Second-Depth Shadow Mapping

Yulan Wang and Steven Molnar

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

Abstract

Depth-map algorithms for rendering antialiased shadows are computationally efficient and accommodate a wide variety of primitives. They have two drawbacks, however, that make them impractical for everyday use: They require the user to specify a bias that must be optimized for each particular scene and view; and they generally require the use of large shadow maps—frequently much larger than the final image that is to be rendered. This paper presents an improved depth-map shadow algorithm that does not require a bias for scenes composed of solids and produces accurate shadows with smaller depth maps. It consists of two novel changes to the basic depth-map algorithm. First, the depth map samples the depth values of surfaces that are *second* nearest the light source. Second, *virtual* samples are created on local tangent planes of surfaces that are visible from the camera's point of view to match real samples of the depth map. These two changes allow depth comparisons to be done accurately and without the need for a compensating bias value. They also allow the algorithm to produce superior results when using small depth maps. The new algorithm incorporates percentage-closer filtering to antialias shadow boundaries and can be accelerated using *z*-buffer hardware in the same manner as previous algorithms.

1 INTRODUCTION

Shadows are a ubiquitous feature of our visual environment. In computer graphics, shadows help us understand the spatial relationship between objects, offer depth cues, and enhance the realism of synthetic imagery. Unfortunately, computing shadows is a global illumination problem; shadows cast by any object can affect any other object in the scene. Although many algorithms for rendering shadows have been published [Crow77, Max86], most either are restricted to a limited class of modeling primitive or else are computationally too expensive to use in interactive systems.

Ray tracing algorithms [Whitted80] compute shadows naturally, but are computationally expensive. In order to produce anti-aliased shadows, multiple secondary rays must be traced. Although many ray tracing acceleration techniques have been published [Arvo89], none of them yet offers interactive performance on substantial scenes, even running on large multiprocessors.

Shadow volumes [Crow77, Bergeron86] and area subdivision algorithms [Atherton78] are restricted to polygonal data. They also are inefficient for complex environments because the number of polygons to be rendered increases rapidly with the size of the scene.

Depth-map shadow algorithms, introduced by Williams [Williams78], have many advantages: they support all types of primitives; they are relatively efficient; and they can be accelerated using standard *z*-buffer hardware. Williams' original algorithm has limitations, however: it is prone to aliasing and requires the careful selection of a bias when doing *z* comparisons to avoid incorrect self-shadowing.

Subsequent depth-map algorithms have attacked the aliasing problem by testing for shadow at multiple points in each pixel and filtering the results, so-called *percentage-closer filtering* [Reeves87]. These algorithms, however, still require large depth maps to avoid aliasing and require a scene-dependent bias.

This paper attacks the remaining problems. It introduces second-depth shadow rendering, a novel twist to Williams' original algorithm that eliminates the need for a bias when rendering scenes composed of solids. It also introduces *virtual sampling,* computing samples on a tangent-plane approximation to the surface that align with samples in the shadow depth map. Virtual sampling reduces approximation errors in depth values, allowing shadows to be computed accurately with lower-resolution depth-maps. The new algorithm accommodates percentage-closer filtering to antialias shadow boundaries and, like previous algorithms, can make use of fast *z*-buffer hardware.

The remainder of the paper is organized as follows: Section 2 summarizes related work and explains the bias problem. Section 3 introduces the new algorithm. Section 4 discusses implementation issues and contains sample images. Section 5 summarizes the paper and presents conclusions.

2 RELATED WORK AND THE BIAS PROBLEM

The depth-map shadow algorithm developed by Williams operates in two passes, as shown in Figure 1. In the first pass, the scene is rendered from the light source's point of view to obtain a depth map containing the depth of the surface nearest the light source at each pixel. In the second pass, the scene is rendered from the camera's point of view. For each pixel in the camera image, the (x, y, z) location of the sample point on the visible surface is transformed into the light-source coordinate system. This transformed depth value is then compared against the depth value with the nearest (x, y) coordinates in the light-source depth map. If the stored depth is nearer the light source, the pixel color is attenuated—it is in shadow.

Shadows determined from a single depth comparison at one point-sample exhibit aliasing artifacts at shadow boundaries. Reeves' percentage-closer filtering algorithm improves this dramatically by looking up multiple depth values in the neighborhood of each

