Rendering Antialiased Shadows with Depth Maps

William T. Reeves

David H. Salesin†

Robert L. Cook

Pixar
San Rafael, CA

ABSTRACT

We present a solution to the aliasing problem for shadow algorithms that use depth maps. The solution is based on a new filtering technique called percentage closer filtering. In addition to antialiasing, the improved algorithm provides soft shadow boundaries that resemble penumbrae. We describe the new algorithm in detail, demonstrate the effects of its parameters, and analyze its performance.

CR Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation - Display algorithms; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism - Color, shading, shadowing, and texture

General Terms: Algorithms, Performance Analysis

Key Words: shadows, depth maps, antialiasing, percentage closer filtering

1. Introduction

Shadows enhance the images synthesized by computers. Although many algorithms for rendering shadows have been published, most have been either restricted to a limited class of modeling primitives or are computationally expensive. Max [Max86] has classified these shadow rendering techniques as ray tracing, preprocessing, shadow volumes, area subdivision, and z-buffer algorithms.

Ray tracing algorithms [Whi80] [Kay79] [CPC84] [HaG86] produce excellent shadows and are easy to implement, but they are expensive. In order to make ray tracing more tractable, many techniques have been developed for quickly determining which object a secondary ray hits [Ama84] [HeH84] [RuW80] [KaK86]. However, this does not completely solve the problem, since once the object is determined it must still be accessed from the database. As models become more complex, the need to access any part of the model at any stage becomes more expensive; model and texture paging can dominate the rendering time.

Non ray tracing algorithms produce shadows without tracing secondary rays. Because objects can be sorted into buckets or scan lines according to the part of the screen they affect, the model can be accessed efficiently. But these algorithms also have serious limitations. Shadow α maps [ReB85] [Coo84] provide only a 2½-D solution, not a general 3-D solution. Preprocessing algorithms [BoK70] are suitable mostly for static environments. Shadow volumes [Cro77] [Ber86] [Max86] [NON85] [BrB84] and area subdivision algorithms [AWG78] are restricted to polygonal data and are inefficient for complex environments.

The z-buffer shadow algorithm developed by Williams [Wil78] does not have these problems. It can support all types of primitives; it is not excessively expensive, even for complex environments; and it is easy to implement. Its most serious drawback is a severe aliasing problem; it also requires additional memory for the z-buffer.

The z-buffer algorithm's singular versatility, efficiency, and simplicity make it tempting to look for ways to overcome its drawbacks, particularly the more serious aliasing problem. Storing floating point values in the depth buffer instead of 16-bit integers (as Williams did) reduces but does not solve this problem. An approach proposed by Hourcade and Nicolas [HoN85] stores object tags instead of depth values, but a limitation is that surfaces may not cast shadows on themselves.

In this paper, we introduce percentage closer filtering, a new sampling technique that can be used to eliminate the aliasing problem in Williams's z-buffer shadow algorithm. In addition to providing antialiased shadows, our new technique can be used to render soft shadow edges that resemble penumbrae.

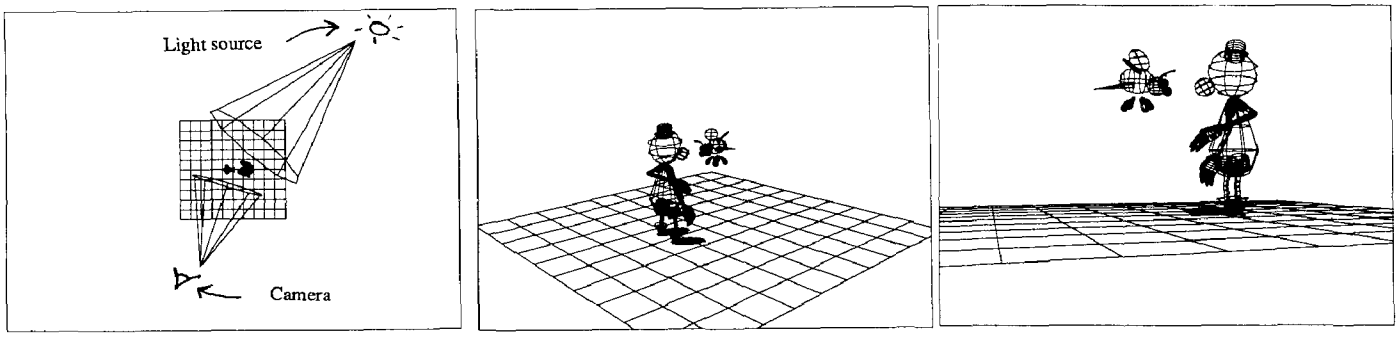
2. Percentage Closer Filtering

The z-buffer algorithm presented in [Wil78] operates in two passes, as illustrated for a simple scene in Figure 1. In the first pass, a view of the scene is computed from the light source's point of view, and the z values for objects nearest the light are stored in a z-buffer (also known as a *depth map*). In the second pass, the scene is rendered from the camera's position. At each pixel, a point on the surface is transformed to light source space, and its transformed z is compared against the z of the object nearest the light, as recorded in the depth map. If the transformed z is behind the stored z, the point is considered to be in shadow.

This algorithm has two aliasing problems: one in creating the depth maps, and the other in sampling them. The first aliasing problem can be solved by creating the depth maps with stochastic sampling [Coo86]. We solve the second problem by introducing a new technique called *percentage closer filtering*.

