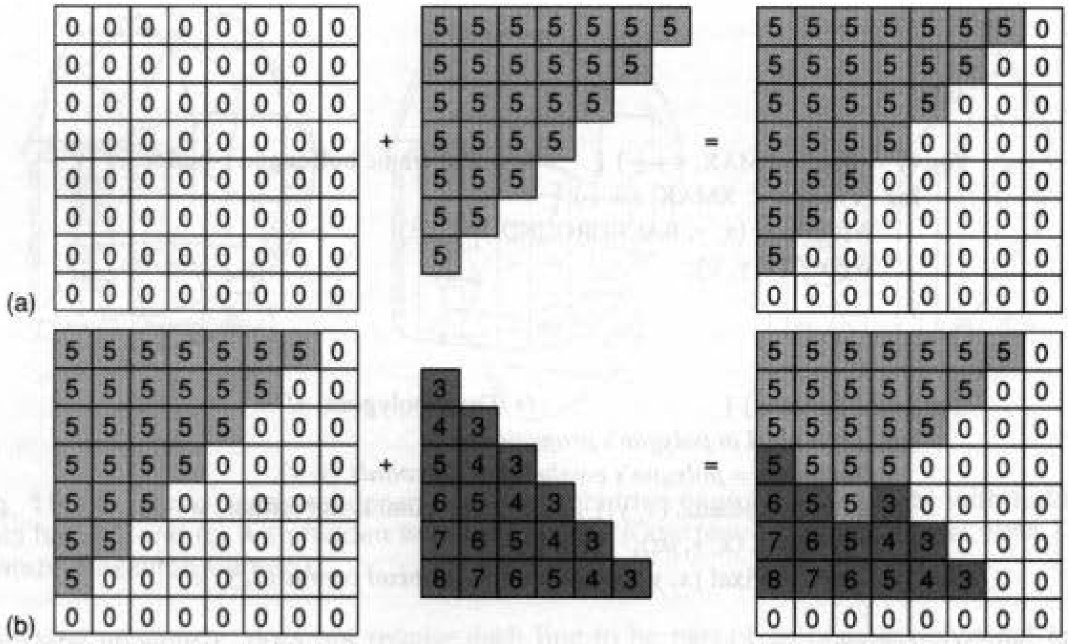


(a)

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

+

5	5	5	5	5	5	5
5	5	5	5	5	5	
5	5	5	5	5		
5	5	5	5			
5	5	5				
5	5					
5						

=

5	5	5	5	5	5	5	0
5	5	5	5	5	5	0	0
5	5	5	5	5	0	0	0
5	5	5	5	0	0	0	0
5	5	5	0	0	0	0	0
5	5	0	0	0	0	0	0
5	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(b)

5	5	5	5	5	5	5	0
5	5	5	5	5	5	0	0
5	5	5	5	5	0	0	0
5	5	5	5	0	0	0	0
5	5	5	0	0	0	0	0
5	5	0	0	0	0	0	0
5	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

+

3					
4	3				
5	4	3			
6	5	4	3		
7	6	5	4	3	
8	7	6	5	4	3

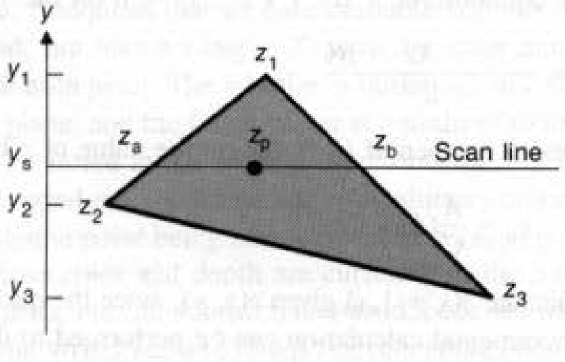
=

5	5	5	5	5	5	5	0
5	5	5	5	5	5	0	0
5	5	5	5	5	0	0	0
5	5	5	5	0	0	0	0
6	5	5	3	0	0	0	0
7	6	5	4	3	0	0	0
8	7	6	5	4	3	0	0
0	0	0	0	0	0	0	0

Fig. 15.22 The *z*-buffer. A pixel's shade is shown by its color, its *z* value is shown as a number. (a) Adding a polygon of constant *z* to the empty *z*-buffer. (b) Adding another polygon that intersects the first.

the surface has not been determined or if the polygon is not planar (see Section 11.1.3), $z(x, y)$ can be determined by interpolating the z coordinates of the polygon's vertices along pairs of edges, and then across each scan line, as shown in Fig. 15.23. Incremental calculations can be used here as well. Note that the color at a pixel does not need to be computed if the conditional determining the pixel's visibility is not satisfied. Therefore, if the shading computation is time consuming, additional efficiency can be gained by performing a rough front-to-back depth sort of the objects to display the closest objects first.

The z-buffer algorithm does not require that objects be polygons. Indeed, one of its most powerful attractions is that it can be used to render any object if a shade and a z-value



$$z_a = z_1 - (z_1 - z_2)\,\frac{y_1 - y_s}{y_1 - y_2}$$

$$z_b = z_1 - (z_1 - z_3)\,\frac{y_1 - y_s}{y_1 - y_3}$$

$$z_p = z_b - (z_b - z_a)\,\frac{x_b - x_p}{x_b - x_a}$$

Fig. 15.23 Interpolation of z values along polygon edges and scan lines. z_a is interpolated between z_1 and z_2; z_b between z_1 and z_3; z_p between z_a and z_b.

can be determined for each point in its projection; no explicit intersection algorithms need to be written.

The *z*-buffer algorithm performs radix sorts in *x* and *y*, requiring no comparisons, and its *z* sort takes only one comparison per pixel for each polygon containing that pixel. The time taken by the visible-surface calculations tends to be independent of the number of polygons in the objects because, on the average, the number of pixels covered by each polygon decreases as the number of polygons in the view volume increases. Therefore, the average size of each set of pairs being searched tends to remain fixed. Of course, it is also necessary to take into account the scan-conversion overhead imposed by the additional polygons.

Although the *z*-buffer algorithm requires a large amount of space for the *z*-buffer, it is easy to implement. If memory is at a premium, the image can be scan-converted in strips, so that only enough *z*-buffer for the strip being processed is required, at the expense of performing multiple passes through the objects. Because of the *z*-buffer's simplicity and the lack of additional data structures, decreasing memory costs have inspired a number of hardware and firmware implementations of the *z*-buffer, examples of which are discussed in Chapter 18. Because the *z*-buffer algorithm operates in image precision, however, it is subject to aliasing. The A-buffer algorithm [CARP84], described in Section 15.7, addresses this problem by using a discrete approximation to unweighted area sampling.

The *z*-buffer is often implemented with 16- through 32-bit integer values in hardware, but software (and some hardware) implementations may use floating-point values. Although a 16-bit *z*-buffer offers an adequate range for many CAD/CAM applications, 16 bits do not have enough precision to represent environments in which objects defined with millimeter detail are positioned a kilometer apart. To make matters worse, if a perspective projection is used, the compression of distant *z* values resulting from the perspective divide has a serious effect on the depth ordering and intersections of distant objects. Two points that would transform to different integer *z* values if close to the view plane may transform to the same *z* value if they are farther back (see Exercise 15.13 and [HUGH89]).

The *z*-buffer's finite precision is responsible for another aliasing problem. Scan-conversion algorithms typically render two different sets of pixels when drawing the common part of two collinear edges that start at different endpoints. Some of those pixels shared by the rendered edges may also be assigned slightly different *z* values because of numerical inaccuracies in performing the *z* interpolation. This effect is most noticeable at the shared edges of a polyhedron's faces. Some of the visible pixels along an edge may be part of one polygon, while the rest come from the polygon's neighbor. The problem can be fixed by inserting extra vertices to ensure that vertices occur at the same points along the common part of two collinear edges.

Even after the image has been rendered, the *z*-buffer can still be used to advantage. Since it is the only data structure used by the visible-surface algorithm proper, it can be saved along with the image and used later to merge in other objects whose *z* can be computed. The algorithm can also be coded so as to leave the *z*-buffer contents unmodified when rendering selected objects. If the *z*-buffer is masked off this way, then a single object can be written into a separate set of overlay planes with hidden surfaces properly removed (if the object is a single-valued function of *x* and *y*) and then erased without affecting the contents of the *z*-buffer. Thus, a simple object, such as a ruled grid, can be moved about the image in x, y, and z, to serve as a "3D cursor" that obscures and is obscured by the objects in the environment. Cutaway views can be created by making the z-buffer and frame-buffer writes contingent on whether the z value is behind a cutting plane. If the objects being displayed have a single z value for each (x, y), then the z-buffer contents can also be used to compute area and volume. Exercise 15.25 explains how to use the z-buffer for picking.

Rossignac and Requicha [ROSS86] discuss how to adapt the z-buffer algorithm to handle objects defined by CSG. Each pixel in a surface's projection is written only if it is both closer in z and on a CSG object constructed from the surface. Instead of storing only the point with closest z at each pixel, Atherton suggests saving a list of all points, ordered by z and accompanied by each surface's identity, to form an *object buffer* [ATHE81]. A postprocessing stage determines how the image is displayed. A variety of effects, such as transparency, clipping, and Boolean set operations, can be achieved by processing each pixel's list, without any need to re–scan convert the objects.

15.5 LIST-PRIORITY ALGORITHMS

List-priority algorithms determine a visibility ordering for objects ensuring that a correct picture results if the objects are rendered in that order. For example, if no object overlaps another in z, then we need only to sort the objects by increasing z, and to render them in that order. Farther objects are obscured by closer ones as pixels from the closer polygons overwrite those of the more distant ones. If objects overlap in z, we may still be able to determine a correct order, as in Fig. 15.24(a). If objects cyclically overlap each other, as Fig. 15.24(b) and (c), or penetrate each other, then there is no correct order. In these cases, it will be necessary to split one or more objects to make a linear order possible.

List-priority algorithms are hybrids that combine both object-precision and image-precision operations. Depth comparisons and object splitting are done with object precision. Only scan conversion, which relies on the ability of the graphics device to overwrite the pixels of previously drawn objects, is done with image precision. Because the list of sorted objects is created with object precision, however, it can be redisplayed correctly at any resolution. As we shall see, list-priority algorithms differ in how they determine the sorted order, as well as in which objects get split, and when the splitting occurs. The sort need not be on z, some objects may be split that neither cyclically overlap nor penetrate others, and the splitting may even be done independent of the viewer's position.

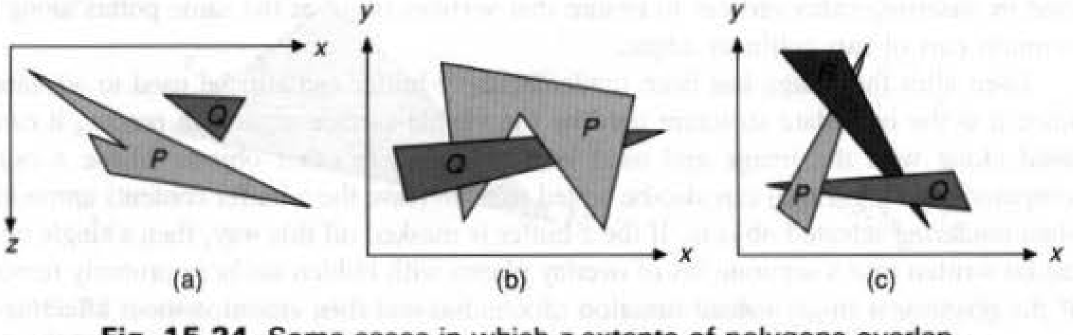


Fig. 15.24 Some cases in which z extents of polygons overlap.

15.5.1 The Depth-Sort Algorithm

The basic idea of the *depth-sort algorithm*, developed by Newell, Newell, and Sancha [NEWE72], is to paint the polygons into the frame buffer in order of decreasing distance from the viewpoint. Three conceptual steps are performed:

1. Sort all polygons according to the smallest (farthest) z coordinate of each
2. Resolve any ambiguities this may cause when the polygons' z extents overlap, splitting polygons if necessary
3. Scan convert each polygon in ascending order of smallest z coordinate (i.e., back to front).

Consider the use of explicit priority, such as that associated with views in SPHIGS. The explicit priority takes the place of the minimum z value, and there can be no depth ambiguities because each priority is thought of as corresponding to a different plane of constant z. This simplified version of the depth-sort algorithm is often known as the *painter's algorithm*, in reference to how a painter might paint closer objects over more distant ones. Environments whose objects each exist in a plane of constant z, such as those of VLSI layout, cartography, and window management, are said to be $2\frac{1}{2}$D and can be correctly handled with the painter's algorithm. The painter's algorithm may be applied to a scene in which each polygon is not embedded in a plane of constant z by sorting the polygons by their minimum z coordinate or by the z coordinate of their centroid, ignoring step 2. Although scenes can be constructed for which this approach works, it does not in general produce a correct ordering.

Figure 15.24 shows some of the types of ambiguities that must be resolved as part of step 2. How is this done? Let the polygon currently at the far end of the sorted list of polygons be called P. Before this polygon is scan-converted into the frame buffer, it must be tested against each polygon Q whose z extent overlaps the z extent of P, to prove that P cannot obscure Q and that P can therefore be written before Q. Up to five tests are performed, in order of increasing complexity. As soon as one succeeds, P has been shown not to obscure Q and the next polygon Q overlapping P in z is tested. If all such polygons pass, then P is scan-converted and the next polygon on the list becomes the new P. The five tests are

1. Do the polygons' x extents not overlap?
2. Do the polygons' y extents not overlap?
3. Is P entirely on the opposite side of Q's plane from the viewpoint? (This is not the case in Fig. 15.24(a), but is true for Fig. 15.25.)
4. Is Q entirely on the same side of P's plane as the viewpoint? (This is not the case in Fig. 15.24(a), but is true for Fig. 15.26.)
5. Do the projections of the polygons onto the (x, y) plane not overlap? (This can be determined by comparing the edges of one polygon to the edges of the other.)

Exercise 15.6 suggests a way to implement tests 3 and 4.

If all five tests fail, we assume for the moment that P actually obscures Q, and therefore test whether Q can be scan-converted before P. Tests 1, 2, and 5 do not need to be repeated,

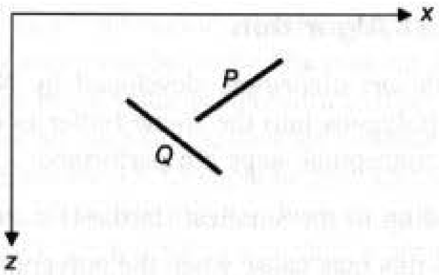


Fig. 15.25 Test 3 is true.

but new versions of tests 3 and 4 are used, with the polygons reversed:

3′. Is Q entirely on the opposite side of P's plane from the viewpoint?

4′. Is P entirely on the same side of Q's plane as the viewpoint?

In the case of Fig. 15.24(a), test 3′ succeeds. Therefore, we move Q to the end of the list and it becomes the new P. In the case of Fig 15.24(b), however, the tests are still inconclusive; in fact, there is no order in which P and Q can be scan-converted correctly. Instead, either P or Q must be split by the plane of the other (see Section 3.14 on polygon clipping, treating the clip edge as a clip plane). The original unsplit polygon is discarded, its pieces are inserted in the list in proper z order, and the algorithm proceeds as before.

Figure 15.24(c) shows a more subtle case. It is possible for P, Q, and R to be oriented such that each polygon can always be moved to the end of the list to place it in the correct order relative to one, but not both, of the other polygons. This would result in an infinite loop. To avoid looping, we must modify our approach by marking each polygon that is moved to the end of the list. Then, whenever the first five tests fail and the current polygon Q is marked, we do not try tests 3′ and 4′. Instead, we split either P or Q (as if tests 3′ and 4′ had both failed) and reinsert the pieces.

Can two polygons fail all the tests even when they are already ordered correctly? Consider P and Q in Fig. 15.27(a). Only the z coordinate of each vertex is shown. With P and Q in their current position, both the simple painter's algorithm and the full depth-sort algorithm scan convert P first. Now, rotate Q clockwise in its plane until it begins to obscure P, but do not allow P and Q themselves to intersect, as shown in Fig. 15.27(b). (You can do this nicely using your hands as P and Q, with your palms facing you.) P and Q

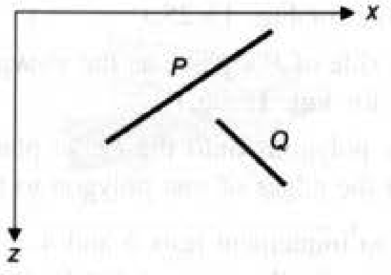


Fig. 15.26 Test 3 is false, but test 4 is true.

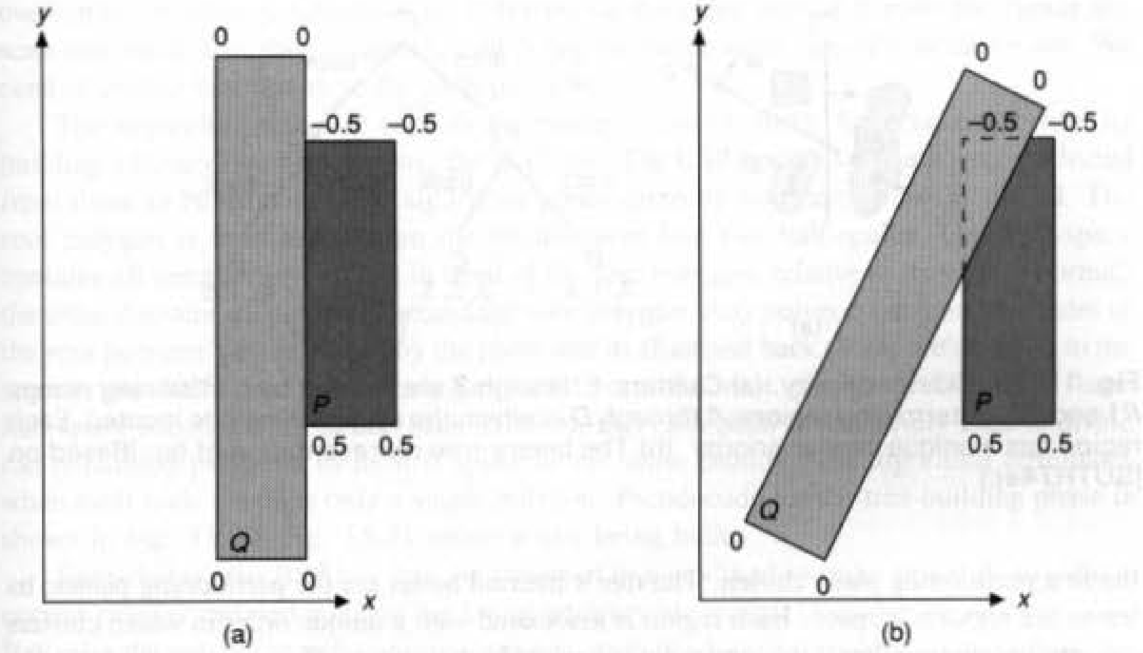


Fig. 15.27 Correctly ordered polygons may be split by the depth-sort algorithm. Polygon vertices are labeled with their z values. (a) Polygons P and Q are scan-converted without splitting. (b) Polygons P and Q fail all five tests even though they are correctly ordered.

have overlapping z extents, so they must be compared. Note that tests 1 and 2 (x and y extent) fail, tests 3 and 4 fail because neither is wholly in one half-space of the other, and test 5 fails because the projections overlap. Since tests 3′ and 4′ also fail, a polygon will be split, even though P can be scan-converted before Q. Although the simple painter's algorithm would correctly draw P first because P has the smallest minimum z coordinate, try the example again with $z = -0.5$ at P's bottom and $z = 0.5$ at P's top.