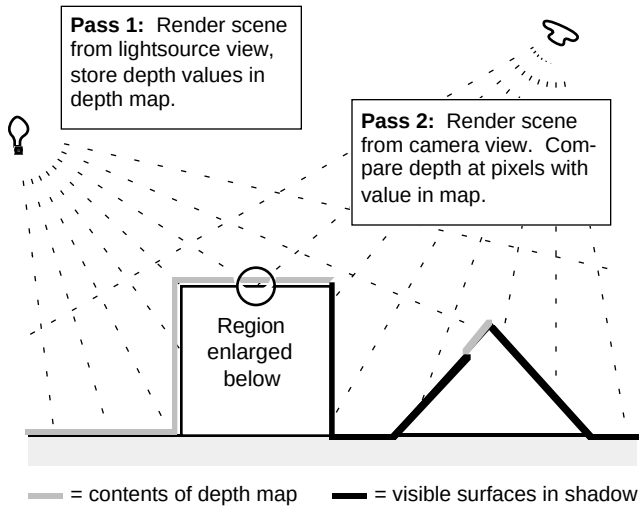


Figure 1: Basic depth-map shadow algorithm.

transformed camera-image sample point, doing a depth comparison with each of these, and computing a fractional shadow value based on the number of comparisons that indicate the sample point is in shadow. This smooths shadow boundaries and produces a penumbra-like effect.

Both Williams' and Reeves' algorithms suffer from a tendency for surfaces to falsely cast shadows on themselves. This arises because pixel samples in the camera image, when transformed into lightsource coordinates, do not align with samples in the lightsource image. Even if both samples lie on the same surface, the depth values may differ, as shown in Figure 2.

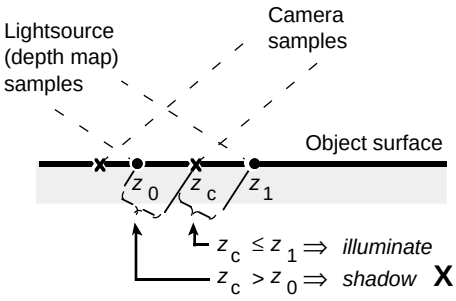


Figure 2: Self-shadowing due to sample mismatches.

Self-shadowing normally is countered using a *bias*, a small constant added to the depths of lightsource samples stored in the depth map to ensure that they lie behind nearby samples in the camera image (Figure 3). Unfortunately, a bias indiscrimately moves shadow boundaries away from the lightsource by the amount of the bias. Choosing a bias, therefore, is a tradeoff between eliminating self-shadowing artifacts and moving shadows from their true positions.

The choice of bias value depends on the scene, including the metric of objects within the scene, the position of the lightsource and the camera, and the resolution of the shadow depth map. Reeves, et. al., claimed that they were always able to find an

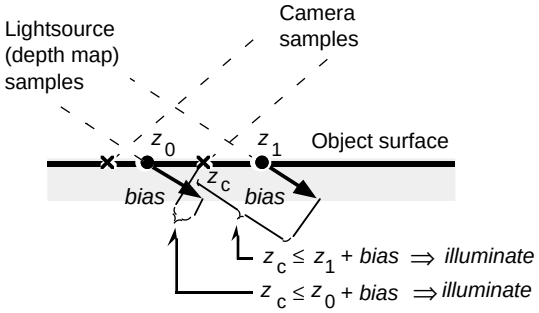


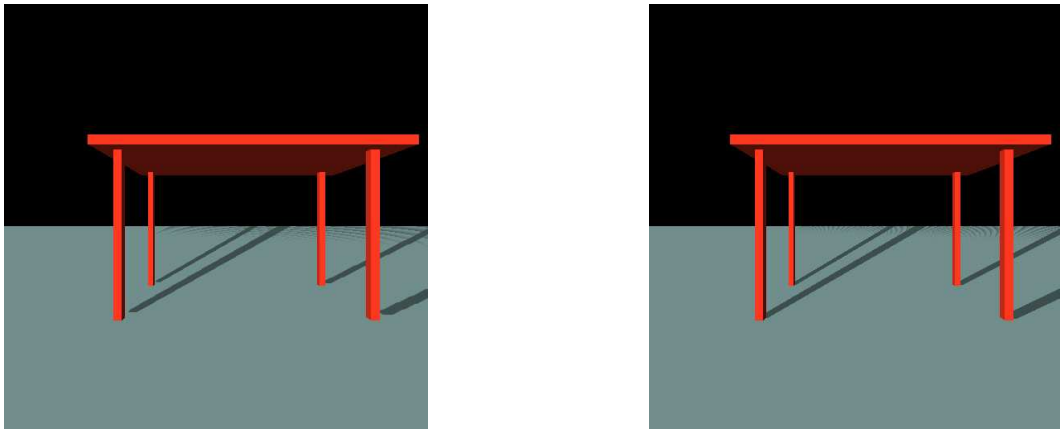
Figure 3: Eliminating self-shadowing using a bias.

acceptable bias for images they rendered. We have found the opposite to be true: simple scenes exist for which no acceptable bias can be found. Figure 4a shows one example. With a 512x512 depth map and a bias of 15, self-shadowing appears on the floor while shadows move away from the legs of the table (the legs touch the floor in the object description). Adjusting the bias in either direction just makes one of the two artifacts worse. However, with a 2048x2048 depth map and a bias of 5, we obtained the better (but still not perfect) image in Figure 4b. Since a larger map produces smaller sampling errors, a larger depth map means a smaller bias can be used, keeping shadow boundaries closer to their true positions.

