solutions for static geometry (with soft shadows) can be represented in texture maps, which can be used for interactive walkthroughs of a static scene, but these should not be considered real-time methods, since their precomputation phase can take minutes or even hours for each change in geometry, when suitably high-quality meshing is used.

Secondly, several algorithms described can produce "soft-edged" shadows by blurring hard shadows. While this is often a visual improvement over hard shadow discontinuities, it does not provide even a qualitatively accurate simulation of most 3-D scenes, since the width of the penumbra is constant, regardless of the positions of the objects in a scene.

2.1 Shadows in Practice

2.1.1 Real-Time Methods

On current hardware, every method currently known to generate real-time shadows generates them from a single point light, generally resulting in hard-edged shadows. Among these, several are in common use.

Blinn [4] introduced a simple method that projects geometry to planar receivers using a 3D to 2D projection, in which shadows are drawn in black or in a stippled pattern. While this method is very straightforward to implement, it suffers from a lack of generality. This technique is commonly used on one or several large planes (e.g., the walls and floor in a room). For non-infinite receivers, it requires some difficult clipping. Additionally, while this technique can produce shadows that are geometrically correct, it cannot produce shadows that correctly reveal the relative intensities of shadows and other shading (lighting and texture, for example) across the surface. Nevertheless, since its implementation literally involves only computing one perspective transform, this technique is the most widely used for shadows drawn on a single surface.

The shadow volume approach has been used for real-time display by several researchers. This technique computes the volume caused by occluders, oriented with respect to a specific light source. Using either analytic intersection of these volumes with scene geometry or a more discrete approach (generally pixel-based), shadows can be drawn as the interiors of their intersections with receiver polygons. Using stencil buffers on high-end hardware, an approach introduced by Fuchs et al [14] and later improved by Heidmann [19] can generate hard shadows in real-time, and is a popular technique.

The shadow z-buffer is an efficient and commonly-used technique in both hardware and software. A shadow z-buffer is constructed by viewing a scene from the point of view of a light source and drawing each polygon or surface in the scene using a color that uniquely identifies it, using standard z-buffering. This shadow z-buffer can later be consulted at every point rendered in the scene, with a point being in shadow if its projection into the shadow z-buffer references another object (i.e. it cannot be seen from the light source). Reeves et al. [26] introduced a more robust version of the algorithm for software renderers, which was used in versions of Pixar's prman. This method can blur hard shadows to create a "soft" effect, but this should not be mistaken for a correct soft shadow, as it only accounts for visibility from a point source. Segal et al. [27] has demonstrated how texture-mapping hardware can be used to support the shadow z-buffer technique. As a result, Silicon Graphics has added extensions to their version of the OpenGL interface that use hardware (when available) to produce such shadows quickly. The shadow z-buffer technique can suffer from sampling artifacts due to lack of resolution, and has a per-frame cost that is linear in the number of light sources, but is a widely-used way to produce real-time hard shadows for an entire scene.

2.1.2 Non-Real-Time Methods

Brotman and Badler [5] presented a depth-buffer based soft shadow algorithm that produces reasonable results but which is slow and takes large amounts of memory. Extending Brotman's and Heidmann's techniques, Diefenbach [12] produced soft shadows quickly on graphics hardware, but his work fails to be real-time in practice.

Haeberli [15] introduced a brute-force technique for producing real-time soft shadows that scales well with increases in geometry. This technique simply averages together full-scene renderings due to point-light sources generated by one of the above hard shadow techniques. This technique has a high cost per frame, but it is fully general. The availability of hardware that will make this method real-time is several years off.

Area light sources can be sampled using ray tracing, by sending multiple shadow rays to each area light source for each object intersection [11]. This technique produces the highest-quality images of any known techniques, but its computational cost is more than prohibitive for real-time simulation.

Straightforward radiosity techniques can also simulate soft shadows, with a corresponding high computational cost. Visibility and shadow computation are equivalent problems in the context of finding radiosity solutions. Radiosity typically computes visibility using ray casting or hemicube methods for form-factor computation, and the visibility calculations often dominate the computation time. Incremental radiosity can in some cases produce real-time shadows [6], but the mesh resolutions required for high-quality shadows are typically out of the reach of real-time computation.

2.2 Alternate Methods

2.2.1 Geometric Methods

Chin [7] has used BSP trees to quickly compute an analytic soft-shadow solution. All combinations of projections of vertices and edges of a polygonal light source with respect to polygonal occluders are used to divide the scene. The areas between these discontinuities can be interpolated using Gouraud shading, which is a reasonable approximation. Chin's method is not real-time. Drettakis [13] has used a technique called backprojection to generate shadows in several seconds to minutes for simple scenes. Drettakis's algorithm scales linearly with scene geometry, a particular advantage of the technique. Both of these methods are reputedly difficult to implement.

2.2.2 Analytic Methods

Several researchers [3, 29] use analytic methods to compute shadows. Tanaka claims relatively quick speeds. In general, these techniques do not scale well with large amounts of geometry.

2.3 This Work

In this work and in previous work with Paul Heckbert [18, 20], we draw from several of the above techniques to describe a way to produce realistic radiance texture maps. Blinn's technique of projecting objects to surfaces is similar, as is the shadow z-buffer technique, since ours considers an eye-point projection. The use of the accumulation buffer is similar to Diefenbach's.

Möller [22] suggests some additional speedups for our earlier technique, including an approximation that translates texture coordinates to simulate motion, and blurring hard textures to simulate soft shadowing. These optimizations make questionable quality tradeoffs. If blurred hard shadows are the goal, it is probably more advantageous to take an approach like Reeves [26], but these should not be considered correct soft shadows. Blurring hard shadows using Reeves's and Segal's techniques will produce images much more quickly and for every object in a scene. Möller also suggests an alpha-blending technique similar to that suggested in the software section of [20] and presented more formally in this report, but also introduces unacceptable error by blending multiple shadows, approximating an arithmetic series using a geometric one.

In the earlier work[18], we focus on an algorithm suitable for high-end graphics hardware. In this work, new techniques are introduced that allow a similar computation to be done efficiently in software.

Chapter 3

The Basic Algorithm

This description is taken in part from [18] to give the reader appropriate background for sections that follow.

3.1 Registration of Hard Shadows

To compute a soft shadow from multiple hard shadows, we want a way to accurately combine the hard shadow images into a soft shadow texture map.

One might assume that pointing a perspective camera at each receiver from the point of view of each light sample might work, but the change in viewpoint creates a registration problem, where the projected geometry of the receiver polygon changes from viewpoint to viewpoint. We show that for a parallelogram receiver (or one that has such as its bounding box), registering these projections can be accomplished as a simple perspective projection.

We derive a method to transform from the volume formed by a light sample point and a parallelogram bounding the receiver polygon to a unit box. Clipping to this box will leave only the polygon fragments between the light source and the receiver, and these occluding fragments will all be rendered with the same ambient color, so it suffices to render them orthographically projected with a painter's algorithm in unsorted order.

