UCRL-JC-132898
PREPRINT

Shadow Penumbras for Complex Objects by Depth-Dependent Filtering of Multi-Layer Depth Images

B. Keating
N. Max

This paper was prepared for submittal to the
Association for Computing Machinery (ACM)
Special Interest Group on Graphics (SIGGRAPH) '99 Conference
Los Angeles, CA
August 8-13, 1999

January 12, 1999

Lawrence Livermore National Laboratory

Shadow Penumbras for Complex Objects by Depth-Dependent Filtering of Multi-Layer Depth Images

Brett Keating and Nelson Max
Lawrence Livermore National Laboratory and UC Davis

Abstract. This paper presents an efficient algorithm for filtering multi-layer depth images (MDIs) in order to produce approximate penumbras. The filtering is performed on a MDI that represents the view from the light source. The algorithm is based upon both ray tracing and the z-buffer shadow algorithm, and is closely related to convolution methods. The method's effectiveness is demonstrated on especially complex objects such as trees, whose soft shadows are expensive to compute by other methods. The method specifically addresses the problem of light-leaking that occurs when tracing rays through discrete representations, and the inability of convolution methods to produce accurate self-shadowing effects.

1 Introduction and Background

1.1 Introduction

Depth images are images that contain the depth of a visible object at each pixel, possibly along with other relevant scene information such as surface color. They are extremely common in computer graphics; the well-known and often-used z-buffer is a special case of a depth image. Depth images can be obtained synthetically by using a 3-D rendering engine, or they can be captured from real-life scenes by using special range-finding technologies or by finding correspondences between photographs.

An image-based rendering technique known as *view interpolation* uses depth images as input to synthesize new images [4]. This technique enjoys a rendering cost independent of geometric complexity. One major problem with view interpolation is the occurrence of holes in the synthesized images. To help solve this, Max extended the view interpolation technique to handle multiple depths at each pixel in a *multi-layer depth image* (MDI) [16]. MDIs contain additional important information about hidden surfaces at each viewpoint, which allows for reprojections that contain far less holes. Shade et al. proposed *layered depth images* (LDIs), which are multi-layer depth images optimized for rendering speed [24]. Each pixel in a layered depth image is called a *layered depth pixel*, each of which is a sorted collection of *depth pixels*.

The success of image-based rendering encouraged researchers to extend the technique so that it could produce general rendering effects, such as global illumination [14,17]. Also, image-based objects are beginning to be used in place of geometric objects [24]. This provides the motivation for the current work, which presents a technique for computing penumbras cast by image-based objects.

We propose a method for calculating penumbras that operates on a single multi-layer depth image, rendered from the point of view of the light source. Although image-based rendering motivates this work, the proposed method can be used in conjunction with virtually any rendering paradigm. We use an efficient incremental ray-tracing algorithm, similar to that used in [14]. The rays essentially perform an approximate integration operation on the visibility function, and this can be re-interpreted as a filtering scheme. Viewing the algorithm as a filtering scheme allows us to merge our algorithm with percentage-closer filtering [22], make direct comparisons with convolution methods [15,25], and weight the contribution of each ray to the final shadow.

1.2 Previous Work in Soft Shadow Calculation

The calculation of a penumbra can be viewed as the problem of computing the partial visibility of a light source from points on a surface. The darkness of the shadow cast by a light source at a point corresponds exactly to the occluded fraction of the source. Penumbra calculation is a difficult problem that has given rise to numerous different solutions, each with their own set of advantages and disadvantages [31].

Perhaps the most obvious approach to solving this problem would be to construct a frustum from a point on the surface back towards the light source. By doing so, one can clip objects to the frustum, effectively clipping objects against the extent of the light source. Tanaka et al. propose a fairly fast way to accomplish this calculation [27]. They use a ray-oriented buffer to cull much of the geometry in the scene. Then for the remaining objects, they use a fast silhouette generation algorithm in combination with a sophisticated clipping technique to quickly find the occluded area of the light source.

Discontinuity meshing is another exact method for calculating penumbra regions [7,11,13,26]. This technique partitions surfaces into regions within which all points view the light source as having the same topological configuration of vertices and edges. The visible area of the light source can be interpolated exactly within each of these regions. Collectively, these regions form a mesh known as a discontinuity mesh. Although this method is accurate, it can suffer from numerical instability and is also expensive to compute, especially for scenes containing a large number of objects. Discontinuity meshing methods are often used in a radiosity setting. There are several ways to compute soft shadows in radiosity, notably [5,21].

Since this problem tends to be expensive and difficult to solve exactly, much research has been done on finding approximate solutions [25,31]. Most of these solutions approximate an area light source by sampling its extent. One such approximation technique that can yield accurate soft shadows is distributed ray tracing [6]. This popular technique fires shadow rays from a point on a surface to sample points on a light source. The fraction of shadow at the surface point is equal to the proportion of rays that hit an intermediate object before reaching the light. This method is intuitive and completely general, however many rays are necessary to compute accurate penumbras. The current technique is closely related to this method, as is cone tracing [1].

The accumulation buffer is another method that samples the extent of the light source [9]. It does so by approximating it as a collection of point light sources. A visibility algorithm, such as a z-buffer, calculates a hard shadow (no penumbra) from the point of view of each point light source. These hard shadows are averaged in order to produce a soft shadow. This method requires many sample lights for adequate shadow quality, however view interpolation can be applied between samples to increase efficiency [4]. Other closely related approaches include [2,10].

Soft shadows have also been approximated using convolution techniques. These techniques use the fast Fourier transform to efficiently convolute an image of the light source with an image of the blockers, and the resulting blurred blocker image is projected onto receiver objects. Max applied the convolution technique to render penumbras caused by occluded sunlight and skylight underneath trees [15]. Soler et al. extended the convolution technique to more general situations, and utilized hardware to create soft shadows relatively quickly [25]. However, surfaces cannot accurately shadow themselves by this method.