In our experience, to achieve an artifact-free image one has to increase the size of the depth map until the maximum approximation error between samples is smaller than the thickness of the thinnest object in the scene, and then choose a matching bias. This increases the cost of rendering, since a high-resolution depth map is needed, and requires a scene-specific selection of depth-map resolution and bias value. Percentage-closer filtering algorithm, while reducing aliasing, compounds the bias problem: the larger the sampling area (the more depth-map samples) used for filtering, the larger the approximation errors that the bias needs to offset.

Because of the drawbacks of using a bias, researchers have sought other solutions to the self-shadowing problem. Hourcade [Hourcade85] described a variation of the depth-map algorithm called the *P-buffer*, in which the shadow depth map contains surface tags rather than depth values. If the surface tag in the P-buffer agrees with the surface tag at the current pixel, the surface is assumed to be visible by the light source (not in shadow). If the tags disagree, the surface is assumed to be blocked from the light source (in shadow). This algorithm has problems with curved surfaces that are tesselated, however: if two adjacent polygons have different tags, the algorithm will falsely shadow pixels that lie on the edge that divides the two polygons. If the polygons share a common tag, the surface will not be able to cast shadows on itself. This algorithm, therefore, restricts the type of primitives that can be rendered, one of the major advantages of the original algorithm.

The algorithm described here originated in our attempts to write a robust shadow routine for a real-time shading library. Depth-map algorithms were an obvious choice, but we found that the user-intervention needed to select a bias and the need for large depth maps makes them burdensome in practice. The new algorithm overcomes these disadvantages using a clever twist and some additional information that is often available at rasterization time.



(a) Mapsize = 512x512, bias = 15.

(b) Mapsize = 2048x2048, bias = 5.

Figure 4: Simple scene for which no adequate bias can be found.

3 THE SECOND-DEPTH ALGORITHM

Previous depth buffer shadow algorithms operate by asking the question: Is the depth of the visible surface equal to the depth of the first surface from the lightsource? An equivalent question is to ask: Is the depth of the visible surface *less* than the depth of the *second* surface from the lightsource? The first question leads to standard depth-map algorithms that require a bias. The second question leads to a new algorithm that, for a surprising reason, does not.

3.1 First depth vs. second depth

If we ask the second question, we must store the depth of the second surface from the lightsource in the depth buffer. This is striaghtforward to do. We could use two depth buffers, one for the first surface, and one for the second, and update them appropriately as each primitive is rasterized. There are more efficient methods that are applicable to many scenes, as we will see in Section 4.

Consider the changes that are necessary in Pass 2. For each pixel in the camera image, we must compute z_{camera}, the depth of the transformed camera-image sample, and compare it with z_{second}, the second surface depth stored in the depth buffer. The two possible outcomes are:

- $z_{camera} < z_{second}$. The camera sample probably lies on the first surface and should be illuminated. However, it could lie on the second surface (because of sampling errors) and be falsely illuminated.
- $z_{camera} \geq z_{second}$. The camera sample lies on the second or greater surface and should be in shadow.

We now have an ambiguity on the second surface similar to the ambiguity that causes self-shadowing in first-depth algorithms. But here the advantage of the second-depth algorithm becomes clear—we know more information about the second surface. If the dataset is constructed of solids, the second surface from the lightsource will face away from the lightsource and illumination calculations will indicate that it is in shadow, regardless of the result of depth comparison. *We don't need a correct depth comparison for points on the second surface.*

We can see more clearly how the algorithm operates using an example. Figure 5 shows a 2D slice through a moderately complicated environment. The second surface from the lightsource is represented by bold gray lines. Visible surfaces in shadow are represented by bold black lines.

