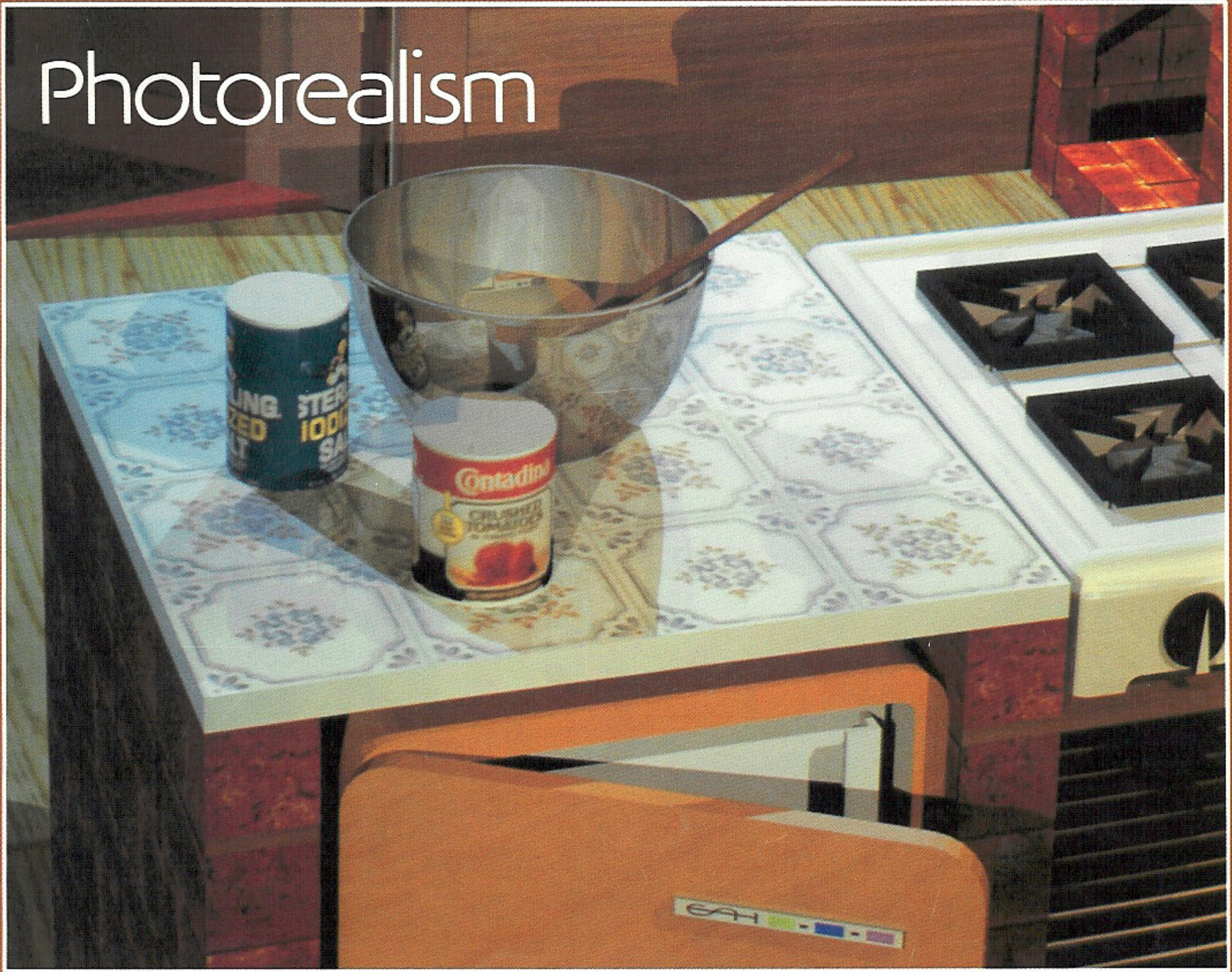
IEEE

Computer Graphics and applications



Photorealism

SEPTEMBER 1986

Also in this issue:

- Shadow Casting in *Tony de Peltrie*
- Two-Part Texture Mapping
- Depth Buffer for Solid Geometry

IEEE COMPUTER SOCIETY

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

The Light Buffer: A Shadow-Testing Accelerator

Eric A. Haines and Donald P. Greenberg
Cornell University

In one area of computer graphics, realistic image synthesis, the ultimate goal is to produce a picture indistinguishable from a photograph of a real environment. A particularly powerful technique for simulating light reflection—an important element in creating this realism—is called ray tracing. This method produces images of excellent quality, but suffers from lengthy computation time that limits its practical use.

This article presents a new method to reduce shadow testing time during ray tracing. The technique involves generating light buffers, each of which partition the environment with respect to an individual light source. These partition descriptions are then used during shadow testing to quickly determine a small subset of objects that may have to be tested for intersection.

The results of timing tests illustrate the beneficial performance of these techniques. The tests compare the standard ray-tracing algorithm to light buffers of varying resolution.

Ray-tracing algorithms are not new. Early versions were presented by Appel[1] in 1968 and Goldstein and Nagel,[2] in 1971. By 1980, a refined ray-tracing procedure had been implemented.[3] Unfortunately, the shadow-testing and ray-intersection routines were taking up nearly 95 percent of total computational time—a portion obviously unacceptable in practical environments. Thus, subse-

0272-1716/86/0900-0006$01.00 © 1986 IEEE

quent work in ray-tracing algorithms was devoted to finding ways to speed up the process. This article presents a method of accelerating shadow testing, which is the most computationally intensive part of ray tracing. To illustrate its significance, let us look at ray tracing more closely.

A bit about ray tracing

Ray tracing approximates the global illumination of a scene by tracing rays from the eye through the viewing plane and into the 3D environment. For each ray the closest surface intersection point is determined. Depending on the material properties, this surface may spawn reflected and refracted rays, which are recursively traced in the same fashion. As the ray is propagated through the environment, a tree of intersection locations is constructed for each sample point. The final image intensity of this point is determined by traversing the tree and computing the contribution of each node according to a shading model.