3.2 Projective Transformation of a Pyramid to a Box

We want a projective (perspective) transformation that maps a pyramid with parallelogram base into a rectangular parallelepiped. Note that the pyramid need not be right (axis need not be perpendicular to base). The pyramid lies in object space, with coordinates $(x_{\mathrm{o}}, y_{\mathrm{o}}, z_{\mathrm{o}})$.

It has apex $\mathbf{a}$ and its parallelogram base has one vertex at $\mathbf{b}$ and edge vectors $\mathbf{e}_{\mathrm{x}}$ and $\mathbf{e}_{\mathrm{y}}$ (bold lower case denotes a 3-D point or vector). The parallelepiped lies in what we will call unit screen space, with coordinates $(x_{\mathrm{u}}, y_{\mathrm{u}}, z_{\mathrm{u}})$. Viewed from the apex, the left and right

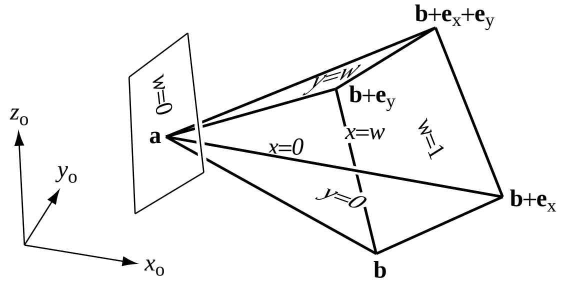

Figure 3.1: *Pyramid with parallelogram base. Faces of pyramid are marked with their plane equations.*

sides of the pyramid map to the parallel planes $x_\mathrm{u}=0$ and $x_\mathrm{u}=1$, the bottom and top map to $y_\mathrm{u}=0$ and $y_\mathrm{u}=1$, and the base plane and a plane parallel to it through the apex map to $z_\mathrm{u}=1$ and $z_\mathrm{u}=\infty$, respectively. See figure 3.1.

A 4×4 homogeneous matrix effecting this transformation can be derived from these conditions. It will have the form:

$$\mathbf{M}=\begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ 0 & 0 & 0 & 1 \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix},$$

and the homogeneous transformation and homogeneous division to transform object space to unit screen space are:

$$\begin{pmatrix} x \\ y \\ 1 \\ w \end{pmatrix}=\mathbf{M}\begin{pmatrix} x_\mathrm{o} \\ y_\mathrm{o} \\ z_\mathrm{o} \\ 1 \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} x_\mathrm{u} \\ y_\mathrm{u} \\ z_\mathrm{u} \end{pmatrix}=\begin{pmatrix} x/w \\ y/w \\ 1/w \end{pmatrix}.$$

The third row of matrix $\mathbf{M}$ takes this simple form because a constant z_u value is desired on the base plane. The homogeneous screen coordinates x, y, and w are each affine functions of x_o, y_o, and z_o (that is, linear plus translation). The constraints above specify the value of each of the three coordinates at four points in space – just enough to uniquely determine the twelve unknowns in $\mathbf{M}$.

The w coordinate, for example, has value 1 at the points $\mathbf{b}$, $\mathbf{b}+\mathbf{e}_\mathrm{x}$, and $\mathbf{b}+\mathbf{e}_\mathrm{y}$, and value 0 at $\mathbf{a}$. Therefore, the vector $\mathbf{n}_\mathrm{w}=\mathbf{e}_\mathrm{y}\times\mathbf{e}_\mathrm{x}$ is normal to any plane of constant w, thus fixing the first three elements of the last row of the matrix within a scale factor: $(m_{30},m_{31},m_{32})^T=\alpha_\mathrm{w}\mathbf{n}_\mathrm{w}$. Setting w to 0 at $\mathbf{a}$ and 1 at $\mathbf{b}$ constrains $m_{33}=-\alpha_\mathrm{w}\mathbf{n}_\mathrm{w}\cdot\mathbf{a}$ and $\alpha_\mathrm{w}=1/\mathbf{n}_\mathrm{w}\cdot\mathbf{e}_\mathrm{w}$, where $\mathbf{e}_\mathrm{w}=\mathbf{b}-\mathbf{a}$. The first two rows of $\mathbf{M}$ can be derived similarly (see figure 3.1). The result is:

$$\mathbf{M}=\begin{pmatrix} \alpha_\mathrm{x}n_\mathrm{xx} & \alpha_\mathrm{x}n_\mathrm{xy} & \alpha_\mathrm{x}n_\mathrm{xz} & -\alpha_\mathrm{x}\mathbf{n}_\mathrm{x}\cdot\mathbf{b} \\ \alpha_\mathrm{y}n_\mathrm{yx} & \alpha_\mathrm{y}n_\mathrm{yy} & \alpha_\mathrm{y}n_\mathrm{yz} & -\alpha_\mathrm{y}\mathbf{n}_\mathrm{y}\cdot\mathbf{b} \\ 0 & 0 & 0 & 1 \\ \alpha_\mathrm{w}n_\mathrm{wx} & \alpha_\mathrm{w}n_\mathrm{wy} & \alpha_\mathrm{w}n_\mathrm{wz} & -\alpha_\mathrm{w}\mathbf{n}_\mathrm{w}\cdot\mathbf{a} \end{pmatrix},$$

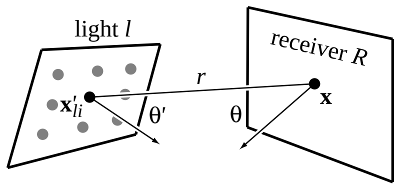

Figure 3.2: *Geometry for direct illumination. The radiance at point* $\mathbf{x}$ *on the receiver is being calculated by summing the contributions from a set of point light sources at* $\mathbf{x}'_{li}$ *on light* l*.*

where

$$\begin{aligned} \mathbf{n}_\mathrm{x} &= \mathbf{e}_\mathrm{w} \times \mathbf{e}_\mathrm{y} & & & \alpha_\mathrm{x} &= 1/\mathbf{n}_\mathrm{x} \cdot \mathbf{e}_\mathrm{x} \\ \mathbf{n}_\mathrm{y} &= \mathbf{e}_\mathrm{x} \times \mathbf{e}_\mathrm{w} & &\text{and} & \alpha_\mathrm{y} &= 1/\mathbf{n}_\mathrm{y} \cdot \mathbf{e}_\mathrm{y} \ . \\ \mathbf{n}_\mathrm{w} &= \mathbf{e}_\mathrm{y} \times \mathbf{e}_\mathrm{x} & & & \alpha_\mathrm{w} &= 1/\mathbf{n}_\mathrm{w} \cdot \mathbf{e}_\mathrm{w} \end{aligned}$$

Both Blinn [4] and Diefenbach [12] use a related projective transformation for the generation of shadows on a plane, but both are projections (they collapse 3-D to 2-D), while the above is 3-D to 3-D. The third dimension allows efficient axis-aligned clipping, since the projection maps to an orthographic space.