15.5.2 Binary Space-Partitioning Trees

The binary space-partitioning (BSP) tree algorithm, developed by Fuchs, Kedem, and Naylor [FUCH80; FUCH83], is an extremely efficient method for calculating the visibility relationships among a static group of 3D polygons as seen from an arbitrary viewpoint. It trades off an initial time- and space-intensive preprocessing step against a linear display algorithm that is executed whenever a new viewing specification is desired. Thus, the algorithm is well suited for applications in which the viewpoint changes, but the objects do not.

The BSP tree algorithm is based on the work of Schumacker [SCHU69], who noted that environments can be viewed as being composed of *clusters* (collections of faces), as shown in Fig. 15.28(a). If a plane can be found that wholly separates one set of clusters from another, then clusters that are on the same side of the plane as the eyepoint can obscure, but cannot be obscured by, clusters on the other side. Each of these sets of clusters can be recursively subdivided if suitable separating planes can be found. As shown in Fig. 15.28(b), this partitioning of the environment can be represented by a binary tree rooted at

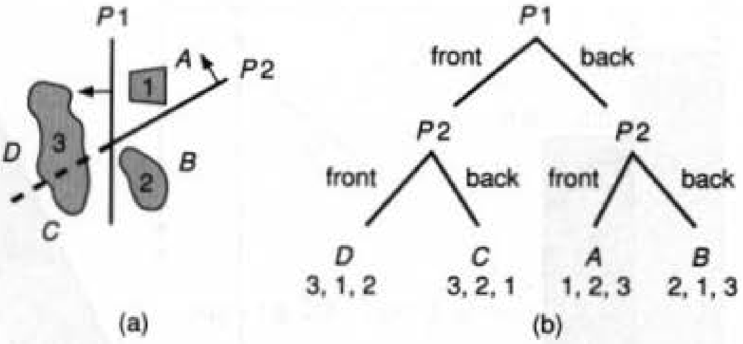


Fig. 15.28 Cluster priority. (a) Clusters 1 through 3 are divided by partitioning planes *P*1 and *P*2, determining regions *A* through *D* in which the eyepoint may be located. Each region has a unique cluster priority. (b) The binary-tree representation of (a). (Based on [SUTH74a].)

the first partitioning plane chosen. The tree's internal nodes are the partitioning planes; its leaves are regions in space. Each region is associated with a unique order in which clusters can obscure one another if the viewpoint is located in that region. Determining the region in which the eyepoint lies involves descending the tree from the root and choosing the left or right child of an internal node by comparing the viewpoint with the plane at that node.

Schumacker selects the faces in a cluster so that a priority ordering can be assigned to each face independent of the viewpoint, as shown in Fig. 15.29. After back-face culling has been performed relative to the viewpoint, a face with a lower priority number obscures a face with a higher number wherever the faces' projections intersect. For any pixel, the correct face to display is the highest-priority (lowest-numbered) face in the highest-priority cluster whose projection covers the pixel. Schumacker used special hardware to determine the frontmost face at each pixel. Alternatively, clusters can be displayed in order of increasing cluster priority (based on the viewpoint), with each cluster's faces displayed in order of their increasing face priority. Rather than take this two-part approach to computing an order in which faces should be scan-converted, the BSP tree algorithm uses a generalization of Schumacker's approach to calculating cluster priority. It is based on the observation that a polygon will be scan-converted correctly (i.e., will not incorrectly overlap or be incorrectly

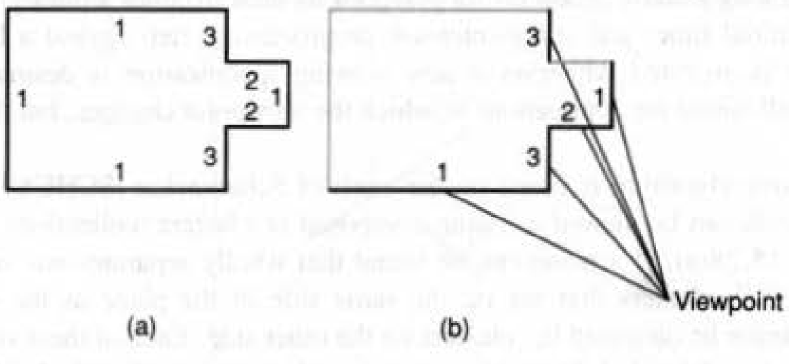


Fig. 15.29 Face priority. (a) Faces in a cluster and their priorities. A lower number indicates a higher priority. (b) Priorities of visible faces. (Based on [SUTH74a].)

overlapped by other polygons) if all polygons on the other side of it from the viewer are scan-converted first, then it, and then all polygons on the same side of it as the viewer. We need to ensure that this is so for each polygon.

The algorithm makes it easy to determine a correct order for scan conversion by building a binary tree of polygons, the *BSP tree*. The BSP tree's root is a polygon selected from those to be displayed; the algorithm works correctly no matter which is picked. The root polygon is used to partition the environment into two half-spaces. One half-space contains all remaining polygons in front of the root polygon, relative to its surface normal; the other contains all polygons behind the root polygon. Any polygon lying on both sides of the root polygon's plane is split by the plane and its front and back pieces are assigned to the appropriate half-space. One polygon each from the root polygon's front and back half-space become its front and back children, and each child is recursively used to divide the remaining polygons in its half-space in the same fashion. The algorithm terminates when each node contains only a single polygon. Pseudocode for the tree-building phase is shown in Fig. 15.30; Fig. 15.31 shows a tree being built.

Remarkably, the BSP tree may be traversed in a modified in-order tree walk to yield a correct priority-ordered polygon list for an arbitrary viewpoint. Consider the root polygon. It divides the remaining polygons into two sets, each of which lies entirely on one side of the root's plane. Thus, the algorithm needs to only guarantee that the sets are displayed in the correct relative order to ensure both that one set's polygons do not interfere with the other's and that the root polygon is displayed properly and in the correct order relative to the others. If the viewer is in the root polygon's front half-space, then the algorithm must first display all polygons in the root's rear half-space (those that could be obscured by the root), then the root, and finally all polygons in its front half-space (those that could obscure the root). Alternatively, if the viewer is in the root polygon's rear half-space, then the algorithm must first display all polygons in the root's front half-space, then the root, and finally all polygons in its rear half-space. If the polygon is seen on edge, either display order suffices. Back-face culling may be accomplished by not displaying a polygon if the eye is in its rear half-space. Each of the root's children is recursively processed by this algorithm. Pseudocode for displaying a BSP tree is shown in Fig. 15.32; Fig. 15.33 shows how the tree of Fig.15.31 (c) is traversed for two different projections.

Each polygon's plane equation can be transformed as it is considered, and the polygon's vertices can be transformed by the displayPolygon routine. The BSP tree can also assist in 3D clipping. Any polygon whose plane does not intersect the view volume has one subtree lying entirely outside of the view volume that does not need to be considered further.

Which polygon is selected to serve as the root of each subtree can have a significant effect on the algorithm's performance. Ideally, the polygon selected should cause the fewest splits among all its descendants. A heuristic that is easier to satisfy is to select the polygon that splits the fewest of its children. Experience shows that testing just a few (five or six) polygons and picking the best provides a good approximation to the best case [FUCH83].

Like the depth-sort algorithm, the BSP tree algorithm performs intersection and sorting entirely at object precision, and relies on the image-precision overwrite capabilities of a raster device. Unlike depth sort, it performs all polygon splitting during a preprocessing step that must be repeated only when the environment changes. Note that more

```
typedef struct {
    polygon root;
    BSP_tree *backChild, *frontChild;
} BSP_tree;

BSP_tree *BSP_makeTree (polygon *polyList)
{
    polygon root;
    polygon *backList, *frontList;
    polygon p, backPart, frontPart;        /* We assume each polygon is convex. */

    if (polyList == NULL)
        return NULL;
    else {
        root = BSP_selectAndRemovePoly (&polyList);
        backList = NULL;
        frontList = NULL;
        for (each remaining polygon p in polyList) {
            if (polygon p in front of root)
                BSP_addToList (p, &frontList);
            else if (polygon p in back of root)
                BSP_addToList (p, &backList);
            else {                  /* Polygon p must be split. */
                BSP_splitPoly (p, root, &frontPart, &backPart);
                BSP_addToList (frontPart, &frontList);
                BSP_addToList (backPart, &backList);
            }
        }
        return BSP_combineTree (BSP_makeTree (frontList),
                        root,
                        BSP_makeTree (backList));
    }
}   /* BSP_makeTree */
```

Fig. 15.30 Pseudocode for building a BSP tree.

polygon splitting may occur than in the depth-sort algorithm.

List-priority algorithms allow the use of hardware polygon scan converters that are typically much faster than are those that check the z at each pixel. The depth-sort and BSP tree algorithms display polygons in a back-to-front order, possibly obscuring more distant ones later. Thus, like the z-buffer algorithm, shading calculations may be computed more than once for each pixel. Alternatively, polygons can instead be displayed in a front-to-back order, and each pixel in a polygon can be written only if it has not yet been.

If a list-priority algorithm is used for hidden-line removal, special attention must be paid to the new edges introduced by the subdivision process. If these edges are

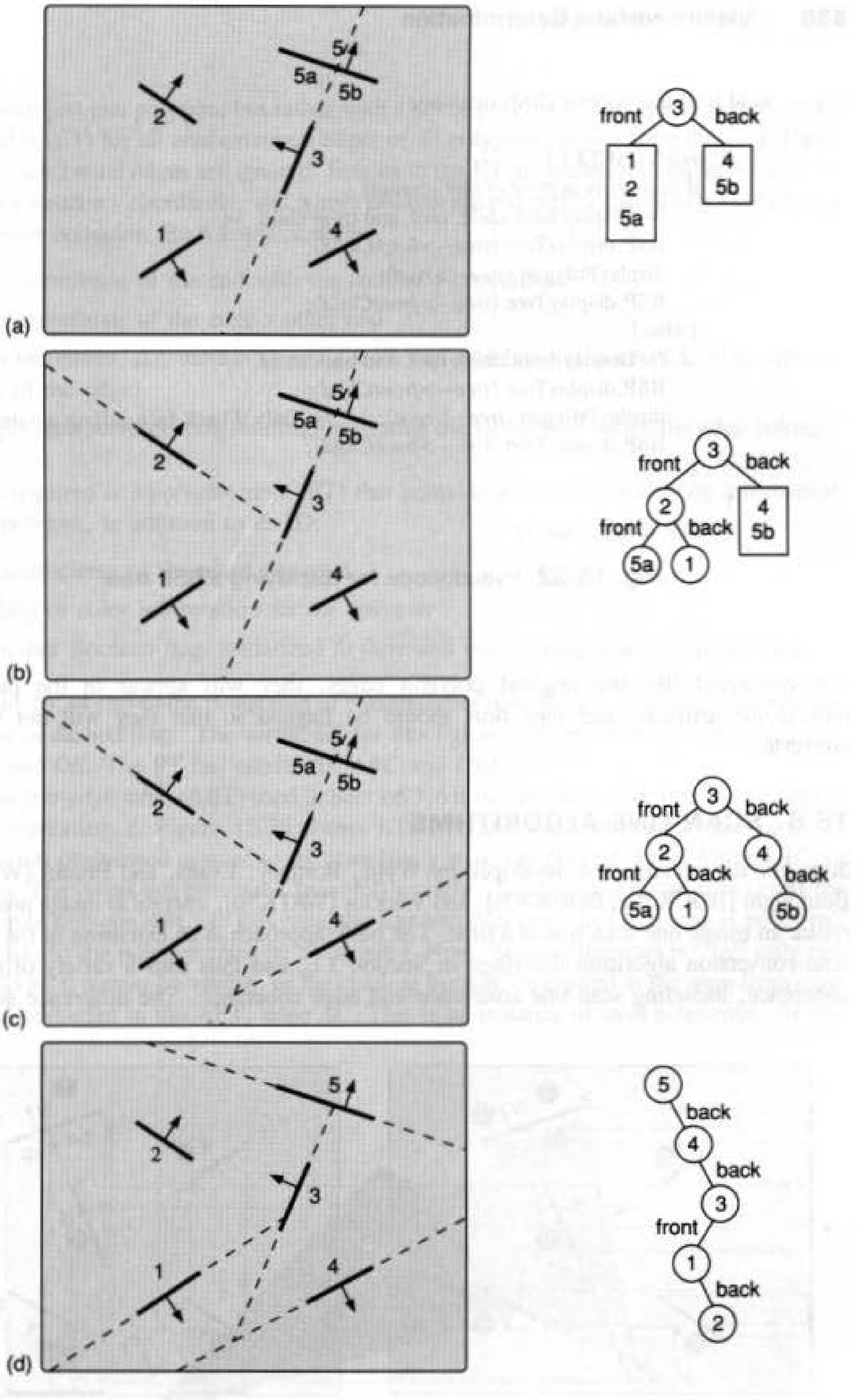


Fig. 15.31 BSP trees. (a) Top view of scene with BSP tree before recursion with polygon 3 as root. (b) After building left subtree. (c) Complete tree. (d) Alternate tree with polygon 5 as root. (Based on [FUCH83].)

```
void BSP_displayTree (BSP_tree *tree)
{
    if (tree != NULL) {
        if (viewer is in front of tree->root) {
            /* Display back child, root, and front child. */
            BSP_displayTree (tree->backChild);
            displayPolygon (tree->root);
            BSP_displayTree (tree->frontChild);
        } else {
            /* Display front child, root, and back child. */
            BSP_displayTree (tree->frontChild);
            displayPolygon (tree->root);        /* Only if back-face culling not desired */
            BSP_displayTree (tree->backChild);
        }
    }
} /* BSP_displayTree */
```

Fig. 15.32 Pseudocode for displaying a BSP tree.

scan-converted like the original polygon edges, they will appear in the picture as unwelcome artifacts, and they thus should be flagged so that they will not be scan-converted.

15.6 SCAN-LINE ALGORITHMS

Scan-line algorithms, first developed by Wylie, Romney, Evans, and Erdahl [WYLI67], Bouknight [BOUK70a; BOUK70b], and Watkins [WATK70], operate at image precision to create an image one scan line at a time. The basic approach is an extension of the polygon scan-conversion algorithm described in Section 3.6, and thus uses a variety of forms of coherence, including scan-line coherence and edge coherence. The difference is that we

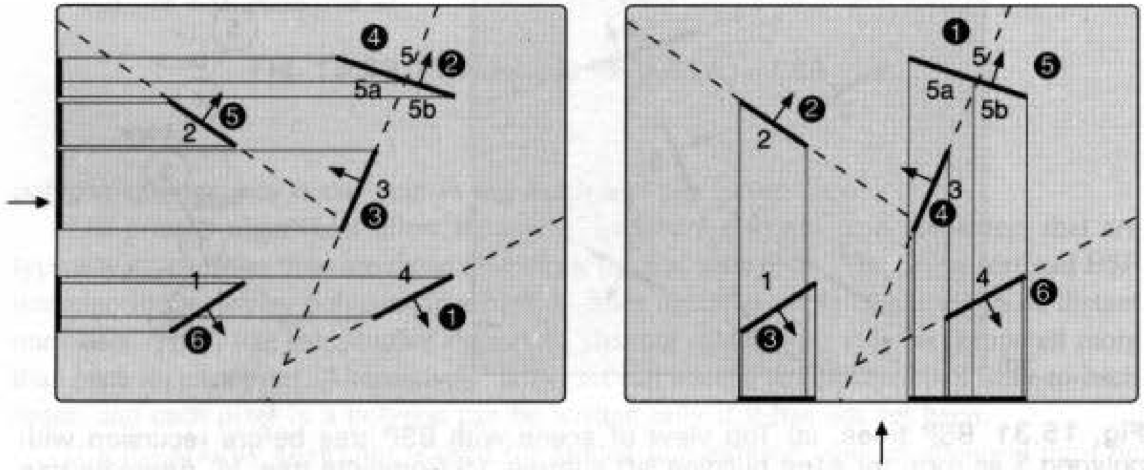


Fig. 15.33 Two traversals of the BSP tree corresponding to two different projections. Projectors are shown as thin lines. White numbers indicate drawing order.

deal not with just one polygon, but rather with a set of polygons. The first step is to create an *edge table* (ET) for all nonhorizontal edges of all polygons projected on the view plane. As before, horizontal edges are ignored. Entries in the ET are sorted into buckets based on each edge's smaller y coordinate, and within buckets are ordered by increasing x coordinate of their lower endpoint. Each entry contains

1. The x coordinate of the end with the smaller y coordinate
2. The y coordinate of the edge's other end
3. The x increment, Δx, used in stepping from one scan line to the next (Δx is the inverse slope of the edge)
4. The polygon identification number, indicating the polygon to which the edge belongs.

Also required is a *polygon table* (PT) that contains at least the following information for each polygon, in addition to its ID:

1. The coefficients of the plane equation
2. Shading or color information for the polygon
3. An in–out Boolean flag, initialized to *false* and used during scan-line processing.

Figure 15.34 shows the projection of two triangles onto the (x, y) plane; hidden edges are shown as dashed lines. The sorted ET for this figure contains entries for AB, AC, FD, FE, CB, and DE. The PT has entries for ABC and DEF.

The *active-edge table* (AET) used in Section 3.6 is needed here also. It is always kept in order of increasing x. Figure 15.35 shows ET, PT, and AET entries. By the time the algorithm has progressed upward to the scan line $y = \alpha$, the AET contains AB and AC, in that order. The edges are processed from left to right. To process AB, we first invert the in–out flag of polygon ABC. In this case, the flag becomes *true*; thus, the scan is now "in" the polygon, so the polygon must be considered. Now, because the scan is "in" only one polygon (ABC), it must be visible, so the shading for ABC is applied to the *span* from edge AB to the next edge in the AET, edge AC. This is an instance of span coherence. At this

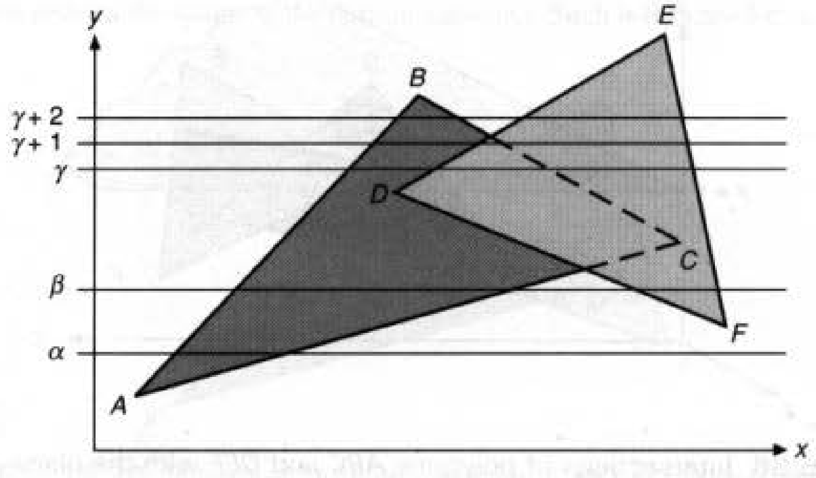
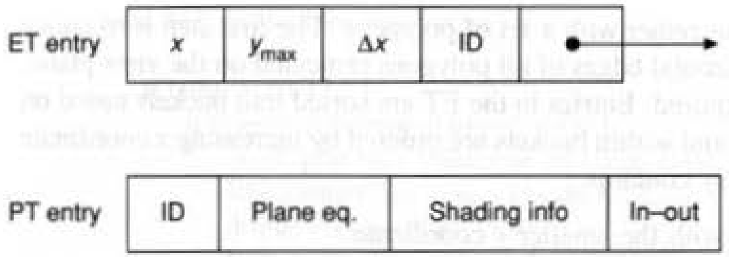


Fig. 15.34 Two polygons being processed by a scan-line algorithm.