The intensity calculation for each node consists of two parts. The first part determines if the intersection point can be seen from each light source. If no opaque object blocks the light from the intersection point, then the second part consists of using the attributes of the surface and the light source to calculate the contribution to the final color intensity of the pixel.

The first operation, called *shadow testing*, is normally the most computationally expensive process of the ray-tracing algorithm. The reason is that each object in the entire environment must be tested to see if it occludes each light source for every ray intersection point. For scenes with complex lighting schemes the percentage of time required for occlusion testing can increase to over 80 percent of the total computation time.[4]

A number of methods, based primarily on using object and image coherence to speed processing, have been used to accelerate ray tracing calculations. With object coherence, we use environmental properties such as bounding volumes or partitions to speed up ray intersection. Image coherence, which is view dependent, relies on the fact that a pixel of a given image is likely to be similar to its neighbors.

The idea of surrounding intricate objects with simpler bounding volumes is widely accepted.[3-6] The expensive intersection time for complex polyhedrons or parametric surfaces is reduced by using simpler objects such as boxes or spheres as enclosures. If the bounding volume is not intersected, we can avoid the test for the more intricate object.

Another set of methods is based on partitioning the environment in a variety of fashions. Rubin and Whitted[5] decreased the intersection calculation time significantly by subdividing the environment into small orthogonal volumes, which they further subdivided and accessed hierarchically. Glassner[7] and Kaplan[8] independently researched procedures based on an octree encoding of the environment that saved time through an efficient access and intersection of rectangular volumes. Dippe also presented a concept for subdividing an environment for parallel computations.[9]

One method, introduced by Weghorst et al.,[4] uses a visible surface preprocessor from the observer's position. This preprocess took advantage of image coherence by using a scan-line algorithm to create an item buffer, which contained the first node of each intersection tree. Since the average tree depth in a complex image is often quite low,[10] the savings were substantial.

The light buffer

Our method for reducing shadow testing time during ray tracing involves generating a *light buffer*.[11] The concept is based on the idea that a point can be determined to be in shadow, without having to find which object first occludes it. Such knowledge can decrease occlusion testing time. When data is referenced using the direction of the light ray, a light buffer results. Thus, an idealized light buffer is defined as having two algorithms:

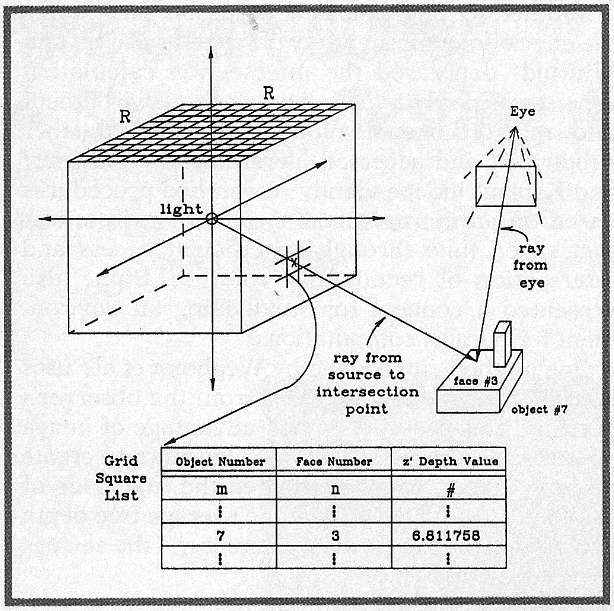
1. a procedure to partition the environment with respect to each light's position
2. a procedure to test if a given point is in shadow by using this partitioned definition of the environment

Bouknight and Kelley[12] and Williams[13] have used similar approaches to perform shadow generation under limited conditions.

In this section we will describe the partitioning and shadow-testing routines for environments consisting entirely of opaque polygonal surfaces. Techniques for handling nonpolygonal and transparent surfaces will be presented later.

Opaque polygonal surfaces

For each light source a light buffer is constructed. These buffers are modeled on the "hemicube" buffer concept developed for use in radiosity calculations.[14] A light buffer can be idealized as a cube-shaped frame surrounding a light source. The cube is then aligned with the environment's coordinate axes, and the cube faces form six windows sur-



Object Number	Face Number	z' Depth Value
m	n	#
⋮	⋮	⋮
7	3	6.811758
⋮	⋮	⋮

Figure 1. Schematic representation of a light buffer.

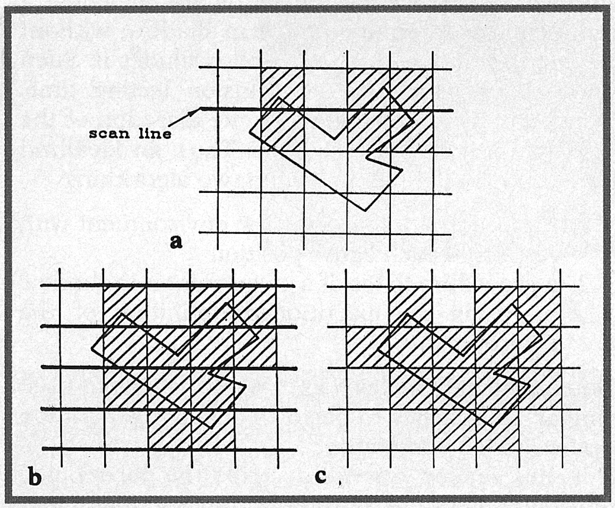


Figure 2. Polygon conversion process. In (a) grid squares are marked for one horizontal scan line; in (b) they are marked after entire horizontal scan-line process; and in (c) they are marked after tracing polygon edges.

rounding the light (Figure 1). Each face is associated with one of the orthogonal axes within the environment. The axis perpendicular to a particular cube face is designated the z' direction, with the edges of the face defining the x' and y' axes. The exact orientation of the x' and y' axes is arbitrary and has no effect on the algorithm.

Each cube face has a gridwork of squares, with the number of squares on an edge referred to as the light buffer's resolution, or R. Each cell within this gridwork contains a list of object surfaces that can be seen from the light. Each grid square is thought of as a small window through which the light source can look. The objects that the light can see through this grid square are saved in that cell's list. Franklin[15] used a similar concept to reduce the sorting time of his hidden-surface algorithm.