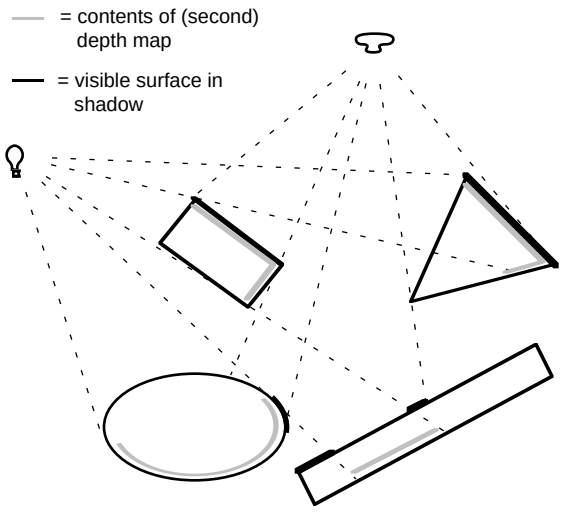
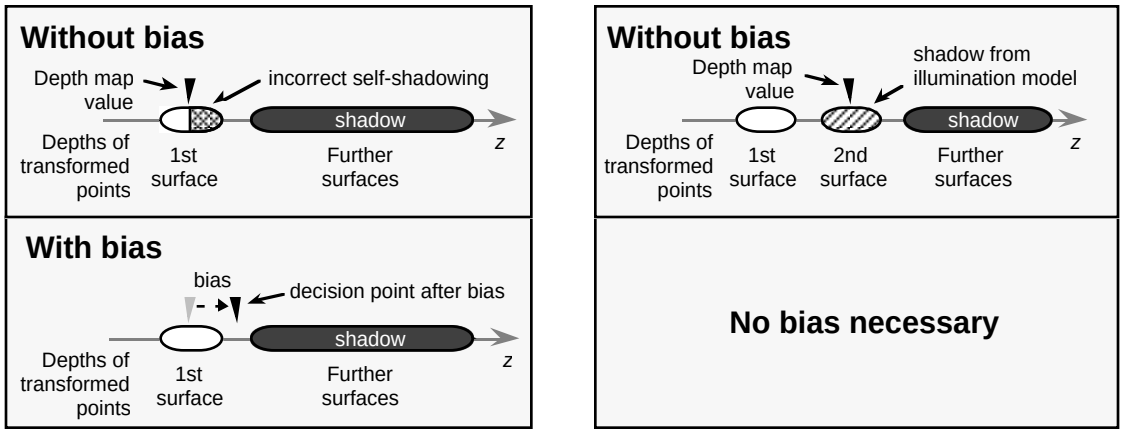


Figure 5: Example of second-depth algorithm.

Figure 6 shows a more abstract view of the difference between first-depth and second-depth algorithms. The ovals represent the range of depth values that each successive surface in the lightsource view could have if we take sampling approximation errors into account. The vertical wedge represents the depth value stored in the depth map.

In first-depth algorithms, the decision point for illumination/shadow is at the first surface. To avoid self-shadowing, we must translate the decision point beyond the first surface using a bias. But we must not translate it so far that pixels from the second surface are wrongly classified. Herein lies the tricky nature of choosing a bias. With small depth maps, the sample errors become larger, so surfaces are harder to distinguish.



(a) First-depth algorithm (b) Second-depth algorithm

Figure 6: Difference between first-depth and second-depth algorithms.

In second-depth algorithms, the decision point at the second surface cleanly partitions the first surface (illuminated) from the third and succeeding surfaces (shadowed). Ambiguities at the second surface do not matter because it is already known to be in shadow. Of course, this algorithm can misclassify surfaces too if sampling errors are so large that the pixels from the first or third surface cross the decision point.

So, although both algorithms have trouble distinguishing the surface stored in the depth buffer, first-depth algorithms must do so (and therefore require a bias), whereas second-depth algorithms do not. Since the second-depth algorithm is nearly as easy to implement as the first-depth algorithm, we conclude that it is the superior choice.

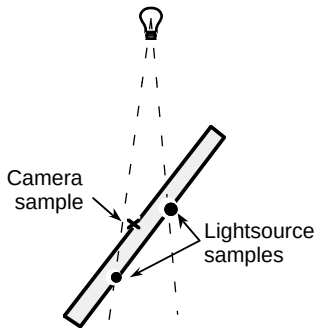


Figure 7: Ambiguity between first and second surface of a thin object.

3.2 Virtual samples

Depth comparisons without a bias depend on an adequate separation between first and second surfaces. This can be violated for very thin primitives, as shown in Figure 7. Here a camera sample on the (first) surface of a thin object lies behind a lightsource sample on the second surface of the object, falsely indicating that the pixel is in shadow.

If we knew the orientation of the surface at each pixel, we might be able to prevent this kind of artifact. We could perturb the (x, y) coordinates of the pixel sample to line up with lightsource samples and adjust the depth values accordingly.

Fortunately, this information often is available. Shading models typically require knowledge of a surface-normal vector. In some cases (*e.g.* Phong shading) this information is available at each pixel, but in other cases (*e.g.* flat or Gouraud shading), an approximation to the surface-normal is available at each pixel. The normal vector, together with the (x, y, z) location of the sample point define a tangent plane that touches the surface at the sample point and closely approximates it in the neighborhood of the sample point.

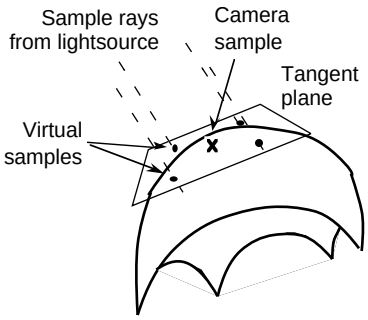


Figure 8: Virtual samples on local tangent plane.

