Simulating Soft Shadows with Graphics Hardware

Paul S. Heckbert and Michael Herf

January 15, 1997
CMU-CS-97-104

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

email: ph@cs.cmu.edu, herf+@cmu.edu
World Wide Web: http://www.cs.cmu.edu/∼ph

This paper was written in April 1996. An abbreviated version appeared in [Michael Herf and Paul S. Heckbert, Fast Soft Shadows, *Visual Proceedings, SIGGRAPH 96*, Aug. 1996, p. 145].

Abstract

This paper describes an algorithm for simulating soft shadows at interactive rates using graphics hardware. On current graphics workstations, the technique can calculate the soft shadows cast by moving, complex objects onto multiple planar surfaces in about a second. In a static, diffuse scene, these high quality shadows can then be displayed at 30 Hz, independent of the number and size of the light sources.

For a diffuse scene, the method precomputes a *radiance texture* that captures the shadows and other brightness variations on each polygon. The texture for each polygon is computed by creating registered projections of the scene onto the polygon from multiple sample points on each light source, and averaging the resulting hard shadow images to compute a soft shadow image. After this precomputation, soft shadows in a static scene can be displayed in real-time with simple texture mapping of the radiance textures. All pixel operations employed by the algorithm are supported in hardware by existing graphics workstations. The technique can be generalized for the simulation of shadows on specular surfaces.

This work was supported by NSF Young Investigator award CCR-9357763. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF or the U.S. government.

Keywords: penumbra, texture mapping, graphics workstation, interaction, real-time, SGI Reality Engine.

1 Introduction

Shadows are both an important visual cue for the perception of spatial relationships and an essential component of realistic images. Shadows differ according to the type of light source causing them: point light sources yield hard shadows, while linear and area (also known as *extended*) light sources generally yield soft shadows with an umbra (fully shadowed region) and penumbra (partially shadowed region).

The real world contains mostly soft shadows due to the finite size of sky light, the sun, and light bulbs, yet most computer graphics rendering software simulates only hard shadows, if it simulates shadows at all. Excessive sharpness of shadow edges is often a telltale sign that a picture is computer generated.

Shadows are even less commonly simulated with hardware rendering. Current graphics workstations, such as Silicon Graphics (SGI) and Hewlett Packard (HP) machines, provide z-buffer hardware that supports real-time rendering of fairly complex scenes. Such machines are wonderful tools for computer aided design and visualization. Shadows are seldom simulated on such machines, however, because existing algorithms are not general enough, or they require too much time or memory. The shadow algorithms most suitable for interaction on graphics workstations have a cost per frame proportional to the number of point light sources. While such algorithms are practical for one or two light sources, they are impractical for a large number of sources or the approximation of extended sources.

We present here a new algorithm that computes the soft shadows due to extended light sources. The algorithm exploits graphics hardware for fast projective (perspective) transformation, clipping, scan conversion, texture mapping, visibility testing, and image averaging. The hardware is used both to compute the shading on the surfaces and to display it, using texture mapping. For diffuse scenes, the shading is computed in a preprocessing step whose cost is proportional to the number of light source samples, but while the scene is static, it can be redisplayed in time independent of the number of light sources. The method is also useful for simulating the hard shadows due to a large number of point sources. The memory requirements of the algorithm are also independent of the number of light source samples.

1.1 The Idea

For diffuse scenes, our method works by precomputing, for each polygon in the scene, a *radiance texture* [12,14] that records the color (outgoing radiance) at each point in the polygon. In a diffuse scene, the radiance at each surface point is view independent, so it can be precomputed and re-used until the scene geometry changes. This radiance texture is analogous to the mesh of radiosity values computed in a radiosity algorithm. Unlike a radiosity algorithm, however, our algorithm can compute this texture almost entirely in hardware.

The key idea is to use graphics hardware to determine visibility and calculate shading, that is, to determine which portions of a surface are occluded with respect to a given extended light source, and how brightly they are lit. In order to simulate extended light sources, we approximate them with a number of *light sample points*, and we do visibility tests between a given surface point and each light sample. To keep as many operations in hardware as possible, however, we do not use a hemicube [7] to determine visibility. Instead, to compute the shadows for a single polygon, we render the scene into a scratch buffer, with all polygons except the one being shaded appropriately blackened, using a special projective projection from the point of view of each light sample. These views are registered so that corresponding pixels map to identical points on the polygon. When the resulting hard shadow images are averaged, a soft shadow image results (figure 1). This image is then used directly as a texture on the polygon in order to simulate shadows correctly. The textures so computed are used for real-time display until the scene geometry changes.

In the remainder of the paper, we summarize previous shadow algorithms, we present our method for diffuse scenes in more detail, we discuss generalizations to scenes with specular and general reflectance, we present our implementation and results, and we offer some concluding remarks.