Each list record contains an object number, surface number, and a relative depth value. The object number, m, is an integer from 1 to M, where M is the number of objects in an environment. The surface number, n, is an integer from 1 to N, where N is the number of polygons defining an object. The relative depth from the light is stored as a floating point number for each list record. The sign bit of these numbers is used to mark various types of list records. The records in a grid square list are sorted in ascending order by depth.

To rapidly determine which object surfaces are seen within which grid squares, a modified scan-line algorithm is used. A display scan-line algorithm determines which polygon is seen at each point on a grid overlaying the image. For a light buffer the important determination is which polygonal surfaces can possibly be projected onto each grid square. Because of this difference, the standard scan-line algorithm must be modified to avoid any aliasing.

The basic procedure is to cast all polygons onto each face of the light buffer cube. Transformation, clipping, and culling are performed for each polygon, six times for each light buffer. Casting consists of a number of steps. First, each polygon undergoes a view transformation so that it is in the coordinate system of the light. Polygons that do not lie within the viewing frustum are ignored. This test is performed by using the Cohen-Sutherland algorithm[16] to determine whether a polygon is possibly within the viewing frustum.

The polygons facing away from the light are then culled by determining the polygon's normal. After culling, the remaining polygons undergo a perspective transformation and are projected onto the light buffer face. An edge table is created for each polygon that has survived clipping and culling. A scan-line algorithm is then used to project the polygons onto the image plane.[17] For each horizontal scan line the exact edge intersection points are ignored; instead, to avoid aliasing, all grid squares adjoining the part of the scan line covered by the polygon are marked. This process is shown for one horizontal scan line in Figure 2a and for all relevant scan lines in Figure 2b.

In Figure 2b two squares are not marked by the scan-line process. One method to avoid this aliasing is to trace along the polygon's perimeter. Each edge is used to mark grid squares until it hits a horizontal scan line. The result is shown in Figure 2c.

A final test is made to check if each polygonal surface is marked into at least one grid square. If not, the polygon is entirely enclosed by a single grid square, signifying that it was missed by the scan-line procedures. To correct the oversight, any vertex of the polygon is cast on the light buffer cube face, and the grid square intersected is marked.

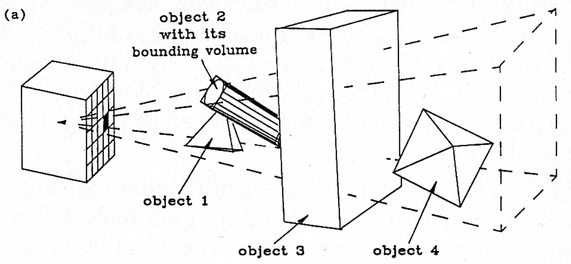
For each marked grid square, the closest depth for each polygon is calculated. This depth is used as a rough sorting key during ray tracing to determine when shadow testing is terminated. To increase preprocessing speed, computations are first performed on the transformed (image space) data, then converted back to true depth (object space) values. The polygonal records are inserted in the list associated with each grid square in ascending order by depth.

The algorithm is repeated for each polygon of each object. At the end of this process, we have information about one face of the basic light buffer cube. The process is repeated as needed for each of the six light buffer faces for each light source. When all light buffers are completed and ready for use, the environment has been effectively partitioned and greatly reduced shadow testing times are now possible. To illustrate this procedure for one grid square, the environment of Figure 3a has been processed, with the basic list shown in Figure 3b.

Cube coherence can be used to reduce the number of calculations by casting an object onto only those light buffer faces that can view it. To begin the process, a point within the object is cast on the buffer, and the face hit is determined. The object is cast on this face, and the edges of the light buffer that were used for clipping are recorded. The object is then cast on only the appropriate contiguous faces. Cube coherence reduces the time needed to create a light buffer by approximately half that ordinarily required.

During shadow testing the light buffers are accessed for each view ray intersection point, and a list is retrieved of all surfaces that might occlude the light ray. Each polygon in the list must be tested until an occlusion is found or until the depth of the potentially occluding polygon is greater than the intersection point.

As with standard ray-tracing algorithms, intersection calculations can be accelerated by creating simple bounding volumes that encompass complicated objects.[4,5] If the bounding volume is not intersected, then the complex object within will not be intersected. A similar bounding volume approach can be used to create the light buffer lists. Within the light buffer, bounding volumes are added when the number of polygons affecting a grid square for an object exceeds a given value. A threshold of six was used in the implementation presented.



		z'
Obj	Face	Depth
1	2	1.28
1	3	1.28
2	1	1.56
2	6	1.56
2	7	1.99
2	4	2.05
2	5	2.10
2	8	2.20
3	4	3.69
4	1	4.48
4	2	4.48
4	7	4.48
4	8	4.48

		z'
Obj	Face	Depth
1	2	1.28
1	3	1.28
-2	0	1.56
2	1	1.56
2	6	1.56
2	7	1.99
2	4	2.05
2	5	2.10
2	8	2.20
3	4	3.69
4	1	4.48
4	2	4.48
4	7	4.48
4	8	4.48

		z'
Obj	Face	Depth
1	2	1.28
1	3	1.28
-2	0	1.56
2	1	1.56
2	6	1.56
2	7	1.99
2	4	2.05
2	5	2.10
2	8	2.20
3	4	3.69
3	0	-4.15

(b) ——→ (c) ——→ (d)

Figure 3. Creation of a grid square list. We begin with (a) the basic environment and (b) corresponding basic list; (c) is the result of inserting a bounding volume record into the basic list; and (d) is the result of inserting maximum *z′* depth records and pruning the list.

Except for a special negative identifier, a bounding volume record is similar to a polygonal surface list record. The depth is set to the depth of the first (and closest) surface of the enclosed object and the bounding volume record is inserted immediately before this surface. All subsequent records of polygons belonging to that object must then be moved to form a contiguous group following the bounding volume record, regardless of their depth values. An insertion of a bounding volume record into a basic list is shown in Figure 3c.