3.3 Direct Illumination

Direct, diffuse illumination from area light sources [10, eq. 2.54] has the form

$$L(\mathbf{x}) = \rho(\mathbf{x}) \left(L_\mathrm{a} + \int_{\text{lights}} \frac{\cos\theta \ \cos\theta'}{\pi r^2} \, v(\mathbf{x}, \mathbf{x}') \, L(\mathbf{x}') \, dA' \right), \tag{3.1}$$

where $v(\mathbf{x}, \mathbf{x}')$ is a Boolean visibility function, $L(\mathbf{x})$ is outgoing radiance at point $\mathbf{x}$ due to either emission or reflection, L_a is ambient radiance, and $\rho(\mathbf{x})$ is reflectance. The latter three are generally RGB vectors. (See figure 3.2.)

In this formula, v, the visibility function, is difficult to compute, since it requires global knowledge of the scene, while the other factors require only local knowledge. Of course, v is necessary for the computation of shadows.

3.4 Hardware Shading

For graphics hardware supporting scan conversion, lighting, and clipping of polygons, and an accumulation buffer that can be used to average a set of images, we can approximate each area light source with n_l point light sources $\mathbf{x}'_{li}$ each with associated area a_{li}:

$$L(\mathbf{x}) \approx \rho(\mathbf{x}) \, L_\mathrm{a} + \sum_l \sum_{i=1}^{n_l} \left(\frac{a_{li} \, \rho(\mathbf{x}) \cos\theta_{li} \ \cos\theta'_{li} \, L(\mathbf{x}'_{li})}{\pi r_{li}^2} \right) v(\mathbf{x}, \mathbf{x}'_{li}). \tag{3.2}$$

Each term in the inner summation can be regarded as a hard shadow image where $\mathbf{x}$ is a function of screen (x, y). These terms consist of the product of two factors. The first factor can be calculated in hardware simply by rendering the receiver (the polygon being shadowed) as illuminated by a point spotlight with quadratic attenuation. The receiver should be subdivided into small polygons so that $\cos\theta\cos\theta'/r^2$ will be well approximated. The second factor is visibility, which can be computed by projecting all polygons between light source point $\mathbf{x}'_{li}$ and the receiver onto the receiver and by drawing all such polygons in the ambient color for the scene.

3.5 Implementation

This technique was implemented by the author using the OpenGL API on a Silicon Graphics IRIS Crimson with RealityEngine graphics. The performance overall was reasonable for small numbers of receivers in simple scenes. A scene with approximately 500 moving polygons, with a single receiver, can be redrawn at 5 Hz with decent light sampling (64 samples). This performance is significantly less than the hardware's capabilities, due to limitations imposed by the API.

At the time of this writing, GLX context sharing is not supported on high-end SGI machines, which means that texture objects cannot be passed between different-sized drawing contexts efficiently. What would be ideal is to generate textures in a small buffer the exact size of the texture, which could be passed directly to texture memory after computation. The context-sharing limitation prevents this.

For shadow computation, there are two options for implementation under OpenGL. First, we can compute textures in a separate context, then pull them through host memory and load them into texture memory, a very inefficient process that relies on host memory bandwidth. This is the only option when working under OpenGL 1.0.

Under OpenGL 1.1, we can let a texture object share the same context as a displayed scene. However, the inflexibility of the accumulation buffer (and the lack of optimization of `glScissor` for small buffers) requires that the hardware execute the accumulation buffer operation on a region the size of the scene's output size, not the size of the texture map! This makes the accumulation buffer the bottleneck for a scene displayed in a large window, since a huge amount of unnecessary computation is done.

Either efficient context-sharing or optimization of `glScissor` would alleviate this problem, and the results presented here would favor graphics hardware much more than they do.

The hardware technique is a better approximation than the technique that follows, supporting colored light sources and a better approximation to the shading formula. For the highest-quality images, or for scenes with extreme light geometries, the hardware technique or its software equivalent should be used.

Chapter 4

Software-Directed Improvements

4.1 Cheaper Shading

To generate soft shadows in software, we may wish to have a simpler approximation to the above formula (3.2). The outer summation above translates to adding radiances at each $\mathbf{x}$ for each light sample, which requires significant computation and memory cost, and $O(tl)$ operations, for t the number of texels, and l the number of light sample points. In preparation for later optimizations, we introduce here a set of assumptions that approximate (3.1).

In certain scenes and light source geometries, we can approximate (3.1) using equal weighting of light samples. It can be shown that these restrictions fit the conditions encountered in many scenes, so we do not sacrifice too much generality.

To achieve the equal weighting simplification, we consider the case with

1. a light source of a single color (i.e. $L(\mathbf{x}')$ constant),
2. subtending a small solid angle with respect to the receiver,
3. and with distance from the receiver having small variance.

The first assumption allows us to compute only intensity across a Lambertian surface, modulating by color as a separate step. (Colored shadows can be computed by applying this approach multiple times.) Most scenes have one or several light sources, generally of uniform color. This assumption is exactly the case with a typical fluorescent or incandescent lamp in a room, for example. By the second two assumptions, we see that the contribution of each light sample point will be roughly equal, since $\cos\theta$ has small variance by our assumption, as does $1/r^2$.

In (3.1), this means that incident radiance and visibility are approximately independent, and so visibility can be pulled out of the integral:

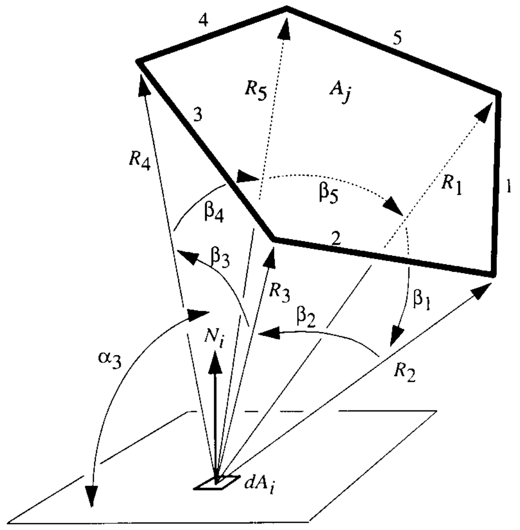

Figure 4.1: *Excerpted from [10] to illustrate quantities used in differential polygon form factor equations.*

$$L(\mathbf{x}) \approx \rho(\mathbf{x})\, L_{\text{a}} + \rho(\mathbf{x}) \sum_l \frac{L(\mathbf{x}_l')}{n_l} \int_{\text{light}\, l} \frac{\cos\theta\, \cos\theta'}{\pi r^2}\, dA' \sum_{i=1}^{n_l} v(\mathbf{x}, \mathbf{x}_{li}'). \tag{4.1}$$

The radiance integral above has an easily-computable closed form given in [10, p. 72] (See figure 4.1 for variable definitions):

$$F_{dA_i \to A_j} = \frac{1}{2\pi} \sum_{i=1}^{n} \beta_i \cos\alpha_i$$

which can be computed as:

$$F_{dA_i \to A_j} = \frac{1}{2\pi} \sum_{i=1}^{n} \beta_i N_i \cdot \overline{(R_i \times R_{(i+1)\%n})} \tag{4.2}$$

"where n is the number of sides on the polygon, β_i is the angle between R_i and $R_{(i+1)\%n}$ in radians, α_i is the angle between the plane of differential area dA_i and the triangle formed by dA_i and the ith edge of the polygon, and N_i is the unit normal to dA_i [10]."