2 Previous Work

2.1 Shadow Algorithms

Woo *et al.* surveyed a number of shadow algorithms [19]. Here we summarize soft shadows methods and methods that run at interactive rates. Shadow algorithms can be divided into three categories: those that compute everything on the fly, those that precompute just visibility, and those that precompute shading.

Computation on the Fly. Simple ray tracing computes everything on the fly. Shadows are computed on a point-by-point basis by tracing rays between the surface point and a point on each light source to check for occluders. Soft shadows can be simulated by tracing rays to a number of points distributed across the light source [8].

The shadow volume approach is another method for computing shadows on the fly. With this method, one constructs imaginary surfaces that bound the shadowed volume of space with respect to each point light source. Determining if a point is in shadow then reduces to point-in-volume testing. Brotman and Badler used an extended z-buffer algorithm with linked lists at each pixel to support soft shadows using this approach [4].

The shadow volume method has also been used in two hardware implementations. Fuchs *et al.* used the pixel processors of the Pixel Planes machine to simulate hard shadows in real-time [10]. Heidmann used the stencil buffer in advanced SGI machines [13]. With Heidmann's algorithm, the scene must be rendered through the stencil created from each light source, so the cost per frame is proportional to the number of light sources times the number of polygons. On 1991 hardware, soft shadows in a fairly simple scene required several seconds with his algorithm. His method appears to be one of the algorithms best suited to interactive use on widely available graphics hardware. We would prefer, however, an algorithm whose cost is sublinear in the number of light sources.

A simple, brute force approach, good for casting shadows of objects onto a plane, is to find the projective transformation that projects objects from a point light onto a plane, and to use it to draw each squashed, blackened object on top of the plane [3], [15, p. 401]. This algorithm effectively multiplies the number of objects in the scene by the number of light sources times the number of *receiver* polygons onto which shadows are being cast, however, so it is typically practical only for very small numbers of light sources and receivers. Another problem with this method is that occluders behind the receiver will cast erroneous shadows, unless extra clipping is done.

Precomputation of Visibility. Instead of computing visibility on the fly, one can precompute visibility from the point of view of each light source.

The z-buffer shadow algorithm uses two (or more) passes of z-buffer rendering, first from the light sources, and then from the eye [18]. The z-buffers from the light views are used in the final

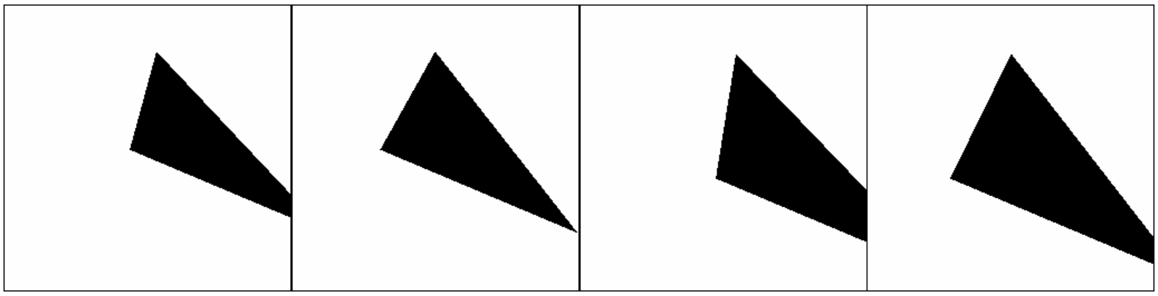

Figure 1: Hard shadow images from 2×2 grid of sample points on light source.

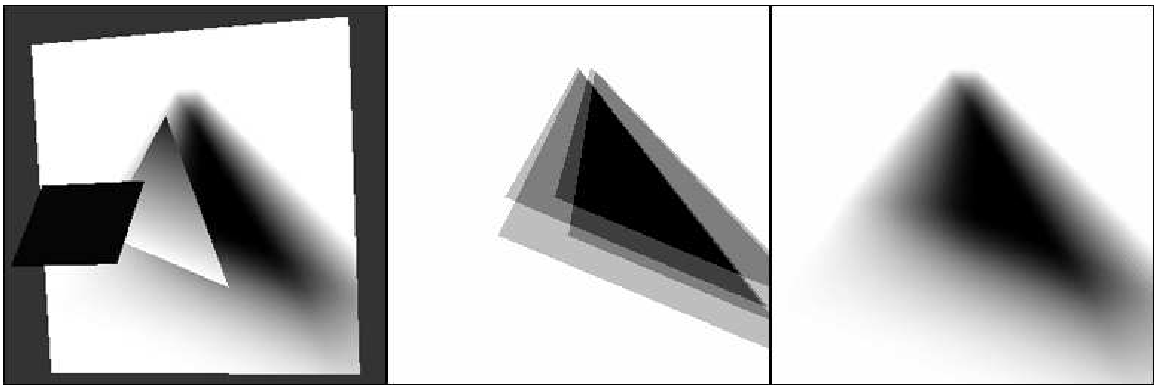

Figure 2: Left: scene with square light source (foreground), triangular occluder (center), and rectangular receiver (background), with shadows on receiver. Center: Approximate soft shadows resulting from 2×2 grid of sample points; the average of the four hard shadow images in Figure 1. Right: Correct soft shadow image (generated with 16×16 sampling). This image is used as the texture on the receiver at left.

pass to determine if a given 3-D point is illuminated with respect to each light source. The transformation of points from one coordinate system to another can be accelerated using texture mapping hardware [17]. This latter method, by Segal *et al.*, achieves real-time rates, and is the other leading method for interactive shadows. Soft shadows can be generated on a graphics workstation by rendering the scene multiple times, using different points on the extended light source, averaging the resulting images using accumulation buffer hardware [11].