A second method to speed intersection testing is based on the observation that if an object fully covers a grid square, all intersection points beyond that object will be in shadow. The method is similar to the "surrounder" test performed in the Warnock algorithm concerning the area subdivision of visible surfaces.[18] This test is performed on all convex objects and all polygons to determine the minimum depth of full occlusion. A full occlusion record is then inserted into the list using this depth. Full occlusion records are flagged by assigning a negative z' depth. The effect of a full occlusion record is to create a shadow volume for a cell that begins at this full occlusion depth.

When all objects have been projected on the cube face, the lists can be scanned and simplified. This simplification saves both memory and time during shadow testing. If there is only one polygon in a list, then the list can be deleted, since that polygon has no other surfaces to cast a shadow on, nor can it occlude itself. Similarly, all polygons beyond a full occlusion record can be deleted, since all objects beyond this record are automatically in shadow. If there are no deeper polygons, the full occlusion record itself can be deleted. The result of inserting maximum z'-depth records in the list and pruning it is shown in Figure 3d.

Accessing light buffers

When an environment is ray traced, the light buffers are accessed during shadow testing. A light vector is generated from each light source to the intersection point of the view ray and the surface (Figure 1). The surface normal is checked to determine if it points away from the light. If the surface faces away from the light, it is in shadow, and testing ends. Otherwise, the list from the appropriate light buffer grid square is retrieved and used to test for occlusion. The first step is to find which grid square the light vector passes through. Because of the cube geometry, the face is quickly determined by finding the largest component of the ray and noting its sign (for example, a ray with vector $[x,y,z] = [3.11,-5.04,-1.66]$ will hit the $-y$ face). The remaining two vector components are converted to grid coordinates of the face, and the appropriate grid square list is retrieved and used to test for shadowing.

The last record in the list is examined. If it is a full occlusion record and its depth is less than the intersection point's z' depth, then the intersection point is in shadow, and no more testing is necessary. If this test fails, then the list is traversed from the beginning. If a record is a polygonal surface, it is tested for intersection. Intersection tests are not performed in two cases. The first is when the polygon in the list is the surface intersected, called the *base surface*. Since this surface can never occlude itself it is not checked.

The second case is when the polygon and the base surface are part of the same object, and the object is convex. A convex object cannot cast a shadow on itself and so can be ignored.

Records of the bounding volume list are handled differently. The bounding volume is tested for intersection. If there is no intersection, then the surfaces within the bounding volume are skipped. In the special case when the bounding volume is the parent of the base surface, then the bounding volume has automatically been hit and the list examination continues normally.

After a polygonal surface or bounding volume record has been examined, the process continues and the next list record is retrieved. A test is made to determine if the new record is beyond the intersection point. If the z' depth of the new record is greater than the base point intersection depth, one of two events occurs. If the record is not a part of a bounding volume list, testing is finished. If the record tested is within a bounding volume, the testing skips to the first record not within the bounding volume. Records within a bounding volume are not sorted by depth with respect to the rest of the list, so testing must continue. Records are retrieved until either a surface is hit, the z' depth of list records is beyond the base point, or the end of the list is reached. In the first case the base point is in shadow; otherwise it is illuminated by the light.

Nonpolygonal objects

Our discussion so far has dealt only with polygonal objects. The techniques used to insert polyhedrons into a light buffer can also be generalized to nonpolygonal objects. To insert a nonpolygonal object, we begin by forming a polygonal hull that encloses the nonpolygonal object (Figure 4a). Casting this polygonal hull onto each light buffer identifies the grid squares that may contain the complex object. The polygonal hulls created for this task cannot be used for full occlusion testing, however, because each occludes an area larger than the object it encompasses. To take advantage of the increased efficiency from using full occlusion records, a polygonal mesh enclosed by the object can also be generated and tested. These full occlusion polygonal meshes are often generated by the same methods that created the enclosing hulls. Figure 4b is a standard list, and Figure 4c is the list after an exterior polygonal hull for a nonpolygonal object has been inserted.

Transparent objects

Transparent objects can also be added to the light buffer scheme with minor changes in the algorithms to insert an object and access the list (Figure 4a). Transparent objects are inserted into the light buffer in a similar fashion to opaque objects. For polygonal objects, no faces can be culled because all their surfaces are visible from the light source. Also, because the objects cannot fully block the light, no full occlusion tests are made. Finally, all list records of these objects are flagged as transparent. Figure 4d shows how a transparent surface is inserted into a light buffer grid list.

To use transparent records, the procedure for processing the light buffer list during shadow testing must be modified. No transparent objects are tested before opaque objects, since if the intersection point is in shadow, testing the transparent objects is superfluous. Whenever a record of a transparent object is encountered, the data is copied to a temporary list. If no opaque object occludes the light, the list of transparent objects is intersected and used to calculate the absorption of light.

Object coherence

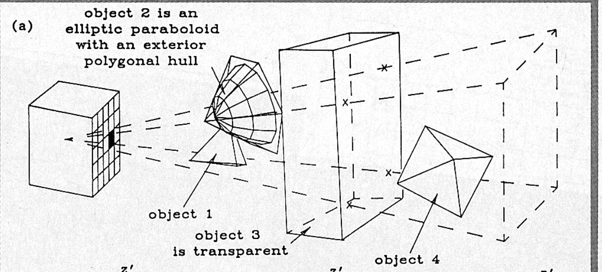
Object coherence can also be used to decrease shadow testing time. It is based on the observation that most object shadows in a scene have a horizontal width of more than one pixel. If an object blocks a light source from contributing to a pixel's intensity, then the same object is likely to block the same light for an adjacent pixel.

The algorithm requires saving and updating an object identity number for each light during processing. If a point is shadowed from a light by some object, the identity of the object casting the shadow is saved for that light. The next time the light is tested for occlusion, the associated shadowing object is first checked for intersection. If intersection occurs, testing ends, with the light source still occluded by the object. If the object is not hit, then the identity number is reset to the null flag and testing continues normally. When this algorithm was used with light buffers of 10 × 10-inch resolution, shadow testing times were reduced by up to 33 percent over standard methods.

An interesting effect of object coherence is that the efficiency increases as the resolution of the image increases. The reason is that, when an environment is rendered at a higher resolution, its objects cast shadows on a larger number of pixels.