Most of the above methods attempt to be completely general in their treatment of light sources. Other work has been done on specific types of light sources, such as sunlight or linear sources [20,28]. Our work addresses light sources that are moderately small in extent. A single MDI only gives completely accurate visibility from a single viewpoint, but the visibility is approximately correct for nearby viewpoints, as has been demonstrated by Max in [16] and Shade in [24]. Thus, a single MDI is adequate to approximate the visibility for all points on the light source if the light source occupies a reasonably small area. In this sense, our algorithm is not completely general and will not handle an arbitrarily large light source.

The remainder of the paper is organized as follows: Section 2 reviews the z-buffer shadow algorithm and percentage-closer filtering. Section 3 outlines the new method, and explains how filtering of depth buffer samples corresponds to the partial visibility of the light source. Section 4 provides implementation details and provides examples of penumbras rendered by this method. Section 5 concludes the paper with a discussion of the method's advantages and disadvantages, and directions for future work.

2 Z-Buffer Shadows and Percentage Closer Filtering

The z-buffer shadow algorithm is a two-pass technique originally conceived by Williams [29]. It is a hard shadow algorithm in that it only works for point light sources and does not produce penumbras. The scene is rendered into a z-buffer from the point of view of the light source, using coordinates we call *shadow space*. The depth is all that is entered into each z-buffer element during rendering, and a greater z-value indicates greater distance from the light. Assuming the scene has already been rendered into a z-buffer in camera space, we can compute shadows as a post-process by mapping each camera z-buffer point to a location in the shadow map. The z-value at that shadow map location is compared to the z-value of the mapped point. If the point's z-value is greater, the point is considered occluded and the illumination of the corresponding camera z-buffer entry is attenuated appropriately.

This shadowing technique enjoys great advantages while simultaneously suffering from major disadvantages. Its main advantage is that it can produce shadows for anything that can be represented in a depth buffer, which is almost everything. Even implicit surfaces, which are difficult to scan convert, can be placed in a depth buffer by ray casting or they can be tesselated. Another advantage is the algorithm's capability to be implemented in hardware, which was demonstrated by Segal et al. using texture-mapping hardware [23].

However, the shadows produced by the naïve approach are of poor quality. This is due to two different aliasing artifacts. The first artifact is a stair-stepping effect along the edge of the shadow, which is due to inadequate shadow buffer resolution. The second artifact is known as “surface acne,” and it is due to inappropriate self-shadowing. In addition to these problems, the z-buffer shadow algorithm only works for point light sources, and cannot be used to generate accurate penumbras except as part of an accumulation-buffer type of scheme.

The above-mentioned aliasing problems can be reduced to a great extent by percentage-closer filtering [22]. This method samples the bounding box in shadow space of the transformed camera space pixel’s area using jitter. Each sample’s z-value is compared to the z-value of the transformed point, with a random bias added. If the sample’s z-value is found to be lower, the sample is considered to be occluding the point. After the comparisons have been made, the proportion of occluding samples determines the fraction of shadowing.

The end result of the filtering is an antialiased shadow edge with the stair-stepping artifact removed. The random bias helps to remove the self-shadowing. The algorithm works well because of the properties of the stochastic sampling process. Jittering the samples effectively replaces the aliasing with random noise, which is a much less disturbing visual artifact.

One main criticism of the algorithm is the large amount of sampling parameters. Perhaps the most difficult parameters to adjust are the upper and lower bounds of the random bias. There is a fine line between too much surface acne and too much shadow displacement. Grant et al. developed an interesting solution to this problem involving the storage of normal vectors along with z-values in the depth buffer [8]. With the added normal information, the depth variation over a shadow map pixel can be modeled more accurately.

Woo proposed another solution to the bias problem, in which the average of the two closest depths is stored at each pixel instead of the closest depth [32]. By doing this, depth values are offset enough to prevent self-shadowing, but not so much as to prevent shadows on other possible receivers. Unfortunately this solution is not easily incorporated into the new method presented here due to the importance of knowing the actual surface depths.

Modifications to the percentage closer filtering algorithm can simulate penumbras by accounting for distances between objects in the scene. As a blocker’s distance from a receiver increases, the shadow edge can be blurred more to account for the increased separation. Partial success in generating penumbras from a single z-buffer using this idea was presented in [12]. Another implementation of this idea was used to create the shadows in the movie ANTZ by Pacific Data Images [30].

3 Depth-Dependent Filtering

3.1 Overview

At this point, we will give a brief overview of the steps taken in the depth-dependent filtering algorithm. The next few subsections will explain in detail the reasons for each step of the process. Then, in section 4, the algorithm will be outlined in greater detail.

First, an MDI is obtained that represents the view from the center of the light source. This MDI is preprocessed by dividing the entire depth range uniformly into disjoint *depth buckets* along the direction of light propagation. Next, a pixel in camera space is transformed to the coordinate frame of the MDI. The pixel area is transformed as well. Points are chosen on the bounding box of the transformed pixel area using jittered sampling on a regular grid. After testing for self-shadowing by using a z-buffer shadow algorithm, a ray is traced to the light source from each of these sample points. The subdivision of the MDI helps make the ray tracing more efficient and more accurate.

3.2 Ray Tracing the MDI

The process of tracing shadow rays through a parallel-projected MDI is shown on the left in Fig. 1. In the figure, each column represents a layered depth pixel of the

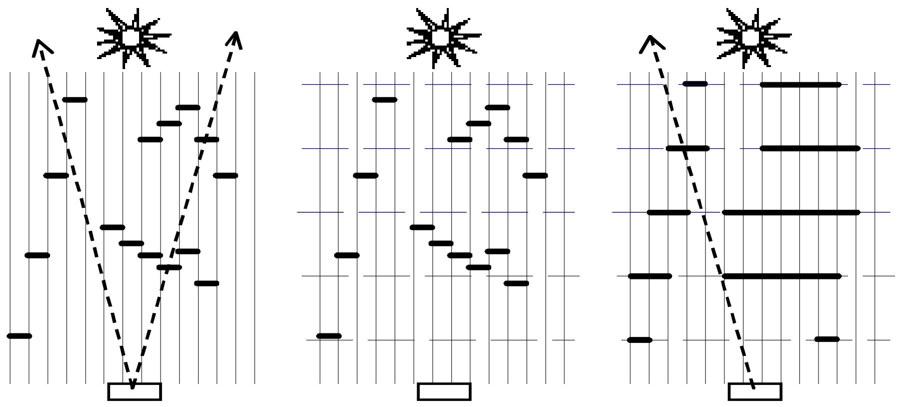
Fig. 1 The light leaking problem and reducing it by rounding to discrete depth levels

MDI. A thick black line across a layered depth pixel represents an individual depth pixel. This is a 2-D cross-section of the MDI, with depth varying vertically and the x-coordinate of the image varying horizontally. The shadow rays go to the light source from the shadow receiver, which is a pixel transformed from camera space to the MDI coordinate frame. Following the image-based ray tracing method used by Lischinski et al. [14], a ray intersects a depth pixel if the pixel's depth falls within the Z_{min} and Z_{max} of the ray as it crosses the layered depth pixel. The figure shows two rays, one that intersects a depth pixel and one that misses all depth pixels.