ET entry	x	y_{max}	Δx	ID	●⟶
PT entry	ID	Plane eq.	Shading info	In–out	

AET contents

Scan line	Entries
α	*AB AC*
β	*AB AC FD FE*
$\gamma, \gamma + 1$	*AB DE CB FE*
$\gamma + 2$	*AB CB DE FE*

Fig. 15.35 ET, PT, AET for the scan-line algorithm.

edge the flag for *ABC* is inverted to false, so that the scan is no longer "in" any polygons. Furthermore, because *AC* is the last edge in the AET, the scan-line processing is completed. The AET is updated from the ET and is again ordered on x because some of its edges may have crossed, and the next scan line is processed.

When the scan line $y = \beta$ is encountered, the ordered AET is *AB, AC, FD,* and *FE*. Processing proceeds much as before. There are two polygons on the scan line, but the scan is "in" only one polygon at a time.

For scan line $y = \gamma$, things are more interesting. Entering *ABC* causes its flag to become *true*. *ABC*'s shade is used for the span up to the next edge, *DE*. At this point, the flag for *DEF* also becomes *true*, so the scan is "in" two polygons. (It is useful to keep an explicit list of polygons whose in–out flag is *true*, and also to keep a count of how many polygons are on the list.) We must now decide whether *ABC* or *DEF* is closer to the viewer, which we determine by evaluating the plane equations of both polygons for z at $y = \gamma$ and with x equal to the intersection of $y = \gamma$ with edge *DE*. This value of x is in the AET entry for *DE*. In our example, *DEF* has a larger z and thus is visible. Therefore, assuming nonpenetrating polygons, the shading for *DEF* is used for the span to edge *CB*, at which point *ABC*'s flag becomes *false* and the scan is again "in" only one polygon *DEF* whose shade continues to be used up to edge *FE*. Figure 15.36 shows the relationship of the two polygons and the $y = \gamma$ plane; the two thick lines are the intersections of the polygons with the plane.

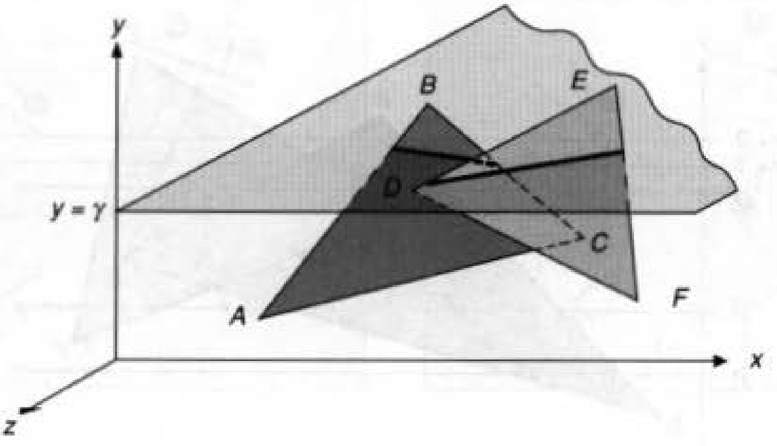


Fig. 15.36 Intersections of polygons *ABC* and *DEF* with the plane $y = \gamma$.

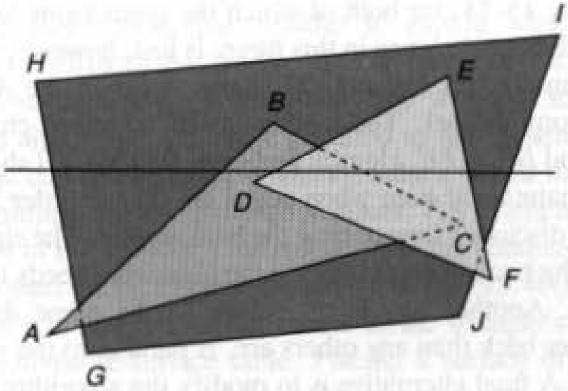


Fig. 15.37 Three nonpenetrating polygons. Depth calculations do not need to be made when scan line γ leaves the obscured polygon *ABC*, since nonpenetrating polygons maintain their relative *z* order.

Suppose there is a large polygon *GHIJ* behind both *ABC* and *DEF*, as in Fig. 15.37. Then, when the $y = \gamma$ scan line comes to edge *CB*, the scan is still "in" polygons *DEF* and *GHIJ*, so depth calculations are performed again. These calculations can be avoided, however, if we assume that none of the polygons penetrate another. This assumption means that, when the scan leaves *ABC*, the depth relationship between *DEF* and *GHIJ* cannot change, and *DEF* continues to be in front. Therefore, depth computations are unnecessary when the scan leaves an obscured polygon, and are required only when it leaves an obscuring polygon.

To use this algorithm properly for penetrating polygons, as shown in Fig. 15.38, we break up *KLM* into *KLL'M'* and *L'MM'*, introducing the *false edge M'L'*. Alternatively, the algorithm can be modified to find the point of penetration on a scan line as the scan line is processed.

Another modification to this algorithm uses *depth coherence*. Assuming that polygons do not penetrate each other, Romney noted that, if the same edges are in the AET on one scan line as are on the immediately preceding scan line, and if they are in the same order, then no changes in depth relationships have occurred on any part of the scan line and no new depth computations are needed [ROMN68]. The record of visible spans on the previous scan line then defines the spans on the current scan line. Such is the case for scan lines $y = \gamma$

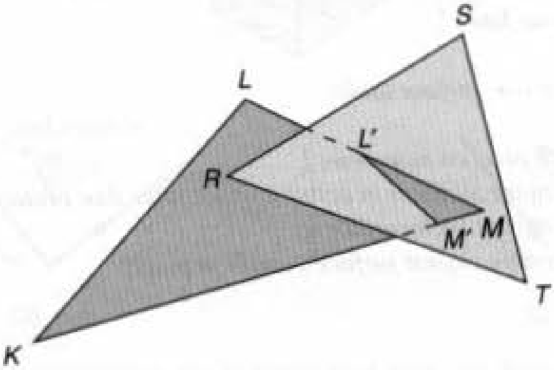


Fig. 15.38 Polygon *KLM* pierces polygon *RST* at the line *L'M'*.

and $y = \gamma + 1$ in Fig. 15.34, for both of which the spans from *AB* to *DE* and from *DE* to *FE* are visible. The depth coherence in this figure is lost, however, as we go from $y = \gamma + 1$ to $y = \gamma + 2$, because edges *DE* and *CB* change order in the AET (a situation that the algorithm must accommodate). The visible spans therefore change and, in this case, become *AB* to *CB* and *DE* to *FE*. Hamlin and Gear [HAML77] show how depth coherence can sometimes be maintained even when edges do change order in the AET.

We have not yet discussed how to treat the background. The simplest way is to initialize the frame buffer to the background color, so the algorithm needs to process only scan lines that intersect edges. Another way is to include in the scene definition a large enough polygon that is farther back than any others are, is parallel to the projection plane, and has the desired shading. A final alternative is to modify the algorithm to place the background color explicitly into the frame buffer whenever the scan is not “in” any polygon.

Although the algorithms presented so far deal with polygons, the scan-line approach has been used extensively for more general surfaces, as described in Section 15.9. To accomplish this, the ET and AET are replaced by a *surface table* and *active-surface table*, sorted by the surfaces' (x, y) extents. When a surface is moved from the surface table to the active-surface table, additional processing may be performed. For example, the surface may be decomposed into a set of approximating polygons, which would then be discarded when the scan leaves the surface's y extent; this eliminates the need to maintain all surface data throughout the rendering process. Pseudocode for this general scan-line algorithm is shown in Fig. 15.39. Atherton [ATHE83] discusses a scan-line algorithm that renders polygonal objects combined using the regularized Boolean set operations of constructive solid geometry.

A scan-line approach that is appealing in its simplicity uses a z-buffer to resolve the visible-surface problem [MYER75]. A single-scan-line frame buffer and z-buffer are cleared for each new scan line and are used to accumulate the spans. Because only one scan line of storage is needed for the buffers, extremely high-resolution images are readily accommodated.

```
add surfaces to surface table;
initialize active-surface table;

for (each scan line) {

    update active-surface table;

    for (each pixel on scan line) {
        determine surfaces in active-surface table that project to pixel;
        find closest such surface;
        determine closest surface's shade at pixel;
    }
}
```

Fig. 15.39 Pseudocode for a general scan-line algorithm.

Crocker [CROC84] uses a scan-line z-buffer to exploit what he calls *invisibility coherence,* the tendency for surfaces that are invisible on one scan line to be invisible on the next. When the active-surface table is made up for a given scan line, a separate *invisible-surface table* is also built. A surface is added to the invisible-surface table if its maximum z value for the current scan line is less than the z values in the previous line's z buffer at the surface's minimum and maximum x values. For example, given the cube and contained triangle shown in Fig 15.40(a), the triangle and the contents of the previous scan line's z-buffer projected onto the (x, z) plane are shown in Fig. 15.40(b). The triangle's $z_{\max}$ is less than the previous scan line's z-buffer values at the triangle's $x_{\min}$ and $x_{\max}$, so the triangle is added to the invisible-surface table. Placing a surface in the invisible-surface table eliminates it from much of the visible-surface processing. Some surfaces in the invisible-surface table may not belong there. To remedy this, as each pixel on the current scan line is processed, surfaces are removed from the invisible-surface table and are added to the active-surface table if their maximum z value is greater than the z value of what is currently determined to be the visible surface at the pixel. For example, even though the triangle shown in Fig. 15.40(c) was placed in the invisible-surface table, it is actually visible because the cube has been clipped, and it will be removed and added to the active-surface table.

Sechrest and Greenberg [SECH82] have developed an object-precision algorithm for nonintersecting polygons that is somewhat in the spirit of a scan-line algorithm. Their algorithm relies on the fact that the visibility of edges can change only at vertices and edge crossings. It sorts vertices and edge crossings by y, effectively dividing up the scene into horizontal bands inside of which the visibility relationships are constant (see Fig. 15.41). Object-precision coordinates of the edge segments visible in each strip are output as the strip is processed and are supplemented with information from which the contours of visible polygons can be reconstructed for scan conversion. Initially, only vertices that are local minima are inspected in sorted order. An AET is kept, and is modified whenever a vertex is encountered in the scan. Edge crossings are determined on the fly by testing only the active edges for intersections.

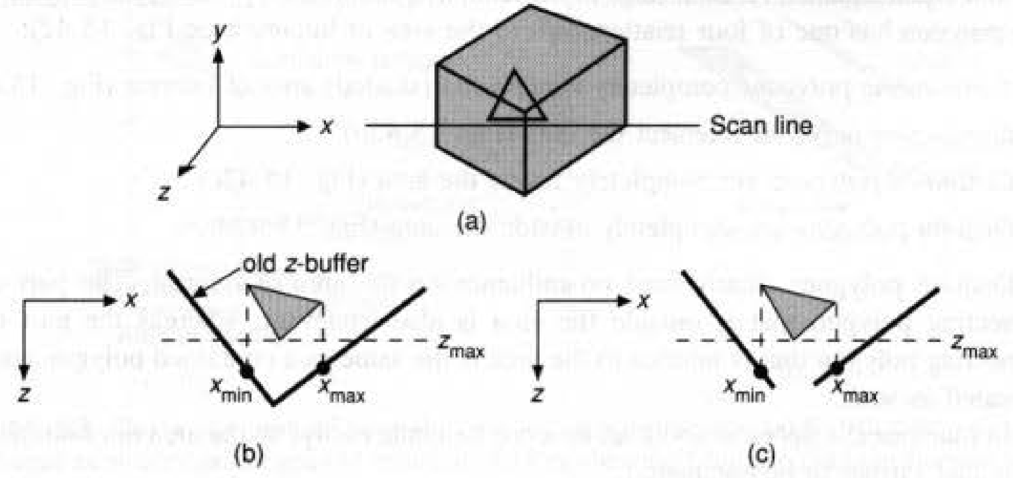


Fig. 15.40 Invisibility coherence. (a) Triangle in a box. (b) Triangle is correctly placed in invisible table. (c) Triangle is incorrectly placed in invisible-surface table. (Based on [CROC84].)

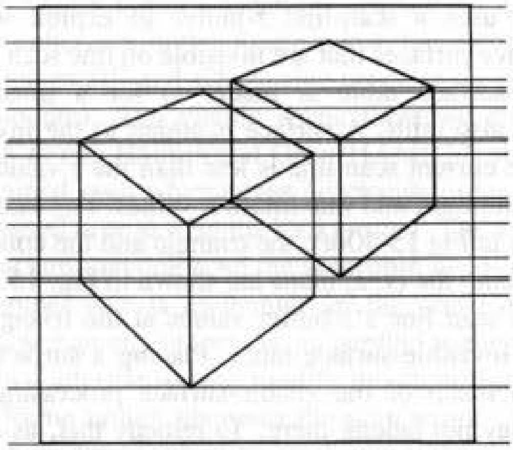


Fig. 15.41 The Sechrest and Greenberg object-precision algorithm divides the picture plane into horizontal strips at vertices and edge crossings. (Courtesy of Stuart Sechrest and Donald P. Greenberg, Program of Computer Graphics, Cornell Univeristy, 1982.)

15.7 AREA-SUBDIVISION ALGORITHMS

Area-subdivision algorithms all follow the divide-and-conquer strategy of spatial partitioning in the projection plane. An area of the projected image is examined. If it is easy to decide which polygons are visible in the area, they are displayed. Otherwise, the area is subdivided into smaller areas to which the decision logic is applied recursively. As the areas become smaller, fewer polygons overlap each area, and ultimately a decision becomes possible. This approach exploits *area coherence*, since sufficiently small areas of an image will be contained in at most a single visible polygon.

15.7.1 Warnock's Algorithm

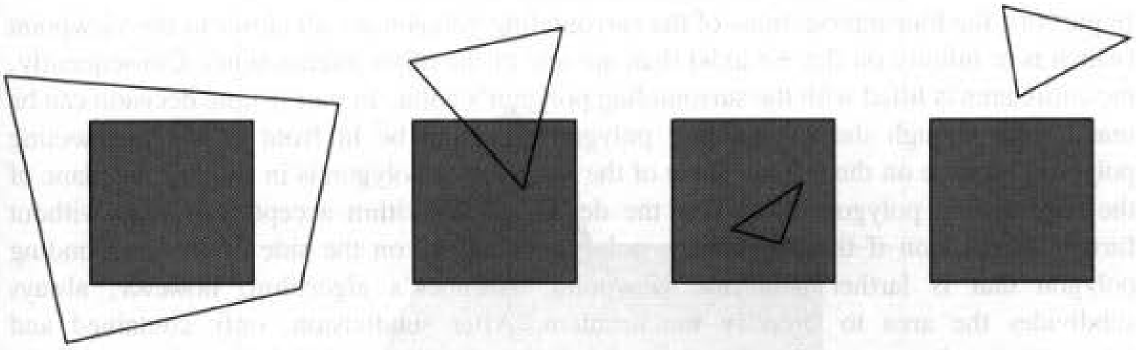
The area-subdivision algorithm developed by Warnock [WARN69] subdivides each area into four equal squares. At each stage in the recursive-subdivision process, the projection of each polygon has one of four relationships to the area of interest (see Fig. 15.42):

1. *Surrounding polygons* completely contain the (shaded) area of interest (Fig. 15.42a)
2. *Intersecting polygons* intersect the area (Fig. 15.42b)
3. *Contained polygons* are completely inside the area (Fig. 15.42c)
4. *Disjoint polygons* are completely outside the area (Fig. 15.42d).

Disjoint polygons clearly have no influence on the area of interest. The part of an intersecting polygon that is outside the area is also irrelevant, whereas the part of an intersecting polygon that is interior to the area is the same as a contained polygon and can be treated as such.

In four cases, a decision about an area can be made easily, so the area does not need to be divided further to be conquered:

1. All the polygons are disjoint from the area. The background color can be displayed in the area.



(a) Surrounding (b) Intersecting (c) Contained (d) Disjoint

Fig. 15.42 Four relations of polygon projections to an area element: (a) surrounding, (b) intersecting, (c) contained, and (d) disjoint.

2. There is only one intersecting or only one contained polygon. The area is first filled with the background color, and then the part of the polygon contained in the area is scan-converted.
3. There is a single surrounding polygon, but no intersecting or contained polygons. The area is filled with the color of the surrounding polygon.
4. More than one polygon is intersecting, contained in, or surrounding the area, but one is a surrounding polygon that is in front of all the other polygons. Determining whether a surrounding polygon is in front is done by computing the *z* coordinates of the planes of all surrounding, intersecting, and contained polygons at the four corners of the area; if there is a surrounding polygon whose four corner *z* coordinates are larger (closer to the viewpoint) than are those of any of the other polygons, then the entire area can be filled with the color of this surrounding polygon.

Cases 1, 2, and 3 are simple to understand. Case 4 is further illustrated in Fig. 15.43.

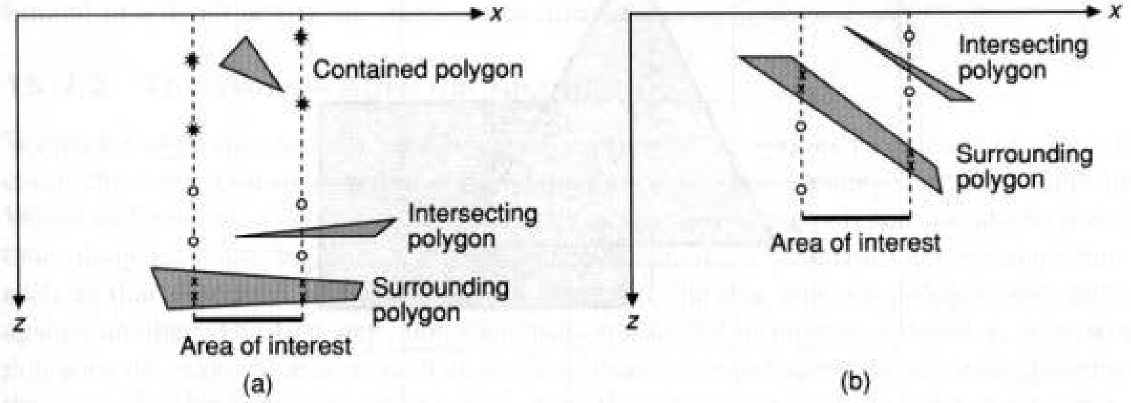


Fig. 15.43 Two examples of case 4 in recursive subdivision. (a) Surrounding polygon is closest at all corners of area of interest. (b) Intersecting polygon plane is closest at left side of area of interest. × marks the intersection of surrounding polygon plane; ○ marks the intersection of intersecting polygon plane; * marks the intersection of contained polygon plane.