Results and timing tests

Figures 5 through 11 show the environments that were modeled and rendered to compare the light buffer algorithm with the traditional shadow testing method. These environments differ greatly in complexity, item shapes, number of light sources, and specular properties. For each ray-traced image, we provided processing statistics for the creation of light buffers of various resolutions, and we compare them to the statistics of an enhanced ray-tracing algorithm.


(a)
object 2 is an elliptic paraboloid with an exterior polygonal hull
object 1
object 3 is transparent
object 4

Obj	Face	z′ Depth
1	2	1.28
1	3	1.28
4	1	4.48
4	2	4.48
4	7	4.48
4	8	4.48

Obj	Face	z′ Depth
1	2	1.28
1	3	1.28
-2	-1	1.33
4	1	4.48
4	2	4.48
4	7	4.48
4	8	4.48

Obj	Face	z′ Depth
1	2	1.28
1	3	1.28
-2	-1	1.33
3	4	-3.69
3	2	-4.10
4	1	4.48
4	2	4.48
4	7	4.48
4	8	4.48

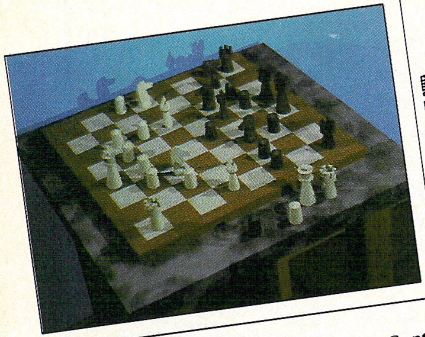
(b) → (c) → (d)

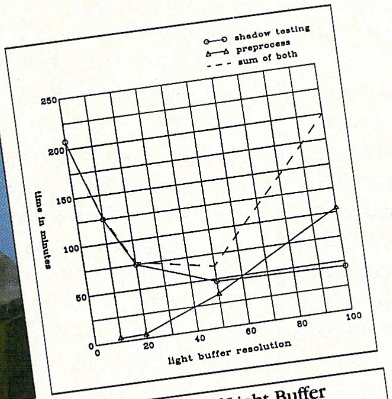
Figure 4. Insertion of nonpolygonal and transparent objects. In (a) we form a polygonal hull to enclose the nonpolygonal object. Into list (b) we (c) insert the hull and then (d) a transparent surface.

Intersection trees were generated using the visible-surface preprocess discussed by Weghorst et al.[4] Rectangular or spherical bounding volumes were placed around all objects, and hierarchical clustering was performed as applicable. The combination of these techniques alone reduced computation times for typical environments by a factor of approximately four or five when compared to standard ray-tracing algorithms.

The light buffer approach further reduced the computation times by a significant amount. The overall processing time of the ray-tracing algorithm was generally reduced by an additional factor of two to four. For each test, more detailed observations are made about the timing results. As the light buffer resolution increased, the number of polygons per square went down, fewer bounding volume records were generated, and more full occlusion records were created. Thus, the shadow tests needed per ray drops noticeably as the light buffer resolution rises, although the overall shadow testing time does not fall to the same levels because of the additional overhead involved.

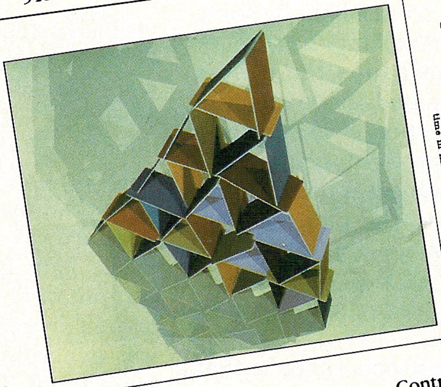
All computation times are listed in minutes of CPU time on a VAX 11/780. Test images were calculated at a resolution of 512 × 480, except where noted. No antialiasing was performed for the tests, nor was the object coherence algorithm performed. "Miscellaneous" reflects the time spent primarily on bookkeeping and file I/O. The average tree size is the mean number of intersection nodes generated for each ray from the eye. "Shadow tests

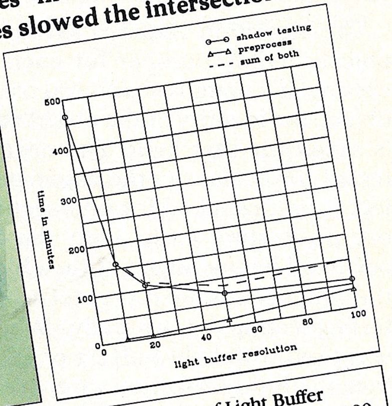




		Resolution of Light Buffer			
	Control	10×10	20×20	50×50	100×100
Shadow testing	216	125	75	45	41
Light buffer preprocess	—	4	6	35	109
Tree generation	32	32	32	32	32
Color calculation	30	30	30	30	30
Miscellaneous	16	16	16	16	16
Total time (in minutes)	294	207	159	158	228
Shadow tests per ray	99	20.9	8.9	2.9	1.5
Total number of list items	—	10613	14608	27857	50768
Number of list items per square	—	4.42	1.52	0.46	0.21
Nonzero grid squares percentage	—	11.8%	8.7%	6.9%	5.5%
Average maximum of list items	—	512	282	145	83

Figure 5. Timing test for "Chessboard." Specifications are 132 objects (4944 polygons), four light sources, and an average tree size of 1.02. Total time was reduced by a factor of 1.9 for the 50 × 50 light buffer. The large number of faces in a few light buffer grid squares slowed the intersection testing.