Notice that the depth pixels on the left seem to lie in a fairly straight line, indicating that they may all correspond to the same object. For example, a polygon that appears extremely slanted from the viewpoint of the light source may exhibit this behavior. With traditional ray tracing, the ray would intersect the object. However, note that with image-based ray tracing the object may be missed completely. This problem is known as *light-leaking*. To help reduce the light-leaking problem, we subdivide the MDI uniformly into separate regions of depth. We call these regions *depth buckets*. This is shown in the center of Fig. 1.

Instead of testing rays against depth pixels, we compute the intersection of a ray with the boundary between two buckets. Depth pixels from both buckets contribute to the boundary and form a blocker image. This essentially rounds each depth pixel to two discrete depths. The effect of doing this is shown at the right of Fig. 1, and notice that the light leak is removed. Although this changes the geometry of the scene, the net result is usually an imperceptible increase in the size of the shadow boundary. This problem is far less visually disturbing than having light appear to leak through a solid object.

In terms of implementation, we do not actually produce blocker images. Instead, we store a bitmask *at each layered depth pixel* that indicates which buckets contain depth pixels. For example, a 32-bit word can be used to describe 32 depth buckets at a pixel. The ray-object intersection method relies upon these bitmasks. When a ray intersects a depth bucket boundary, the intersection point occurs at a particular layered depth pixel. At this pixel, we check the two bits in the bitmask corresponding to the two buckets forming the boundary. Checking these two bits essentially checks for a ray-object intersection, and can be done with a single logic operation.

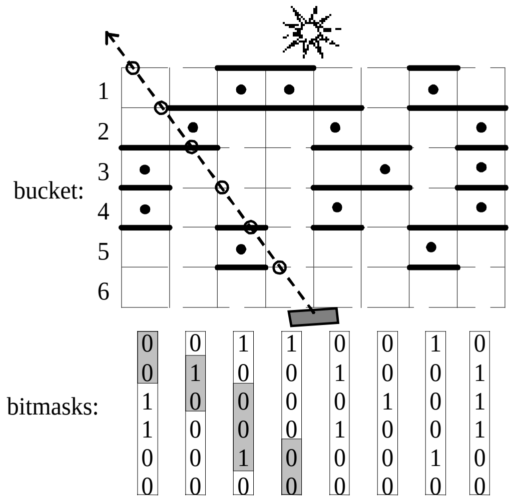

Fig. 2 Using bitmasks to perform ray-object intersections. Shaded bits are the ones checked at each layered depth pixel.

Fig. 2 graphically illustrates this process. Each black dot indicates that depth pixels are present in the relevant bucket at the relevant layered depth pixel. Thicker black lines indicate the depth pixel contributions to the boundary blocker images. Ray crossings at the boundaries are shown as small circles in the figure. The bits that are checked at each layered depth pixel are shaded. If the bit for either bucket is 1, we count that as a ray-object intersection.

Organizing the MDI into buckets in this fashion leads to several optimizations. First, it dramatically reduces the number of depth comparisons necessary for ray intersections at a layered depth pixel. Rather than iterate through the entire list of depths, we merely check the appropriate bits in a bitmask. Second, since all objects are assumed to be at uniformly spaced depths, we can step the rays through the MDI with uniform increments. These advantages, in concert with the reduction of light leaks, make this approach attractive despite the approximations made. A third important optimization uses the bitmasks to avoid unnecessary memory accesses. The range of layered depth pixels rays could hit is found, and the bitmasks of these pixels are *or*-ed together into a local variable. Since usually only a small proportion of buckets contain depth pixels, this variable can be used to cull a lot of processing and memory accesses.

The traced rays emanate from the transformed pixel area. We choose the emanation points by taking jittered samples of the transformed pixel area's bounding box. In our implementation, we trace one ray per jittered sample. We have found that for a given number of rays, the shadows visually look better if we do not also select the ray direction in a random fashion. In traditional distributed ray tracing, rays emanate from a surface with a random direction determined by the location of a random sample on the light source. In our implementation, we *deterministically* choose the ray direction based upon its starting location on the transformed pixel area. We choose the direction so that the rays intersect each blocker image with scaled versions of the same intersection pattern. By doing this, we are essentially correlating points on the pixel area with points on the light source, which is akin to a mapping operation. This is shown in Fig. 3. In the figure, the bottom line represents the receiver and the top line represents the light source.

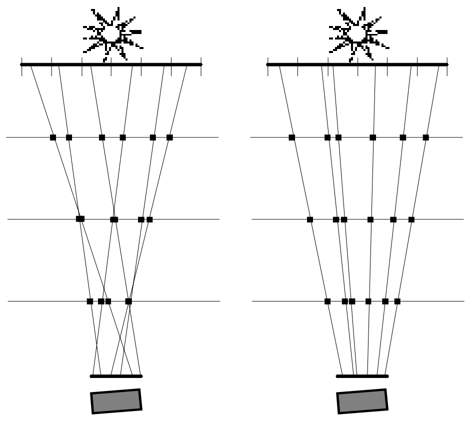
Fig. 3 Deterministic rays create similar patterns on blocker images.

Our method can be viewed as a solution to the problem of how to combine the effect of multiple blocker images. In [15,25], ad-hoc methods were used to combine occlusion from different overlapping blocker images. By using ray tracing, we are proposing a consistent and intuitive solution to the problem.