In part (a), the four intersections of the surrounding polygon are all closer to the viewpoint (which is at infinity on the $+z$ axis) than are any of the other intersections. Consequently, the entire area is filled with the surrounding polygon's color. In part (b), no decision can be made, even though the surrounding polygon seems to be in front of the intersecting polygon, because on the left the plane of the intersecting polygon is in front of the plane of the surrounding polygon. Note that the depth-sort algorithm accepts this case without further subdivision if the intersecting polygon is wholly on the side of the surrounding polygon that is farther from the viewpoint. Warnock's algorithm, however, always subdivides the area to simplify the problem. After subdivision, only contained and intersecting polygons need to be reexamined: Surrounding and disjoint polygons of the original area are surrounding and disjoint polygons of each subdivided area.

Up to this point, the algorithm has operated at object precision, with the exception of the actual scan conversion of the background and clipped polygons in the four cases. These image-precision scan-conversion operations, however, can be replaced by object-precision operations that output a precise representation of the visible surfaces: either a square of the area's size (cases 1, 3, and 4) or a single polygon clipped to the area, along with its Boolean complement relative to the area, representing the visible part of the background (case 2). What about the cases that are not one of these four? One approach is to stop subdividing when the resolution of the display device is reached. Thus, on a 1024 by 1024 raster display, at most 10 levels of subdivision are needed. If, after this maximum number of subdivisions, none of cases 1 to 4 have occurred, then the depth of all relevant polygons is computed at the center of this pixel-sized, indivisible area. The polygon with the closest z coordinate defines the shading of the area. Alternatively, for antialiasing, several further levels of subdivision can be used to determine a pixel's color by weighting the color of each

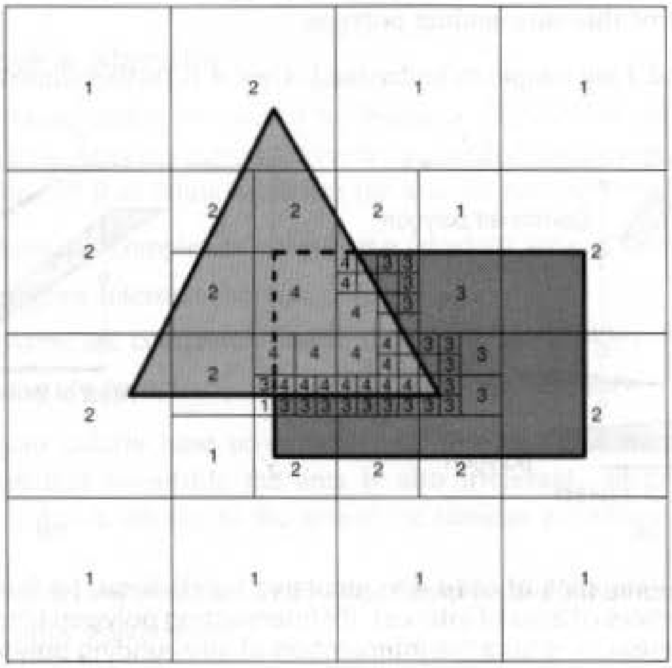


Fig. 15.44 Area subdivision into squares.

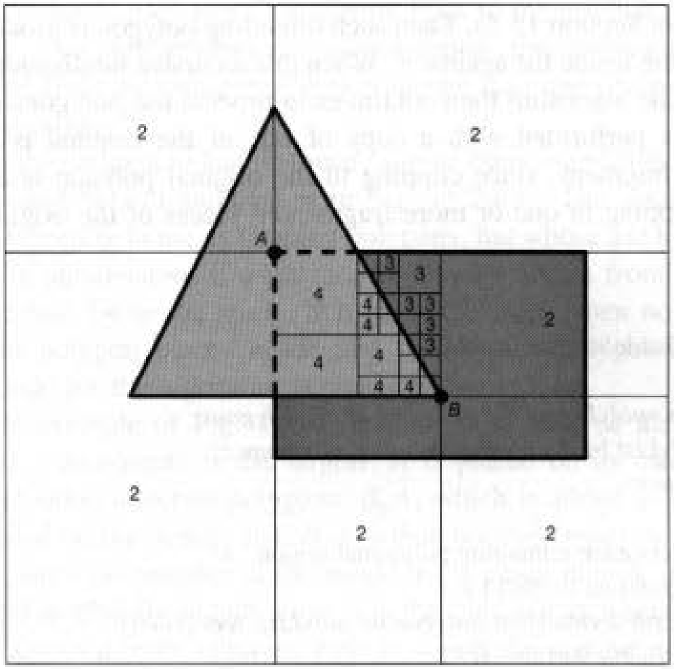


Fig. 15.45 Area subdivision about circled polygon vertices. The first subdivision is at vertex *A*; the second is at vertex *B*.

of its subpixel-sized areas by its size. It is these image-precision operations, performed when an area is not one of the simple cases, that makes this an image-precision approach.

Figure 15.44 shows a simple scene and the subdivisions necessary for that scene's display. The number in each subdivided area corresponds to one of the four cases; in unnumbered areas, none of the four cases are true. Compare this approach to the 2D spatial partitioning performed by quadtrees (Section 12.6.3). An alternative to equal-area subdivision, shown in Fig. 15.45, is to divide about the vertex of a polygon (if there is a vertex in the area) in an attempt to avoid unnecessary subdivisions. Here subdivision is limited to a depth of five for purposes of illustration.

15.7.2 The Weiler–Atherton Algorithm

Warnock's algorithm was forced to use image-precision operations to terminate because it could clip a polygon only against a rectangular area. Another strategy, developed later by Weiler and Atherton [WEIL77], subdivides the screen area along polygon boundaries rather than along rectangle boundaries. This approach requires a powerful clipping algorithm, such as that described in Section 19.1.4, that can clip one concave polygon with holes against another. The first step, not mandatory but useful to improve efficiency, is to sort polygons on some value of z, such as on their nearest z coordinate. The polygon closest to the viewer by this criterion is then used to clip all the polygons, including the clip polygon, into two lists containing the pieces inside and outside the clip polygon. All polygons on the inside list that are behind the clip polygon are then deleted, since they are invisible. If any polygon on the inside list is closer than is the clip polygon, the initial sort did not give a correct priority order (in fact, such a priority order does not exist in the case of cyclic overlap discussed in Section 15.5). Each such offending polygon is processed recursively to clip the pieces on the inside list against it. When this recursive subdivision is over, the inside list is displayed. The algorithm then continues to process the polygons on the outside list. Clipping is always performed with a copy of one of the original polygons as the clip polygon, never a fragment, since clipping to the original polygon is assumed to be less expensive than clipping to one or more fragmented pieces of the original polygon. Thus,

```
void WA_visibleSurface (void)
{
    polygon *polyList = list of copies of all polygons;
    sort polyList by decreasing value of maximum z;
    clear stack;

    /* Process each remaining polygonal region. */
    while (polyList != NULL)
        WA_subdivide (first polygon on polyList, &polyList);
}   /* WA_visibleSurface */

void WA_subdivide (polygon clipPolygon, polygon **polyList)
{
    polygon *inList;          /* Fragments inside clipPolygon */
    polygon *outList;         /* Fragments outside clipPolygon */

    inList = NULL;
    outList = NULL;

    for (each polygon in *polyList)
        clip polygon to ancestor of clipPolygon, placing inside pieces on
            inList, outside pieces on outList;

    remove polygons behind clipPolygon from inList;

    /* Process incorrectly ordered fragments recursively. */
    for (each polygon in inList that is not on stack and not a part of clipPolygon) {
        push clipPolygon onto stack;
        WA_subdivide (polygon, &inList);
        pop stack;
    }

    /* Display remaining polygons inside clipPolygon. */
    for (each polygon in inList)
        display polygon;

    *polyList = outList;      /* Subtract inList from *polyList. */
}   /* WA_subdivide */
```

Fig. 15.46 Pseudocode for the Weiler–Atherton visible surface algorithm.

when a polygon is clipped, each piece must point back to the original input polygon from which it is derived. As a further efficiency consideration, the clipping algorithm can treat any polygon derived from the clip polygon as a special case and place it on the inside list with no additional testing.

The algorithm uses a stack to handle those cases of cyclic overlap in which one polygon is both in front of and behind another, as in Fig. 15.24(b). The stack contains a list of polygons that are currently in use as clipping polygons, but whose use has been interrupted because of recursive subdivision. If a polygon is found to be in front of the current clip polygon, it is searched for in the stack. If it is on the stack, then no more recursion is necessary since all polygon pieces inside and behind that polygon have already been removed. Pseudocode for the algorithm is shown in Fig. 15.46.

For the simple example of Fig. 15.47, triangle A is used as the first clip polygon because its nearest z coordinate is the largest. A is placed on its own inside list; next, rectangle B is subdivided into two polygons: $B_{in}A$, which is added to the inside list, and $B_{out}A$, which is placed on the outside list. $B_{in}A$ is then removed from the inside list, since it is behind A. Now, since no member of the inside list is closer than A, A is output. $B_{out}A$ is processed next, and is trivially output since it is the only polygon remaining.

Figure 15.48 shows a more complex case in which the original sorted order (or any other order) is incorrect. Part (a) depicts four polygons whose vertices are each marked with their z value. Rectangle A is considered to be closest to the viewer because its maximum z coordinate is greatest. Therefore, in the first call to WA_subdivide, A is used to clip all the polygons, as shown in part (b). The inside list is A, $B_{in}A$, $C_{in}A$, and $D_{in}A$; the outside list is $B_{out}A$, $C_{out}A$, and $D_{out}A$. $B_{in}A$ and $D_{in}A$ are discarded because they are farther back than A is, leaving only A and $C_{in}A$ on the inside list. Since $C_{in}A$ is found to be on the near side of A's plane, however, it is apparent that the polygons were ordered incorrectly. Therefore, recursive subdivision is accomplished by calling WA_subdivide to clip the current inside list against C, the ancestor of the offending polygon, as shown in part (c). The new inside list for this level of recursion is $A_{in}C$ and $C_{in}A$; the new outside list is $A_{out}C$. $A_{in}C$ is removed from the inside list because it is behind C. Only $C_{in}A$ is left on the inside list; since it is a part of the clip polygon, it is displayed. Before returning from the recursive call to WA_subdivide, *polyList* is set to the new outside list containing only $A_{out}C$. Since *polyList* is the caller's *inList*, $A_{out}C$ is displayed next, as shown in part (d), in which displayed

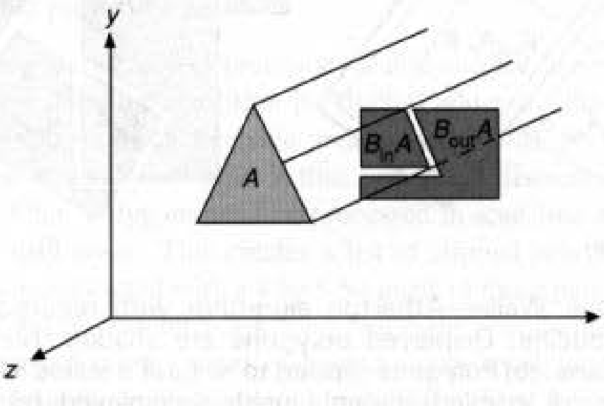


Fig. 15.47 Subdivision of a simple scene using the Weiler–Atherton algorithm.

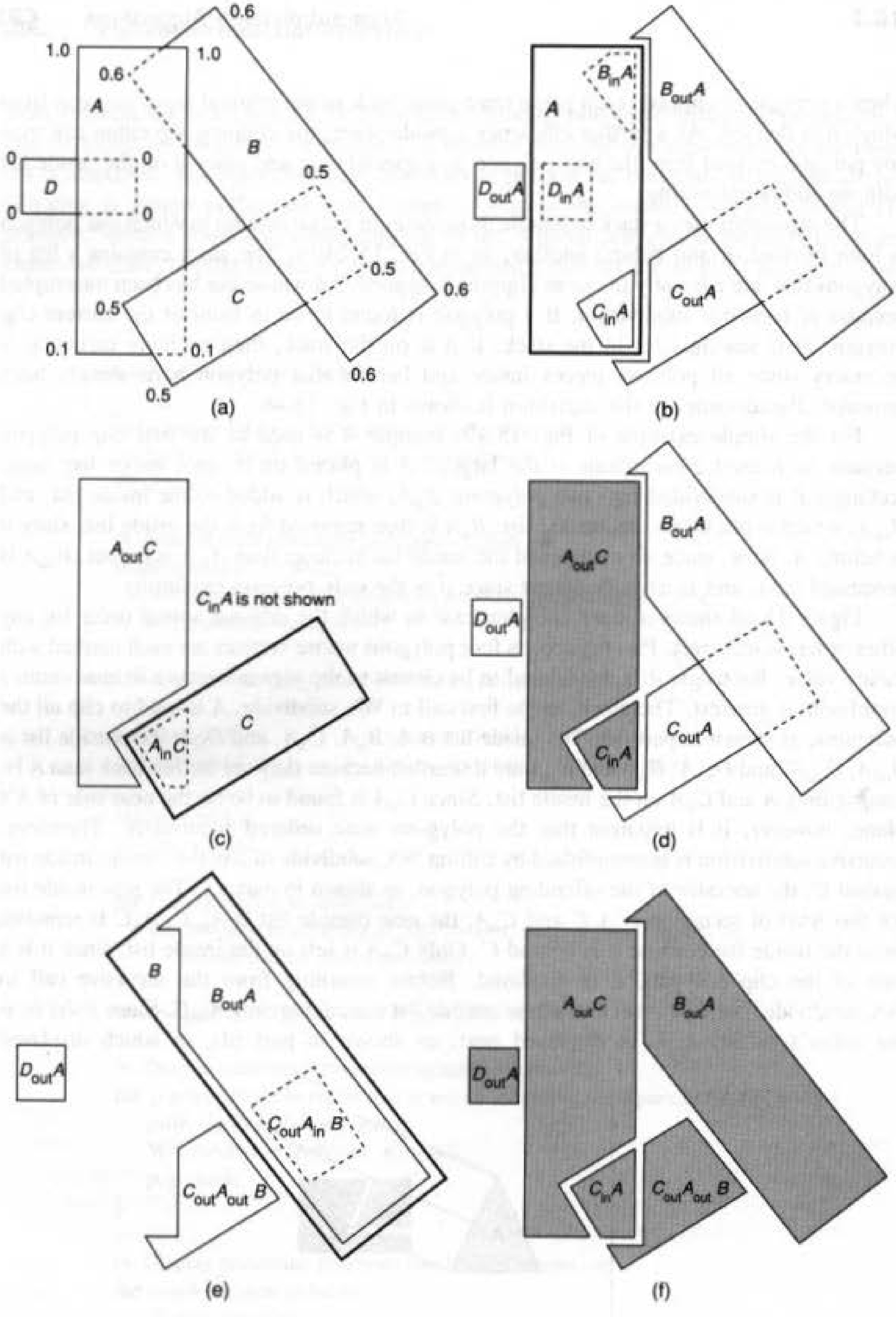


Fig. 15.48 Using the Weiler–Atherton algorithm with recursion. Clip polygon is shown with heavy outline. Displayed polygons are shaded. Numbers are vertex z values. (a) Original scene. (b) Polygons clipped to *A*. (c) *A*'s inside list clipped to *C* during recursive subdivision. (d) Visible fragments inside *A* displayed. (e) Polygons clipped to *B*. (f) All visible fragments displayed at completion.

fragments are shaded. The initial invocation of WA_subdivide then sets *polyList* to its outside list ($B_{\text{out}}A$, $C_{\text{out}}A$, and $D_{\text{out}}A$) and returns.

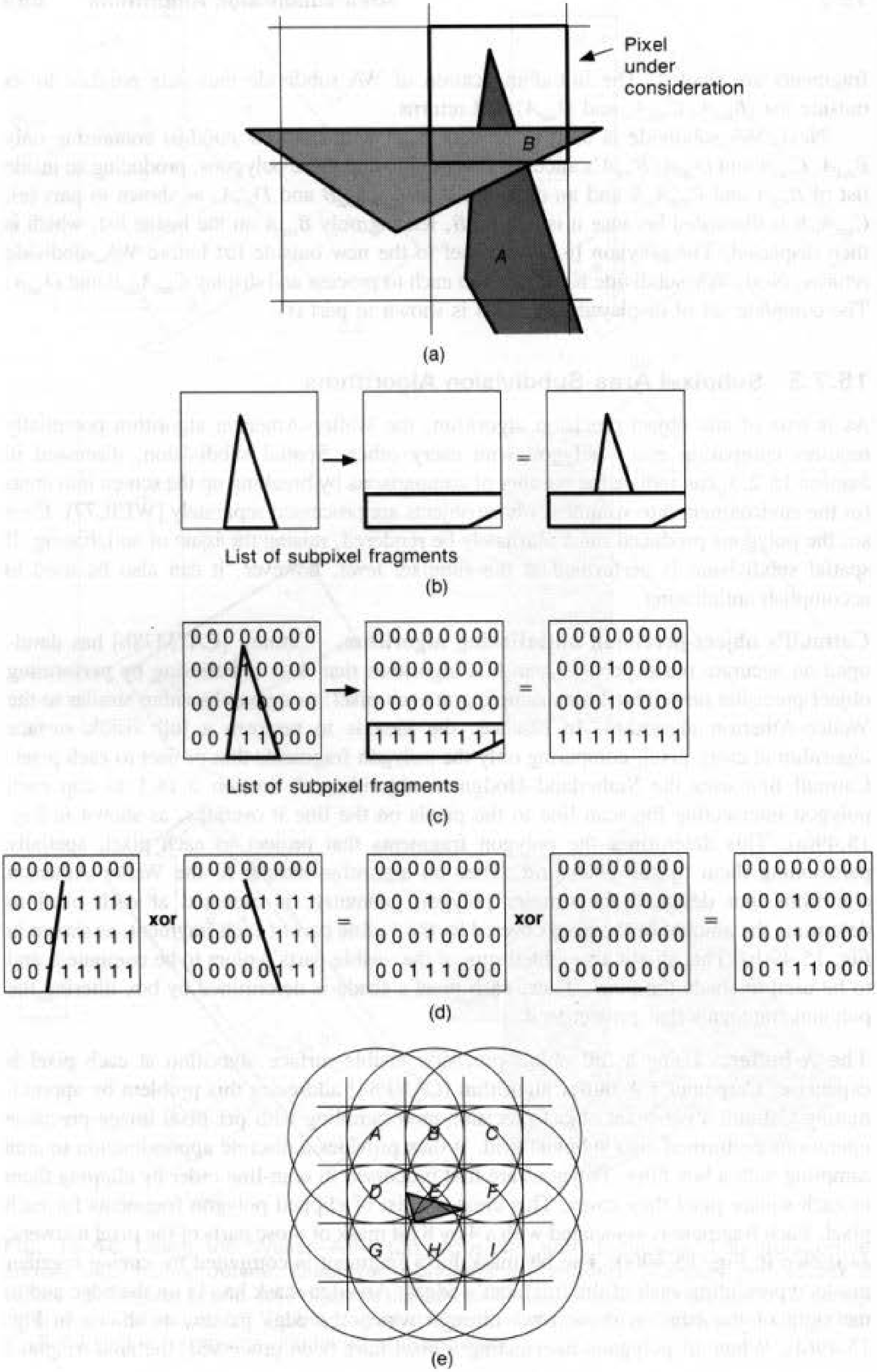
Next, WA_subdivide is used to process $B_{\text{out}}A$ with the new *polyList* containing only $B_{\text{out}}A$, $C_{\text{out}}A$ and $D_{\text{out}}A$. $B_{\text{out}}A$'s ancestor B is used to clip these polygons, producing an inside list of $B_{\text{out}}A$ and $C_{\text{out}}A_{\text{in}}B$ and an outside list of $C_{\text{out}}A_{\text{out}}B$ and $D_{\text{out}}A$, as shown in part (e). $C_{\text{out}}A_{\text{in}}B$ is discarded because it is behind B, leaving only $B_{\text{out}}A$ on the inside list, which is then displayed. The polygon list is then set to the new outside list before WA_subdivide returns. Next, WA_subdivide is called once each to process and display $C_{\text{out}}A_{\text{out}}B$ and $D_{\text{out}}A$. The complete set of displayed fragments is shown in part (f).