We can resample this tangent-plane approximation to obtain approximate depth values at points near the pixel sample. In particular, we can resample at points to match samples in the depth map. We call these points "virtual samples" because they sample a virtual (tangent-plane) surface (Figure 8).

For planar primitives (polygons, etc.), the tangent plane coincides with the actual surface, so the depths computed at virtual samples are exact. For curved surfaces (splines, NURBS, or finely tesselated polygonal surfaces), the surface may deviate from the tangent plane, leading to errors in depth values. These errors normally are small, but can become large if the tangent plane is almost parallel to the light sample ray, as shown in Figure 9.

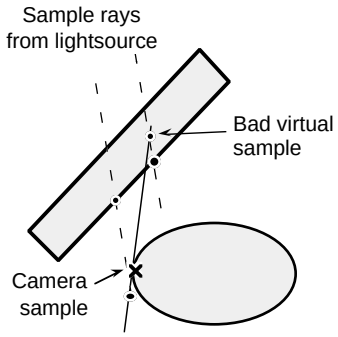


Figure 9: Potential error when virtual sampling.

Note that in this case, virtual sampling is not required to get the correct result—normal depth comparisons indicate the surface is in shadow. To prevent errors like this, we have adopted the heuristic of only computing virtual samples for pixels whose normal depth comparisons are ambiguous. We use the following rules to process each pixel:

- If $z_{camera} < z_{second}$ for all nearby samples, the pixel is fully illuminated.
- If $z_{camera} \geq z_{second}$ for all nearby samples, the pixel is fully in shadow.
- If comparisons are mixed, compute tangent plane and virtual samples; compare depth at virtual samples with depth at corresponding location in depth map; do percentage filtering on the results.

This scheme does not use all the information that is potentially available for each pixel, so it may cause errors that a more sophisticated heuristic could catch. It has the advantage of minimizing the number of pixels that need virtual sampling, however, which reduces the cost of the algorithm.

3.3 Non-solid primitives

Thus far we have assumed that all objects that can cast shadows have thickness. We can extend the algorithm to non-solids by considering them to be solids with infinitesimal thickness—the "first" and "second" surface are the same. By virtual sampling, we can detect such two-sided surfaces if they are planar. We change the 'less-than' comparison in the illumination test to 'less-than-or-equal'. If the surface is planar, the depth value at virtual samples will be the same as the ones in the depth map. If two-sided surfaces are curved, however, we will need a bias to achieve the correct answer.

This removes one of the main advantages of the algorithm. In fact, for scenes composed entirely of non-solids, the second-depth algorithm degenerates to a first-depth one.

All is not lost, however. We can implement the second-depth algorithm with a bias parameter that defaults to zero. If a scene is composed of solids, it can be rendered without further attention. If the scene contains non-planar, non-solids, the user must select an appropriate bias. Therefore, a single routine can compute shadows for any type of scene.

3.4 Implementation details

The two passes in the second-depth algorithm are similar to the two passes in conventional depth-map shadow algorithms. Only now the first pass computes the depth of the second surface from the lightsource view and the second pass performs a more complicated set of depth comparisons.

Pass 1: Computing the second-depth map

The only modification to this pass is that the depth map must store the second surface from the lightsource point of view. This can be done in at least two ways. The most straightforward is to maintain two depth buffers when rendering this image: one stores the depth of surface closest to the lightsource; the other stores the depth of the second surface. Each time a primitive is rasterized, one or both of these depth buffers may need to be updated. At the end of the pass, the front surface can be discarded.

An alternate method, useful on scenes composed of polygons, is to cull out polygons that face the lightsource before they are rasterized and perform a standard *z*-buffer algorithm on the remaining polygons. In scenes composed of solids, the resulting surface will be the first shadowed surface, or the second surface overall. This method of *front-face culling* can be adapted to scenes with curved surfaces if surface-normal vectors are available at each pixel; only pixels whose normal vectors face away from the lightsource ($N \bullet L < 0$) should participate in *z*-buffer calculations. If the scene contains two-sided surfaces, the side facing away from the lightsource must be rendered in Pass 1, and the side facing the lightsource rendered in Pass 2.

```
// Return percent illumination for a pixel
float percent_illumination()
{
   int IllumCnt     = 0;
   int VirtIllumCnt = 0;

   // Do normal depth comparisons (test is ≤
   // to handle two-sided polygons correctly)
   for each nearby sample in depth map
      if (zcamera ≤ zsecond)
         IllumCnt++;

   // Do trivial shadow/illuminate test
   if (IllumCnt == 0)
      return(0.0);
   if (IllumCnt == NUM_SAMPLES)
      return(1.0);

   // Compute virtual samples and filter
   Fit local tangent plane to surface;
   Transform plane to lightsource coords;
   for each nearby sample in depth map
      if (depth at virtual sample < zsecond)
         VirtIllumCnt++;

   // Percentage closer filtering
   return(VirtIllumCnt / NUM_SAMPLES);
}
```

Figure 10: Algorithm for determining percent illumination for a pixel.