3.3 Filtering the MDI

Now that we have a method for tracing rays through the MDI, we can see how that method can be interpreted as a filtering method. This will be useful to us in several ways. First, we can easily incorporate pixel-area antialiasing by combining percentage closer filtering with our partial visibility algorithm. Second, we can adjust filter weights to approximate shaped light sources. Third, we can more directly compare our method with convolution methods for producing penumbras.

As mentioned in the previous section, the ray directions are chosen so that the intersection pattern is the same on each blocker image. However, as shown in Fig. 3, the pattern is scaled according to the relative distance from the transformed point. We can view this pattern as a *filter* that is performing area-averaging on each blocker image. Due to the way our ray directions have been chosen, the filters applied to the blocker images are all scaled versions of the same filter. Throughout the paper, the term *filter* will be used often to describe the area of a blocker image through which rays could possibly travel.

For the case of the parallel MDI, filter sizes for each blocker image are computed using the solid angle of the light source. The solid angle is used because the light source area is constant from all viewpoints, as with sunlight. In this case, visibility rays are confined to a cone of the same solid angle. For a particular transformed pixel, the volume of all possible rays will be the union of all such cones whose apex is a point on the transformed pixel area. The intersection of this volume with the blocker image plane yields the appropriate filter size for that blocker image.

This method can now be directly compared to convolution methods for calculating soft shadows. Convolution methods would convolute each blocker image with an image of the light source. We are convoluting each of these images with scaled versions of the same filter. Thus, by making the comparison between methods, our filter can be interpreted roughly as a representation of the image of the light source (it is really more like an approximate convolution of the pixel area and the light source image). If we change the weights on the filter, we can change the apparent shape of the light source. For example, if we set all corner weights on a filter to 1 and the remaining weights to 0, this would model four separate light sources at once. As another example, setting a row of filter weights equal to 1 and the rest 0 would model a linear source. An isotropic filter was used to model sunlight in the images provided in the color plates.

3.4 Self-Shadowing Surfaces

We do not discard the original depths contained in the MDI because we want to accurately produce self-shadowing effects. The z-buffer shadow algorithm is an excellent algorithm for computing self-shadows on complex surfaces, especially when percentage closer filtering is used. Since our algorithm is essentially based upon the z-buffer shadow algorithm (since we are using a multi-layer depth buffer), accurate self-shadows are possible if the actual depth pixels are accessed.

We mix the ray-tracing/filtering methods outlined above with a simple z-buffer shadow algorithm. The transformed pixel is mapped to a layered depth pixel location, as is done in the z-buffer shadow algorithm. Also, its depth occurs within the range of a particular depth bucket. Depth comparisons are made as in the z-buffer shadow algorithm, however only the depth pixels that are in that bucket and the next bucket are considered. The depth bias used in the z-buffer shadow algorithm may push the transformed point into the next bucket, in which case only depth pixels from the next two buckets are considered.

The depth pixels in these two buckets normally combine at their mutual boundary to form a blocker image. However, some of those depth pixels may actually be behind the transformed point. The depth comparisons serve to remove these pixels from the blocker image. To be consistent with our ray-tracing scheme, the appropriate filter size for the blocker image depth is used. Recall that the filter size on a blocker image determines the area through which visibility rays travel. In terms of implementation, this filter is applied using a method that closely mimics the percentage closer filtering algorithm.

3.5 Finite Light Sources

A perspective-projected MDI is used for the case of finite light sources. The actual 3-D Euclidean distance from the center of the light is stored in each depth pixel. This is done to ease the combination of the self-shadowing algorithm with the ray-tracing algorithm. If an orthographic projection was used, the z-buffer shadow

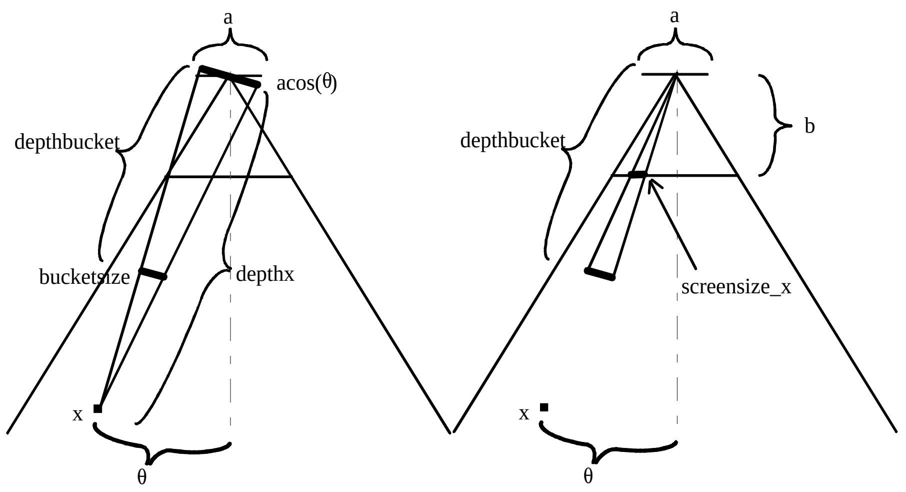

Fig. 4 Similar triangles argument for stepping rays through a perspective MDI.

algorithm would not accurately compute self-shadowing. Also, a perspective projection that stored a non-linear depth would complicate the ray-tracing algorithm.

By using a perspective projection and storing Euclidean depths, the algorithm is essentially the same as in the parallel case. The major difference is in the computation of the filter sizes. Intuitively, a square light on the ceiling will not appear to be square when viewed at an angle. Also, the filter will not grow linearly with depth as it did with a parallel light source.

We present a 2-D argument based on similar triangles to calculate the filter sizes for each depth bucket. We assume that the argument is independently valid for both dimensions of the filter. This is not entirely accurate, however it guarantees that the filter will remain rectangular.

At the left of Fig. 4 is a 2-D representation of the situation for finite light sources. The light is represented at the top of the figure by a line of length *a*. The transformed point occurs at **x**, which is an angle θ from the center of the projection. The line across the upper part of the light source frustum is the screen onto which all depth pixels are projected. Each pixel address on the projection screen represents a layered depth pixel, each of which contains a list of depth pixels and a bucket bitmask. The volume containing all rays from **x** to points on the light source is drawn, and the line across it represents a bucket filter located at the minimum bucket depth.