A variation of the shadow volume approach is to intersect these volumes with surfaces in the scene to precompute the umbra and penumbra regions on each surface [16]. During the final rendering pass, illumination integrals are evaluated at a sparse sampling of pixels.

Precomputation of Shading. Precomputation can be taken further, computing not just visibility but also shading. This is most relevant to diffuse scenes, since their shading is view-independent. Some of these methods compute visibility continuously, while others compute it discretely.

Several researchers have explored continuous visibility methods for soft shadow computation and radiosity mesh generation. With this approach, surfaces are subdivided into fully lit, penumbra, and umbra regions by splitting along lines or curves where visibility changes. In Chin and Feiner's soft shadow method, polygons are split using BSP trees, and these sub-polygons are then pre-shaded [6]. They achieved rendering times of under a minute for simple scenes. Drettakis and Fiume used more sophisticated computational geometry techniques to precompute their subdivision, and reported rendering times of several seconds [9].

Most radiosity methods discretize each surface into a mesh of elements and then use discrete methods such as ray tracing or hemicubes to compute visibility. The hemicube method computes visibility from a light source point to an entire hemisphere by projecting the scene onto a half-cube [7]. Much of this computation can be done in hardware. Radiosity meshes typically do not resolve shadows well, however. Typical artifacts are Mach bands along the mesh element boundaries and excessively blurry shadows. Most radiosity methods are not fast enough to support interactive changes to the geometry, however. Chen's incremental radiosity method is an exception [5].

Our own method can be categorized next to hemicube radiosity methods, since it also precomputes visibility discretely. Its technique for computing visibility also has parallels to the method of flattening objects to a plane.

2.2 Graphics Hardware

Current graphics hardware, such as the Silicon Graphics Reality Engine [1], can projective-transform, clip, shade, scan convert, and texture tens of thousands of polygons in real-time (in 1/30 sec.). We would like to exploit the speed of this hardware to simulate soft shadows.

Typically, such hardware supports arbitrary 4×4 homogeneous transformations of planar polygons, clipping to any truncated pyramidal frustum (right or oblique), and scan conversion with z-buffering or overwriting. On SGI machines, Phong shading (once per pixel) is not possible, but faceted shading (once per polygon) and Gouraud shading (once per vertex) are supported. Phong shading can be simulated by splitting polygons into small pieces on input. A common, general form for hardware-supported illumination is diffuse reflection from multiple point spotlight sources, with a texture mapped reflectance function and attenuation:

$$I_c(x,y) = T_c(u,v) \sum_l \frac{\cos\theta_l \cos\theta_l'^{e}\, L_{lc}}{\alpha + \beta r_l + \gamma r_l^2}$$

where c is color channel index (= r, g, or b), $I_c(x,y)$ is the pixel value at screen space (x,y), $T_c(u,v)$ is a texture parameterized by texture coordinates (u,v), which are a projective transform of (x,y), θ_l is the polar angle for the ray to light source l, θ_l' is the angle away from the directional axis of the light source, e is the spotlight exponent, L_{lc} is the radiance of light l, r_l is distance to light source l, and α, β, and γ are constants controlling attenuation. Texture mapping, lights, and attenuation can be turned on and off independently on a per-polygon basis. Most systems also support Phong illumination, which has an additional specular term that we have not shown. The most advanced, expensive machines support all of these functions in hardware, while the cheaper machines do some of these calculations in software. Since the graphics subroutine interface, such as OpenGL [15], is typically identical on any machine, these differences are transparent to the user, except for the dramatic differences in running speed. So when we speak of a computation being done "in hardware", that is true only on high end machines.

The accumulation buffer [11], another feature of some graphics workstations, is hardware that allows a linear combination of images to be easily computed. It is capable of computing expressions of the general form:

$$A_c(x,y) = \sum_i \alpha_i I_{ic}(x,y)$$

where I_{ic} is a channel of image i, and A_c is a channel of the accumulator array.

3 Diffuse Scenes

Our shadow generation method for diffuse scenes takes advantage of these hardware capabilities.