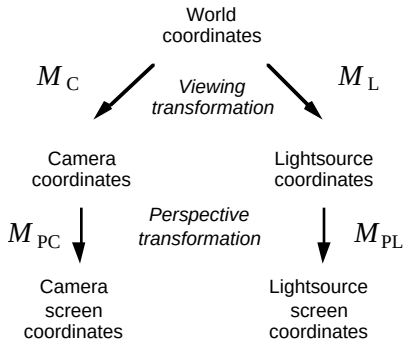


Figure 11: Coordinate systems for shadow calculations.

Pass 2: Computing the camera view

In the second pass, the shadow test is modified to do depth comparisons with the second surface and to compute virtual samples when necessary. Figure 10 gives the algorithm for determining the percent illumination for a given pixel.

For pixels that require virtual sampling, the first step is to fit a local tangent plane to the surface at the pixel. It is most convenient to do this in the coordinate system in which surface normal vectors are defined. Assume for now that normal vectors are defined in world space and that 4x4 transformation matrices for transforming points from world space into camera and light coordinates are as shown in Figure 11. (Certain details of the algorithm vary slightly if different normal vectors are defined in a different coordinate system, but the essence of the algorithm remains the same).

If a pixel has camera-space screen coordinates $P_C = (x_C, y_C, z_C)$ and a world-space normal vector $N_W = (n_{XW}, n_{YW}, n_{ZW})$, the first step is to transform P_C into world coordinates. This is done by transforming it by the inverse of the world-to-camera screen coordinate matrix:

$$P_{\mathbf{W}} = \left(M_{\mathbf{PC}} M_{\mathbf{C}}\right)^{-1} P_{\mathbf{C}}.$$

The local tangent plane in world coordinates satisfies the equation:

$$a_{\mathbf{W}} x + b_{\mathbf{W}} y + c_{\mathbf{W}} z + d_{\mathbf{W}} = 0,$$

where $a_W = n_{XW}$, $b_W = n_{YW}$, $c_W = n_{ZW}$, and $d_W = -a_W \bullet x_W - b_W \bullet y_W - c_W \bullet z_W$. The plane can be written as a column vector:

$$T_{\mathbf{W}} = \begin{pmatrix} a_{\mathbf{W}} \\ b_{\mathbf{W}} \\ c_{\mathbf{W}} \\ d_{\mathbf{W}} \end{pmatrix}.$$

The next step is to transform T_W into lightsource coordinates. We know from linear algebra that if M is a point transformation matrix, $(M^{-1})^T$ is the corresponding plane transformation matrix. The composite matrix $M_{PL}M_L$ transforms points from world coordinates to lightsource screen coordinates, so the matrix

$$\left(\left(M_{\mathbf{PL}} M_{\mathbf{L}}\right)^{-1}\right)^{\mathbf{T}}$$

transforms a plane from world to lightsource screen coordinates. The tangent plane in lightsource screen coordinates, therfore, is given by:

$$T_{\mathbf{L}} = \left(\left(M_{\mathbf{PL}} M_{\mathbf{L}}\right)^{-1}\right)^{\mathbf{T}} T_{\mathbf{W}}.$$

To determine depth values at virtual sample points (x_L, y_L), we solve the following equation for z_L, the depth of the virtual sample:

$$a_{\mathbf{L}} x_{\mathbf{L}} + b_{\mathbf{L}} y_{\mathbf{L}} + c_{\mathbf{L}} z_{\mathbf{L}} + d_{\mathbf{L}} = 0.$$

The solution is a linear expression in (x_L, y_L):

$$z_{\mathbf{L}} = \left(\frac{-a_{\mathbf{L}}}{c_{\mathbf{L}}}\right) x_{\mathbf{L}} + \left(\frac{-b_{\mathbf{L}}}{c_{\mathbf{L}}}\right) y_{\mathbf{L}} + \left(\frac{-d_{\mathbf{L}}}{c_{\mathbf{L}}}\right),$$

We can evaluate this linear expression for each virtual sample, or, since lightsource samples lie on a regular grid, we can evaluate this expression for one virtual sample and use forward differences to calculate the remaining virtual samples using only additions. Using this formulation, the arithmetic needed to compute virtual samples reduces to two matrix-vector multiplies, three divisions, and just a few additions and multiplications per pixel.

4 DISCUSSION AND EXAMPLES

4.1 Robustness

This new shadow algorithm is still a discrete sampling algorithm and can produce errors when the scene is under-sampled—for example, if the depth map is too small or if objects in the scene fit between depth-map samples. Storing the second rather than the first depth helps, as does virtual sampling. When artifacts still occur, the only solution is to increase the size of the depth map.