Since depth is stored as Euclidean distance, uniform subdivisions in depth result in shell-like bucket regions. Thus, the filter size tangent to the shell is desired. Using the projected size of the light source, the filter size is found by similar triangles:

$$bucketsize = a\cos(\Theta)\cdot\left(\frac{depth_x - depth_{bucket}}{depth_x}\right)$$

However, this doesn't indicate how many layered depth pixels the filter covers. To find this, we use another similar triangles argument (see the right of Fig. 4). The projected size of the filter is given by:

$$bucketsize = depth_{bucket} \cdot \frac{screensize_x}{b}$$

Eliminating the intermediate variable *bucketsize*, we discover the relationship between *depth*$_x$, *depth*$_{bucket}$ and the parameters of the MDI. Using a geometric substitution for cos(θ):

$$screensize_x = \frac{ab}{\sqrt{x^2+1}} \cdot \left(\frac{depth_x - depth_{bucket}}{depth_x \cdot depth_{bucket}} \right)$$

To find the screen size in the y direction, we simply substitute y for x in the above equation. Obviously, there is a non-linear dependence of the projected filter size on the minimum bucket depth. Thus, constant additive increments cannot be used to step rays from boundary to boundary. However, the difference in depth between filters is kept constant. Let *D* denote this depth difference. Also, the minimum depth of a particular bucket will be some multiple of *D*, plus the distance from the light source to the minimum depth *B* of the closest bucket. If the minimum depth of a previous bucket is given by *KD*+*B*, then *screensize* of the next bucket can be calculated from the *screensize* of the previous bucket by using the following iteration:

$$screensize_{next} = screensize_{previous} + \frac{C}{(KD+B)((K-1)D+B)}$$

where *C*=*abDcos*(θ). The number *K* is simply the enumeration of the bucket, where *K*=0 indicates the closest bucket to the light source. This is a convenient representation, since *C* only needs to be calculated once per transformed pixel, and the enumeration of the buckets is readily available.

The above calculations compute the intersection of the volume of rays from a point with a blocker image. Recall that there are several sample points on the transformed pixel area from which rays can emanate. The actual filter size is found by considering all ray volumes emanating from the pixel. We calculate the ray intersection area from the center of the pixel and add half to each side of the pixel area. This is a good approximation to the actual area of intersection if the pixel area is small enough, which is usually the case.

4 Implementation Details and Results

4.1 Implementation Details

The entire visibility-testing algorithm is contained in a single routine. A skeletal pseudocode description is contained in Fig. 5. The input is the point to be shadowed in shadow space coordinates, and the bounding box of the transformed pixel area. The output is a percentage of shadowing, which is used to attenuate the illumination of the camera-space pixel or subpixel fragment. It is assumed that *pixel3D* occurs in the middle of the box defined by *pixelsize*. The variables *ray_x* and *ray_y* are coordinates of jittered samples in the box.

For clarity, one simple optimization was left out of the pseudocode. Before iterating through all depth pixels in the second *for* loop, it is a simple matter to check if any depth pixels exist in the current or next depth bucket by first checking the bitmask at *MDI*[*ray_x*][*ray_y*]. If the bitmask indicates there are no depth pixels in those buckets, there is no need to iterate through the depth pixels and the z-buffer self-shadowing test can be skipped.

The filter calculation details were also left out of the pseudocode for clarity. To compute the appropriate filter size at a boundary, the volume of possible rays must be considered. The distance of the transformed point to the boundary is used to increase the size of the filter according to the volume. For the example of an infinite light source with a specified solid angle, the filter would be increased on each side by this distance times the tangent of the angle. For finite light sources, the arguments outlined in Section 3.5 would be applied. Filters are recalculated for each transformed pixel, in order to guarantee smooth variation of penumbra width as the transformed pixel moves from one bucket to the next.

```
Shadow( MDI[len][wid], pixel3D, pixelsize )
  Find bucket that encloses pixel3D based upon pixel3D.z;
  Find filter size at closest boundary using pixelsize;
  For(i=all jittered samples on pixel=(ray_x,ray_y))
    Compute random bias;
    Adjust sample location and current bucket due to bias;
    For(d=all depth pixels in MDI[ray_x][ray_y])
      If(d.z<pixel3D.z-bias)&(d in current or next bucket)
        inshadow += filterweight[i]; next i;
      else shoot a ray
        while(bucket>=0)
          Increment ray_x, ray_y;
          If(MDI[ray_x][ray_y].bitmask shows intersection)
             inshadow += filterweight[i]; next i;
          Decrement bucket;
  return inshadow/total_filter_weight;
```

Fig. 5 Pseudocode description

A few words should be said about how the depth pixels are stored. As in [24], we compute the MDI by first inserting depths into a linked list at each layered depth pixel. This allows for quick insertion without copying or re-ordering. Then, we collapse the depth pixels into a compact data structure. We use an array of structs, and each struct contains 1) a memory offset into a linear depth array, which is given the value of –1 if the layered depth pixel is empty, 2) the number of depths attributed to this layered depth pixel, and 3) the bitmask describing the presence of depth pixels in each bucket. For a MDI with resolution P by Q with R bytes used in the bucket bitmask (R=4 for 32 buckets), the memory requirement is then $PQ(R+8)$ bytes, where it is assumed that 4-byte words are used for the memory offset and the depth count. Every MDI requires at least this much memory.

The depths are compactly stored in a linear depth array, and are only accessed as needed by using the memory offset. To iterate through all depths at a layered depth pixel, we first add the offset to the start of the array. To get the remaining depths, we increment this pointer a number of times equal to the number of depths stored at that layered depth pixel. The depth array has a memory requirement of 4*S*, where a 4-byte word is used to store each depth and there are a total of *S* depth pixels in the entire MDI. Thus, the total memory requirement of the processed MDI is *PQ*(*R*+8)+4*S*. As an example, a 512x512 MDI with 32 buckets and an average of 4 depth pixels per layered depth pixel requires about 7 megabytes.