Direct illumination in a scene of opaque surfaces that emit or reflect light diffusely is given by the following formula:

$$L_c(\mathbf{x}) = \rho_c(\mathbf{x}) \left(L_{\text{ac}} + \int_{\text{lights}} \frac{\cos_+\theta \cos_+\theta'}{\pi r^2}\, v(\mathbf{x},\mathbf{x}')\, L_c(\mathbf{x}')\, d\mathbf{x}' \right),$$

where, as shown in Figure 3,

- $\mathbf{x} = (x,y,z)$ is a 3-D point on a reflective surface, and $\mathbf{x}'$ is a point on a light source,
- θ is polar angle (angle from normal) at $\mathbf{x}$, θ' is the angle at $\mathbf{x}'$,
- r is the distance between $\mathbf{x}$ and $\mathbf{x}'$,
- θ, θ', and r are functions of $\mathbf{x}$ and $\mathbf{x}'$,
- $L_c(\mathbf{x})$ is outgoing radiance at point $\mathbf{x}$ for color channel c, due to either emission or reflection, L_{ac} is ambient radiance,
- $\rho_c(\mathbf{x})$ is reflectance,
- $v(\mathbf{x},\mathbf{x}')$ is a Boolean visibility function that equals 1 if point $\mathbf{x}$ is visible from point $\mathbf{x}'$, else 0,
- $\cos_+\theta = \max(\cos\theta, 0)$, for backface testing, and
- the integral is over all points on all light sources, with respect to $d\mathbf{x}'$, which is an infinitesimal area on a light source.

The inputs to the problem are the geometry, the reflectance $\rho_c(\mathbf{x})$, and emitted radiance $L_c(\mathbf{x}')$ on all light sources, the ambient radiance L_{ac}, and the output is the reflected radiance function $L_c(\mathbf{x})$.

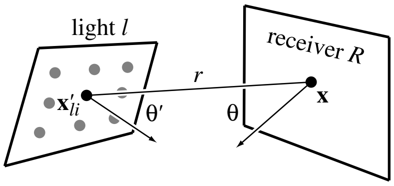

Figure 3: Geometry for direct illumination. The radiance at point $\mathbf{x}$ on the receiver is being calculated by summing the contributions from a set of point light sources at $\mathbf{x}'_{li}$ on light l.

3.1 Approximating Extended Light Sources

Although such integrals can be solved in closed form for planar surfaces with no occlusion ($v \equiv 1$), the complexity of the visibility function makes these integrals intractable in the general case. We can compute approximations to the integral, however, by replacing each extended light source l by a set of n_l point light sources:

$$L_c(\mathbf{x}') \approx \sum_l \sum_{i=1}^{n_l} a_{li}\, L_c(\mathbf{x}')\, \delta(\mathbf{x}' - \mathbf{x}'_{li}),$$

where $\delta(\mathbf{x})$ is a 3-D Dirac delta function, $\mathbf{x}'_{li}$ is sample point i on light source l, and a_{li} is the area associated with this sample point. Typically, each sample on a light source has equal area: $a_{li} = a_l/n_l$, where a_l is the area of light source l.

With this approximation, the radiance of a reflective surface point can be computed by summing the contributions over all sample points on all light sources:

$$L_c(\mathbf{x}) = \rho_c(\mathbf{x})\, L_{\text{ac}} + \rho_c(\mathbf{x}) \sum_l \sum_{i=1}^{n_l} a_{li} \frac{\cos_+\theta_{li} \cos_+\theta'_{li}}{\pi r_{li}^2}\, v(\mathbf{x},\mathbf{x}'_{li})\, L_c(\mathbf{x}'_{li}). \tag{1}$$

The formulas above can be generalized to linear and point light sources, as well as area light sources.

The most difficult and expensive part of the above calculation is evaluation of the visibility function v, since it requires global knowledge of the scene, whereas the remaining factors require only local knowledge. But computing v is necessary in order to simulate shadows. The above formula could be evaluated using ray tracing, but the resulting algorithm would be slow due to the large number of light source samples.

3.2 Soft Shadows in Hardware

Equation (1) can be rewritten in a form suitable to hardware computation:

$$L_c(\mathbf{x}) = \rho_c(\mathbf{x})\, L_{\text{ac}} + \sum_l \sum_{i=1}^{n_l} \big(a_{li}\, \rho_c(\mathbf{x})\big) \left(\frac{\cos_+\theta_{li} \cos_+\theta'_{li}\, L_c(\mathbf{x}'_{li})}{\pi r_{li}^2} \right) v(\mathbf{x},\mathbf{x}'_{li}). \tag{2}$$

Each term in the inner summation can be regarded as a hard shadow image resulting from a point light source at $\mathbf{x}'_{li}$, where $\mathbf{x}$ is a function of screen (x,y).