† Current address: Computer Science Dept., Stanford University, Stanford, CA 94305

 ACM-0-89791-227-6/87/007/0283 $00.75



(a) View from high above the scene. (b) View from the light source. (c) View from the camera.

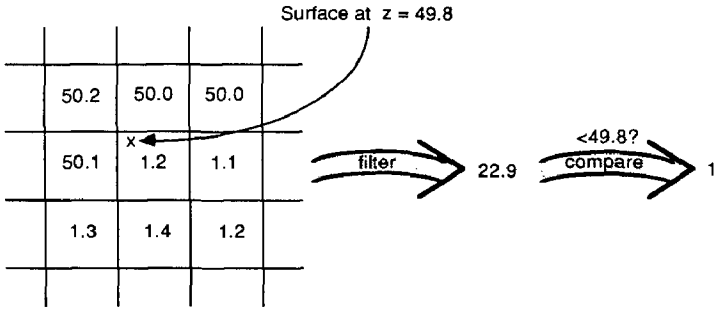
Figure 1. Points of view for a simple scene.

Ordinarily, texture maps are accessed by filtering the texture values over some region of the texture map. However, depth maps for shadow calculations cannot be accessed in this manner. The main problem is that the filtered depth value would be compared to the depth of the surface being rendered to determine whether or not the surface is in shadow at that point. The result of this comparison would be binary, making soft antialiased edges impossible. Another problem is that filtered depth values along the edges of objects would bear no relation to the geometry of the scene.

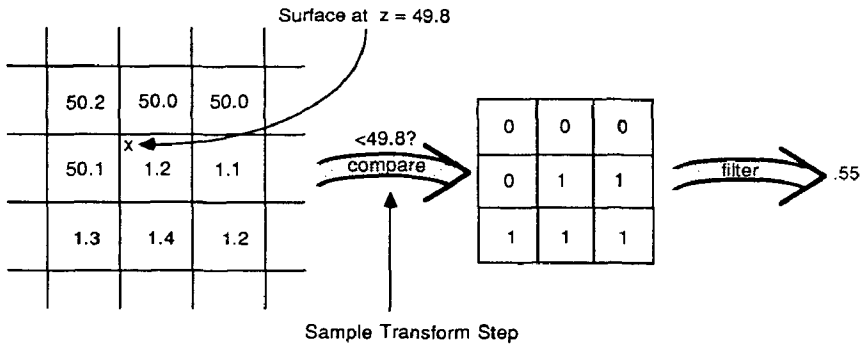
Our solution reverses the order of the filtering and comparison steps. The z values of the depth map across the entire region are first compared against the depth of the surface being rendered. This *sample transformation* converts the depth map under the region into a binary image, which is then filtered to give the proportion of the region in shadow. The resulting shadows have soft, antialiased edges.

The difference between ordinary texture map filtering and percentage closer filtering is shown schematically in Figure 2. In this example, the distance from the light source to the surface to be shadowed is $z = 49.8$. The region in the depth map that it maps onto (shown on the left in the figures) is a square measuring 3 pixels by 3 pixels.* Ordinary filtering would filter the depth map values to get 22.9 and then compare that to 49.8 to end up with a value of 1 meaning that 100% of the surface was in shadow. Percentage closer filtering compares each depth map value to 49.8 and then filters the array of binary values to arrive at a value of .55 meaning that 55% of the surface is in shadow.

* A square region and box filtering are used to simplify this example. The real algorithm, as described in subsequent sections, uses more sophisticated techniques.



a) Ordinary texture map filtering. Does not work for depth maps.



b) Percentage closer filtering.

Figure 2. Ordinary filtering versus percentage closer filtering.

In ordinary texture map applications, the cost of examining every pixel in a region can be avoided by saving the texture in some prefiltered format such as resolution pyramids [Wil83] or summed-area tables [Cro84]. Because our sample transformation depends on the *unfiltered* depth values, we cannot apply any such prefiltering technique here. But we can limit the number of texture pixel accesses in another way. By employing Monte Carlo techniques [Coo86], we can use a small, constant number of samples to approximate the result of transforming every sample in the region.

This can be done in one of several ways (Figure 3):

(a) choose samples randomly from a bounding box for the region;

(b) choose samples under some distribution, such as a Gaussian, from the same bounding box;

(c) partition the bounding box into subregions and sample each one with jitter;

(d) sample only positions inside the geometric boundary of the region.

Method (c), jitter sampling, approximates a Poisson disk distribution to produce shadows that are less noisy than those produced with either (a) or (b). All figures in this paper use (c), though (a) was used successfully in *Luxo Jr.* [Pix86]. Images made with (b) did not appear substantially different from those made with (a). We have not implemented (d), which is potentially more accurate, but also more complex and expensive.

Increasing the size of the sample region diffuses a shadow's edge over a larger area, resulting in a soft shadow edge resembling a penumbra. Although the shadow edges produced are not true penumbrae (in that their sizes do not depend on the relative distances between objects and light source), we have nevertheless found them convincing enough for many applications.

3. Implementation of the Algorithm

We now describe in detail how percentage closer filtering can be used with a depth buffer to create shadows. As in Williams's original z-buffer algorithm, we use two passes: one to create the depth map for each light source; and one to render the scene, using the depth maps to determine portions of objects in shadow.