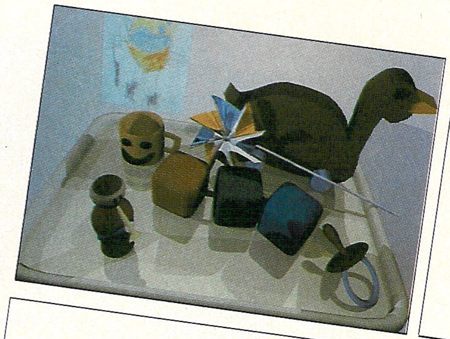


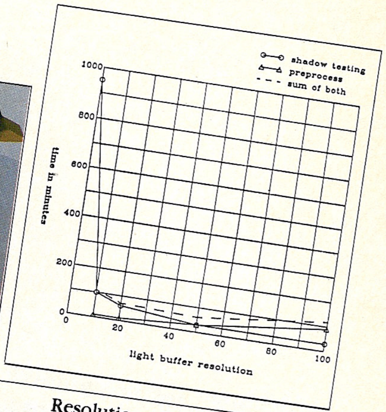


		Resolution of Light Buffer			
	Control	10×10	20×20	50×50	100×100
Shadow testing	466	160	112	69	58
Light buffer preprocess	—	3	5	17	40
Tree generation	62	62	62	62	62
Color calculation	58	58	58	58	58
Miscellaneous	18	18	18	18	18
Total time (in minutes)	604	301	255	224	236
Shadow tests per ray	392	7.0	2.5	0.6	0.3
Total number of list items	—	9443	11158	17134	30730
Number of list items per square	—	3.14	0.93	0.23	0.10
Nonzero grid squares percentage	—	5.9%	4.6%	3.6%	3.4%
Average maximum of list items	—	234	130	81	59

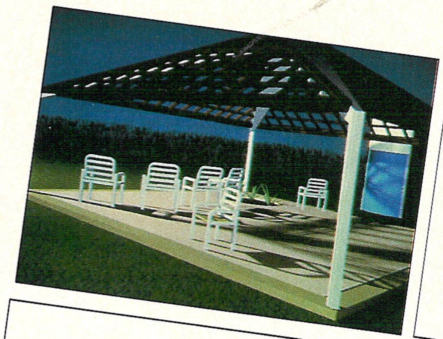
Figure 6. Timing test for "House of Cards." Specifications are 41 objects (3286 polygons), five light sources, and an average tree size of 1.38. Total time was reduced by a factor of 2.7 for the 50 × 50 light buffer. This image was a challenge for the algorithm, since each card consists of 82 polygons, and a large number of these tend to fall into single grid squares. Bounding volume records shortened testing time by a large factor.

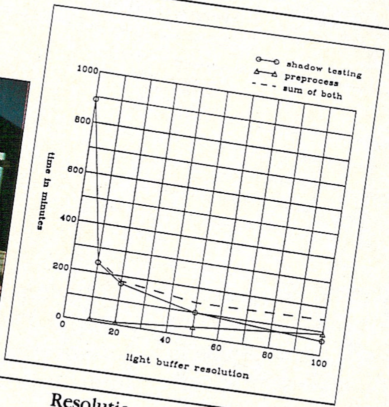
Figure 7. Timing test for "Sarah's Highchair." Specifications are 33 objects (6702 polygons, two cylinders, eight quadrics) three light sources, and an average tree size of 1.26. Total time was reduced by a factor of 3.8 for the 50 × 50 light buffer. This image has the largest number of polygonal faces and the smallest number of objects. Because of this object complexity, the greatest amount of time was spent testing each ray against a large number of faces. The light buffer approach was highly effective in reducing the testing time.





	Control	Resolution of Light Buffer 10×10	20×20	50×50	100×100
Shadow testing	950	99	60	34	30
Light buffer preprocess	—	5	10	26	89
Tree generation	215	215	215	215	215
Color calculation	25	25	25	25	25
Miscellaneous	16	16	16	16	16
Total time (in minutes)	1206	351	326	316	375
Shadow tests per ray	1270	17.2	8.3	2.6	1.2
Total number of list items	—	12897	15721	25492	48484
Number of list items per square	—	7.17	2.18	0.57	0.27
Nonzero grid squares percentage	—	8.2%	6.3%	5.5%	5.2%
Average maximum of list items	—	638	385	219	128





	Control	Resolution of Light Buffer 10×10	20×20	50×50	100×100
Shadow testing	888	237	172	100	61
Light buffer preprocess	—	5	14	39	93
Tree generation	52	52	52	52	52
Color calculation	45	45	45	45	45
Miscellaneous	14	14	14	14	14
Total time (in minutes)	999	353	297	250	265
Shadow tests per ray	182	19.0	11.6	4.9	1.3
Total number of list items	—	20526	29544	61485	130778
Number of list items per square	—	4.27	1.54	0.51	0.27
Nonzero grid squares percentage	—	7.0%	6.0%	5.1%	4.8%
Average maximum of list items	—	391	242	98	57

Figure 8. Timing test for "Gazebo at Night." Specifications are 80 objects (3222 polygons), eight light sources, and an average tree size of 1.03. Total time was reduced by a factor of 4.0 for the 50 × 50 light buffer. In this scene seven of the light sources are in a cluster, which creates an interesting shadow pattern. Although little full occlusion occurs in the scene the light buffers perform well in dividing the environment.

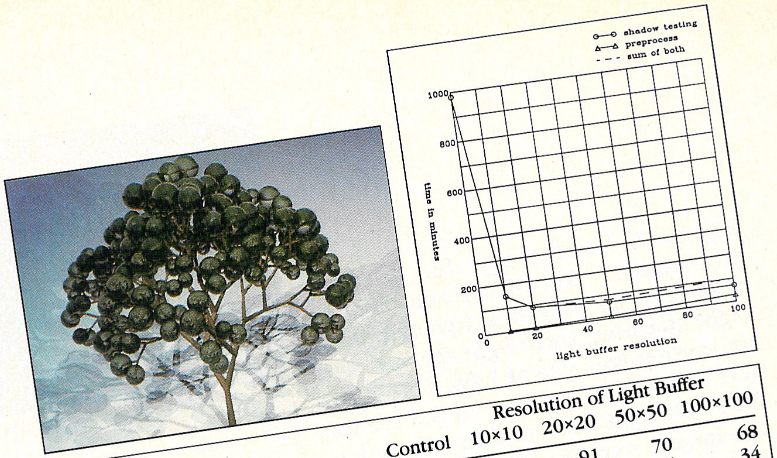
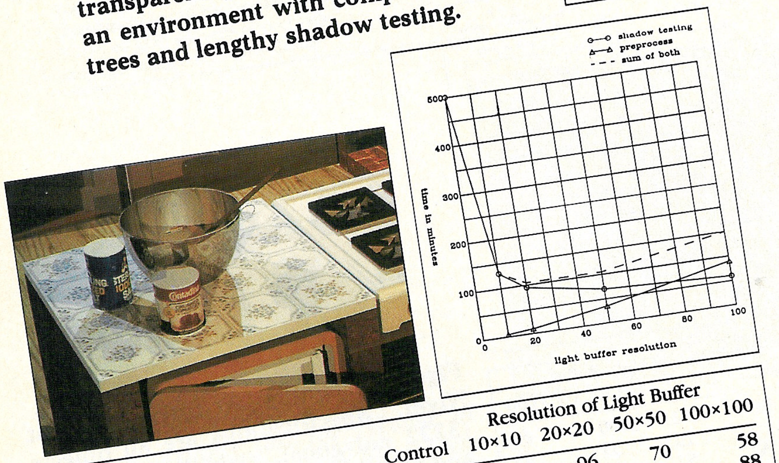