That summand consists of the product of three factors. The first one, which is an area times the reflectance of the receiving polygon, can be calculated in software. The second factor is the cosine of the angle on the receiver, times the cosine of the angle on the light

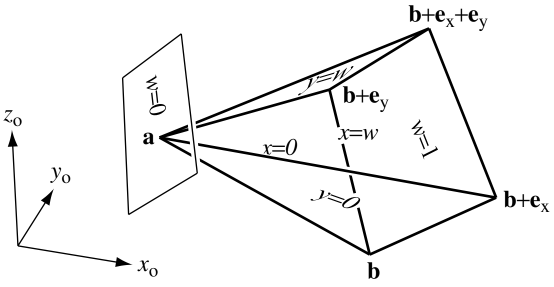

Figure 4: Pyramid with parallelogram base. Faces of pyramid are marked with their plane equations.

source, times the radiance of the light source, divided by r^2. This can be computed in hardware by rendering the receiver polygon with a single spotlight at $\mathbf{x}'_{li}$ turned on, using a spotlight exponent of $e = 1$ and quadratic attenuation. On machines that do not support Phong shading, we will have to finely subdivide the polygon. The third factor is visibility between a point on a light source and each point on the receiver. Visibility can be computed by projecting all polygons between light source point $\mathbf{x}'_{li}$ and the receiver onto the receiver.

We want to simulate soft shadows as quickly as possible. To take full advantage of the hardware, we can precompute the shading for each polygon using the formula above, and then display views of the scene from moving viewpoints using real-time texture mapping and z-buffering.

To compute soft shadow textures, we need to generate a number of hard shadow images and then average them. If these hard shadow images are not registered (they would not be, using hemi-cubes), then it would be necessary to resample them so that corresponding pixels in each hard shadow image map to the same surface point in 3-D. This would be very slow. A faster alternative is to choose the transformation for each projection so that the hard shadow images are perfectly registered with each other.

For planar receiver surfaces, this is easily accomplished by exploiting the capabilities of projective transformations. If we fit a parallelogram around the receiver surface of interest, and then construct a pyramid with this as its base and the light point as its apex, there is a 4×4 homogeneous transformation that will map such a pyramid into an axis-aligned box, as described shortly.

The hard shadow image due to sample point i on light l is created by loading this special transformation matrix and rendering the receiver polygon. The polygon is illuminated by the ambient light plus a single point light source at $\mathbf{x}'_{li}$, using Phong shading or a good approximation to it. The visibility function is then computed by rendering the remainder of the scene with all surfaces shaded as if they were the receiver illuminated by ambient light: $(r, g, b) = (\rho_{\mathrm{r}} L_{\mathrm{ar}}, \rho_{\mathrm{g}} L_{\mathrm{ag}}, \rho_{\mathrm{b}} L_{\mathrm{ab}})$. This is most quickly done with z-buffering off, and clipping to a pyramid with the receiver polygon as its base. Drawing each polygon with an unsorted painter's algorithm suffices here because all polygons are the same color, and after clipping, the only polygon fragments remaining will lie between the light source and the receiver, so they all cast shadows on the receiver. To compute the weighted average of the hard shadow images so created, we use the accumulation buffer.

3.3 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. 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 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 4.

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 4). The result is:

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

where

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

Blinn [3] uses a related projective transformation for the generation of shadows on a plane, but his is a projection (it collapses 3-D to 2-D), while ours is 3-D to 3-D. We use the third dimension for clipping.

3.4 Using the Transformation

To use this transformation in our shadow algorithm, we first fit a parallelogram around the receiver polygon. If the receiver is a rectangle or other parallelogram, the fit is exact; if the receiver is a triangle, then we fit the triangle into the lower left triangle of the parallelogram; and for more general polygons with four or more sides, a simple 2-D bounding box in the plane of the polygon can be used. It is possible to go further with projective transformations, mapping arbitrary planar quadrilaterals into squares (using the homogeneous texture transformation matrix of OpenGL, for example). We assume for simplicity, however, that the transformation between texture space (the screen space in these light source projections) and object space is affine, and so we restrict ourselves to parallelograms.

3.5 Soft Shadow Algorithm for Diffuse Scenes

To precompute soft shadow radiance textures:

```
turn off z-buffering
for each receiver polygon R
  choose resolution for receiver's texture (s_x × s_y pixels)
  clear accumulator image of s_x × s_y pixels to black
  create temporary image of s_x × s_y pixels
  for each light source l
    first backface test: if l is entirely behind R
      or R is entirely behind l, then skip to next l
    for each sample point i on light source l
      second backface test: if x'_li is behind R then skip to next i
      compute transformation matrix M, where a = x'_li,
        and the base parallelogram fits tightly around R
      set current transformation matrix to scale(s_x, s_y, 1)·M
      set clipping planes to z_u,near = 1 − ε and z_u,far = big
      draw R with illumination from x'_li only, as described in
        equation (2), into temp image
      for each other object in scene
        draw object with ambient color into temp image
      add temp image into accumulator image with weight a_l/n_l
  save accumulator image as texture for polygon R
```

A hard shadow image is computed in each iteration of the i loop. These are averaged together to compute a soft shadow image, which is used as a radiance texture. Note that objects casting shadows need not be polygonal; any object that can be quickly scan converted will work well.