The visibility summation is the fraction of light sample points visible from a point $\mathbf{x}$ on a receiver. These quantities can be multiplied in a final pass to produce a radiance texture. To proceed, we need an efficient way to compute the visibility map.

We first describe how to compute the visibility map using a brute-force approach, very similar to the hardware technique described above. Following that, we start again with a technique that computes the same quantities more efficiently, but with more implementation effort.

4.2 Brute-Force Approach

We can start by taking an approach quite similar to that taken in hardware. Most importantly, we wish to avoid the overhead of simulating an accumulation buffer to average images. The accumulation buffer maintains twice as many bits per channel as the output image to avoid quantization. Also, it can handle arbitrary linear combinations of images. Without the "equal weighting" assumption made above, these capabilities would be necessary. But with it, our problem is a much simpler one. We simply add n binary images to form an 8-bit image. (Of course, more bits may be necessary to accurately represent large numbers of samples.)

We do not have to worry about scaling this image, since we use it to scale unoccluded radiance as described above. In this modulation step (a pointwise scaling of a second image), we can "build in" all scaling factors that we need. The quantity we are computing is the number of visible light samples at each point, so we start with n visible light sample points, and subtract to represent occlusion.

```
For each receiver polygon
  Clear final visibility buffer to intensity n
  Choose a set of n samples
  For each light sample
    Clear intermediate output buffer to intensity 0
    Compute projection matrix
    Draw all polygons in color 1
      Clip to [0, 1], [0, 1], [1, infty]
    Subtract intermediate image from visibility buffer
  Compute unoccluded point-to-area form factor pointwise over receiver
  Multiply the visibility buffer pointwise by the form factor buffer
Store the final radiance texture map for later display
```

Newer memory architectures (those using Synchronous DRAM [25], for example) can provide much higher bandwidth to memory when accessed sequentially than when accessed randomly. It is reasonable to assume that even if raw memory speeds increase, the gap between sequential and random access will remain substantial. Improvements in microprocessor technology, such as Intel's MMX [24] make this memory bandwidth usable, by allowing many additions or multiplications to be accomplished in a single SIMD instruction operating on 64-bit data. This advantage benefits image processing algorithms in general, and specifically benefits a pixel-wise addition and blending of many bitmaps, as is required in the brute-force version of this soft shadow algorithm.

The application of this technique should be sufficient to achieve real-time rates, especially for small texture maps in scenes of very low complexity. The next section describes how to reduce required memory bandwidth and increase drawing rates. In most cases the following method will be much faster than this brute-force approach.

4.3 More Efficient Drawing and Compositing

Even if newer hardware can do brute-force compositing quickly, the algorithm still requires that we fill and transform many polygons to create the hard shadow texture maps. This can be very slow, especially for models of high complexity with many shaded receivers. In this section, a substantially faster implementation is described that reduces the required memory bandwidth and computation for producing soft shadow textures. Some of the described optimizations could be applied to either the hardware or to the brute-force solution presented above as well.

4.3.1 Edge-Based Methods

Recall that the inner summation of the visibility function in (4.1) is the number of visible light samples from each point (x, y) on the receiver's surface. This function, point-sampled at each texel, we call $V(x, y)$. For a finite number of samples, this quantity changes discretely as we move across a texture-mapped surface. In fact, for many regions (e.g., non-shadowed regions and umbra regions), $V(x, y)$ is constant.

This corresponds well with the observation that one of the time sinks in the brute-force algorithm is in filling polygons to a raster buffer. As an alternative, we discuss several edge-based schemes that can dramatically speed up computation of the desired visibility factor. We hope to find methods that allow the insertion of edges into a single buffer, so the compositing step can be greatly simplified, or even eliminated.

Span Buffers

First, we might try a span-based method, inserting spans into scanline-sized x-sorted buckets. Using a linear data structure, this method requires a $O(n)$ search or insertion step (for a scanline with n edge events), which can perform very badly for a scene with many small polygons. All of the cases that were empirically tested used a $O(n)$ step to coalesce overlapping spans into single spans, in order to save on memory usage. If we can afford to use more memory, we could store all spans in arbitrary order, sort them in $O(n \log n)$ time, and then draw them (without coalescing) into an output buffer.

On very well-tuned code in common cases, the speeds achieved using span buffers are just marginally faster than those achieved by simply filling the polygons. The compositing step can of course be greatly simplified, but the total benefit is not very great, especially for small texture maps.

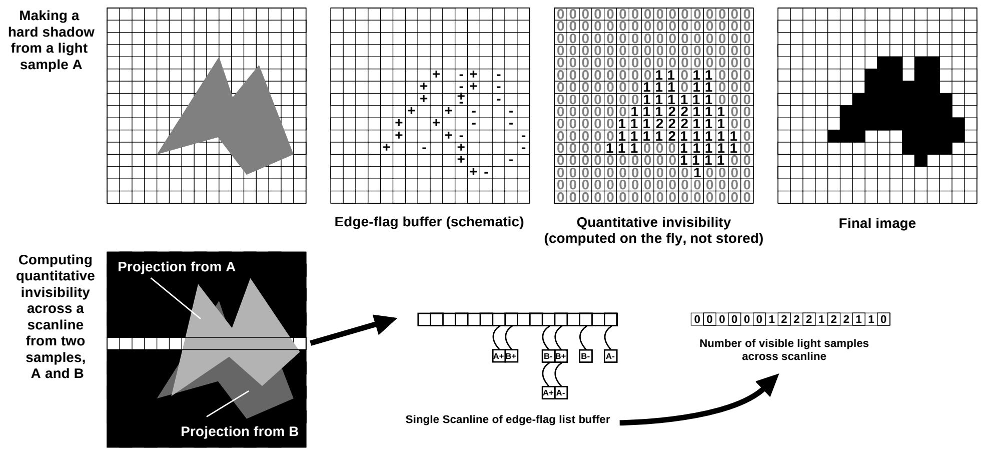
Making a hard shadow from a light sample A
Edge-flag buffer (schematic)
Quantitative invisibility (computed on the fly, not stored)
Final image
Computing quantitative invisibility across a scanline from two samples, A and B
Projection from A
Projection from B
A+
B+
B-
B+
B-
A-
A+
A-
Single Scanline of edge-flag list buffer
0 0 0 0 0 0 1 2 2 2 1 2 2 1 1 0
Number of visible light samples across scanline

Figure 4.2: *Edge flag fill explained. Top explains how a shadow texture from just one light sample with overlapping geometry would be computed, using edge flags and quantitative invisibility. Though quantitative invisibility is not binary here, a binary image is produced by using the δ-function. The bottom demonstrates for a highlighted scanline how to combine projections from two light sources into a buffer that indicates the number of visible light samples. Here, edge flags from different light sources are properly distinguished, and the result is an image with three levels of shading.*

Modified Edge-Flag Fill

Instead, we consider a method based on Ackland's edge-flag fill algorithm [1]. Ackland used hardware to fill polygons for animation, drawing only edges to indicate the beginning and end of a span. He drew edges by using a flag with a bit devoted to each color, and then computed the combination using an XOR operation. The method here generalizes this method to an arbitrary number of "colors", or changes in intensity at each pixel by use of a linked list, and computes a running sum rather than an XOR.