15.7.3 Subpixel Area-Subdivision Algorithms

As is true of any object-precision algorithm, the Weiler–Atherton algorithm potentially requires comparing every polygon with every other. Spatial subdivision, discussed in Section 15.2.5, can reduce the number of comparisons by breaking up the screen into areas (or the environment into volumes) whose objects are processed separately [WEIL77]. Even so, the polygons produced must ultimately be rendered, raising the issue of antialiasing. If spatial subdivision is performed at the subpixel level, however, it can also be used to accomplish antialiasing.

Catmull's object-precision antialiasing algorithm. Catmull [CATM78b] has developed an accurate but expensive scan-line algorithm that does antialiasing by performing object-precision unweighted area sampling at each pixel, using an algorithm similar to the Weiler–Atherton algorithm. In essence, the idea is to perform a full visible-surface algorithm at every pixel, comparing only the polygon fragments that project to each pixel. Catmull first uses the Sutherland–Hodgman algorithm of Section 3.14.1 to clip each polygon intersecting the scan line to the pixels on the line it overlaps, as shown in Fig. 15.49(a). This determines the polygon fragments that project to each pixel, spatially partitioning them by the pixel grid. Then an algorithm similar to the Weiler–Atherton algorithm, but designed for simpler polygon geometry, is executed at each pixel to determine the amount of the pixel covered by the visible part of each fragment, as shown in Fig. 15.49(b). This allows a weighted sum of the visible parts' colors to be computed, and to be used to shade the pixel. Thus, each pixel's shade is determined by box filtering the polygon fragments that project to it.

The A-buffer. Using a full object-precision visible-surface algorithm at each pixel is expensive! Carpenter's A-buffer algorithm [CARP84] addresses this problem by approximating Catmull's per-pixel object-precision area sampling with per-pixel image-precision operations performed on a subpixel grid. It thus provides a discrete approximation to area sampling with a box filter. Polygons are first processed in scan-line order by clipping them to each square pixel they cover. This creates a list of clipped polygon fragments for each pixel. Each fragment is associated with a 4 by 8 bit mask of those parts of the pixel it covers, as shown in Fig. 15.49(c). The bit mask for a fragment is computed by **xor**ing together masks representing each of the fragment's edges. An edge mask has 1s on the edge and to the right of the edge in those rows through which the edge passes, as shown in Fig. 15.49(d). When all polygons intersecting a pixel have been processed, the area-weighted


Pixel
under
consideration
B
A
(a)
List of subpixel fragments
(b)
00000000
00010000
00011000
00111000
00000000
00000000
00000000
11111111
=
00000000
00010000
00010000
11111111
List of subpixel fragments
(c)
00000000
00011111
00011111
00111111
xor
00000000
00001111
00000111
00000011
=
00000000
00010000
00010000
00111000
xor
00000000
00000000
00000000
00000000
=
00000000
00010000
00010000
00111000
(d)
A
B
C
D
E
F
G
H
I
(e)

average of the colors of the pixel's visible surfaces is obtained by selecting fragments in depth-sorted order and using their bit masks to clip those of farther fragments. The bit masks can be manipulated efficiently with Boolean operations. For example, two fragment bit masks can be **and**ed together to determine the overlap between them. The A-buffer algorithm saves only a small amount of additional information with each fragment. For example, it includes the fragment's z extent, but no information about which part of the fragment is associated with these z values. Thus, the algorithm must make assumptions about the subpixel geometry in cases in which fragment bit masks overlap in z. This causes inaccuracies, especially where multiple surfaces intersect in a pixel.

Using precomputed convolution tables for better filtering. Both subpixel area-subdivision algorithms described so far use unweighted area sampling; thus, they restrict a fragment's influence to the single pixel to which it is clipped. If we would like to use filters with wider support, however, we must take into account that each fragment lies within the support of a number of filters positioned over nearby pixels, as shown in Fig. 15.49(e). Abram, Westover, and Whitted [ABRA85] describe how to incorporate better filters in these algorithms by classifying each visible fragment into one of a number of classes, based on the fragment's geometry. For each pixel within whose filter the fragment lies (pixels *A* through *I* in Fig 15.49e), the fragment's class and its position relative to the pixel are used to index into a look-up table. The look-up table contains the precomputed convolution of the desired filter kernel with prototype fragments at a set of different positions. The selected entry is multiplied by the fragment's intensity value and is added to an accumulator at that pixel. Those fragments that do not fit into one of the classes are approximated either as sums and differences of simpler fragments or by using bit masks.

15.8 ALGORITHMS FOR OCTREES

Algorithms for displaying octree-encoded objects (see Section 12.6.3) take advantage of the octree's regular structure of nonintersecting cubes. Since the octree is spatially presorted, list-priority algorithms have been developed that yield a correct display order for parallel projections [DOCT81; MEAG82a; GARG86]. In a back-to-front enumeration, nodes are listed in an order in which any node is guaranteed not to obscure any node listed after it. For an orthographic projection, a correct back-to-front enumeration can be determined from the VPN alone. One approach is to display the farthest octant first, then those three neighbors that share a face with the farthest octant in any order, then those three neighbors of the closest octant in any order, then the closest octant. In Fig. 15.50, one such enumeration for a VPN from 0 to V is 0, 1, 2, 4, 3, 5, 6, 7. No node in this enumeration can obscure any node enumerated after it. As each octant is displayed, its descendants are displayed recursively in this order. Furthermore, because each leaf node is a cube, at most three of its faces are visible, the identities of which may also be determined from the VPN.

Fig. 15.49 Subpixel area-subdivision algorithms. (a) Sample pixel contents. (b) Catmull algorithm subpixel geometry. (c) A-buffer algorithm subpixel geometry. (d) A-buffer algorithm subpixel mask for a fragment is computed by **xor**ing together masks for its edges. (e) Abram, Westover, and Whitted algorithm adds polygon's contribution to all pixels it affects. (Part e is based on [ABRA85].)

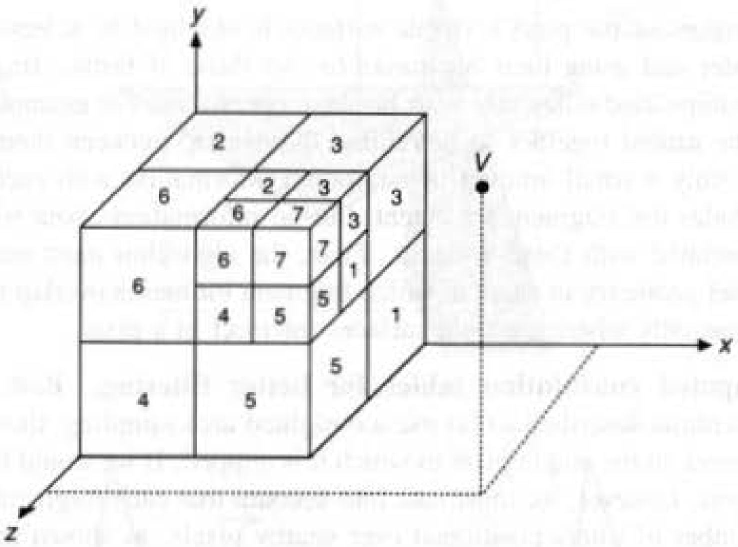


Fig. 15.50 Octree enumeration for back-to-front display. (Node 0 is at the lower-left back corner.) For a VPN from the origin to *V*, nodes may be recursively displayed using several different ordering systems.

Table 15.1 shows eight different back-to-front orders as determined by the signs of the three coordinates of the VPN and the visible octant faces associated with each. (Note that only the first and last octants in each order are fixed.) A positive or negative VPN *x* coordinate means the right (R) or left (L) face is visible, respectively. Similarly, the *y* coordinate determines the visibility of the up (U) and down (D) faces, and the *z* coordinate controls the front (F) and back (B) faces. If any VPN coordinate is zero, then neither of the faces associated with it is visible. Only the VPN's nonzero coordinates are significant in determining an order. Since all octree nodes are identically oriented, the visible faces and their relative polygonal projections for all nodes need to be determined only once. Arbitrary parallel projections can be accommodated by considering the DOP instead of the VPN.

Another approach to back-to-front enumeration for orthographic projections iterates through the octants in slices perpendicular to one of the axes, and in either rows or columns

TABLE 15.1 BACK-TO-FRONT ENUMERATION AND VISIBLE FACES

VPN				
z	*y*	*x*	Back-to-front order	Visible faces*
−	−	−	7,6,5,3,4,2,1,0	B,D,L
−	−	+	6,7,4,2,5,3,0,1	B,D,R
−	+	−	5,4,7,1,6,0,3,2	B,U,L
−	+	+	4,5,6,0,7,1,2,3	B,U,R
+	−	−	3,2,1,7,0,6,5,4	F,D,L
+	−	+	2,3,0,6,1,7,4,5	F,D,R
+	+	−	1,0,3,5,2,4,7,6	F,U,L
+	+	+	0,1,2,4,3,5,6,7	F,U,R

*R = right, L = left; U = up; D = down; F = front; B = back.

within each slice. The sign of each component of the VPN determines the direction of iteration along the corresponding octree axis. A positive component indicates increasing order along its axis, whereas a negative component indicates decreasing order. The order in which the axes are used does not matter. For example, in Fig. 15.50, one such enumeration for a VPN with all positive coordinates is 0, 4, 2, 6, 1, 5, 3, 7, varying first *z*, then *y*, and then *x*. This approach is easily generalized to operate on voxel arrays [FRIE85].

It is not necessary to display all an object's voxels, since those that are surrounded entirely by others will ultimately be invisible; more efficient scan conversion can be accomplished by rendering only the voxels on the octree's border [GARG86]. The set of border voxels can be determined using the algorithm presented in Section 12.6.3. Further improvement may be obtained by noting that, even when only border voxels are displayed, some faces may be drawn and then overwritten. Gargantini [GARG86] uses the information obtained during border extraction to identify for each voxel those faces that abut another voxel. These faces need not be drawn, since they will always be obscured. Rather than scan convert each voxel as a small cube, it is also possible to approximate each voxel with a single upright rectangle (a pixel in the limiting case).

Meagher [MEAG82b] describes a front-to-back algorithm that uses the reverse of the back-to-front order described previously. It represents the image being rendered as a quadtree that is initially empty. Each full or partially full octree node is considered in front-to-back order and is compared with the quadtree nodes that its projection intersects. Those octree nodes whose projections intersect only full quadtree nodes are invisible; they and their descendants are not considered further. If a partially full octree node's projection intersects one or more partially full quadtree nodes, then the octree node's children are compared with the children of these quadtree nodes. If a full octree node's projection intersects partially full quadtree nodes, then only these partially full quadtree nodes are further subdivided to determine the previously empty nodes that are covered by the projection. Any empty quadtree node enclosed by a full octree node's projection is shaded with the octree node's value.

As shown in Fig. 15.51, Meagher bounds each octree-node projection with an upright rectangular extent. Any extent needs to be compared with only four of the lowest-level quadtree nodes whose edge size is at least as great as the extent's largest dimension. These

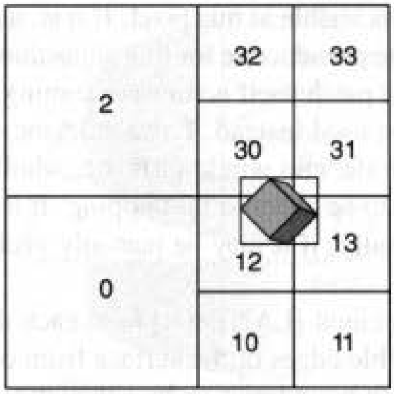


Fig. 15.51 Front-to-back octree scan conversion. Each octree node's projection and its rectangular extent are compared with four quadtree nodes, here 12, 13, 30, and 31.

are the quadtree node containing the extent's lower-left corner and the three adjacent quadtree nodes in the N, E, and NE directions. In Fig. 15.51, for example, these are quadtree nodes 12, 13, 30, and 31. If the rectangular extent intersects a rectangular quadtree node, then whether the octree node's projection (a convex polygon) also intersects the quadtree node can be determined efficiently [MEAG82b]. In contrast to the list-priority back-to-front algorithms, this front-to-back algorithm operates at image precision because it relies on an image-precision, quadtree representation of the projections on the image plane.

15.9 ALGORITHMS FOR CURVED SURFACES

All the algorithms presented thus far, with the exception of the z-buffer, have been described only for objects defined by polygonal faces. Objects such as the curved surfaces of Chapter 11 must first be approximated by many small facets before polygonal versions of any of the algorithms can be used. Although this approximation can be done, it is often preferable to scan convert curved surfaces directly, eliminating polygonal artifacts and avoiding the extra storage required by polygonal approximation.

Quadric surfaces, discussed in Section 11.4, are a popular choice in computer graphics. Visible-surface algorithms for quadrics have been developed by Weiss [WEIS66], Woon [WOON71], Mahl [MAHL72], Levin [LEVI76], and Sarraga [SARR83]. They all find the intersections of two quadrics, yielding a fourth-order equation in x, y, and z whose roots must be found numerically. Levin reduces this to a second-order problem by parameterizing the intersection curves. Spheres, a special case of quadrics, are easier to work with, and are of particular interest because molecules are often displayed as collections of colored spheres (see Color Plate II.19). A number of molecular display algorithms have been developed [KNOW77; STAU78; MAX79; PORT79; FRAN81; MAX84]. Section 15.10 discusses how to render spheres using ray tracing.

Even more flexibility can be achieved with the parametric spline surfaces introduced in Chapter 11, because they are more general and allow tangent continuity at patch boundaries. Catmull [CATM74b; CATM75] developed the first display algorithm for bicubics. In the spirit of Warnock's algorithm, a patch is recursively subdivided in s and t into four patches until its projection covers no more than one pixel. A z-buffer algorithm determines whether the patch is visible at this pixel. If it is, a shade is calculated for it and is placed in the frame buffer. The pseudocode for this algorithm is shown in Fig. 15.52. Since checking the size of the curved patch itself is time consuming, a quadrilateral defined by the patch's corner vertices may be used instead. Extra efficiency may be gained by comparing each patch (or its extent) with the clip window. If it is wholly inside the window, then no patch generated from it needs to be checked for clipping. If it is wholly outside the window, then it may be discarded. Finally, if it may be partially visible, then each patch generated from it must be checked.

Since then, Blinn and Whitted [LANE80b] have each developed scan-line algorithms for bicubics that track the visible edges of the surface from one scan line to the next. Edges may be defined by actual patch boundaries or by *silhouette edges*, as shown in Fig. 15.53. At a silhouette edge, the z component of the surface normal in the 3D screen coordinate system is zero as it passes between positive and negative values.

```
for (each patch) {
    push patch onto stack;

    while (stack not empty) {
        pop patch from stack;

        if (patch covers ≤ 1 pixel) {
            if (patch's pixel closer in z)
                determine shade and draw
        } else {
            subdivide patch into 4 subpatches;
            push subpatches onto stack;
        }
    }
}
```

Fig. 15.52 Pseudocode for the Catmull recursive-subdivision algorithm.

Blinn deals directly with the parametric representation of the patch. For the scan line $y = \alpha$, he finds all s and t values that satisfy the equation

$$y(s, t) - \alpha = 0. \tag{15.8}$$

These values of s and t are then used to evaluate $x(s, t)$ and $z(s, t)$. Unfortunately, Eq. (15.8) does not have a closed-form solution and its roots are therefore found numerically using Newton–Raphson iteration (see Appendix). Since the root-finding algorithm requires an initial value, coherence can be exploited by beginning with the previous scan line's solution for the current scan line. There are also special cases in which the roots cannot be found, causing the algorithm to fail. Similarly, Whitted uses numerical methods plus some approximations to the curve in the (x, z) plane defined by the intersection of the $y = \alpha$ plane

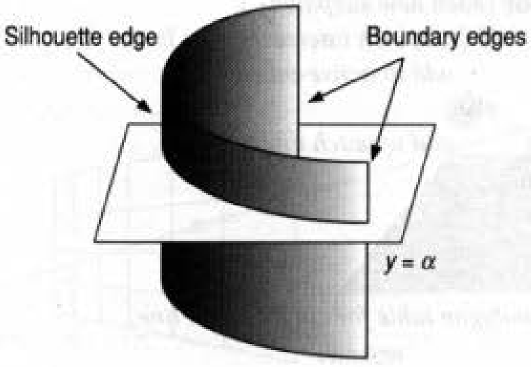


Fig. 15.53 The visible edges of a patch are defined by its boundary edges and silhouette edges.

with the bicubic surface patch. Whitted's algorithm fails to handle certain silhouette edges properly, however; an algorithm that does a more robust job of silhouette-edge detection is described in [SCHW82].

One highly successful approach is based on the adaptive subdivision of each bicubic patch until each subdivided patch is within some given tolerance of being flat. This tolerance depends on the resolution of the display device and on the orientation of the area being subdivided with respect to the projection plane, so unnecessary subdivisions are eliminated. The patch needs to be subdivided in only one direction if it is already flat enough in the other. Once subdivided sufficiently, a patch can be treated like a quadrilateral. The small polygonal areas defined by the four corners of each patch are processed by a scan-line algorithm, allowing polygonal and bicubic surfaces to be readily intermixed.

Algorithms that use this basic idea have been developed by Lane and Carpenter [LANE80b], and by Clark [CLAR79]. They differ in the choice of basis functions used to derive the subdivision difference equations for the surface patches and in the test for flatness. The Lane–Carpenter algorithm does the subdivision only as required when the scan line being processed begins to intersect a patch, rather than in a preprocessing step as does Clark's algorithm. The Lane–Carpenter patch subdivision algorithm is described in Section 11.3.5. Pseudocode for the Lane–Carpenter algorithm is shown in Fig. 15.54.

```
add patches to patch table;
initialize active-patch table;

for (each scan line) {
    update active-patch table;

    for (each patch in active-patch table) {
        if (patch can be approximately by planar quadrilateral)
            add patch to polygon table;
        else {
            split patch into subpatches;
            for (each new subpatch) {
                if (subpatch intersects scan line)
                    add to active-patch table;
                else
                    add to patch table;
            }
        }
    }

    process polygon table for current scan line;
}
```

Fig. 15.54 Pseudocode for the Lane–Carpenter algorithm.