4.2 Discussion of Deterministic Ray Tracing

We have compared the results of our deterministic ray-tracing approach to that of tracing a random ray at each sample. We found that although intuitively it is more accurate to trace random rays, the shadows look worse due to additional noisiness in the shadow. This is a typical bias vs. variance problem. By preventing valid ray directions that may (or may not) reach the light source, we bias the size of the shadow in order to reduce the variance. This bias should not be confused with the depth bias used in the z-buffer shadow algorithm.

One way to understand the bias is to view it in terms of the direct correlation between points on the pixel and points on the light source. Essentially, we approximate the four-dimensional integral over the light source and the pixel as a two-dimensional integral, by constructing a one-to-one mapping between points in the two areas. The correlation introduced by the mapping makes the two-dimensional integral a biased approximation of the four-dimensional integral. Mathematically, the approximation can be described as follows:

$$\int\int V(\vec{x},\vec{y})d\vec{x}d\vec{y} \approx \int V(\vec{x}, f(\vec{x}))d\vec{x}$$

In the above equation, *V*(*x*,*y*) is the visibility function between points *x* on the pixel and points *y* on the light source. In the case of parallel light sources, *y* would represent a direction within the cone of the appropriate solid angle. The function *f*(*x*) is a one-to-one function that maps each pixel point *x* to a unique point (or direction) *y* on the light source.

We chose *f*(*x*) to be the rectangle-to-rectangle mapping [*Au*,*Bv*]→[*Cu*,*Dv*] where [*u*,*v*] are in the range [0,1], the pixel is a rectangle of dimensions *A*x*B*, and the light source is a rectangle (or a rectangular approximation to a cone of ray directions) of dimensions *C*x*D*. This mapping generates a fan-like distribution of rays emanating from the pixel area. It has several desirable properties. First, it is a one-to-one mapping. Second, the set of rays generated by this mapping fills the same volume as the original set of possible rays from the four-dimensional case. Third, it creates scaled versions of the same intersection pattern on each blocker image, as mentioned in Section 3.2.

By reducing the dimensionality of the integral, we are able to obtain a lower variance result with the same number of samples. We essentially perform Monte Carlo integration on the right hand side of the above equation by using stratified sampling over the pixel area. Stratification helps to reduce the variance compared to uniformly random sampling patterns [19]. Discussions of the relationship between the dimensionality of the sample space and the variance in computer graphics problems can be found in [18,19]. A variance measurement experiment involving four-dimensional double-area integration can be found in [18].

A comparison between tracing one random ray at each pixel sample versus tracing one correlated ray at each pixel sample is given in the color plate. By looking at the shadows of the poles, the variance difference is most noticeable. The variance presents itself as a noisy pattern in the penumbra. By comparing the shadows of the bush, the bias difference is most noticeable; certain areas that are illuminated in the random case are not present in the deterministic case, and the shape of illuminated areas are slightly different. Based upon comparisons like these, we decided that the bias error is more tolerable than the variance error for a given number of samples.

4.3 Results

All images were rendered using a program developed in-house in C++. The visibility algorithm used for rasterizing polygons in camera space was a modified version of the A-buffer [3], with 64 bits per pixel used for subpixel antialiasing. This algorithm was chosen for its close relationship to the MDI, and for its antialiasing. The program was executed on a 300Mhz Pentium II-based machine running Windows NT4, with 128 megabytes of RAM. All shadow computation times do not include the time spent rendering the MDI, since the MDI rendering process is independent of the shadow algorithm. We used a polygon-based renderer that typically produced an MDI in about a second. All times were found using the `clock` function. All rendering was done in software.

Each image was rendered at 256x256 resolution. Since our algorithm computes the shadow for each pixel individually, it is essentially an $O(pqNR)$ algorithm for each light source, where p and q are the dimensions of the final image, N is the number of buckets, and R is the number of rays per pixel. Thus, our algorithm fits in well with most of image-based rendering, since it is independent of geometric complexity.

We used 32 buckets in all situations, and 36 samples/rays per pixel. Increasing the number of buckets helps to increase accuracy to a point, but light-leaking becomes more of a problem and the efficiency begins to drop as the number of buckets gets higher. Using fewer buckets helps make the method faster, however the approximations made in geometry start to become more noticeable. We chose to use 32 buckets because it works well in most situations and fits neatly into a 32-bit word.

A standard resolution of 512x512 was used for all of the shadow MDIs. The speed of the algorithm is fairly insensitive to MDI resolution, however it can be sensitive to the number of depth pixels contained at a particular layered depth pixel. It is especially sensitive when there is a lot of self-shadowing, since it is for those cases that we must iterate through the entire list of depth values at a layered depth pixel.

To demonstrate the running time of the algorithm on a typical bad case, we used a dense tree, shown in the color plates. The filter size was selected to model sunlight. The MDI was allowed to contain any number of depth pixels at each layered depth pixel. For this model, the maximum number of depth pixels at a layered depth pixel was 112, and the average number over non-empty layered depth pixels was 19. The shadow was computed in 18.6 seconds. We feel that this is a fast time, since we are tracing rays through an enormous MDI. This shadow would be extremely difficult to compute using geometry-based methods, since the model contains 1,237,614 polygons.

The algorithm also tends to run slower if the size of the light source is larger. This is due to increased numbers of shadowed pixels and random memory accesses; thus it is not included in the algorithmic complexity argument given above. The color plates show a sunlight image, rendered in 2.5 seconds, and an image containing a parallel light with 10 times the sunlight solid angle, rendered in 9.5 seconds. For the finite light source examples, the light is modeled as a square source directly above the bush in the images. The poles are 30 units high, and the light source is positioned 25 units above them. The images show square light sources that are 4x4 and 8x8 in size, with shadows computed in 11.8 and 20.6 seconds, respectively. The field of view of the light sources was approximately 77 degrees.

A simulation of multiple light sources using a single filter is shown in the color plates as well, with a shadow computation time of about 4 seconds. The filter was an 8x8 filter, with each 3x3 corner isotropically weighted. Rays corresponding to zero filter weights were ignored.

5 Conclusions and Future Work

We have implemented a soft shadow algorithm that filters an MDI that represents the visibility from the light source. We have demonstrated how this algorithm efficiently traces rays through the MDI, and how the ray tracing corresponds to a filtering process. We have also shown how our algorithm relates to convolution methods, and how our algorithm intuitively blurs shadow edges according to the geometry of the scene.