If we draw edges rather than polygons into a buffer with the same size as our output buffer, we can indicate easily where the number of visible light samples changes (to the nearest texel). This approach is analogous to bucket-sorting, whereas the span-based approach is similar to insertion sort (with extra work to coalesce edges) or a quicksort in the less memory-efficient case. Drawing all projected polygon edges for each light sample will give an edge-flag buffer, indicating changes in shading due to an approximated area light source. After all such projections, we reconstruct $V(x, y)$ by computing a running sum of all edge events across each scanline.

Since geometry can be oriented arbitrarily with respect to the receiver polygon, a potentially unbounded number of edge events could happen at each point in the edge-flag buffer. We use a linked list at each pixel so that we can store any number of events at a given pixel. Also, between a receiver and a light source, there can be many occluding surfaces. To account for this, we store two items in each edge event (represented here as a linked list):

```
Flag
  Direction : int
  LightID   : int
  Next      : ˆFlag
```

Direction and Polygon Facing

The `Direction` above is typically the integer 1 or -1, to indicate whether the edge is a left or right edge for the polygon that we are drawing. These values are stored directly to make computing the running sum very simple.

To compute the value of `Direction` for each edge event, we make a pass over the geometry, after transforming projected polygons by M. First we compute the "facing" F. The following is for three non-collinear projected vertices of a polygon, p_1, p_2, and p_3:

$$F := \text{sign}\left((p_{3_x} - p_{1_x})(p_{2_y} - p_{1_y}) - (p_{3_y} - p_{1_y})(p_{2_x} - p_{1_x})\right)$$

This is simply the sign of the z-component of the cross product, which is equivalent in sign to the dot product $\mathbf{k} \cdot \mathbf{N}$, where $\mathbf{k}$ is the z basis vector and $\mathbf{N}$ is the normal to the face in question. This is correct since the projection is at this point orthographic, so we do not need to compute the other parts of the cross product. However, this is not yet the value `Direction` that we seek.

We now have two "convention" issues to clear up before we can determine which side of a polygon an edge lies on. First, there is the polygon edge-ordering convention. When you look from the "outside" of a mesh, some polygons have vertices ordered in clockwise order, and some have vertices ordered counterclockwise. We account for either case by simply flipping sign based on the convention of the mesh we use.

Secondly, we consider our coordinate system. On many displays, scanlines are ordered top-to-bottom, and if we compute coordinates in screen space, we must be careful to adjust for this case as well. Again this is another sign flip.

We multiply these two sign flips, and then find $D := \pm\text{sign}(\Delta y)$ for the edge we are drawing, where Δy is the directed y-component of the edge in question.

Finally, to obtain the value of `Direction`, we simply multiply to find DF. The sign flips above are constant for a given mesh and computing environment, the facing computations to find F are done at most once per polygon, and `Direction` is constant over an edge.

LightID and Sample Uniqueness

The `LightID` is simply the number of the light sample $(1..n_l)$ that uniquely identifies which point source was used to project geometry to that edge event. This allows us to

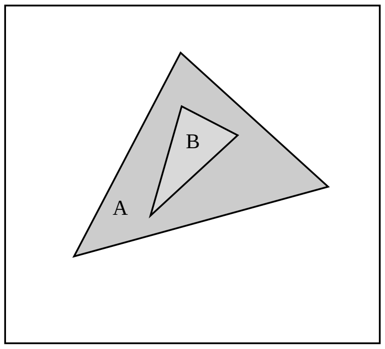

Figure 4.3: *Two polygons projected from the same light sample point. We do not want to count the edges of B when considering visibility from this light sample.*

evaluate at every edge event whether or not this event should cause a change in visibility. For example, consider two polygons A and B projected from a single light sample such that polygon B lies entirely within A. In this case, when we encounter an edge of A we should increment or decrement the number of visible samples, but we should ignore the edges of B.

To generalize this observation, we define a counter array S. (S_i is the counter for light sample i.) We use each counter as part of the postprocess that computes $V(\mathbf{x})$ across a receiver as a sum of edge events. This array has one element per light sample, and space enough to count the maximum number of events that may be encountered in a scanline. This counter dynamically computes the number referred to by Appel as [2] as *quantitative invisibility*, which is a quantity equivalent to the number of surfaces a ray traced between $\mathbf{x}$ and the light sample l would intersect. We now can represent $V(x,y)$ as follows:

$$V(x,y) = \sum_{i \in l} \delta(S_i)$$

Here $\delta(t)$ is the Kronecker (discrete) delta function. ($\delta(0) = 1$ and $\delta(t) = 0$ for $t \neq 0$.) To make this computation incremental, we would allow changes of S_i from the value $0 \rightarrow 1$ to decrement the dynamically-computed $V(x,y)$, and changes from $1 \rightarrow 0$ to increment $V(x,y)$.

Rasterization Details and the Final Pass

In the final pass, we loop over all pixels, accumulating edge events at each pixel and then writing the result incrementally to a framebuffer.

To draw these edges, we must be very careful, since exactly one pixel must be drawn each time a scanline is intersected by an edge. For vertices that fall on scanlines that are shared by two edges, we must draw them only for only one of the two segments, typically the lower one.

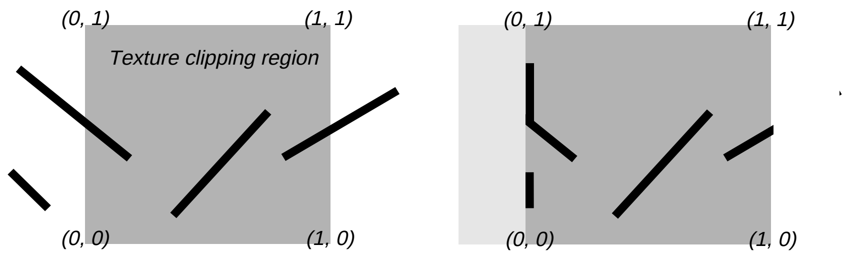

Figure 4.4: *Left side shows segments before clipping, right side shows segments afterwards. Notice the segments clamped to the left edge on the right-side image.*

First, notice that a Bresenham line-drawing algorithm will not work for drawing such lines, since it may draw multiple points per scanline. We find edges in a way similar to how correctly-drawn polygons are filled[17]. To achieve a pixel-perfect line drawer, correctly rasterized edges must do sub-pixel correct computations, and must pay attention to rounding conventions in the x direction as well as well as in y. Surprisingly, we can round *up* all x coordinates to the nearest integer in this case because of our drawing method. In a correct polygon renderer, left edges would be rounded up, while right edges would be rounded down, but the right edges would be drawn. Here we add edge events at each pixel and then draw, meaning that one pixel to the left of the right edge, but not the edge itself, is drawn.

Correctness is critical in this implementation, since anything less than a pixel-perfect renderer will cause not just single-pixel errors, but “leaks” that appear as incorrect shading across an entire scanline.