To display a static scene from moving viewpoints, simply:

```
turn on z-buffering
for each object in scene
  if object receives shadows, draw it textured but without illumination
  else draw object with illumination
```

3.6 Backface Testing

The cases where $\cos_+\theta \cos_+\theta' = 0$ can be optimized using backface testing.

To test if polygon p is behind polygon q, compute the signed distances from the plane of polygon q to each of the vertices of p (signed positive on the front of q and negative on the back). If they are all positive, then p is entirely in front of q, if they are all nonpositive, p is entirely in back, otherwise, part of p is in front of q and part is in back.

To test if the apex $\mathbf{a}$ of the pyramid is behind the receiver R that defines the base plane, simply test if $\mathbf{n}_\mathrm{w} \cdot \mathbf{e}_\mathrm{w} \leq 0$.

The above checks will ensure that $\cos\theta > 0$ at every point on the receiver, but there is still the possibility that $\cos\theta' \leq 0$ on portions of the receiver (i.e. that the receiver is only partially illuminated by the light source). This final case should be handled at the polygon level or pixel level when shading the receiver in the algorithm above. Phong shading, or a good approximation to it, is needed here.

3.7 Sampling Extended Light Sources

The set of samples used on each light source greatly influences the speed and quality of the results. Too few samples, or a poorly chosen sample distribution, result in penumbras that appear stepped, not continuous. If too many samples are used, however, the simulation runs too slowly.

If a uniform grid of sample points is used, the stepping is much more pronounced in some cases. For example, if a uniform grid of $m \times m$ samples is used on a parallelogram light source, an occluder edge coplanar with one of the light source edges will cause m big steps, while an occluder edge in general position will cause m^2 small steps.

Stochastic sampling [8] with the same number of samples yields smoother penumbra than a uniform grid, because the steps no longer coincide. We use a jittered uniform grid because it gives good results and is very easy to compute.

Using a fixed number of samples on each light source is inefficient. Fine sampling of a light source is most important when the light source subtends a large solid angle from the point of view of the receiver, since that is when the penumbra is widest and stepping artifacts would be most visible. A good approach is to choose the light source sample resolution such that the solid angle subtended by the light source area associated with each sample is below a user-specified threshold.

The algorithm can easily handle diffuse (non-directional) light sources whose outgoing radiance varies with position, such as stained glass windows. For such light sources, importance sampling might be preferable: concentration of samples in the regions of the light source with highest radiance.

3.8 Texture Resolution

The resolution of the shadow texture should be roughly equal to the resolution at which it will be viewed (one texture pixel mapping to one screen pixel); lower resolution results in visible artifacts such as blocky shadows, and higher resolution is wasteful of time and memory. In the absence of information about probable views, a reasonable technique is to set the number of pixels on a polygon's texture, in each dimension, proportional to its size in world space using a "desired pixel size" parameter. With this scheme, the required texture memory, in pixels, will be the total world space surface area of all polygons in the scene divided by the square of the desired pixel size.

Texture memory for triangles can be further optimized by packing the textures for two triangles into one rectangular texture block.

If there are too many polygons in the scene, or the desired pixel size is too small, the texture memory could be exceeded, causing paging of texture memory and slow performance.

Radiance textures can be antialiased by supersampling: generating the hard and initial soft shadow images at several times the desired resolution, and then filtering and downsampling the images before creating textures. Textured surfaces should be rendered with good texture filtering.

Some polygons will contain penumbral regions with respect to a light source, and will require high texture resolution, but others will be either totally shadowed (umbral) or totally illuminated by each light source, and will have very smooth radiance functions. Sometimes these functions will be so smooth that they can be adequately approximated by a single Gouraud shaded polygon. This optimization saves significant texture memory and speeds display.

This idea can be carried further, replacing the textured planar polygon with a mesh of coplanar Gouraud shaded triangles. For complex shadow patterns and radiance functions, however, textures may render faster than the corresponding Gouraud approximation, depending on the relative speed of texture mapping and Gouraud-shaded triangle drawing, and the number of triangles required to achieve a good approximation.

3.9 Complexity

We now analyze the expected complexity of our algorithm (worst case costs are not likely to be observed in practice, so we do not discuss them here). Although more sophisticated schemes are possible, we will assume for the purposes of analysis that the same set

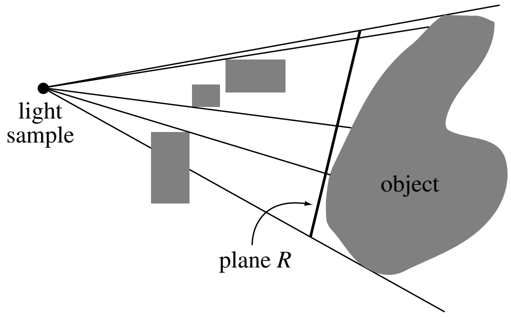

Figure 5: Shadows are computed on plane R and projected onto the receiving object at right.

of light samples are used for shadowing all polygons. Suppose we have a scene with s surfaces (polygons), a total of $n = \sum_l n_l$ light source samples, a total of t radiance texture pixels, and the output images are rendered with p pixels. We assume the depth complexity of the scene (the average number of surfaces intersecting a ray) is bounded, and that t and p are roughly linearly related. The average number of texture pixels per polygon is t/s.