Our algorithm has several advantages over other methods. It correctly produces self-shadowing for complex objects. In addition, the time complexity is independent of geometry. Also, it was designed to be applicable to image-based rendering situations, unlike most other soft shadow algorithms. The shadow computations are also reasonably fast given that the implementation is entirely in software.

Our algorithm also has several disadvantages. First of all, it is an approximation. However, approximate shadows are acceptable for many situations, as was argued in [25]. Another disadvantage is that our algorithm only works for light sources of moderate size. The fact that we are using MDIs makes this a high-memory-cost algorithm. The algorithm does not compute shadows at interactive rates, and the solution cannot be redisplayed interactively as in [10]. We feel that the shadow computation times could be improved if we could better exploit memory coherence.

Although we have reduced the light-leaking problem significantly, it has not been completely removed. The possibility still exists that light may leak through. A complete solution would treat the MDI used in the examples as a 512x512x32 collection of voxel-like slabs, and test against the sides of the slabs as well as the top and bottom. This would slow our implementation significantly, since it would be difficult to incorporate into our efficient bit-based scheme. We have yet to see light leaks happen for the types of light sources for which the algorithm was designed.

Like the z-buffer shadow algorithm, this algorithm suffers from resolution issues. If a particular area in the scene is sampled at high resolution in camera space (for example, objects very close to the camera in a perspective projection) but is sampled at a significantly lower rate in the MDI, the shadows will be of poor quality. Antialiasing over the pixel area is usually sufficient, but not in extreme situations. As a subject of future study, we are looking at ways to adaptively adjust the MDI resolution according to the relative positions and orientations of the camera, the light source and the surfaces in the scene.

The most noticeable artifact in the shadows is the random noise. We reduced the noise to a degree by putting a bias on the shadow size, which is less noticeable. Allowing multiple rays per pixel sample can reduce the noise without increasing bias, however this is significantly more expensive. We are currently working on image processing approaches based upon the wavelet transform to reduce noise as a post-process. This will hopefully allow us to use an unbiased integration scheme with fewer samples than is typically necessary, and still obtain penumbras with low noise levels.

6 Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract W-7405-ENG-48. Oliver Deussen provided the bush model and the tree model. Ross Gaunt and Jan Nunes recorded the video tape. We would like to also thank the reviewers, who added quite a bit to this paper through their constructive insights.

References

[1] John Amanatides. Ray tracing with cones. *Computer Graphics*, 18(3):129-135, July 1984. Proc. SIGGRAPH '84.

[2] Lynne Shapiro Brotman and Norman Badler. Generating soft shadows with a depth buffer algorithm. *IEEE CG&A*, 4(10):5-24, Oct. 1984.

[3] Loren C. Carpenter. The A-Buffer, an anti-aliased hidden surface method. *Computer Graphics*, 18(3):103-8. Proc. SIGGRAPH '84.

[4] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. *Computer Graphics*, 27:279-88, August 1993. Proc. SIGGRAPH '93.

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

[6] Robert Cook, Thomas Porter and Loren Carpenter. Distributed Ray Tracing. *Computer Graphics*, 18(3):137-145, July 1984. Proc. SIGGRAPH '84.

[7] George Drettakis and Eugene Fiume. A fast algorithm for area light sources using backprojection. *Computer Graphics*, pp.223-30, 1994. Proc. SIGGRAPH '94.

[8] Charles Grant and Michael Allison. Improvements on the depth buffer shadow algorithm. *Technical Report UCRL-102856*, Lawrence Livermore National Laboratory, January 1990.

[9] Paul Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. *Computer Graphics*, 24(4):309-18, August 1990. Proc. SIGGRAPH '90.

[10] Paul S. Heckbert and Michael Herf. Simulating soft shadows with graphics hardware. Technical report TR CMU-CS-97-104, Carnegie Mellon University, January 1997.

[11] Paul S. Heckbert. Discontinuity meshing for radiosity. *Rendering Techniques '92*, pp. 203-16, May 1992. Proc 3rd Eurographics Workshop on Rendering.

[12] Brett Keating. Extracting approximate sunlight penumbras from a single 3-D image (shadow z-buffer). Informal poster session, *Image-based rendering workshop*, Stanford University, March 23-25, 1998.

[13] Daniel Lischinski, Filippo Tampieri, and Donald P. Greenberg. Discontinuity meshing for accurate radiosity. *IEEE CG&A*, 12(6):25-39, November 1992.

[14] Dani Lischinski and Ari Rappoport. Image-based rendering for non-diffuse synthetic scenes. *Rendering Techniques '98*, pp. 301-14, June 1998. Proc. 9th Eurographics Workshop on Rendering.

[15] Nelson Max. Unified sun and sky illumination for shadows under trees. *CVGIP: Graphical Models and Image Processing*. 53(3):223-30, May, 1991.

[16] Nelson Max and Keiichi Ohsaki. Rendering trees from precomputed z-buffer views. *Rendering Techniques '95*, pp.74-81, June 1995. Proc. 6th Eurographics Workshop on Rendering.

[17] Nelson Max, Curtis Mobley, Brett Keating and En-Hua Wu. Plane-parallel radiance transport for global illumination in vegetation. *Rendering Techniques '97*, pp. 239-50, June 1997. Proc. 8th Eurographics Workshop on Rendering.

[18] Don P. Mitchell. Spectrally Optimal Sampling for Distribution Ray Tracing. *Computer Graphics*, 25(4):157-64, July 1991. Proc. SIGGRAPH '91.

[19] Don P. Mitchell. Consequences of Stratified Sampling in Graphics. *Computer Graphics*, pp. 277-80, August 1996. Proc. SIGGRAPH '96.

[20] Tomoyuki Nishita and Eihachiro Nakamae. Shading models for point and linear sources. *ACM Transactions on Graphics*, 14(2), 124-26, 1985.

[21] Tomoyuki Nishita and Eihachiro Nakamae. Continuous tone representation of three-dimensional objects taking account of shadows and interreflection. *Computer Graphics*, 19(3):23-30, July 1985. Proc. SIGGRAPH '85.