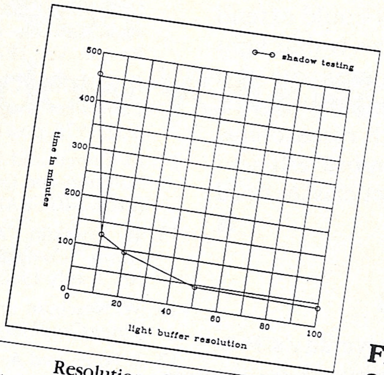
Figure 9. Timing test for "Glass-Leaved Tree." Specifications are 768 objects (six polygons, 256 transparent spheres, 511 quadrics), three light sources, and an average tree size of 2.02 (maximum tree depth is 5). Total time was reduced by a factor of 3.2 for the 50 × 50 light buffer. The tree was generated using Aono and Kunii's GMT model.[19] It has transparent spheres for leaves, creating an environment with complicated ray trees and lengthy shadow testing.

		Resolution of Light Buffer			
	Control	10×10	20×20	50×50	100×100
Shadow testing	975	149	91	70	68
Light buffer preprocess	—	1	2	6	34
Tree generation	281	281	281	281	281
Color calculation	39	39	39	39	39
Miscellaneous	16	16	16	16	16
Total time (in minutes)	1311	486	429	412	438
Shadow tests per ray	427	185	162	153	145
Total number of list items	—	3912	5886	14208	35595
Number of list items per square	—	2.17	0.82	0.32	0.20
Nonzero grid squares percentage	—	5.4%	4.4%	3.8%	3.4%
Average maximum of list items	—	113	68	33	20



		Resolution of Light Buffer			
	Control	10×10	20×20	50×50	100×100
Shadow testing	503	131	96	70	58
Light buffer preprocess	—	4	11	33	88
Tree generation	124	124	124	124	124
Color calculation	51	51	51	51	51
Miscellaneous	15	15	15	15	15
Total time (in minutes)	693	325	297	293	336
Shadow tests per ray	62	25.4	23.9	22.7	21.7
Total number of list items	—	8075	16548	58499	102063
Number of list items per square	—	2.69	1.38	0.67	0.34
Nonzero grid squares percentage	—	44.9%	42.6%	39.5%	38%
Average maximum of list items	—	79	56	43	37

Figure 10. Timing test for "Countertop" (see cover for image). Specifications are 224 objects (1298 polygons, four spheres, 76 cylinders, 35 quadrics), five light sources, and an average tree size of 1.69. Total time was reduced by a factor of 2.4 for the 50 × 50 light buffer. The large number of objects within a few grid squares increased the number of intersection tests.



		Resolution of Light Buffer			
	Control	10×10	20×20	50×50	100×100
Shadow testing	456	123	94	67	54
Light buffer preprocess	—	0	0	0	0
Tree generation	86	86	86	86	86
Color calculation	44	44	44	44	44
Miscellaneous	16	16	16	16	16
Total time (in minutes)	602	269	240	213	200
Shadow tests per ray	226	27.2	11.8	4.2	1.5
Total number of list items	—	8075	16548	50499	102063
Number of list items per square	—	2.69	1.38	0.67	0.34
Nonzero grid squares percentage	—	44.9%	42.6%	39.5%	38.0%
Average maximum of list items	—	79	56	43	37

Figure 11. Timing test for "Kitchen." Specifications are 224 objects (1298 polygons, four spheres, 76 cylinders, 35 quadrics), five light sources, and an average tree size of 1.25. Resolution of the image was at 436 × 479. This image was rendered with the same light buffers used for the image in Figure 10. Thus, we did not perform light buffer preprocessing again. These times are listed as zero (--). Total time was reduced by a factor of 3.2 for the 50 × 50 light buffer.

per light ray" is the average number of polygonal surfaces and nonpolygonal objects tested for intersection per shadow test light ray. "Total number of list items" is the total for all grid squares for all light buffers. "Nonzero grid squares percentage" is the ratio of grid squares with lists containing at least one object divided by the total number of grid squares. "Average maximum of list items" is the mean of the highest number of list records in a grid square for each light buffer.

Note that although statistics are presented for a lower resolution, the images presented in Figures 5 through 11 were rendered at a resolution of 1280 × 1024 with antialiasing performed using pixel subdivision techniques.

Conclusions and future outlook

It is well-known that standard ray-tracing techniques are still computationally too expensive for practical use. By far the largest portion of the computation time is devoted to shadow testing, particularly for complex lighting situations.

For every view ray we must test if the surface intersection is occluded from each light source. The method we have described uses a light buffer to accelerate the shadow testing portion of ray-tracing algorithms. This computationally expensive operation must test all polygons for all rays for each light source. The use of the light buffer substantially improves the performance of shadow testing by efficiently partitioning the environment. For examples tested, shadow testing times were reduced by a factor of four to 30.