With our technique, preprocessing renders the scene ns times. A painter's algorithm rendering of s polygons into an image of t/s pixels takes $O(s+t/s)$ time for scenes of bounded depth complexity. The total preprocessing time is thus $O(ns^2+nt)$, and the required texture memory is $O(t)$. Display requires only z-buffered texture mapping of s polygons to an image of p pixels, for a time cost of $O(s+p)$. The memory for the z-buffer and output image is $O(p)=O(t)$.

Our display algorithm is very fast for complex scenes. Its cost is independent of the number of light source samples used, and also independent of the number of texture pixels (assuming no texture paging).

For scenes of low or moderate complexity, our preprocessing algorithm is fast because all of its pixel operations can be done in hardware. For very complex scenes, our preprocessing algorithm becomes impractical because it is quadratic in s, however. In such cases, performance can be improved by calculating shadows only on a small number of surfaces in the scene (e.g. floor, walls, and other large, important surfaces), thereby reducing the cost to $O(nss_t+nt)$, where s_t is the number of textured polygons.

In an interactive setting, a progressive refinement of images can be used, in which hard shadows on a small number of polygons (precomputation with $n = 1$, s_t small) are rendered while the user is moving objects with the mouse, a full solution (precomputation with n large, s_t large) is computed when they complete a movement, and then top speed rendering (display with texture mapping) is used as the viewer moves through the scene.

More fundamentally, the quadratic cost can be reduced using more intelligent data structures. Because the angle of view of most of the shadow projection pyramids is narrow, only a small fraction of the polygons in a scene shadow a given polygon, on average. Using spatial data structures, entire objects can be culled with a few quick tests [2], obviating transformation and clipping of most of the scene, speeding the rendering of each hard shadow image from $O(s+t/s)$ to $O(s^\alpha+t/s)$, where $\alpha \approx .3$ or so.

An alternative optimization, which would make the algorithm more practical for the generation of shadows on complex curved or many-faceted objects, is to approximate a receiving object with a plane, compute shadows on this plane, and then project the shadows onto the object (figure 5). This has the advantage of replacing many renderings with a single rendering, but its disadvantage is that self-shadowing of concave objects is not simulated.

3.10 Comparison to Other Algorithms

We can compare the complexity of our algorithm to other algorithms capable of simulating soft shadows at near-interactive rates. The main alternatives are the stencil buffer technique by Heidmann, the z-buffer method by Segal *et al.*, and hardware hemicube-based radiosity algorithms.

The stencil buffer technique renders the scene once for each light source, so its cost per frame is $O(ns+np)$, making it difficult to support soft shadows in real-time. With the z-buffer shadow algorithm, the preprocessing time is acceptable, but the memory cost and display time cost are $O(np)$. This makes the algorithm awkward for many point light sources or extended light sources with many samples (large n). When soft shadows are desired, our approach appears to yield faster walkthroughs than either of these two methods, because our display process is so fast.

Among current radiosity algorithms, progressive radiosity using hardware hemicubes is probably the fastest method for complex scenes. With progressive radiosity, very high resolution hemicubes and many elements are needed to get good shadows, however. While progressive radiosity may be a better approach for shadow generation in very complex scenes (very large s), it appears slower than our technique for scenes of moderate complexity because every pixel-level operation in our algorithm can be done in hardware, but this is not the case with hemicubes, since the process of summing differential form factors while reading out of the hemicube must be done in software [7].

4 Scenes with General Reflectance

Shadows on specular surfaces, or surfaces with more general reflectance, can be simulated with a generalization of the diffuse algorithm, but not without added time and memory costs.

Shadows from a single point light source are easily simulated by placing just the visibility function $v(\mathbf{x}, \mathbf{x}')$ in texture memory, creating a Boolean *shadow texture*, and computing the remaining local illumination factors at vertices only. This method costs $O(ss_t+t)$ for precomputation, and $O(s+p)$ for display.

Shadows from multiple point light sources can also be simulated. After precomputing a shadow texture for each polygon when illuminated with each light source, the total illumination due to n light sources can be calculated by rendering the scene n times with each of these sets of shadow textures, compositing the final image using blending or with the accumulation buffer. The cost of this method is nt one-bit texture pixels and $O(ns+np)$ display time.

Generalizing this method to extended light sources in the case of general reflectance is more difficult, as the computation involves the integration of light from polygonal light sources weighted by the bidirectional reflectance distribution functions (BRDFs). Specular BRDF's are spiky, so careful integration is required or the highlights will betray the point sampling of the light sources. We believe, however, that with careful light sampling and numerical integration of the BRDF's, soft shadows on surfaces with general reflectance could be displayed with $O(nt)$ memory and $O(ns+np)$ time.