Since a patch's control points define its convex hull, the patch is added to the active-patch table for processing at the scan line whose y value is that of the minimum y value of its control points. This saves large amounts of memory. The test for flatness must determine whether the patch is sufficiently planar and whether the boundary curves are sufficiently linear. Unfortunately, subdivision can introduce cracks in the patch if the same patch generates one patch that is determined to be flat and an adjacent patch that must be subdivided further. What should be a common shared edge between the patches may, instead, be a single line for the first patch and a piecewise linear approximation to a curve for the subpatches derived from the second patch. This can be avoided by changing the tolerance in the flatness test such that patches are subdivided more finely than necessary. An alternative solution uses Clark's method of subdividing an edge as though it were a straight line, once it has been determined to be sufficiently flat.

15.10 VISIBLE-SURFACE RAY TRACING

Ray tracing, also known as *ray casting*, determines the visibility of surfaces by tracing imaginary rays of light from the viewer's eye to the objects in the scene.[2] This is exactly the prototypical image-precision algorithm discussed at the beginning of this chapter. A center of projection (the viewer's eye) and a window on an arbitrary view plane are selected. The window may be thought of as being divided into a regular grid, whose elements correspond to pixels at the desired resolution. Then, for each pixel in the window, an *eye ray* is fired from the center of projection through the pixel's center into the scene, as shown in Fig. 15.55. The pixel's color is set to that of the object at the closest point of intersection. The pseudocode for this simple ray tracer is shown in Fig. 15.56.

Ray tracing was first developed by Appel [APPE68] and by Goldstein and Nagel [MAGI68; GOLD71]. Appel used a sparse grid of rays used to determine shading, including whether a point was in shadow. Goldstein and Nagel originally used their algorithm to simulate the trajectories of ballistic projectiles and nuclear particles; only later

[2] Although *ray casting* and *ray tracing* are often used synonymously, sometimes *ray casting* is used to refer to only this section's visible-surface algorithm, and *ray tracing* is reserved for the recursive algorithm of Section 16.12.

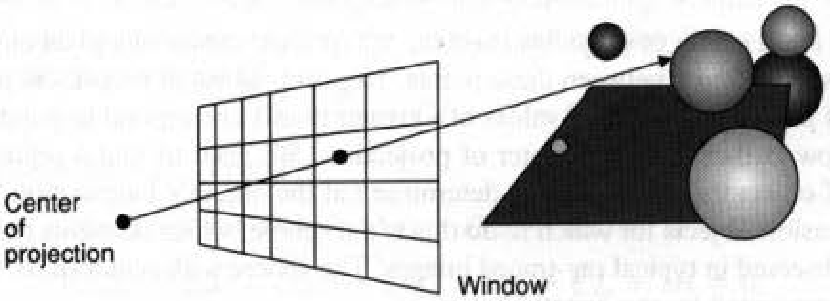


Fig. 15.55 A ray is fired from the center of projection through each pixel to which the window maps, to determine the closest object intersected.

```
select center of projection and window on viewplane;
for (each scan line in image) {
    for (each pixel in scan line) {
        determine ray from center of projection through pixel;
        for (each object in scene) {
            if (object is intersected and is closest considered thus far)
                record intersection and object name;
        }
        set pixel's color to that at closest object intersection;
    }
}
```

Fig. 15.56 Pseudocode for a simple ray tracer.

did they apply it to graphics. Appel was the first to ray trace shadows, whereas Goldstein and Nagel pioneered the use of ray tracing to evaluate Boolean set operations. Whitted [WHIT80] and Kay [KAY79a] extended ray tracing to handle specular reflection and refraction. We discuss shadows, reflection, and refraction—the effects for which ray tracing is best known—in Section 16.12, where we describe a full recursive ray-tracing algorithm that integrates both visible-surface determination and shading. Here, we treat ray tracing only as a visible-surface algorithm.

15.10.1 Computing Intersections

At the heart of any ray tracer is the task of determining the intersection of a ray with an object. To do this task, we use the same parametric representation of a vector introduced in Chapter 3. Each point (x, y, z) along the ray from (x_0, y_0, z_0) to (x_1, y_1, z_1) is defined by some value t such that

$$x = x_0 + t\,(x_1 - x_0), \qquad y = y_0 + t\,(y_1 - y_0), \qquad z = z_0 + t\,(z_1 - z_0). \tag{15.9}$$

For convenience, we define Δx, Δy, and Δz such that

$$\Delta x = x_1 - x_0, \qquad \Delta y = y_1 - y_0, \qquad \Delta z = z_1 - z_0. \tag{15.10}$$

Thus,

$$x = x_0 + t\,\Delta x, \qquad y = y_0 + t\,\Delta y, \qquad z = z_0 + t\,\Delta z. \tag{15.11}$$

If (x_0, y_0, z_0) is the center of projection and (x_1, y_1, z_1) is the center of a pixel on the window, then t ranges from 0 to 1 between these points. Negative values of t represent points behind the center of projection, whereas values of t greater than 1 correspond to points on the side of the window farther from the center of projection. We need to find a representation for each kind of object that enables us to determine t at the object's intersection with the ray. One of the easiest objects for which to do this is the sphere, which accounts for the plethora of spheres observed in typical ray-traced images! The sphere with center (a, b, c) and radius r may be represented by the equation

$$(x - a)^2 + (y - b)^2 + (z - c)^2 = r^2. \tag{15.12}$$

The intersection is found by expanding Eq. (15.12), and substituting the values of x, y, and z from Eq. (15.11) to yield

$$x^2 - 2ax + a^2 + y^2 - 2by + b^2 + z^2 - 2cz + c^2 = r^2, \tag{15.13}$$

$$\begin{aligned}(x_0 + t\Delta x)^2 - 2a(x_0 + t\Delta x) + a^2 + (y_0 + t\Delta y)^2 - 2b(y_0 + t\Delta y) + b^2 \\ + (z_0 + t\Delta z)^2 - 2c(z_0 + t\Delta z) + c^2 = r^2,\end{aligned} \tag{15.14}$$

$$\begin{aligned}x_0^2 + 2x_0\Delta xt + \Delta x^2t^2 - 2ax_0 - 2a\Delta xt + a^2 \\ + y_0^2 + 2y_0\,\Delta yt + \Delta y^2t^2 - 2by_0 - 2b\Delta yt + b^2 \\ + z_0^2 + 2z_0\,\Delta zt + \Delta z^2t^2 - 2cz_0 - 2c\Delta zt + c^2 = r^2.\end{aligned} \tag{15.15}$$

Collecting terms gives

$$\begin{aligned}(\Delta x^2 + \Delta y^2 + \Delta z^2)t^2 + 2t[\Delta x(x_0 - a) + \Delta y(y_0 - b) + \Delta z(z_0 - c)] \\ + (x_0^2 - 2ax_0 + a^2 + y_0^2 - 2by_0 + b^2 + z_0^2 - 2cz_0 + c^2) - r^2 = 0,\end{aligned} \tag{15.16}$$

$$\begin{aligned}(\Delta x^2 + \Delta y^2 + \Delta z^2)t^2 + 2t[\Delta x\,(x_0 - a) + \Delta y(y_0 - b) + \Delta z(z_0 - c)] \\ + (x_0 - a)^2 + (y_0 - b)^2 + (z_0 - c)^2 - r^2 = 0.\end{aligned} \tag{15.17}$$

Equation (15.17) is a quadratic in t, with coefficients expressed entirely in constants derived from the sphere and ray equations, so it can be solved using the quadratic formula. If there are no real roots, then the ray and sphere do not intersect; if there is one real root, then the ray grazes the sphere. Otherwise, the two roots are the points of intersection with the sphere; the one that yields the smallest positive t is the closest. It is also useful to normalize the ray so that the distance from (x_0, y_0, z_0) to (x_1, y_1, z_1) is 1. This gives a value of t that measures distance in WC units, and simplifies the intersection calculation, since the coefficient of t^2 in Eq. (15.17) becomes 1. We can obtain the intersection of a ray with the general quadric surfaces introduced in Chapter 11 in a similar fashion.

As we shall see in Chapter 16, we must determine the surface normal at the point of intersection in order to shade the surface. This is particularly easy in the case of the sphere, since the (unnormalized) normal is the vector from the center to the point of intersection: The sphere with center (a, b, c) has a surface normal $((x - a)/r, (y - b)/r, (z - c)/r)$ at the point of intersection (x, y, z).

Finding the intersection of a ray with a polygon is somewhat more difficult. We can determine where a ray intersects a polygon by first determining whether the ray intersects the polygon's plane and then whether the point of intersection lies within the polygon. Since the equation of a plane is

$$Ax + By + Cz + D = 0, \tag{15.18}$$

substitution from Eq. (15.11) yields

$$A(x_0 + t\Delta x) + B(y_0 + t\Delta y) + C(z_0 + t\Delta z) + D = 0, \tag{15.19}$$

$$t(A\Delta x + B\Delta y + C\Delta z) + (Ax_0 + By_0 + Cz_0 + D) = 0, \tag{15.20}$$

$$t = -\frac{(Ax_0 + By_0 + Cz_0 + D)}{(A\Delta x + B\Delta y + C\Delta z)}. \tag{15.21}$$

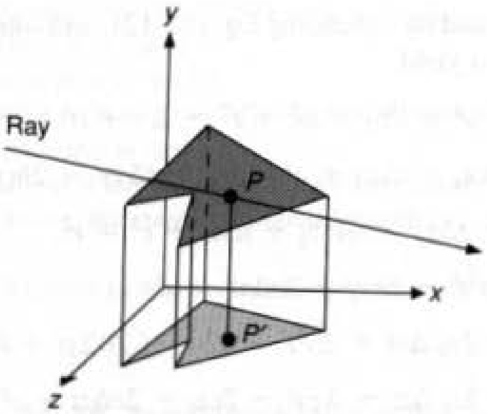


Fig. 15.57 Determining whether a ray intersects a polygon. The polygon and the ray's point of intersection p with the polygon's plane are projected onto one of the three planes defining the coordinate system. Projected point p' is tested for containment within the projected polygon.

If the denominator of Eq. (15.21) is 0, then the ray and plane are parallel and do not intersect. An easy way to determine whether the point of intersection lies within the polygon is to project the polygon and point orthographically onto one of the three planes defining the coordinate system, as shown in Fig. 15.57. To obtain the most accurate results, we should select the axis along which to project that yields the largest projection. This corresponds to the coordinate whose coefficient in the polygon's plane equation has the largest absolute value. The orthographic projection is accomplished by dropping this coordinate from the polygon's vertices and from the point. The polygon-containment test for the point can then be performed entirely in 2D, using the point-in-polygon algorithm of Section 7.12.2.

Like the z-buffer algorithm, ray tracing has the attraction that the only intersection operation performed is that of a projector with an object. There is no need to determine the intersection of two objects in the scene directly. The z-buffer algorithm approximates an object as a set of z values along the projectors that intersect the object. Ray tracing approximates objects as the set of intersections along each projector that intersects the scene. We can extend a z-buffer algorithm to handle a new kind of object by writing a scan-conversion and z-calculation routine for it. Similarly, we can extend a visible-surface ray tracer to handle a new kind of object by writing a ray-intersection routine for it. In both cases, we must also write a routine to calculate surface normals for shading. Intersection and surface-normal algorithms have been developed for algebraic surfaces [HANR83], for parametric surfaces [KAJI82; SEDE84; TOTH85; JOY86], and for many of the objects discussed in Chapter 20. Surveys of these algorithms are provided in [HAIN89; HANR89].

15.10.2 Efficiency Considerations for Visible-Surface Ray Tracing

At each pixel, the z-buffer algorithm computes information only for those objects that project to that pixel, taking advantage of coherence. In contrast, the simple but expensive version of the visible-surface ray tracing algorithm that we have discussed, intersects each of the rays from the eye with each of the objects in the scene. A 1024 by 1024 image of 100 objects would therefore require 100M intersection calculations. It is not surprising that Whitted found that 75 to over 95 percent of his system's time was spent in the intersection routine for typical scenes [WHIT80]. Consequently, the approaches to improving the efficiency of visible-surface ray tracing we discuss here attempt to speed up individual intersection calculations, or to avoid them entirely. As we shall see in Section 16.12, recursive ray tracers trace additional rays from the points of intersection to determine a pixel's shade. Therefore, several of the techniques developed in Section 15.2, such as the perspective transformation and back-face culling, are not in general useful, since all rays do not emanate from the same center of projection. In Section 16.12, we shall augment the techniques mentioned here with ones designed specifically to handle these recursive rays.

Optimizing intersection calculations. Many of the terms in the equations for object–ray intersection contain expressions that are constant either throughout an image or for a particular ray. These can be computed in advance, as can, for example, the orthographic projection of a polygon onto a plane. With care and mathematical insight, fast intersection methods can be developed; even the simple intersection formula for a sphere given in Section 15.10.1 can be improved [HAIN89]. If rays are transformed to lie along the z axis, then the same transformation can be applied to each candidate object, so that any intersection occurs at $x = y = 0$. This simplifies the intersection calculation and allows the closest object to be determined by a z sort. The intersection point can then be transformed back for use in shading calculations via the inverse transformation.

Bounding volumes provide a particularly attractive way to decrease the amount of time spent on intersection calculations. An object that is relatively expensive to test for intersection may be enclosed in a bounding volume whose intersection test is less expensive, such as a sphere [WHIT80], ellipsoid [BOUV85], or rectangular solid [RUBI80; TOTH85]. The object does not need to be tested if the ray fails to intersect with its bounding volume.

Kay and Kajiya [KAY86] suggest the use of a bounding volume that is a convex polyhedron formed by the intersection of a set of infinite *slabs*, each of which is defined by a pair of parallel planes that bound the object. Figure 15.58(a) shows in 2D an object bounded by four slabs (defined by pairs of parallel lines), and by their intersection. Thus, each slab is represented by Eq. (15.18), where A, B, and C are constant, and D is either $D_{\min}$ or $D_{\max}$. If the same set of parameterized slabs is used to bound all objects, each bound can be described compactly by the $D_{\min}$ and $D_{\max}$ of each of its slabs. A ray is intersected with a bound by considering one slab at a time. The intersection of a ray with a slab can be computed using Eq. (15.21) for each of the slab's planes, producing near and far values of t. Using the same set of parameterized slabs for all bounds, however, allows us to simplify Eq.(15.21), yielding

$$t = (S + D)T, \tag{15.22}$$

where $S = Ax_0 + By_0 + Cz_0$ and $T = -1/(A\Delta x + B\Delta y + C\Delta z)$. Both S and T can be calculated once for a given ray and parameterized slab.

Since each bound is an intersection of slabs, the intersection of the ray with an entire bound is just the intersection of the ray's intersections with each of the bound's slabs. This can be computed by taking the maximum of the near values of t and the minimum of the far values of t. In order to detect null intersections quickly, the maximum near and minimum

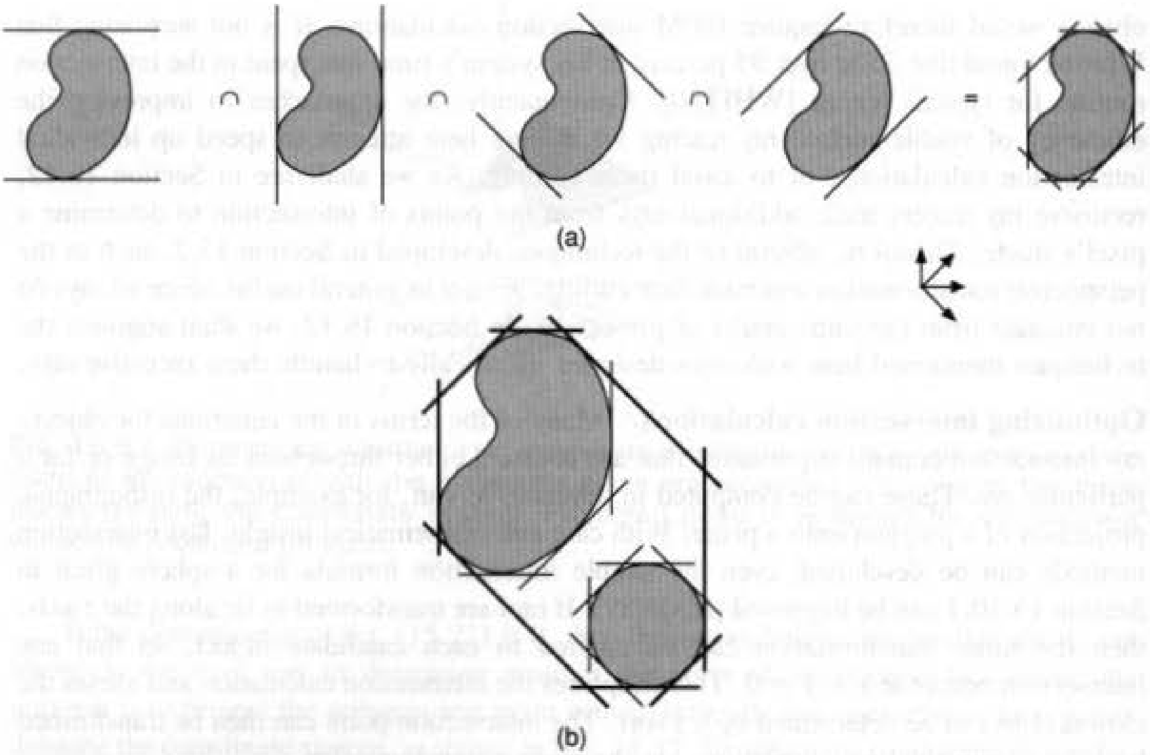


Fig. 15.58 Bounds formed intersection of slabs. (a) Object bounded by a fixed set of parameterized slabs. (b) The bounding volume of two bounding volumes.

far values of *t* for a bound can be updated as each of its slabs is processed, and the processing of the bound terminated if the former ever exceeds the latter.

Avoiding intersection calculations. Ideally, each ray should be tested for intersection only with objects that it actually intersects. Furthermore, in many cases we would like each ray to be tested against only that object whose intersection with the ray is closest to the ray's origin. There is a variety of techniques that attempt to approximate this goal by preprocessing the environment to partition rays and objects into equivalence classes to help limit the number of intersections that need to be performed. These techniques include two complementary approaches introduced in Section 15.2: hierarchies and spatial partitioning.

Hierarchies. Although bounding volumes do not by themselves determine the order or frequency of intersection tests, bounding volumes may be organized in nested hierarchies with objects at the leaves and internal nodes that bound their children [RUBI80; WEGH84; KAY86]. For example, a bounding volume for a set of Kay–Kajiya bounding volumes can be computed by taking for each pair of planes the minimum $D_{\min}$ and the maximum $D_{\max}$ of the values for each child volume, as shown in Fig. 15.58(b).