3.1 First Pass: Creating the Depth Maps

The depth map for each light source is created by computing the depth values for all objects in the image from the light source's point of view. In the screen space of the light, the x- and y-coordinates correspond to pixel locations in the depth map, and the z-coordinate (in floating point) is the distance from the light in world space. We use the term *light space* to refer to these x, y, and z values.

In practice, we use regular sampling instead of stochastic sampling when creating the depth maps. This is because with one sample per pixel and low depth map resolutions, the slight aliasing of depth values with regular sampling is sometimes less objectionable than noise with stochastic sampling. We use a relatively course resolution for our depth maps in order to minimize the memory requirements.

The first pass can be implemented very easily. In our rendering system, we only needed to make one change: instead of computing and storing integer color values in a picture file, the closest z values (which are already being computed for hidden surface calculations) must be stored in floating point in a texture file.* This change amounted to about 40 lines of code out of over 30,000 for the rendering system as a whole.

A depth map can be computed faster than a shaded image for several reasons. First, none of the shading, texturing, and lighting calculations are needed. Second, objects (such as the ground plane) that never cast shadows on any other object can be ignored. Finally, depth maps require only one sample per pixel.

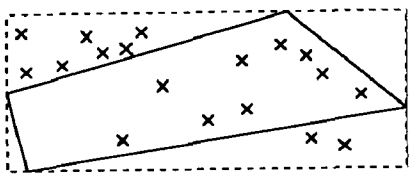
3.2 Second Pass: Rendering the Scene

In the second pass, the scene is rendered from the camera's point of view. Each shading calculation represents some region on a surface. Depending on the rendering system, these regions might represent anything from polygons covering large areas of pixels (e.g., for a painter's algorithm) to exact pixel regions (e.g., for scanline methods) to tiny subpixel regions (e.g., for micropolygons [CCC]). The shadow algorithm presented here is independent of the type of region used.

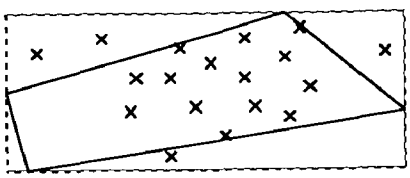
Each region to be shaded is first mapped into light space, giving a region in the depth map. Percentage closer filtering is used to determine the proportion of z values in the region of the depth map that are closer to the light than the surface. This gives the proportion of the surface in shadow over the region. This proportion is then used to attenuate the intensity of the light. If several lights in a scene cast shadows, this process is repeated for every light. The attenuated intensities are then used in the shading calculations.

One problem arises if the transformed region lies outside the extent of the depth map. Generally, we use the convention that regions outside the light source's field of view are *not* considered to be in shadow. For directed lights the distinction is not important, since objects outside a light's field of view are not illuminated by that light anyway. For lights that cast shadows in all directions, a more complex mapping scheme, such as spherical or cubical environment maps, should be used [Gre86].

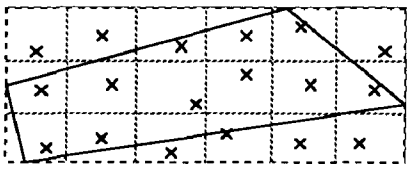
* For pixels containing no visible surface, a very large constant is stored to denote an infinite depth.



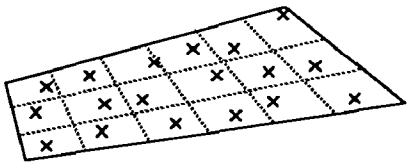
a Bounding box sampled with uniform distribution.



b Bounding box sampled with gaussian distribution.



c Bounding box sampled with jitter.



d Geometric boundary sampled with jitter.

Figure 3. Different methods for choosing samples.

Our implementation of the second pass is simplified by the use of light trees [Coo84]. Light trees are essentially small programs that describe the illumination calculations for individual light sources. They allow us to describe lights with different characteristics. For example, we can control the softness of shadow edges or distinguish lights that cast shadows from lights that do not. As each region is shaded, the light trees are called to determine the proportion of the region illuminated. Each light tree calls the percentage closer filtering routine, passing a pointer to the depth map and the region to be filtered, and uses the result (in conjunction, perhaps, with other parameters controlling the spatial distribution of the light) in computing its contribution to the illumination of the surface.

In our implementation, the second pass required about 370 lines of code. Most of this code served merely to integrate the heart of the percentage closer filtering algorithm, shown in Figure 4, with the rest of our renderer.

The code in Figure 4 provides a number of parameters that we can adjust in computing our shadows. The `NumSamples` parameter controls the number of sample points used per region. The `ResFactor` parameter artificially enlarges the size of the sampling region, allowing lights that cast softer shadows. The `Bias` parameter is used to offset the surface slightly when comparisons to the z values in the depth buffer are made. This prevents a surface from incorrectly shadowing itself if a z value for a point on the surface is compared to the depth map z from a nearby but different position on the same surface.* This incorrect self-shadowing can create Moiré patterns. The effects of these and other parameters are discussed later.

3.3 Storage Issues

Depth maps tend to be large. We store along with each depth map a bounding box of all pixels with finite depths, and pixels outside this box are not actually stored with the depth map. We have not found the working storage requirements of our algorithm to be too great. For a scanline rendering algorithm, locality in the screen space of the camera is correlated with locality in the screen space of the light source. Thus, a simple paging scheme works well for this application.

Still more memory could potentially be saved by dividing the depth map into rectangular tiles, which could be cached to take better advantage of the algorithm's two-dimensional locality. In addition, a tile scheme would allow us to store a maximum z value with each tile, thereby avoiding sampling the depth map altogether if a region were further away from the light than any object in the tile.