4.2 Efficiency

The new algorithm requires very little extra computation compared to previous algorithms. If front-face culling is used, the first pass costs precisely the same as the first pass in first-depth algorithms.

In the second pass, virtual sampling is invoked only when the camera sample lies between the minimum and maximum depths of neighboring samples in the depth map. This only happens near object silhouettes or when objects are very thin—generally a small fraction of the pixels in the entire scene. Decreasing the size of the depth map increases the fraction of pixels that require virtual sampling, so this can be considered a tradeoff of computation for space.

4.3 Depth map size

The new algorithm allows us to reduce the size of depth maps compared to previous algorithms. This occurs for two reasons:

- Depth comparisons are generally between surfaces (first and second) that are disjoint in z, a bias is not required. There is no hard threshold at which self-shadowing takes place.

(a) 512x512 depth map. (b) 256x256 depth map. (c) 128x128 depth map.

Figure 12: Degradation of image quality with decreasing size of depth map.

- Virtual sampling increases the precision of depth comparisons.

With the new algorithm, depth maps of moderate size can produce quite good images, as shown in Figure 12a, a 512x512 image generated with a depth map of 512x512. As the depth map becomes smaller and smaller, shadows begin to leak out and image quality degrades slowly. Figure 12b was generated with a depth map of 256x256. Figure 12c was generated with a depth map of 128x128. We think the image quality is acceptable considering the small size of the depth map.

4.4 Sample images

Figures 13, 14, and 15 are sample images rendered with the second-depth/virtual sampling shadow algorithm. Figure 12 is a 1kx1k image of a procedural dataset. Figure 13 is a 1kx1k scene from an architectural walkthrough. Figure 14 is a 640x512 still from a video sequence designed to demonstrate the real-time shading performance of PixelFlow, the experimental graphics engine we are building. All of these images were rendered without a bias.

5 CONCLUSION

The second depth-map shadow algorithm has many features that make it appealing as a utility shadow routine: It is general and efficient like previous depth-map algorithms. It is more convenient to use, since no bias is required for most scenes. Finally, it reduces depth map size compared to previous algorithms.

The new algorithm derives from the observation that most objects that cast shadows are solid. This allows us to transfer the unavoidable depth ambiguities from a discrete sampling algorithm from the first surface of objects to the second surface, where they can be resolved with the assistance of the illumination model. Virtual sampling further increases the robustness of the algorithm by matching virtual camera samples to real samples in the depth map, rather than attempting to reconstruct depth values from the discrete depth-map samples.

The new algorithm is efficient, since the second-surface depth map can usually be computed as easily as a first-surface depth map. The additional computation cost for virtual sampling is small, since we take advantage of the fact that the screen-space representation of the local tangent plane is still a linear equation. Furthermore, we do virtual sampling only when it is likely to help. Percentage closer filtering is easily incorporated into the algorithm to generate anti-aliased shadow boundaries. The algorithm has proven to be more robust and easy to use in our real-time shading library.

ACKNOWLEDGEMENTS

We wish to thank Gary Bishop, Anselmo Lastra, Turner Whitted, and the members of the PixelFlow team for their ideas and comments. We wish to thank Lee Westover, the UNC Walkthru project, and Pixar for the use of their datasets. Finally, we wish to acknowledge our sponsors, NSF (Grant No. MIP-9306208) and ARPA (ISTO Order No. A410).

REFERENCES

[Arvo89] J. Arvo and D. Kirk, A Survey of Ray Tracing Acceleration Techniques, in *An Introduction to Ray Tracing*, ed. A.S. Glassner, Academic Press, San Diego, (1989), 201-262.

[Atherton78] P. R. Atherton, K. Weiler and D. P. Greenberg, Polygon Shadow Generation, *Computer Graphics* (SIGGRAPH '78 Proceedings) 12, 3 (1978), 275-281.

[Bergeron86] P. Bergeron, A General Version of Crow's Shadow Volumes, *IEEE CG&A*, 6, 9 (Sept. 1986), 17-28.

[Crow77] F. C. Crow, Shadow Algorithms for Computer Graphics, *Computer Graphics* (SIGGRAPH '77 Proceedings) 11, 2 (1977), 442-448.

[Hourcade85] J.C. Hourcade and A.Nicolas, Algorithms for Antialiased Cast Shadows, *Computer & Graphics*, 9, 3 (1985), 259-265.

[Max86] N.L. Max, Atmospheric Illumination and Shadows, *Computer Graphics* (SIGGRAPH '86 Proceedings) 20, 4 (1986), 269-278.

[Reeves87] William T. Reeves, David H. Salesin, Robert L. Cook, Rendering Antialiased Shadows with Depth Maps, *Computer Graphics*, SIGGRAPH '87 Proceedings), 21, 4 (1987), 283-291.