A child volume is guaranteed not to intersect with a ray if its parent does not. Thus, if intersection tests begin with the root, many branches of the hierarchy (and hence many objects) may be trivially rejected. A simple method to traverse the hierarchy is

```
void HIER_traverse (ray r, node n)
{
    if (r intersects n's bounding volume)
        if (n is a leaf)
            intersect r with n's object;
        else
            for (each child c of n)
                HIER_traverse (r, c);
}   /* HIER_traverse */
```

Efficient hierarchy traversal. HIER_traverse explores a hierarchy depth first. In contrast, Kay and Kajiya [KAY86] have developed an efficient method for traversing hierarchies of bounding volumes that takes into account the goal of finding the closest intersection. Note that the intersection of a ray with a Kay–Kajiya bound yields two values of t, the lower of which is a good estimate of the distance to the object. Therefore, the best order in which to select objects for intersection tests is that of increasing estimated distance from the ray's origin. To find the closest object intersected by a ray, we maintain the list of nodes to be tested in a priority queue, implemented as a heap. Initially, the heap is empty. If the root's bound is intersected by the ray, then the root is inserted in the heap. As long as the heap is not empty and its top node's estimated distance is closer than the closest object tested so far, nodes are extracted from the heap. If the node is a leaf, then its object's ray intersection is calculated. Otherwise, it is a bound, in which case each of its children's bounds is tested and is inserted in the heap if it is intersected, keyed by the estimated distance computed in the bound-intersection calculation. The selection process terminates when the heap is empty or when an object has been intersected that is closer than the estimated distance of any node remaining in the heap. Pseudocode for the algorithm is shown in Fig. 15.59.

Automated hierarchy generation. One problem with hierarchies of bounding volumes, such as those used by the Kay–Kajiya algorithm, is that generating good hierarchies is difficult. Hierarchies created during the modeling process tend to be fairly shallow, with structure designed for controlling objects rather than for minimizing the intersections of objects with rays. In addition, modeler hierarchies are typically insensitive to the actual position of objects. For example, the fingers on two robot hands remain in widely separated parts of the hierarchy, even when the hands are touching. Goldsmith and Salmon [GOLD87] have developed a method for generating good hierarchies for ray tracing automatically. Their method relies on a way of determining the quality of a hierarchy by estimating the cost of intersecting a ray with it.

Consider how we might estimate the cost of an individual bounding volume. Assume that each bounding volume has the same cost for computing whether a ray intersects it. Therefore, the cost is directly proportional to the number of times a bounding volume will be hit. The probability that a bounding volume is hit by an eye ray is the percentage of rays from the eye that will hit it. This is proportional to the bounding volume's area projected on the view plane. On the average, for convex bounding volumes, this value is roughly

```
void KayKajiya (void)
{
    object *p = NULL;          /* Pointer to nearest object hit */
    double t = ∞;              /* Distance to nearest object hit */

    precompute ray intersection;
    if (ray hits root's bound) {
        insert root into heap;
    }
    while (heap is not empty and distance to top node < t) {
        node *c = top node removed from heap;

        if (c is a leaf) {
            intersect ray with c's object;
            if (ray hits it and distance < t) {
                t = distance;
                p = object;
            }
        } else {        /* c is a bound */
            for (each child of c) {
                intersect ray with child's bound;
                if (ray hits child's bound)
                    insert child into heap;
            }
        }
    }
}   /* KayKajiya */
```

Fig. 15.59 Pseudocode for using Kay–Kajiya bounds to find the closest object intersected by a ray.

proportional to the bounding volume's surface area. Since each bounding volume is contained within the root's bounding volume, the conditional probability that a ray will intersect the *i*th bounding volume if it intersects the root can be approximated by A_i / A_r, where A_i is the surface area of the *i*th bounding volume and A_r is the surface area of the root.

If a ray intersects a bounding volume, we assume that we must perform an intersection calculation for each of the bounding volume's k children. Thus, the bounding volume's total estimated cost in number of intersections is kA_i / A_r. The root's estimated cost is just its number of children (since $A_i / A_r = 1$), and the cost of a leaf node is zero (since $k = 0$). To compute the estimated cost of a hierarchy, we sum the estimated costs of each of its bounding volumes. Consider, for example, the hierarchy shown in Fig. 15.60 in which each node is marked with its surface area. Assuming the root A is hit at the cost of one intersection, the root's estimated cost is 4 (its number of children). Two of its children (C

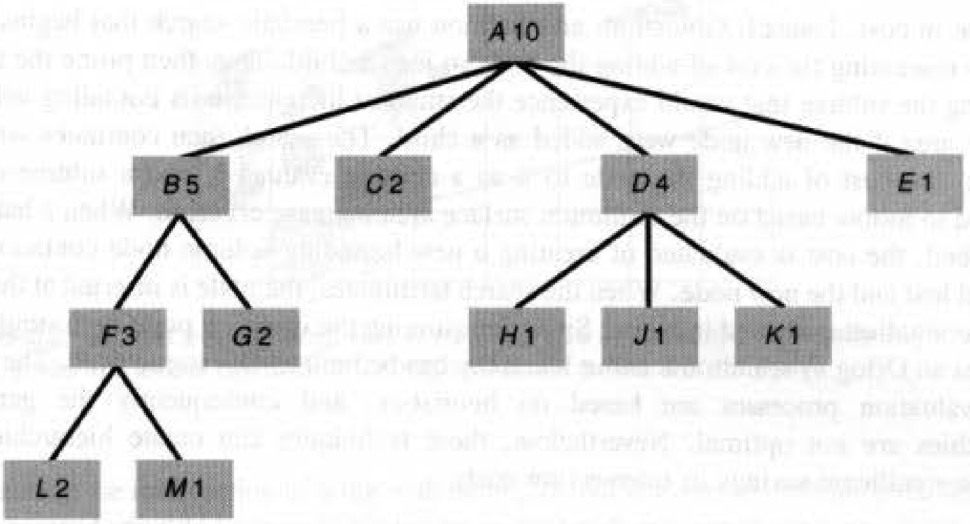


Fig. 15.60 Estimating the cost of a hierarchy. Letter is node name; number is node surface area.

and *E*) are leaves, and therefore have zero cost. *B* has two children and a surface area of 5. Thus, its estimated cost is 2(5/10) = 1.0. *D* has three children and a surface area of 4, so its estimated cost is 3(4/10) = 1.2. The only other nonleaf node is *F*, which has two children and a surface area of 3, giving an estimated cost of 2(3/10) = .6. The total estimated cost is 1 (to hit the root) + 4 + 1.0 + 1.2 + .6 = 7.8 expected intersections.

Since we are interested in only relative values, there is no need to divide by the root's surface area. Furthermore, we do not need the actual surface area of the bounding volume—we need only a value proportional to it. For example, rather than use $2lw + 2lh + 2wh$ for a rectangular prism, we can factor out the 2 and rearrange terms to yield $(w + h)l + wh$.

Goldsmith and Salmon create the hierarchy incrementally, adding one new node at a time. The order in which nodes are added affects the algorithm. The modeler's order can be used, but for many scenes better results can be obtained by randomizing the order by shuffling nodes. Each node may be added by making it a child of an existing node or by replacing an existing node with a new bounding volume node that contains both the original node and the new node. In each case, instead of evaluating the cost of the new tree from scratch, the incremental cost of adding the node can be determined. If the node is being added as a child of an existing node, it may increase the parent's surface area, and it also increases the parent's number of children by 1. Thus, the difference in estimated cost of the parent is $k\,(A_{new} - A_{old}) + A_{new}$, where A_{new} and A_{old} are the parent's new and old surface areas, and k is the original number of children. If the node is added by creating a new parent with both the original and new nodes as children, the incremental cost of the newly created parent is $2A_{new}$. In both cases, the incremental cost to the new child's grandparent and older ancestors must also be computed as $k(A_{new} - A_{old})$, where k, A_{new}, and A_{old} are the values for the ancestor node. This approach assumes that the position at which the node is placed has no effect on the size of the root bounding volume.

A brute-force approach would be to evaluate the increased cost of adding the new node at every possible position in the tree and to then pick the position that incurred the least increase in cost. Instead, Goldsmith and Salmon use a heuristic search that begins at the root by evaluating the cost of adding the node to it as a child. They then prune the tree by selecting the subtree that would experience the smallest increase in its bounding volume's surface area if the new node were added as a child. The search then continues with this subtree, the cost of adding the node to it as a child is evaluated, and a subtree of it is selected to follow based on the minimum surface area increase criterion. When a leaf node is reached, the cost is evaluated of creating a new bounding volume node containing the original leaf and the new node. When the search terminates, the node is inserted at the point with the smallest evaluated increase. Since determining the insertion point for a single node requires an $O(\log n)$ search, the entire hierarchy can be built in $O(n \log n)$ time. The search and evaluation processes are based on heuristics, and consequently the generated hierarchies are not optimal. Nevertheless, these techniques can create hierarchies that provide significant savings in intersection costs.

Spatial partitioning. Bounding-volume hierarchies organize objects bottom-up; in contrast, spatial partitioning subdivides space top-down. The bounding box of the scene is calculated first. In one approach, the bounding box is then divided into a regular grid of equal-sized extents, as shown in Fig. 15.61. Each partition is associated with a list of objects it contains either wholly or in part. The lists are filled by assigning each object to the one or more partitions that contain it. Now, as shown in 2D in Fig. 15.62, a ray needs to be intersected with only those objects that are contained within the partitions through which it passes. In addition, the partitions can be examined in the order in which the ray passes through them; thus, as soon as a partition is found in which there is an intersection, no more partitions need to be inspected. Note that we must consider all the remaining objects in the partition, to determine the one whose intersection is closest. Since the partitions follow a regular grid, each successive partition lying along a ray may be calculated using a 3D version of the line-drawing algorithm discussed in Section 3.2.2, modified to list every partition through which the ray passes [FUJI85; AMAN87].

If a ray intersects an object in a partition, it is also necessary to check whether the intersection itself lies in the partition; it is possible that the intersection that was found may be further along the ray in another partition and that another object may have a closer intersection. For example, in Fig. 15.63, object B is intersected in partition 3 although it is encountered in partition 2. We must continue traversing the partitions until an intersection is found in the partition currently being traversed, in this case with A in partition 3. To avoid

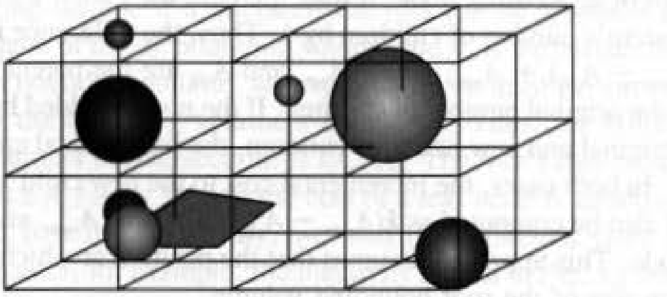


Fig. 15.61 The scene is partitioned into a regular grid of equal-sized volumes.

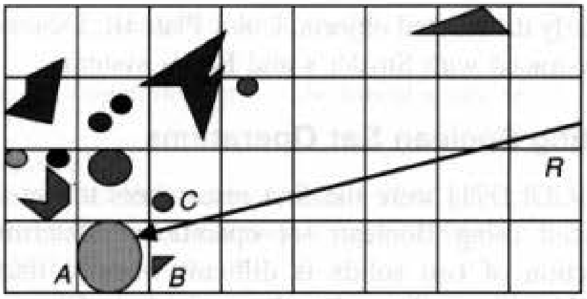


Fig. 15.62 Spatial partitioning. Ray *R* needs to be intersected with only objects *A*, *B*, and *C*, since the other partitions through which it passes are empty.

recalculating the intersection of a ray with an object that is found in multiple partitions, the point of intersection and the ray's ID can be cached with the object when the object is first encountered.

Dippé and Swensen [DIPP84] discuss an adaptive subdivision algorithm that produces unequal-sized partitions. An alternative adaptive spatial-subdivision method divides the scene using an octree [GLAS84]. In this case, the octree neighbor-finding algorithm sketched in Section 12.6.3 may be used to determine the successive partitions lying along a ray [SAME89b]. Octrees, and other hierarchical spatial partitionings, can be thought of as a special case of hierarchy in which a node's children are guaranteed not to intersect each other. Because these approaches allow adaptive subdivision, the decision to subdivide a partition further can be sensitive to the number of objects in the subdivision or the cost of intersecting the objects. This is advantageous in heterogeneous, unevenly distributed environments.

Spatial partitioning and hierarchy can be used together to combine their advantages. Snyder and Barr [SNYD87] describe an approach that uses hand-assembled hierarchies whose internal nodes are either lists or regular 3D grids. This allows the person designing an environment to choose lists for small numbers of sparsely arranged objects and grids for

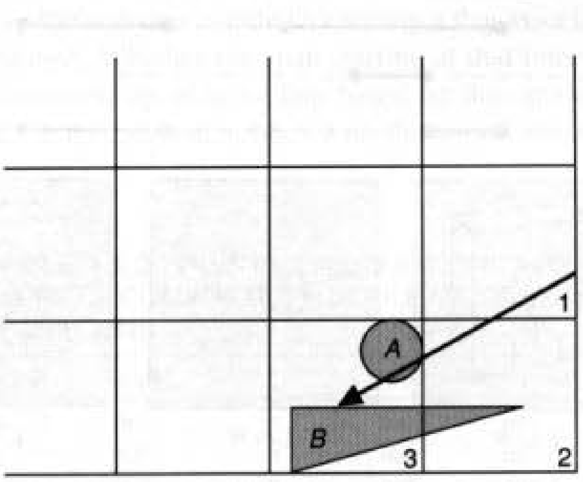


Fig. 15.63 An object may be intersected in a different voxel than the current one.

large numbers of regularly distributed objects. Color Plate III.1 shows a scene with 2 × 10^9 primitives that was ray-traced with Snyder's and Barr's system.

15.10.3 Computing Boolean Set Operations

Goldstein and Nagel [GOLD71] were the first researchers to ray trace combinations of simple objects produced using Boolean set operations. Determining the 3D union, difference, or intersection of two solids is difficult when it must be done by direct comparison of one solid with another using the methods in Chapter 12. In contrast, ray tracing allows the 3D problem to be reduced to a set of simple 1D calculations. The intersections of each ray and primitive object yield a set of t values, each of which specifies a point at which the ray enters or exits the object. Each t value thus defines the beginning of a span in which the ray is either in or out of the object. (Of course, care must be taken if the ray grazes the object, intersecting it only once.) Boolean set operations are calculated one ray at a time by determining the 1D union, difference, or intersection of spans from the two objects along the same ray. Figure 15.64 shows the spans defined by a ray passing through two objects, and the combinations of the spans that result when the set operations are

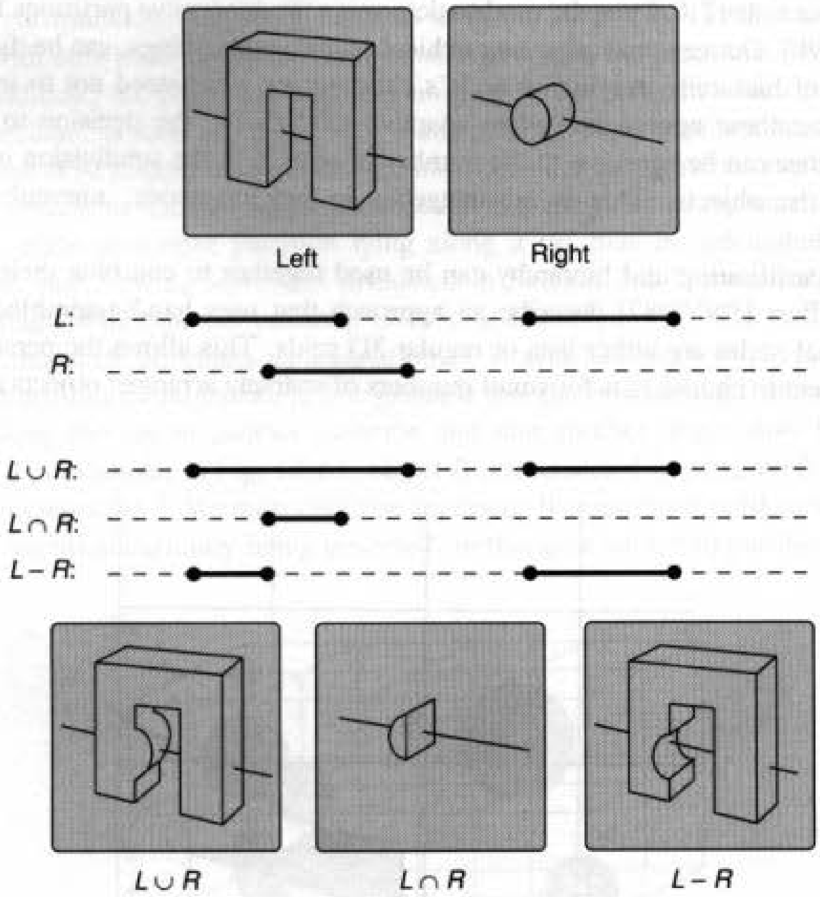


Fig. 15.64 Combining ray–object intersection span lists. (Adapted from [ROTH82] with permission.)

```
span *CSG_intersect (Ray *ray, CSG_node *node)
{
    span *leftIntersect, *rightIntersect;       /* lists of spans */

    if (node is composite) {
        leftIntersect = CSG_intersect (ray, node->leftChild);
        if (leftIntersect == NULL && node->op != UNION)
            return NULL;
        else {
            rightIntersect = CSG_intersect (ray, node->rightChild);
            return CSG_combine (node->op, leftIntersect, rightIntersect);
        }
    } else      /* node is primitive */
        return intersections of object with ray;
} /* CSG_intersect */
```

Fig. 15.65 Pseudocode for evaluating the intersection of a ray with a CSG hierarchy.

performed. The CSG hierarchy is traversed for each ray by evaluating the left and right intersection lists at each node, as shown in the pseudocode of Fig. 15.65. Color Plate III.2 is a ray-traced bowl defined by a CSG hierarchy.

Roth points out that, if there is no intersection with the left side of the tree, then there is no reason to intersect with the right side of the tree if the operation is a difference or intersection [ROTH82]. Only if the operation is a union can the result be nonempty. In fact, if we need to determine only whether or not the compound object is intersected (rather than the actual set of intersections), then the right-hand side need not be evaluated if the left-hand side intersects and the operation is a union.

The CSG_combine function takes two lists of intersection records, each ordered by increasing *t*, and combines them according to the operation being performed. The lists are merged by removing the intersection record that has the next largest value of *t*. Whether the ray is "in" the left list or the right list is noted by setting a flag associated with the list from which the record is removed. Whether the span starting at that intersection point is in the combined object is determined by table lookup based on the operator and the two "in" flags, using Table 15.2. A record is then placed on the combined list only if it begins or

TABLE 15.2 POINT CLASSIFICATION FOR OBJECTS COMBINED BY BOOLEAN SET OPERATIONS

Left	Right	∪	∩	−
in	in	in	in	out
in	out	in	out	in
out	in	in	out	out
out	out	out	out	out

ends a span of the combined object, not if it is internal to one of the combined object's spans. If a ray can begin inside an object, the flags must be initialized correctly.

15.10.4 Antialiased Ray Tracing

The simple ray tracer described so far uses point sampling on a regular grid, and thus produces aliased images. Whitted [WHIT80] developed an adaptive method for firing more rays into those parts of the image that would otherwise produce the most severe aliasing. These additional samples are used to compute a better value for the pixel. His *adaptive supersampling* associates rays with the corners, rather than with the centers, of each pixel, as shown in Fig. 15.66(a) and (b). Thus, at first, only an extra row and an extra column of rays are needed for the image. After rays have been fired through all four corners of a pixel, the shades they determine are averaged; the average is then used for the pixel if the shades differ from it by only a small amount. If they differ by too much, then the pixel is subdivided further by firing rays through the midpoints of its sides and through its center, forming four subpixels (Fig. 15.66c). The rays at the four corners of each subpixel are then compared using the same criterion. Subdivision proceeds recursively until a predefined maximum subdivision depth is reached, as in the Warnock algorithm, or until the ray shades are determined to be sufficiently similar. The pixel's shade is the area-weighted average of its subpixels' shades. Adaptive supersampling thus provides an improved approximation to unweighted area sampling, without the overhead of a uniformly higher sampling rate.