4. Examples

4.1 Effect of Stochastic Percentage Closer Filtering

Our first example, Figure 5, shows a simple scene. A light source off-screen to the left shines on a red sphere, which casts a shadow onto an upright green plane. Figure 6 is the result when the diffuse and specular shading components are removed, revealing only the shadow component of the shading. The image in Figure 6a was rendered with our algorithm but with its parameters set to simulate an ordinary z-buffer algorithm (i.e., the depth map is point sampled without jitter). The image in Figure 6b uses the same depth map as the other, but is rendered using our new technique of percentage closer filtering. This image has antialiased shadow edges.

* We actually specify a minimum and a maximum bias, and at every sample the actual bias is chosen stochastically from that range. This allows a surface to curve over to legitimately shadow itself without aliasing.

```
/* parameters setable in other parts of renderer */
float ResFactor = 3, MinSize = 0, Bias0 = .3, Bias1 = .4;
int NumSamples = 16, MinSamples = 1;

#define MAPRES 1024                 /* size of depth map */
float DepthMap[MAPRES][MAPRES];     /* actual depth map */

#define CLAMP(a, min, max) (a<min?min:(a>max?max:a))
float Rand();          /* returns random numbers in range [0.,1.) */
float ceil();          /* returns smallest integer no less than argument */
float floor();         /* returns largest integer no greater than argument */

typedef struct {
   int r_umin, r_umax;      /* min and max pixels in u dimension */
   int r_vmin, r_vmax;      /* min and max pixels in v dimension */
} TextureRect;

float SampleShadow(s, t, z, sres, tres, bbox)
   float s, t;              /* depth map indices. range [0.,1.) */
   float z;                 /* light space depth */
   float sres, tres;        /* size of sampling rectangle. range [0.,1.) */
   TextureRect *bbox;       /* bounding box on depth map in pixels */
{
   int i, j, inshadow, iu, iv, ns, nt, lu, hu, lv, hv;
   float bias, smin, tmin, ds, dt, js, jt;

   /* if point is behind light source, call it not in shadow */
   if (z < 0.)
       return(0.);

   /* convert to coordinates of depth map */
   sres = MAPRES * sres * ResFactor;
   tres = MAPRES * tres * ResFactor;
   if(sres < MinSize)
       sres = MinSize;
   if(tres < MinSize)
       tres = MinSize;
   s = s * MAPRES; t = t * MAPRES;

   /* cull if outside bounding box */
   lu = floor(s - sres); hu = ceil(s + sres);
   lv = floor(t - tres); hv = ceil(t + tres);
   if(lu>bbox->r_umax || hu<bbox->r_umin
                      || lv>bbox->r_vmax || hv<bbox->r_vmin)
       return(0.);

   /* calculate number of samples */
   if(sres*tres*4. < NumSamples) {
       ns = sres*2.+.5;
       ns = CLAMP(ns, MinSamples, NumSamples);
       nt = tres*2.+.5;
       nt = CLAMP(nt, MinSamples, NumSamples);
   } else {
       nt = sqrt(tres*NumSamples/sres)+.5;
       ns = CLAMP(ns, MinSamples, NumSamples);
       ns = ((float)NumSamples)/nt+.5;
       nt = CLAMP(nt, MinSamples, NumSamples);
   }

   /* setup jitter variables */
   ds = 2*sres/ns; dt = 2*tres/nt;
   js = ds*.5; jt = dt*.5;
   smin = s - sres + js; tmin = t - tres + jt;

   /* test the samples */
   inshadow = 0;
   for (i = 0, s = smin; i < ns; i = i+1, s = s+ds) {
       for (j = 0, t = tmin; j < nt; j = j+1, t = t+dt) {
               /* jitter s and t */
               iu = s + Rand() * js;
               iv = t + Rand() * jt;
               /* pick a random bias */
               bias = Rand()*(Bias1-Bias0)+Bias0;
               /* clip to bbox */
               if(iu>=bbox->r_umin && iu<=bbox->r_umax
                                   && iv>=bbox->r_vmin
                                   && iv<=bbox->r_vmax) {
                       /* compare z value to z from depth map plus bias */
                       if(z > DepthMap[iu][iv] + bias)
                               inshadow = inshadow+1;
               }
       }
   }
   return(((float) inshadow) / (ns*nt));
}
```

Figure 4. Percentage Closer Filtering Algorithm.