Clipping

Finally, slight modifications to a standard parametric clipping algorithm must be made. We assume here that the final computation of $V(x, y)$ above is done by summing edge events left-to-right, though this procedure can be generalized easily. Additionally, we use the notation that a texture map is computed in the region $[0, 1], [0, 1]$.

We must make special accomodation for polygons whose left edges fall to the left of 0, since otherwise quantitative invisibility would be incorrect across a scanline. Specifically, we clip to the box defined by $[-\infty, 1]$ in the upper left and $[1, 0]$ in the lower right. For each segment that intersects the left edge, we make two new segments, one containing portions of the line with non-negative xs and one without. Then we clamp all negative x-coordinates to 0, giving many extra “seed” segments on the left edge of the edge-buffer. This ensures that the clipper will produce correct results when we add edge events later.

4.3.2 Memory Allocation

The described edge-flag algorithm uses linked lists heavily, and as such makes heavy use of the memory allocation functions, so these are a specific target for optimization. A call to a standard `malloc()` once per texel is clearly too much, and the cleanup costs (to re-use the space) are equally expensive.

The solution is to implement a simple memory-management scheme that allocates a large amount of memory for all linked lists, and then to incrementally allocate memory by incrementing a pointer by a fixed amount (since all elements are of constant size). This also allows efficient clears of the buffer, by dropping this pointer to the base of the allocated space and by clearing all pointers that index into this structure (typically one per texel). This can be done very quickly.

4.3.3 Silhouettes

The above edge-based scheme will be correct for complex meshes that have faces sharing many edges. However, a hard shadow can be computed also by just its silhouette edges. In general, there are many fewer silhouette edges than total edges in a mesh.

We pay to draw each edge, so the fast determination of silhouette edges is a very advantageous optimization. For this discussion, we consider open and closed manifolds, a slight extension to most treatments of the matter in most existing shadow literature, most of which consider only closed manifolds.

Notice first that we are finding silhouettes for *projected* geometry, i.e. a mesh after being transformed by M but before being projected to the plane of the receiver. By our assumptions, each edge is shared by at most two faces. We use a simple spatial data structure of polygons, edges, and vertices, where edges have pointers to vertices and (one or two) polygons, and polygons have pointers to edges. This allows a fast mapping of edges to the set of one or two polygons they belong to.

From our discussion of edge-based methods above, we already know how to compute whether an edge is “left” or “right” for a given polygon and a given projection. This quantity, as opposed to F, the facing, which is used in most silhouette methods for closed manifolds, allows us to handle open manifolds easily. In other methods, edges that are shared by both a front-facing and back-facing polygon are considered silhouette edges.

An edge is a silhouette edge if and only if one of the following is true:

1. It is a left edge for two faces
2. It is a right edge for two faces
3. It a left or right edge for exactly one face

The edges of a mesh so chosen comprise the silhouette edge set of the mesh. These edges can be rasterized using the above edge flag method as if they are the edges of the mesh itself, for an identical result in the final visibility map. One modification must be made to the value of `Direction` for edge events, however. Since the first two silhouette cases above effectively combine two edges into one, we simply count edges belonging to two faces as two events. The value of `Direction` in that case is 2 or -2. The other cases remain as described above.

4.3.4 Radiance Integral

Now that we have a fast way to compute $V(x, y)$, we need to consider the other term in (4.1), namely the integral that we claimed has a simple closed form earlier. Cohen and Wallace [10, p. 72] give the unoccluded point-to-area form factor in equation (4.2).

This function can be updated in real-time for each point in a texture map by doing significant optimizations (table lookups, etc.) However, this effort is unnecessary, since the function is very smooth, and so it is sufficient to subsample the function and bilinearly interpolate it, which can be done much more quickly. For most situations, a rather coarse sampling is sufficient to produce good results.

4.4 Unimplemented but Useful Optimizations

All of the optimizations described in this section are likely to have a major impact on performance, and are described here without having been implemented in our test renderer.

4.4.1 Simple Frustum Culling

A simple and commonly-used technique for reducing the amount of drawn geometry is frustum culling. We could use the approach suggested by Clark [8], and represent scene geometry as hierarchical bounding volumes to enable recursive frustum culling. This technique transforms a hierarchy of simple bounding boxes to determine if the more complex meshes that they bound would fall inside the viewing frustum. This technique has enormous advantages for the production of shadow textures, since objects outside of point-light to receiver frusta can be efficiently culled from scene geometry. For a well-distributed scene, the culling operation can be performed in $O(n^{0.33})$ in the average case [9] for uniform subdivision. Haines and Wallace [16] have done similar work for hierarchical bounding volumes, but the expected complexity is much harder to compute. We assume that efficient culling will provide a much-needed speedup over the $O(n)$ worst case for a single texture.

For both clipping and culling, it is more efficient to transform scene geometry (and its bounding hierarchy) using the matrix presented above, rather than using arbitrary-plane clipping. The culling and clipping steps take no longer, since they can be accomplished by using axis-aligned clipping. Most importantly, the transformed vertices can be re-used after clip-testing is done.

This technique was not implemented in our test implementation, since most test scenes were very simple, but this or something like it is of paramount importance for doing real work!

4.4.2 Hierarchical Culling

The naïve implementation of the above will require computing a partial cross product for each face in every mesh that is not culled and transforming each vertex in the mesh by a 4×4 perspective transform. This can be quite expensive, so we could substitute a minor variation of the hierarchical backface computation method described by Kumar [21] to avoid transformation and silhouette determination of all edges in a mesh. This significantly speeds up both the transformation and silhouette computation steps, requiring only $O(\log n)$ operations for each mesh to determine which edges are silhouettes.

The method works by clustering faces that will behave similarly under transformation. If a representative face from a given cluster has a normal in a certain range, the cluster can be considered to be entirely front-facing or entirely back-facing. We have a slightly different goal than Kumar, in that we wish to find silhouette edges that are on the boundary between front and back facing, but this is a rather simple modification to the algorithm. Altogether, this allows a $O(\log n)$ traversal of a hierarchy of clusters, where we only must transform a representative at each step to determine the facing of the cluster. This can improve overall performance significantly, since a linear search to compute silhouette edges often dominates the entire computation.

This method was also not implemented in our test renderer, but its performance on backface culling indicates that the speedup to this algorithm would be great. For a complex mesh of 5804 faces, our implementation spent more than 55% of overall execution time testing for silhouette edges. (Nearly 90% of these edges were eliminated.) We speculate that this fraction of runtime would be reduced substantially by implementing hierarchical silhouette computation.

4.5 Implementation Details

This section describes some additional empirical results and observations, as well as some optional optimizations and user interface suggestions.

4.5.1 Sampling

In the implementation as described, all light sources are parallelograms. Sampling patterns for these objects [11] show that uniform sampling produces visually worse results than samples on a jittered grid across a parallelogram emitter. However, it is necessary to fix a jittered grid before computing textures, or the visible artifacts of changing sample distributions will be distracting when textures are animated.