[22] William T. Reeves, David H. Salesin and Robert L. Cook. Rendering anti-aliased shadows with depth maps. *Computer Graphics*, 21(4):283-90, July 1987. Proc. SIGGRAPH '87.

[23] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran and Paul Haeberli. Fast Shadows and Lighting Effects Using Texture Mapping. *Computer Graphics*, 26(2):249-52, July 1992. Proc. SIGGRAPH '92.

[24] Jonathan Shade, Steven Gortler, Li-wei He and Richard Szeliski. Layered depth images. *Computer Graphics*, pp.231-42, July 1998. Proc. SIGGRAPH '98.

[25] Cyril Soler and François X. Sillion. Fast calculation of soft shadow textures using convolution. *Computer Graphics*, pp.321-32, July 1998. Proc. SIGGRAPH '98.

[26] A. James Stewart and Sherif Ghali. Fast computation of shadow boundaries using spatial coherence and backprojection. *Computer Graphics*, pp. 231-38, July 1994. Proc. SIGGRAPH '94.

[27] Toshimitsu Tanaka and Tokiichiro Takahashi. Fast analytic shading and shadowing for area light sources. Proc. EUROGRAPHICS '97, 16(3):C-231-40, 1997.

[28] Shinichi Takita, Kazufumi Kaneda, Toshio Akinobu, Haruhiko Iriyama, Eihachiro Nakame, and Tomoyuki Nishita. A simple rendering for penumbra caused by sunlight. *The Visual Computer*, 7(5) pp. 259-68, 1991.

[29] Lance Williams. Casting curved shadows on curved surfaces. *Computer Graphics*, 12(3):270-74, August 1978. Proc. SIGGRAPH '78.

[30] Daniel Wexler, Research & Development, Pacific Data Images. Personal communication, 1999.

[31] Andrew Woo, Pierre Poulin and Alain Fournier. A survey of shadow algorithms. *IEEE CG&A*, 10(6):13-32, Nov. 1990.

[32] Andrew Woo. The Shadow Depth Map Revisited. In David Kirk ed. Graphics Gems III. Academic Press, San Diego CA, pp. 338-42, 1992.

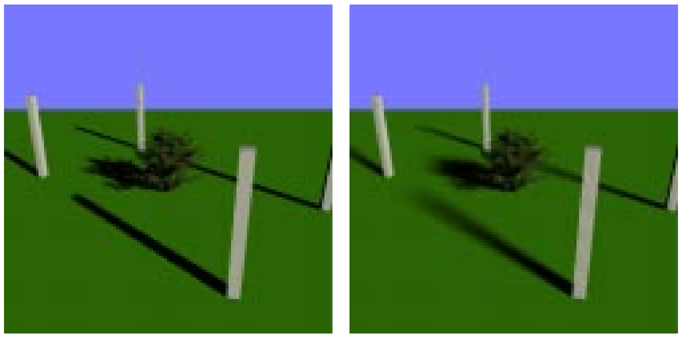
Fig. 5 Images from infinite light sources. A sunlight image is on the left and a 10x sunlight image is on the right. Shadow computation times are 2.5s and 9.5s respectively.

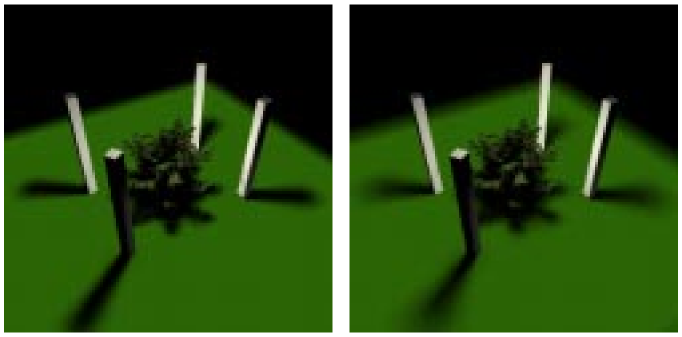
Fig. 6 Images from finite light sources. A 4x4 square light is used on the left and an 8x8 square light is used on the right. Shadow computation times are 11.8s and 20.6s respectively.

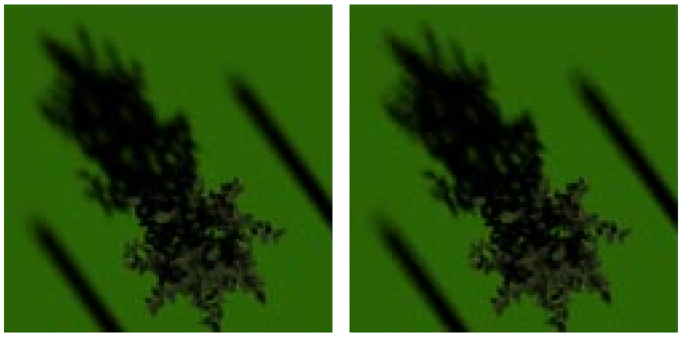
Fig. 7 Deterministic rays on the left vs. random rays on the right. Notice the tradeoff between increased bias on the left and increased variance on the right. Both shadows computed in ~10s.

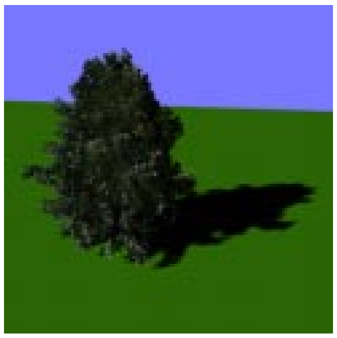
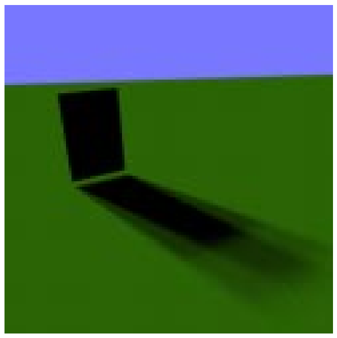
Fig. 8 (left) A demonstration of self-shadowing using a dense tree in sunlight, shadow computed in 18.6s. **Fig. 9 (right)** Using filter weights to simulate multiple light sources with one filter.