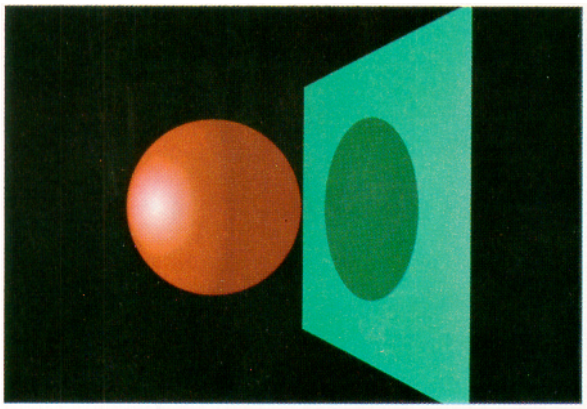
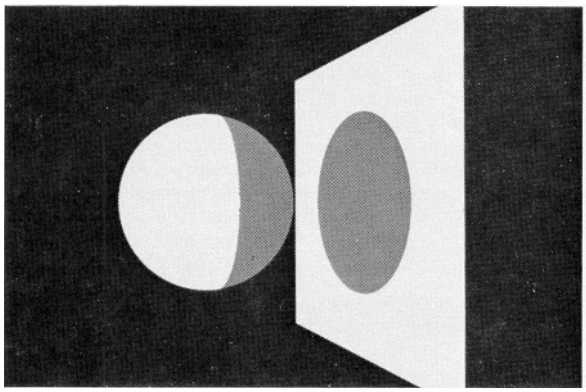
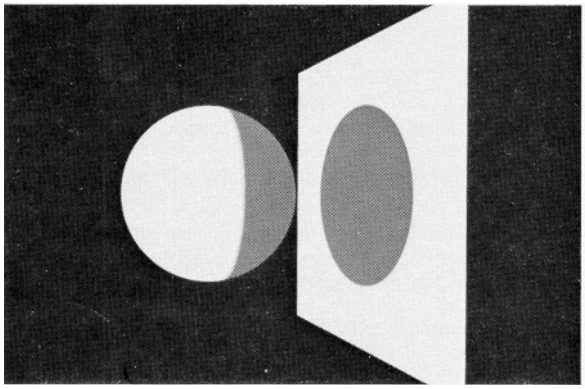


Figure 5. Sphere and plane example.



(a) Ordinary z-buffer algorithm.



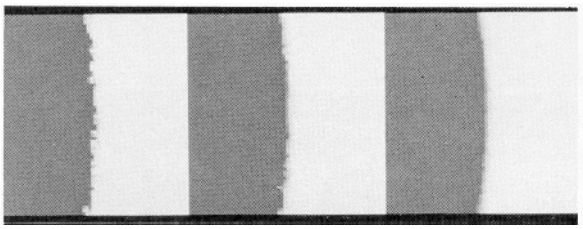
(b) Percentage closer filtering algorithm.

Figure 6. Shadow component.

4.2 Effect of Parameters

We now explore the effects of the shadow algorithm's parameters defined earlier. In most of the following figures, we have zoomed in by pixel replication on either the edge of the shadow cast onto the plane or on the boundary of the self-shadowing region on the sphere.

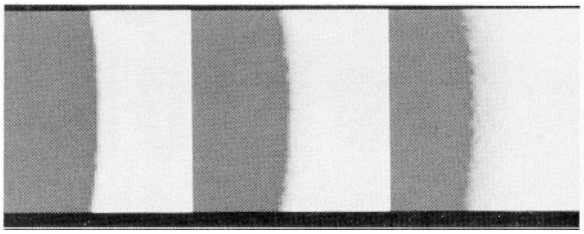
Figure 7 shows the effect of increasing the `NumSamples` parameter. Because the shadow intensity is the result of filtering `NumSamples` binary values, the number of bits in the shadow intensity is equal to the logarithm base 2 of this parameter's value. If `NumSamples` is too small, the shadow edges appear noisy; if it is larger, the filtering becomes better but more expensive. We normally use a `NumSamples` of 16.



(a) `NumSamples` 1. (b) `NumSamples` 4. (c) `NumSamples` 16.

Figure 7. Effect of the `NumSamples` parameter.

Figure 8 shows the effect of increasing the `ResFactor` parameter. In general, increasing `ResFactor` produces softer shadow edges. Note that if `ResFactor` is large, the soft shadow edges are wider, making the quantization of the shadow intensity more apparent. This may become objectionable unless `NumSamples` is increased. `ResFactor` should never be less than one. If it is then any aliases or artifacts in the depth map are reproduced in the shadows. We normally use a `ResFactor` between 2 and 4.

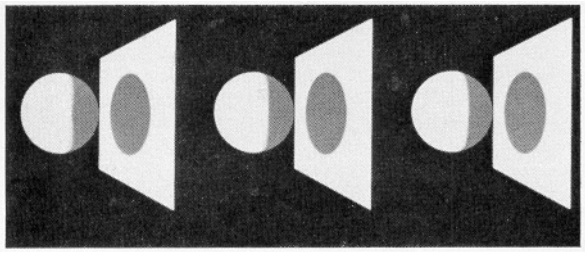


(a) `ResFactor` 1. (b) `ResFactor` 3. (c) `ResFactor` 8.

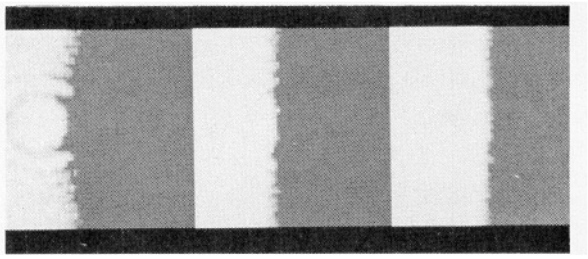
Figure 8. Effect of the `ResFactor` parameter.

Figure 9 shows the effect of increasing the `Bias` parameters. Increasing `Bias` eliminates the Moiré patterns due to incorrect self-shadowing, but moves the shadow boundary away from its true position. In practice, we have not found it difficult to select `Bias` values large enough to eliminate self-shadowing artifacts, yet small enough to avoid any noticeable problems from the offset boundary positions. Note that `Bias` values in our implementation depend on the world space metric of the object, since they are merely added to the z values of the depth map. In worlds measuring 100 by 100 by 100 units, typical bias values we use are (0.3, 0.4).