Importance sampling, or distributing samples where they do the most good, (in this case, where they have equal intensity contribution) has some difficulties with the described algorithm. For example, in the case of large receivers, the importance sampling algorithm has no way to change the sampling based on position on the receiver polygon, which means it must pick an "average" best fit. Also, for changes in the position of the receiver or of the emitter, the re-organization of sample points can cause distracting temporal artifacts.

The number of samples required to produce a realistic result depends primarily on the size of the penumbra in texture space. A penumbra is generally large for occluders close to a light source and also for large light sources. In general, high-quality results can be computed with 30-60 samples for reasonably sized lights. For more extreme geometries, as many as 200 samples may be required for high-quality texture maps.

The tradeoff between quality and speed can be easily extended to a user interface, where the software can maintain a desired frame rate by reducing the number of light samples dynamically, producing progressively higher-quality texture maps when the scene is static.

Since hard shadows are drawn from several very close points in space, we might allow the option of not recomputing the silhouette edges for every light sample point, instead re-using these edges for multiple samples. This can have very bad results in some cases, but for interaction, it can provide a lower-quality image more quickly. This optimization was an user-determined option for the test implementation done of this project.

4.5.2 Better Memory Allocation

At one point, a tiling scheme for the list allocator was implemented, in hopes that the locality of memory accesses would be improved. This attempt did not yield any speed improvements, and made much less efficient use of memory. Memory access patterns are very bad for the described list-based scheme, and this is an area for future improvement.

4.5.3 Curved Objects

The method presented above can provide realistic shadowing for planar surfaces. A natural extension is to produce shadows for curved or faceted surfaces. Segal et al. [27] introduced an efficient method for projecting textures to curved surfaces, where a planar shadow texture can be projected onto a curved surface. For soft shadows, this method produces incorrect shadows, since it re-projects the texture from a point instead of an area. This method can produce “blurred” hard shadows, but correct soft shadows are not possible without further effort.

Chapter 5

Results and Conclusions

5.1 Speed

Of the above optimizations, the edge-flag and linear-time silhouette determination algorithms were implemented, using the memory allocation described above. Hierarchical bounding hierarchies for frustum culling and hierarchical silhouette determination were not implemented.

The results were quite good. Regeneration of 256x256 shadow textures for a single receiver and several hundred occluders with 64 light samples happens at 30 Hz on an Intel Pentium Pro at 200 MHz. This was an unexpected improvement over the earlier Reality Engine implementation, which achieves much lower speeds (5 Hz) using the hardware accumulation buffer. It is expected that further effort could enable regeneration of several textures per frame while amortizing regeneration cost for a scene over several frames.

5.2 Complexity

As was mentioned above, efficient frustum culling of receivers allows this algorithm to scale reasonably well. Hopefully, we do average $O(n^{.33})$ for each texture, giving a texture generation time of $O(n^{1.33})$ for n self-shadowed polygons in the scene. In the worst case, this algorithm scales as $O(n^2)$. Since our implementation did not do frustum culling, we saw performance scale with $O(n^2)$.

5.3 Generality

The optimizations described for this algorithm do restrict it to single-colored light sources, a limitation that can be alleviated by computing multiple passes of color and compositing the final textures as a post-process. (The simplest general case would be red, green, and blue.) Also, arbitrary geometries will produce incorrect, though plausible, illumination due to shadowing because of the equal-contribution assumption for light sample points.

This method is effective at producing realistic shadows for large planar surfaces, especially if the computation for all surfaces in a scene scales with $O(n^{1.33})$ (which theoretically would happen with proper culling). The curved surface problem remains a hard one to solve. Also, rendering shadows for a large number of polygons does not seem to be the most appropriate use of this algorithm. It is recommended that several "important" surfaces be chosen to have realistic shadows generated. This could be automatically prioritized by screen area, perhaps.

5.4 Conclusions

The described algorithm makes large speed improvements over other methods that generate soft shadows. It perhaps most easily serves as a replacement for Blinn's method and several of the other simple schemes for producing real-time hard shadows. It is relatively easy to implement, but some of the optimizations require a certain attention to detail.

This algorithm combines ideas from several places, including the z-buffer shadow algorithm, radiosity computations, Blinn's method of flattening shadows to a plane, as well as computational geometry literature and old frame buffer and memory allocation tricks.

5.5 Future Work

Much more work needs to be done in this area, including a generalization of this technique to curved surfaces, and better use of known techniques for sampling area light sources [28]. Also, the application of better spatial-subdivision schemes could also help the efficiency of the computation.

5.6 Acknowledgements

Thanks go first to my advisor, Paul S. Heckbert. He has been very patient with my various forays over the last two years, and I appreciate his guidance and knowledge of the field. Also, his amazing skills at pulling together large bibliographies have made my job much easier.

Thanks also to Andrew Willmott, for his rendering tips and code snippets, and to Michael Garland for meshes I used. Also, to Dinesh Manocha, for pointing out the applicability of his silhouette ideas to this problem and to Chris Hecker, for interesting discussions about code optimization and graphics in general. Also, to Allen Akin for suggestions and tips on OpenGL limitations and features on SGI machines. Finally, to Peter Shirley, for complexity and sampling discussions.

Bibliography

[1] Bryan D. Ackland and Neil H. Weste. The edge flag algorithm — a fill method for raster scan displays. *IEEE Trans. on Comp.*, C-30:41–47, 1981.

[2] Arthur Appel. The notion of quantitative invisibility and the machine rendering of solids. *Proc. ACM Natl. Mtg.*, pages 387–393, 1967.

[3] James Arvo. Applications of irradiance tensors to the simulation of non-lambertian phenomena. In *SIGGRAPH 95 Proceedings*, pages 335–342, August 1995.

[4] James F. Blinn. Me and my (fake) shadow. *IEEE Computer Graphics and Applications*, 8(1):82–86, Jan. 1988.

[5] Lynne Shapiro Brotman and Norman I. Badler. Generating soft shadows with a depth buffer algorithm. *IEEE Computer Graphics and Applications*, 4(10):5–24, Oct. 1984.

[6] Shenchang Eric Chen. Incremental radiosity: An extension of progressive radiosity to an interactive image synthesis system. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):135–144, August 1990.

[7] Norman Chin and Steven Feiner. Fast object-precision shadow generation for area light sources using BSP trees. In *1992 Symp. on Interactive 3D Graphics*, pages 21–30. ACM SIGGRAPH, Mar. 1992.

[8] James H. Clark. Hierarchical geometric models for visible surface algorithms. *CACM*, 19(10):547–554, Oct. 1976.

[9] John G. Cleary and Geoff Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer*, 4(2):65–83, July 1988.

[10] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press, Boston, 1993.

[11] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Trans. on Graphics*, 5(1):51–72, Jan. 1986.

[12] Paul J. Diefenbach and Norman I. Badler. Pipeline rendering: Interactive refractions, reflections, and shadows. *Displays*, 15(3):173–180, 1994. http://www.cis.upenn.edu/ diefenba/home.html.

[13] George Drettakis and Eugene Fiume. A fast shadow algorithm for area light sources using backprojection. In *SIGGRAPH '94 Proc.*, pages 223–230, 1994. http://safran.imag.fr/Membres/George.Drettakis/pub.html.