[Whitted80] T. Whitted, An Improved Illumination Model for Shaded Display, *CACM*, 23, 6 (June 1980), 343-349.

[Williams78] L. Williams, Casting Curved Shadows on Curved Surfaces, *Computer Graphics* (SIGGRAPH '78 Proceedings), 12, 3 (1978), 270-274.

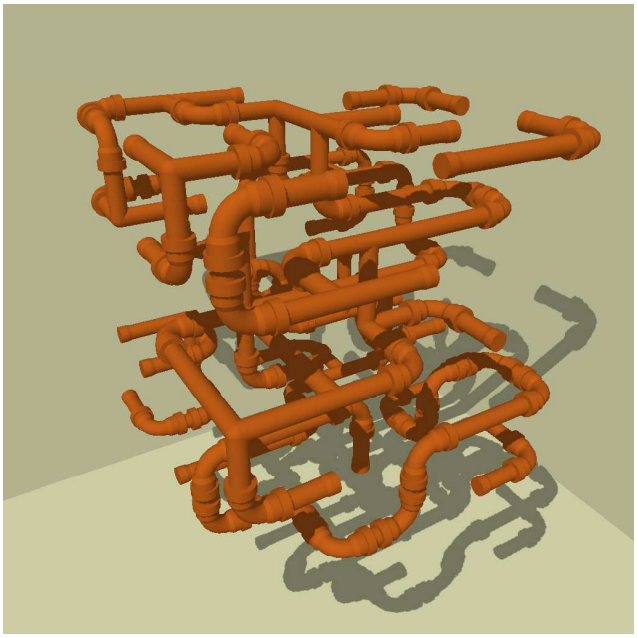


Figure 13: Procedural pipes model. 1024x1024 resolution. 512x512 depth map. (Dataset courtesy of Lee Westover, Sun Microsystems, Inc.)]

Figure 14: Scene from architectural walkthrough. 1024x1024 resolution. 2048x2048 depth map. Dataset courtesy of Building Walkthrough Project, UNC-CH).

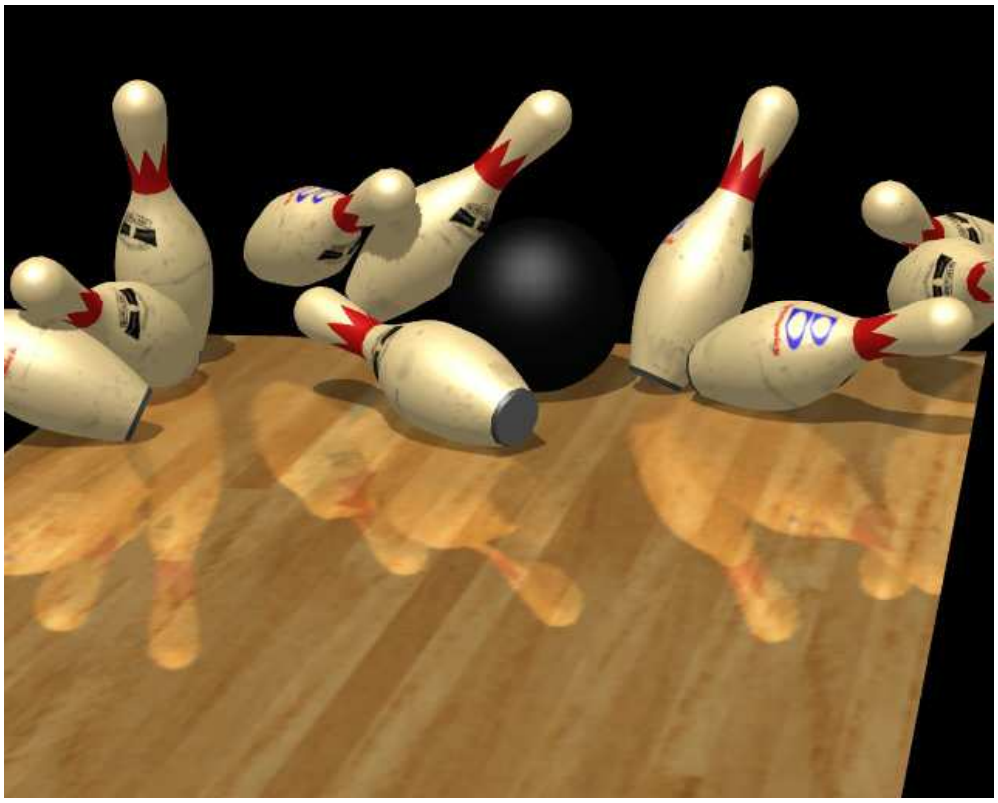


Figure 15: Still from bowling video sequence rendered on PixelFlow simulator. 640x512 resolution. 512x512 depth map. (Textures courtesy of Pixar).