Figure 10 shows the effect of increasing the resolution of the depth map, relative to the resolution of the final image. When the resolution of the depth map is too small, the shadows are blurry and noisy because small detail is not represented in the depth map. When depth maps are too large, storage is wasted. We normally use depth maps whose resolutions are the same size or twice as large (in each dimension) as the resolution of the final image. Typically, we start with a depth map of the same size, make a few test images, and adjust if necessary.

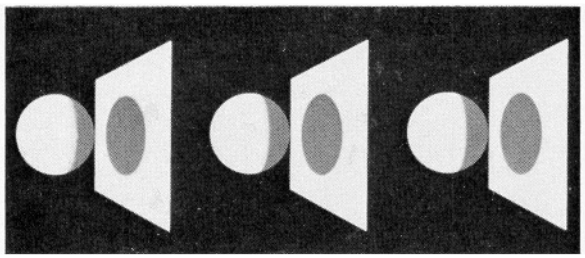


(a) (Bias (0.0, 0.0). (b) (Bias (0.2, 0.3). (c) (Bias (0.3, 0.4).



(d) (Bias (0.0, 0.0). (e) (Bias (0.2, 0.3). (f) (Bias (0.3, 0.4).

Figure 9. Effect of the `Bias` parameter.



(a) Res 256x256. (b) Res 512x512. (c) Res 1024x1024.

Figure 10. Effect of depth map resolution.

Two other parameters that we have not illustrated are `MinSamples` and `MinSize`. These parameters supply absolute minimums for the number of samples taken along each of the region's x and y dimensions, and for the size in pixels of each dimension. As demonstrated in the sample code in Figure 4, they are used to control the effects of sampling oddly-shaped regions of the depth map. We typically set `MinSamples` to 2 or 3 to give us fair coverage in both dimensions of the depth map, and `MinSize` to 2 to give us samples from at least four pixels.

4.3 Colored Lights

The shadow algorithm presented can be used equally well for colored lights. Figure 11 shows a scene illuminated by three colored lights (red, green, and blue) at different positions off-screen to the left. They cast light onto a white sphere before a white plane. Because the light sources are at different positions, their shadows overlap partially, creating a multi-colored pattern on the plane.

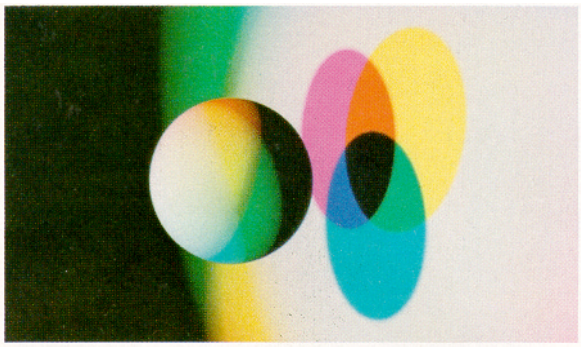


Figure 11. Colored lights example.

4.4 Animation

Our shadow algorithm was used in the computer animated film *Luxo Jr.* [Pix86]. Figure 12 is an example frame from the film. Figure 13 shows the three depth maps created for the three light sources ("Dad", "Junior", and the ceiling light) that illuminate the scene. Here, levels of grey are used to represent the range of floating point z values of the depth map; objects closer to the light are darker in the image. Note that the ground plane does not appear in any of the depth maps. Since it never shadows other objects, it is made invisible, and its depths are never computed.

4.5 Complex Scenes

The shadow algorithm is currently being used in a new animated film called *Red's Dream* [Pix87]. Figure 14 is from a scene that takes place in a dimly lit bicycle store. The scene is illuminated by seven light sources, but only two of these lights actually cast shadows. While shadows are not the most prominent feature of this image, they add a subtlety that is important even in complex scenes.

5. Performance

The following measurements are for the shadow algorithm implemented as a part of the *reyes* rendering system [CCC] running on a CCI Power 6/32 computer under the Berkeley 4.3 UNIX† operating system. A CCI machine executes at about four to six times a VAX 11/780 on typical single precision floating point computations.

The computation time for each example is shown in Table 1. All of the images were computed at a horizontal resolution of 1024 pixels with 16 stochastic samples per pixel. The "Without Shadows" column gives the time to compute the image without shadows. The "Number of Lights" column gives the number of lights casting shadows, i.e., the number of depth maps used. The "First Pass" column gives the total time to compute the depth maps for all the light sources. The "Second Pass" column gives the time to compute the final image with shadows. The "Total With Shadows" column sums the times of the first and second passes. Finally, the "% Increase" column gives the percentage increase in time to compute the image with shadows over the time for the image without shadows.

On average, an image with shadows costs 40% to 50% more than the one without. We consider this increase in computation time to be reasonable. The cost grows as the number of light sources casting shadows grows, but it is significantly less than $(1+NumberOfLights)\times CostWithoutShadows$ which is what might be expected given that $1+NumberOfLights$ images were computed to make it.

† UNIX is a trademark of Bell Laboratories.