[14] Henry Fuchs, Jack Goldfeather, Jeff P. Hultquist, Susan Spach, John D. Austin, Frederick P. Brooks, Jr., John G. Eyles, and John Poulton. Fast spheres, shadows, textures, transparencies, and

image enhancements in Pixel-Planes. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):111–120, July 1985.

[15] Paul Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):309–318, Aug. 1990.

[16] Eric Haines and John Wallace. Shaft culling for efficient ray-traced radiosity. In *Eurographics Workshop on Rendering*, may 1991.

[17] Paul S. Heckbert. Generic convex polygon scan conversion and clipping. In Andrew Glassner, editor, *Graphics Gems I*. Academic Press, Boston, 1990.

[18] Paul S. Heckbert and Michael Herf. Simulating soft shadows with graphics hardware. Technical report, CS Dept., Carnegie Mellon U., Jan. 1997. CMU-CS-97-104, http://www.cs.cmu.edu/ ph.

[19] Tim Heidmann. Real shadows, real time. *Iris Universe*, 18:28–31, 1991. Silicon Graphics, Inc.

[20] Michael Herf and Paul S. Heckbert. Fast soft shadows. In *Visual Proceedings, SIGGRAPH 96*, page 145, Aug. 1996.

[21] Subodh Kumar, Dinesh Manocha, Bill Garrett, and Ming Lin. Hierarchical back-face computation. In *Eurographics '96*, June 1996.

[22] Tomas Möller. Speed-up techniques for soft shadow generation. In *SMCR'97 - Topical Workshop on VR and Advanced Man-Machine Interfaces*, Tampere, Finland, June 1997.

[23] Karol Myszkowski and Tosiyasu L. Kunii. Texture mapping as an alternative for meshing during walkthrough animation. In *Fifth Eurographics Workshop on Rendering*, pages 375–388, June 1994.

[24] A. Peleg, S. Wilkie, and U. Weisner. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1), 1997.

[25] Steven Przybylski. New DRAMS improve bandwidth (part 1). *Microprocessor Reports*, 7(2), 1993.

[26] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):283–291, July 1987.

[27] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):249–252, July 1992.

[28] Peter Shirley. *Physically Based Lighting Calculations for Computer Graphics*. Ph.D. thesis, Dept. of Computer Science, U. of Illinois, Urbana-Champaign, November 1990.

[29] Toshimitsu Tanaka and Tokiichiro Takahashi. Fast analytic shading and shadowing for area light sources. 1996. Submitted for publication.

[30] Andrew Woo, Pierre Poulin, and Alain Fournier. A survey of shadow algorithms. *IEEE Computer Graphics and Applications*, 10(6):13–32, Nov. 1990.

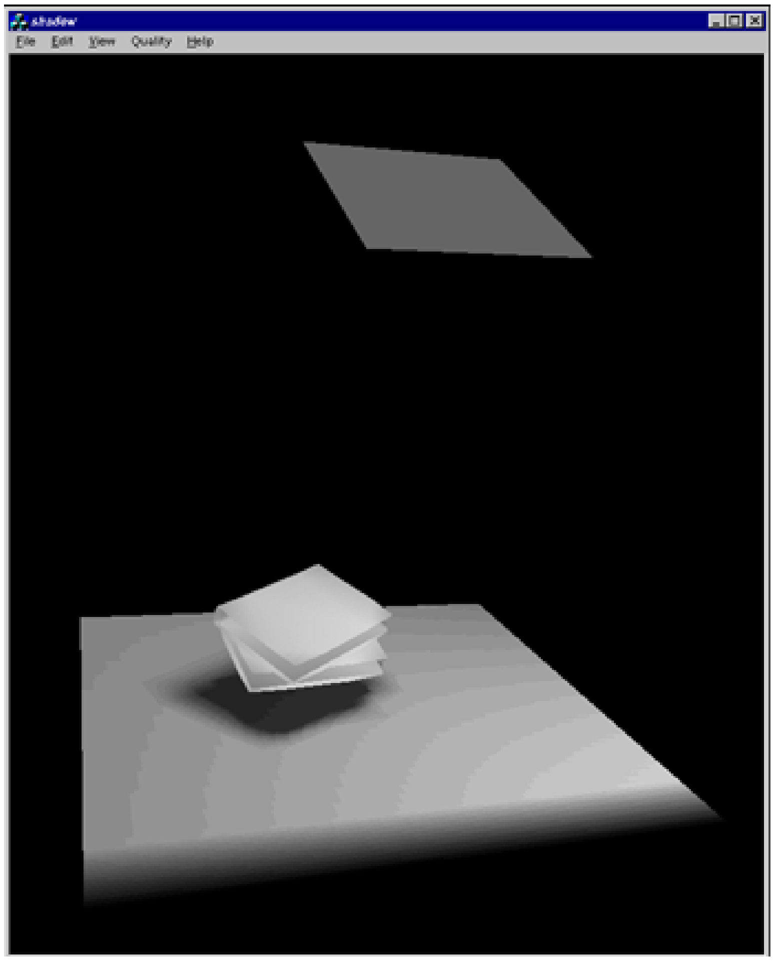

Figure 5.1: *A screenshot from an interactive version of the program with a simple scene, running at 30 Hz on a 200-MHz Intel Pentium-Pro processor.*

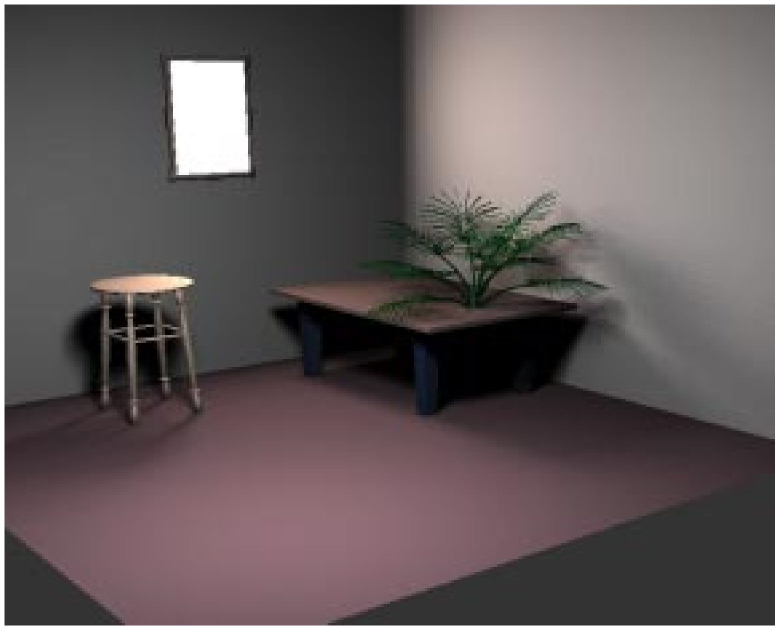

Figure 5.2: *Output from the hardware version of the program, generated in 10 seconds on an Iris Crimson R4000 with RealityEngine graphics.*