5 Implementation

We implemented our diffuse algorithm using the OpenGL subroutine library, running with the IRIX 5.3 operating system on an SGI Crimson with 100 MHz MIPS R4000 processor and Reality Engine graphics. This machine has hardware for texture mapping and an accumulation buffer with 24 bits per channel.

The implementation is fairly simple, since OpenGL supports loading of arbitrary 4×4 matrices, and we intentionally cast our shading formulas in a form that maps cleanly into OpenGL's model. The source code is about 2,000 lines of C++. Our implementation renders at about 900×900 resolution, and uses 24-bit textures at sizes of $2^{k_x} \times 2^{k_y}$ pixels, for $2 \leq k_x, k_y \leq 8$. Phong shading is simulated by subdividing each receiver polygon into a grid of 8×8-pixel parallelograms during preprocessing.

Our software allows interactive movement of objects and the camera. When the scene geometry is changed, textures are recomputed. On a scene with $s = 749$ polygons, $s_t = 3$ of them textured, with two area light sources sampled with $n = 8$ points total, generating textures with about $t = 200,000$ pixels total, and a final picture of about $p = 810,000$ pixels, preprocessing has a redisplay rate of 2 Hz. For simple scenes, the slowest part of preprocessing is the transfer of radiance textures from system memory to texture memory.

When only the view is changed, we simply redisplay the scene with texture mapping. The use of OpenGL display lists helps us achieve 30 Hz rates in most cases. When we allocate more radiance texture memory than the hardware can hold, however, paging slows redisplay.

Since we know the size and perceptual importance of each object at modeling time, we have found it convenient to have each receiver object control the number of light source samples that are used to illuminate it. The floor and walls, for example, might specify many light source samples, while table and chairs might specify a single light source sample. To facilitate further testing of shadow sampling, a slider that acts as a multiplier on the requested number of samples per light source is provided. More automatic and intelligent light sampling schemes are certainly possible.

6 Results

The color figures illustrate high quality results achievable in a few seconds with fine light source sampling. Figure 6 shows a scene with 6,142 polygons, 3 of them shadowed, which was computed in 5.5 seconds using $n = 32$ light samples total on two light sources. Figure 7 illustrates the calculation of shadows on more complex objects, with a total of $s_t = 25$ shadowed polygons. For this image, 7×7 light sampling was used when shadowing the walls and floor, while 3×3 sampling was used to compute shadows on the table top, and 2×2 sampling was used for the table legs. The textures for the table polygons are smaller than those for the walls and floor, in proportion to their world space size. This image was calculated in 13 seconds.

7 Conclusions

We have described a simple algorithm for generating soft shadows at interactive rates by exploiting graphics workstation hardware. Previous shadow generation methods have not supported both the computation and display of soft shadows at these speeds.

To achieve real time rates with our method, one probably needs hardware support for transformation, clipping, scan conversion, texture mapping, and accumulation buffer operations. In coming years, such hardware will only become more affordable, however. Software implementations will also work, of course, but at reduced speeds.

For most scenes, realistic images can be generated by computing soft shadows only for a small set of polygons. This will run quite fast. If it is necessary to compute shadows for every polygon, our preprocessing method has quadratic growth with respect to scene complexity s, but we believe this can be reduced to about $O(s^{1.3})$, using spatial data structures to cull off-screen objects.

Once preprocessing is done, the display cost is independent of the number and size of light sources. This cost is little more than the display cost without shadows.

The method also has potential as a form factor calculation technique for progressive radiosity.

8 Acknowledgments & Notes

We thank Silicon Graphics for the gift of a Reality Engine, which made this work possible. Jeremiah Blatz and Michael Garland provided modeling assistance. This paper grew out of a project by Herf in a graduate course on Rendering taught by Heckbert, Fall 1995.

References

[1] Kurt Akeley. RealityEngine graphics. In *SIGGRAPH '93 Proc.*, pages 109–116, Aug. 1993.

[2] James Arvo and David Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An introduction to ray tracing*, pages 201–262. Academic Press, 1989.

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

[4] 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.

[5] 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.

[6] 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.

[7] Michael F. Cohen and Donald P. Greenberg. The hemi-cube: A radiosity solution for complex environments. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):31–40, July 1985.

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

[9] 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.

[10] 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.

[11] 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.

[12] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):145–154, Aug. 1990.

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

[14] 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.

[15] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading MA, 1993.

[16] Tomoyuki Nishita and Eihachiro Nakamae. Half-tone representation of 3-D objects illuminated by area sources or polyhedron sources. In *COMPSAC '83, Proc. IEEE 7th Intl. Comp. Soft. and Applications Conf.*, pages 237–242, Nov. 1983.

[17] 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.

[18] Lance Williams. Casting curved shadows on curved surfaces. *Computer Graphics (SIGGRAPH '78 Proceedings)*, 12(3):270–274, Aug. 1978.

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