Figure 12. Frame from *Luxo Jr.*

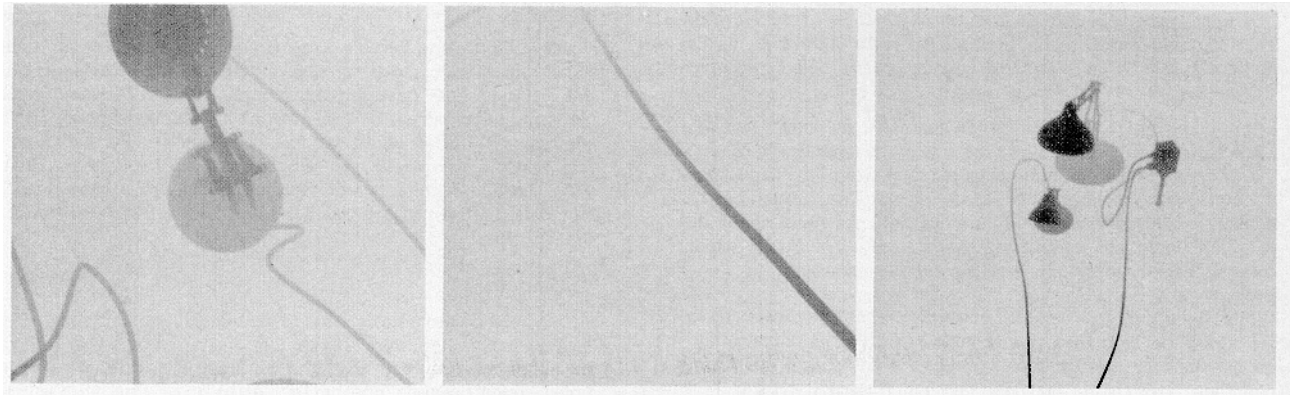


Figure 13. Shadow maps from *Luxo Jr.*

Figure 14. Red's Dream

Computation Time (in minutes)						
Image	Without Shadows	Number of Lights	First pass	Second pass	Total With Shadows	% Increase
Sphere and plane	42.0	1	9.7	48.0	57.7	37.3
Colored lights	64.4	3	14.2	80.8	95.0	47.6
Luxo Jr.	58.8	3	14.6	74.2	88.8	50.9
Red's Dream	261.3	2	55.3	332.1	387.4	48.4

TABLE 1. Shadow Algorithm Performance

Depth Map Storage (in megabytes)				
Image	Resolutions	Raw Depth Maps	Bounding Box Scheme	Tile Scheme
Sphere and plane	1 at 1024^2	4.0	.4	.4
Colored lights	3 at 1024^2	12.0	2.6	2.9
Luxo Jr.	1 at 2048^2, 2 at 1024^2	24.0	13.0	3.7
Bike Store	2 at 1024^2	8.0	5.6	4.2

TABLE 2. Depth Map Storage

A breakdown of the extra time spent in computing our sample frame from *Luxo Jr.* gives an idea of where that extra time is spent. The first pass (creating the three depth maps) accounts for 50% of the extra time. Of the remaining half, the percentage closer filtering accounts for 28%, and the transformation of regions' coordinates into light space accounts for the remaining 22% of the extra time. This last transformation is performed by the light tree, which is implemented with an interpreted language. It could be sped up considerably if we wrote the transformation code in C.

Depth map storage statistics for the examples are shown in Table 2. The "Resolution" column lists the size of the depth maps used. The "Raw Depth Maps" column gives the total size of the depth maps if no storage compaction scheme is used. The "Bounding Box Scheme" column gives the size when only the parts of the depth map containing non-infinite depths are stored. The "Tile Scheme" column gives the size when 32×32-pixel tiles are used.

While the algorithm uses a lot of storage when the raw depth maps are stored, our bounding box scheme can give significant improvements. In some very complex images, such as Red's Dream, the bounding box scheme may not save much storage because objects that can potentially cast shadows are everywhere in the scene. In such images, a tile scheme may give better results.

6. Limitations and Future Work

Although the shadow algorithm described here provides an improvement to Williams's original depth buffer algorithm, it still has many limitations.

The algorithm does not handle motion blur nor compute exact penumbrae. We are currently investigating extensions of the algorithm that we believe will address these limitations. However, it is worth noting that for many situations the algorithm presented here is adequate. For instance, the shadows in *Luxo Jr.* did not have correct penumbrae, and while the lamps themselves were rendered with motion blur, the shadows were not. The algorithm presented here also does not address transparent or translucent objects that cast shadows.

Although the use of bounding boxes reduces the algorithm's storage requirements considerably, we would like to develop more sophisticated schemes for reducing this storage requirement further. The tile-based scheme outlined in this paper would be one approach. An adaptive scheme, where higher or lower resolution areas were used depending on the relative complexities of various regions of the scene, might also be appropriate.

We would like to develop better tools for automating the process of creating the depth maps and generating the light trees. We would also like to build a system that used more intuitive parameters that corresponded more closely with aspects of the appearance of shadows in the scene. This might obviate much of the trial-and-error we have relied upon in choosing parameter values for our scenes.

Finally, we hope to be able to generalize and formalize the sample transformation step in percentage closer filtering. We believe that this technique may have important implications to the use of texture maps for other purposes. For example, in bump mapping [Bli78], specular reflections could be computed before filtering, and the results could be filtered and sampled as ordinary textures. In this way, specular highlights from the microfacets of a bumpy surface would be maintained even as the surface were translated back into the far distance.

7. Conclusions

We have presented a solution to the aliasing problem for the depth buffer shadow algorithm. This solution is based on a new technique for using texture maps called percentage closer filtering. Other improvements to the algorithm include using floating point values in the depth map and including the shadow information as part of the shading calculations instead of as a postprocess. The new technique also provides a penumbra-like effect by providing control over the softness of shadow edges.