The benefits are even greater for dynamic image sequences, since for static environments the light buffers do not have to be recalculated. The method is generally applicable, being suitable for polygonal and nonpolygonal environments with opaque or transparent surfaces. At present, the approach works best for point light sources, and resulting images do not contain soft shadows or penumbrae. Another limitation of the light buffer method is that storage requirements for high resolutions or large numbers of lights can become excessive.

We see the need for further research on a number of topics. The creation of light buffers for linear and area light sources[20,21] would enhance picture quality and make these computationally expensive algorithms much faster. One concept that could extend the use of the method to these sources is that light buffers can be concatenated. Performing some type of adaptive subdivision on grid squares that have a large number of objects is also worth researching. Finally, grid list restructuring and more efficient insertion algorithms for various object classes need exploration. ■

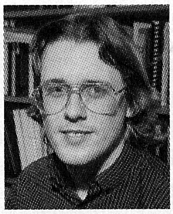
Acknowledgments

Many people from the Cornell University's Program of Computer Graphics contributed to the imaging system used as a basis for this research. Thanks are due to Ted Crane and Channing Verbeck for their technical support, and to Emil Ghinger for his photographic efforts. Our special thanks go to Phil Brock for the use of his gazebo environment and to Rikk Carey for his texturing software.

This research was performed as part of the Program of Computer Graphics at Cornell University with support from NSF grant DCR-8203979 and Digital Equipment Corporation.

References

1. A. Appel, "Some Techniques for Shading Machine Renderings of Solids," *Proc. SJCC*, Thompson Books, Washington, DC, 1968, pp.37-45.
2. R. Goldstein and R. Nagel, "3-D Visual Simulation," *Simulation*, Jan. 1971, pp.25-31.
3. T. Whitted, "An Improved Illumination Model for Shaded Display," *Comm. ACM*, Vol. 23, No. 6, June 1980, pp.343-349.
4. H. Weghorst, G. Hooper, and D. Greenberg, "Improved Computational Methods for Ray Tracing," *ACM Trans. Graphics*, Vol. 3, No. 1, Jan. 1984, pp.52-69.
5. S. Rubin and T. Whitted, "Three-Dimensional Representation for Fast Rendering of Complex Scenes," *Computer Graphics* (Proc. SIGGRAPH 80), Vol. 14, No. 3, July 1980, pp.110-116.
6. S. Roth, "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing*, Vol. 18, No. 2, Feb. 1982, pp.109-144.
7. A. Glassner, "Space Subdivision for Fast Ray Tracing," *IEEE CG&A*, Vol. 4, No. 10, Oct. 1984, pp.15-22.
8. M. Kaplan, "Space-Tracing: A Constant Time Ray-Tracer," course notes from tutorial "State of the Art in Image Synthesis," SIGGRAPH 85.
9. M. Dippe and J. Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," *Computer Graphics* (Proc. SIGGRAPH 84), Vol. 18, No. 3, July 1984, pp.149-158.
10. R. Hall and D. Greenberg, "A Testbed for Realistic Image Synthesis," *IEEE Trans. Computer Graphics and Applications*, Vol. 3, No. 8, Nov. 1983, pp.10-20.
11. E. Haines, *The Light Buffer: A Ray Tracer Shadow Testing Accelerator*, master's thesis, Cornell Univ., 1986.
12. W. Bouknight and K. Kelley, "An Algorithm for Producing Half-Tone Computer Graphics Presentations with Shadows and Movable Light Sources," *Proc. SJCC*, AFIPS Press, Reston, Va., 1970, pp.1-10.
13. L. Williams, "Casting Curved Shadows on Curved Surfaces," *Computer Graphics* (Proc. SIGGRAPH 78), Vol. 12, No. 3, Aug. 1978, pp.270-274.
14. M. Cohen and D. Greenberg, "The Hemi-Cube: A Radiosity Solution for Complex Environments," *Computer Graphics* (Proc. SIGGRAPH 85), Vol. 19, No. 3, July 1985, pp.31-40.
15. W. Franklin, "A Linear Time Exact Hidden Surface Algorithm," *Computer Graphics* (Proc. SIGGRAPH 80), Vol. 14, No. 3, July 1980, pp.117-123.
16. W. Newman and R. Sproull, *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, New York, 1979.
17. J. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass., 1982.
18. J. Warnock, *A Hidden-Surface Algorithm for Computer Generated Half-Tone Pictures*, TR 4-15, Univ. of Utah, CS Dept., 1969.
19. M. Aono and T. Kunii, "Botanical Tree Image Generation," *IEEE CG&A*, Vol. 4, No. 5, May 1984, pp.10-34.
20. C. Verbeck, *A Comprehensive Light Source Description for Computer Graphics*, master's thesis, Cornell Univ., 1984.
21. R. Cook, T. Porter, and L. Carpenter, "Distributed Ray Tracing," *Computer Graphics* (Proc. SIGGRAPH 84), Vol. 18 No. 3, July 1984, pp.137-145.



Eric A. Haines is a senior software engineer at 3D/Eye in Ithaca, New York. His interests and responsibilities include creating a ray-tracing system for image synthesis. Previously, he was a software engineer at RCA Astro-Electronics.

Haines received a BS in computer science from Rensselaer Polytechnic Institute in 1980 and an MS in computer graphics from Cornell University in 1986. He is a member of ACM.

Haines' address is 3D/Eye, 410 E. Upland Rd., Ithaca, NY 14850.

Donald P. Greenberg has been researching and teaching in computer graphics since 1966. At present he is the Jacob Gould Shurman University Professor of Computer Graphics and the director of the Program of Computer Graphics at Cornell, where he is involved in graphics algorithms, geometric models, color science, and realistic images. He also teaches courses as part of the CS Department at Cornell. Previously he was a consulting engineer involved in the design of such projects as the St. Louis Arch and Madison Square Garden.

Greenberg has published extensively in computer graphics, including an article for *Scientific American*. He is on the editorial boards of *Computers and Graphics*, *Computer-Aided Design*, and *ACM Transactions on Graphics*. He is also a member of ACM SIGGRAPH and IEEE.

Greenberg's address is Program of Computer Graphics, Cornell Univ., 120 Rand Hall, Ithaca, NY 14853.