Our method is more expensive than the original, both in terms of time and space. Percentage closer filtering takes more time than evaluating a single sample, and floating point numbers typically consume more space than integers. However, because percentage closer filtering requires only a constant number of samples per region, the extra cost is bounded by a constant factor, and in practice this factor is small. We feel that the extra time and space are justified by the improved image quality. This improved image quality has proven robust in an animated sequence.

8. Acknowledgements

Eben Ostby, Loren Carpenter and Paul Heckbert provided many significant insights while the algorithm was being developed. Eben also found several bugs and optimized parts of the code. Eben and John Lasseter helped test the algorithm by designing animated sequences that depended on the shadows to work. Ricki Blau provided photographic assistance.

9. Bibliography

[Ama84] J. Amanatides, Ray Tracing with Cones, *Computer Graphics (SIGGRAPH '84 Proceedings) 18*, 3 (July 1984), 129-145.

[AWG78] P. R. Atherton, K. Weiler and D. P. Greenberg, Polygon Shadow Generation, *Computer Graphics (SIGGRAPH '78 Proceedings) 12*, 3 (August 1978), 275-281.

[Ber86] P. Bergeron, A General Version of Crow's Shadow Volumes, *IEEE Computer Graphics and Applications 6*, 9 (Sept. 1986), 17-28.

[Bli78] J. F. Blinn, Simulation of Wrinkled Surfaces, *Computer Graphics (SIGGRAPH '78 Proceedings) 12*, 3 (August 1978), 286-292.

[BoK70] J. Bouknight and K. Kelley, An Algorithm for Producing Halftone Computer Graphics Presentations with Shadows and Moving Light Sources, *SJCC, AFIPS 36* (1970), 1-10.

[BrB84] L. S. Brotman and N. I. Badler, Generating Soft Shadows with a Depth Buffer Algorithm, *IEEE CG&A*, October 1984.

[CPC84] R. L. Cook, T. Porter and L. Carpenter, Distributed Ray Tracing, *Computer Graphics (SIGGRAPH '84 Proceedings) 18*, 3 (July 1984), 137-145.

[Coo84] R. L. Cook, Shade Trees, *Computer Graphics (SIGGRAPH '84 Proceedings) 18*, 3 (July 1984), 223-231.

[Coo86] R. L. Cook, Stochastic Sampling in Computer Graphics, *ACM Transactions on Graphics 5*, 1 (January 1986), 51-72.

[CCC] R. L. Cook, L. Carpenter and E. Catmull, An Algorithm for Rendering Complex Scenes, submitted to SIGGRAPH '87.

[Cro77] F. C. Crow, Shadow Algorithms for Computer Graphics, *Computer Graphics (SIGGRAPH '77 Proceedings) 11*, 2 (1977).

[Cro84] F. C. Crow, Summed-Area Tables for Texture Mapping, *Computer Graphics (SIGGRAPH '84 Proceedings) 18*, 3 (July 1984), 207-212.

[Gre86] N. Greene, Applications of World Projections, *Graphics Interface '86*, May 1986, 108-114.

[HaG86] E. A. Haines and D. P. Greenberg, The Light Buffer: A Ray Tracer Shadow Testing Accelerator, *IEEE CG&A 6*, 9 (September 1986), 6-15.

[HeH84] P. S. Heckbert and P. Hanrahan, Beam Tracing Polygonal Objects, *Computer Graphics (SIGGRAPH '84 Proceedings) 18*, 3 (July 1984), 119-127.

[HoN85] J. C. Hourcade and A. Nicolas, Algorithms for Antialiased Cast Shadows, *Computers & Graphics 9*, 3 (1985), 259-265.

[Kay79] D. S. Kay, A Transparency Refraction and Ray Tracing for Computer Synthesized Images, master's thesis, Cornell University, Ithaca, New York, 1979.

[KaK86] T. L. Kay and J. T. Kajiya, Ray Tracing Complex Scenes, *Computer Graphics (SIGGRAPH '86 Proceedings) 20*, 4 (Aug. 1986), 269-278.

[Max86] N. L. Max, Atmospheric Illumination and Shadows, *Computer Graphics (SIGGRAPH '86 Proceedings) 20*, 4 (August 1986), 117-124.

[NON85] T. Nishita, I. Okamura and E. Nakamae, Shading Models for Point and Linear Sources, *ACM Trans. on Graphics 4*, 2 (April 1985), 124-146.

[Pix86] Pixar, Luxo Jr., July 1986.

[Pix87] Pixar, Red's Dream, July 1987.

[ReB85] W. T. Reeves and R. Blau, Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems, *Computer Graphics (SIGGRAPH '85 Proceedings) 19*, 3 (July 1985), 313-322.

[RuW80] S. M. Rubin and T. Whitted, A 3-Dimensional Representation for Fast Rendering of Complex Scenes, *Computer Graphics (SIGGRAPH '80 Proceedings) 14*, 3 (July 1980), 110-116.

[Whi80] T. Whitted, An Improved Illumination Model for Shaded Display, *Communications of the ACM 23* (1980), 343-349.

[Wil78] L. Williams, Casting Curved Shadows on Curved Surfaces, *Computer Graphics 12*, 3 (August 1978), 270-274.

[Wil83] L. Williams, Pyramidal Parametrics, *Computer Graphics 17*, 3 (July 1983), 1-11.