Consider, for example, Fig. 15.66(a), which shows the rays fired through the corners of two adjacent pixels, with a maximum subdivision depth of two. If no further subdivision is needed for the pixel bordered by rays A, B, D, and E in part (b), then, representing a ray's shade by its name, the pixel's shade is $(A + B + D + E)/4$. The adjacent pixel requires further subdivision, so rays G, H, I, J, and K are traced, defining the vertices of four subpixels in part (c). Each subpixel is recursively inspected. In this case, only the lower-right subpixel is subdivided again by tracing rays L, M, N, O, and P, as shown in part (d). At this point, the maximum subdivision depth is reached. This pixel's shade is

$$\frac{1}{4}\left[\frac{B+G+H+I}{4}+\frac{1}{4}\left[\frac{G+L+M+N}{4}+\frac{L+C+N+O}{4}+\frac{M+N+I+P}{4}+\right.\right.$$
$$\left.\left.\frac{N+O+P+J}{4}\right]+\frac{H+I+E+K}{4}+\frac{I+J+K+F}{4}\right].$$

Aliasing problems can also arise when the rays through a pixel miss a small object. This produces visible effects if the objects are arranged in a regular pattern and some are not visible, or if a series of pictures of a moving object show that object popping in and out of view as it is alternately hit and missed by the nearest ray. Whitted avoids these effects by surrounding each object with a spherical bounding volume that is sufficiently large always to be intersected by at least one ray from the eye. Since the rays converge at the eye, the size of the bounding volume is a function of the distance from the eye. If a ray intersects the

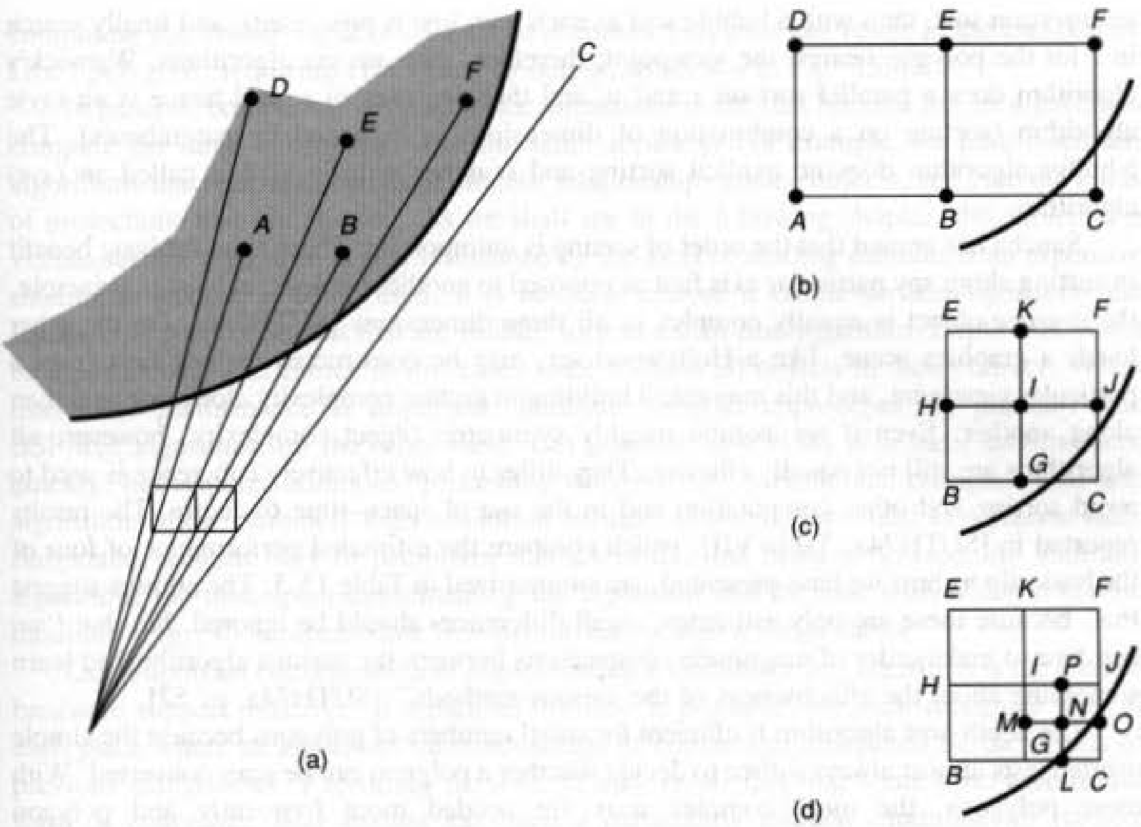


Fig. 15.66 Adaptive supersampling. (a) Two pixels and the rays fired through their corners. (b) The left pixel is not subdivided. (c) The right pixel is subdivided. (d) The lower-right subpixel is subdivided.

bounding volume but does not intersect the object, then all pixels sharing that ray are further subdivided until the object is intersected. Several more recent approaches to antialiased ray tracing are discussed in Section 16.12.

15.11 SUMMARY

Sutherland, Sproull, and Schumacker [SUTH74a] stress that the heart of visible-surface determination is sorting. Indeed, we have seen many instances of sorting and searching in the algorithms, and efficient sorting is vital to efficient visible-surface determination. Equally important is avoiding any more sorting than is absolutely necessary, a goal typically achieved by exploiting coherence. For example, the scan-line algorithms use scan-line coherence to eliminate the need for a complete sort on *x* for each scan line. Hubschman and Zucker use frame coherence to avoid unnecessary comparisons in animation sequences [HUBS82].

Algorithms can be classified by the order in which they sort. The depth-sort algorithm sorts on *z* and then on *x* and *y* (by use of extents in tests 1 and 2); it is thus called a *zxy* algorithm. Scan-line algorithms sort on *y* (with a bucket sort), then sort on *x* (initially with an insertion sort, then with a bubble sort as each scan line is processed), and finally search in *z* for the polygon nearest the viewpoint; therefore, they are *yxz* algorithms. Warnock's algorithm does a parallel sort on *x* and *y*, and then searches in *z*, and hence is an (*xy*)*z* algorithm (sorting on a combination of dimensions is indicated by parentheses). The *z*-buffer algorithm does no explicit sorting and searches only in *z*; it is called an (*xyz*) algorithm.

Sancha has argued that the order of sorting is unimportant: There is no intrinsic benefit in sorting along any particular axis first as opposed to another because, at least in principle, the *average* object is equally complex in all three dimensions [SUTH74a]. On the other hand, a graphics scene, like a Hollywood set, may be constructed to look best from a particular viewpoint, and this may entail building in greater complexity along one axis than along another. Even if we assume roughly symmetric object complexity, however, all algorithms are still not equally efficient: They differ in how effectively coherence is used to avoid sorting and other computation and in the use of space–time tradeoffs. The results reported in [SUTH74a, Table VII], which compare the estimated performance of four of the basic algorithms we have presented, are summarized in Table 15.3. The authors suggest that, because these are only estimates, small differences should be ignored, but that "we feel free to make order of magnitude comparisons between the various algorithms to learn something about the effectiveness of the various methods" [SUTH74a, p. 52].

The depth-sort algorithm is efficient for small numbers of polygons because the simple overlap tests almost always suffice to decide whether a polygon can be scan-converted. With more polygons, the more complex tests are needed more frequently and polygon subdivision is more likely to be required. The *z*-buffer algorithm has constant performance because, as the number of polygons in a scene increases, the number of pixels covered by a single polygon decreases. On the other hand, its memory needs are high. The individual tests and calculations involved in the Warnock area-subdivision algorithm are relatively complex, so it is generally slower than are the other methods.

In addition to these informal estimates, there has been some work on formalizing the visible-surface problem and analyzing its computational complexity [GILO78; FOUR88; FIUM89]. For example, Fiume [FIUM89] proves that object-precision visible-surface algorithms have a lower bound that is worse than that of sorting: Even the simple task of

TABLE 15.3 RELATIVE ESTIMATED PERFORMANCE OF FOUR ALGORITHMS FOR VISIBLE-SURFACE DETERMINATION

	Number of polygonal faces in scene		
Algorithm	100	2500	60,000
Depth sort	1*	10	507
z-buffer	54	54	54
Scan line	5	21	100
Warnock area subdivision	11	64	307

*Entries are normalized such that this case is unity.

computing the visible surfaces of a set of n convex polygons can result in the creation of $\Omega(n^2)$ polygons, requiring $\Omega(n^2)$ time to output, as shown in Fig. 15.67.

In general, comparing visible-surface algorithms is difficult because not all algorithms compute the same information with the same accuracy. For example, we have discussed algorithms that restrict the kinds of objects, relationships among objects, and even the kinds of projections that are allowed. As we shall see in the following chapter, the choice of a visible-surface algorithm is also influenced by the kind of shading desired. If an expensive shading procedure is being used, it is better to choose a visible-surface algorithm that shades only parts of objects that are visible, such as a scan-line algorithm. Depth sort would be a particularly bad choice in this case, since it draws all objects in their entirety. When interactive performance is important, hardware z-buffer approaches are popular. The BSP-tree algorithm, on the other hand, can generate new views of a static environment quickly, but requires additional processing whenever the environment changes. Scan-line algorithms allow extremely high resolution because data structures need to represent fully elaborated versions only of primitives that affect the line being processed. As with any algorithm, the time spent implementing the algorithm and the ease with which it can be modified (e.g., to accommodate new primitives) is also a major factor.

One important consideration in implementing a visible-surface algorithm is the kind of hardware support available. If a parallel machine is available, we must recognize that, at each place where an algorithm takes advantage of coherence, it depends on the results of previous calculations. Exploiting parallelism may entail ignoring some otherwise useful form of coherence. Ray tracing has been a particularly popular candidate for parallel implementation because, in its simplest form, each pixel is computed independently. As we shall see in Chapter 18, there are many architectures that have been designed to execute specific visible-surface algorithms. For example, plummeting memory costs have made hardware z-buffer systems ubiquitous.

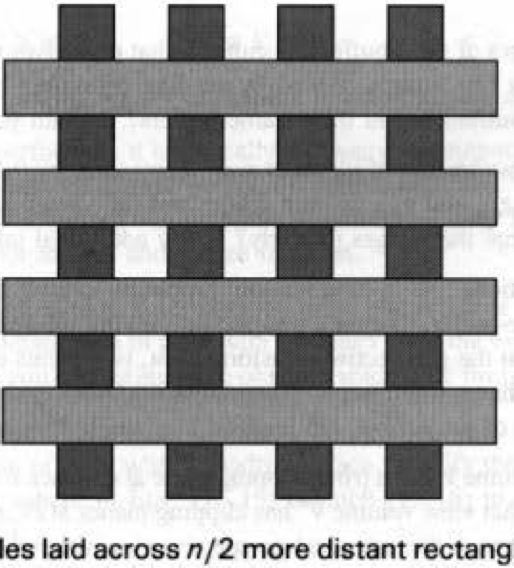


Fig. 15.67 $n/2$ rectangles laid across $n/2$ more distant rectangles can yield $n/2$ visible whole rectangles + $(n/2)$ $(n/2 + 1)$ visible fragments. This is $\Omega(n^2)$ visible polygons. (After [FIUM89].)

EXERCISES

15.1 Prove that the transformation M in Section 15.2.2 preserves (a) straight lines, (b) planes, and (c) depth relationships.

15.2 Given a plane $Ax + By + Cz + D = 0$, apply M from Section 15.2.2 and find the new coefficients of the plane equation.

15.3 How can a scan-line algorithm be extended to deal with polygons with shared edges? Should a shared edge be represented once, as a shared edge, or twice, once for each polygon it borders, with no record kept that it is a shared edge? When the depth of two polygons is evaluated at their common shared edge, the depths will, of course, be equal. Which polygon should be declared visible, given that the scan is entering both?

15.4 Warnock's algorithm generates a quadtree. Show the quadtree corresponding to Fig. 15.44. Label all nodes to indicate how the triangle (T) and the rectangle (R) relate to the node, as (a) surrounding, (b) intersecting, (c) contained, and (d) disjoint.

15.5 For each of the visible-surface algorithms discussed, explain how piercing polygons would be handled. Are they a special case that must be treated explicitly, or are they accommodated by the basic algorithm?

15.6 Consider tests 3 and 4 of the depth-sort algorithm. How might they be implemented efficiently? Consider examining the sign of the equation of the plane of polygon P for each vertex of polygon Q, and vice versa. How do you know to which side of the plane a positive value of the equation corresponds?

15.7 How can the algorithms discussed be adapted to work with polygons containing holes?

15.8 Describe how the visible-line algorithms for functions of two variables, described in Section 15.1, can be modified to work as visible-surface algorithms using the approach taken in the painter's algorithm.

15.9 Why does the Roberts visible-line algorithm not eliminate *all* lines that are edges of a back-facing polygon?

15.10 One of the advantages of the z-buffer algorithm is that primitives may be presented to it in any order. Does this mean that two images created by sending primitives in different orders will have identical values in their z-buffers and in their frame buffers? Explain your answer.

15.11 Consider merging two images of identical size, represented by their frame-buffer and z-buffer contents. If you know the $z_{\min}$ and $z_{\max}$ of each image and the values of z to which they originally corresponded, can you merge the images properly? Is any additional information needed?

15.12 Section 15.4 mentions the z-compression problems caused by rendering a perspective projection using an integer z-buffer. Choose a perspective viewing specification and a small number of object points. Show how, in the perspective transformation, two points near the center of projection are mapped to different z values, whereas two points separated from each other by the same distance, but farther from the center of projection, are mapped to a single z value.

15.13 a. Suppose view volume V has a front clipping plane at distance F and a back clipping plane at distance B, and that view volume V' has clipping planes at F' and B'. After transformation of each view volume to the canonical-perspective view volume, the back clipping plane of V will be at $z = -1$, and the front clipping plane at $z = A$. For V', the front clipping plane will be at $z = A'$. Show that, if $B \,/\, F = B' \,/\, F'$, then $A = A'$.

b. Part (a) shows that, in considering the effect of perspective, we need to consider only the ratio of back-plane to front-plane distance. We can therefore simply study the canonical view volume with various values of the front-plane distance. Suppose, then, that we have a canonical-perspective view volume, with front clipping plane $z = A$ and back clipping plane $z = -1$, and we transform it, through the perspective transformation, to the parallel view volume between $z = 0$ and $z = -1$. Write down the formula for the transformed z coordinate in terms of the original z coordinate. (Your answer will depend on A, of course.) Suppose that the transformed z values in the parallel view volume are multiplied by 2^n, and then are rounded to integers (i.e., they are mapped to an integer z-buffer). Find two values of z that are as far apart as possible, but that map, under this transformation, to the same integer. (Your answer will depend on n and A.)

c. Suppose you want to make an image in which the backplane-to-frontplane ratio is R, and objects that are more than distance Q apart (in z) must map to different values in the z-buffer. Using your work in part (b), write a formula for the number of bits of z-buffer needed.

15.14 Show that the back-to-front display order determined by traversing a BSP tree is not necessarily the same as the back-to-front order determined by the depth-sort algorithm, even when no polygons are split. (Hint: Only two polygons are needed.)

15.15 How might you modify the BSP-tree algorithm to accept objects other than polygons?

15.16 How might you modify the BSP-tree algorithm to allow limited motion?

15.17 Suppose that you are designing a ray tracer that supports CSG. How would you handle a polygon that is not part of a polyhedron?

15.18 Some graphics systems implement hardware transformations and homogeneous-coordinate clipping in X, Y, and Z using the same mathematics, so that clipping limits are

$$-W \le X \le W, \qquad -W \le Y \le W, \qquad -W \le Z \le W,$$

instead of

$$-W \le X \le W, \qquad -W \le Y \le W, \qquad -W \le Z \le 0.$$

How would you change the viewing matrix calculation to take this into account?

15.19 When ray tracing is performed, it is typically necessary to compute only whether or not a ray intersects an extent, not what the actual points of intersection are. Complete the ray–sphere intersection equation (Eq. 15.17) using the quadratic formula, and show how it can be simplified to determine only whether or not the ray and sphere intersect.

15.20 Ray tracing can also be used to determine the mass properties of objects through numerical integration. The full set of intersections of a ray with an object gives the total portion of the ray that is inside the object. Show how you can estimate an object's volume by firing a regular array of parallel rays through that object.

15.21 Derive the intersection of a ray with a quadric surface. Modify the method used to derive the intersection of a ray with a sphere in Eqs. (15.13) through (15.16) to handle the definition of a quadric given in Section 11.4.

15.22 In Eq. (15.5), O, the cost of performing an object intersection test, may be partially underwritten by B, the cost of performing a bounding-volume intersection test, if the results of the bounding-volume intersection test can be reused to simplify the object intersection test. Describe an object and bounding volume for which this is possible.

15.23 Implement one of the polygon visible surface algorithms in this chapter, such as a *z*-buffer algorithm, scan-line algorithm, or BSP tree algorithm.

15.24 Implement a simple ray tracer for spheres and polygons, including adaptive supersampling. (Choose one of the illumination models from Section 16.1.) Improve your program's performance through the use of spatial partitioning or hierarchies of bounding volumes.

15.25 If you have implemented the *z*-buffer algorithm, then add hit detection to it by extending the pick-window approach described in Section 7.12.2 to take visible-surface determination into account. You will need a SetPickMode procedure that is passed a mode flag, indicating whether objects are to be drawn (drawing mode) or instead tested for hits (pick mode). A SetPick Window procedure will let the user set a rectangular pick window. The *z*-buffer must already have been filled (by drawing all objects) for pick mode to work. When in pick mode, neither the frame-buffer nor the *z*-buffer is updated, but the *z*-value of each of the primitive's pixels that falls inside the pick window is compared with the corresponding value in the *z*-buffer. If the new value would have caused the object to be drawn in drawing mode, then a flag is set. The flag can be inquired by calling InquirePick, which then resets the flag. If InquirePick is called after each primitive's routine is called in pick mode, picking can be done on a per-primitive basis. Show how you can use InquirePick to determine which object is actually visible at a pixel.

16 Illumination and Shading

In this chapter, we discuss how to shade surfaces based on the position, orientation, and characteristics of the surfaces and the light sources illuminating them. We develop a number of different *illumination models* that express the factors determining a surface's color at a given point. Illumination models are also frequently called *lighting models* or *shading models*. Here, however, we reserve the term *shading model* for the broader framework in which an illumination model fits. The shading model determines when the illumination model is applied and what arguments it will receive. For example, some shading models invoke an illumination model for every pixel in the image, whereas others invoke an illumination model for only some pixels, and shade the remaining pixels by interpolation.

When we compared the accuracy with which the visible-surface calculations of the previous chapter are performed, we distinguished between algorithms that use the actual object geometry and those that use polyhedral approximations, between object-precision and image-precision algorithms, and between image-precision algorithms that take one point sample per pixel and those that use better filters. In all cases, however, the single criterion for determining the direct visibility of an object at a pixel is whether something lies between the object and the observer along the projector through the pixel. In contrast, the interaction between lights and surfaces is a good deal more complex. Graphics researchers have often approximated the underlying rules of optics and thermal radiation, either to simplify computation or because more accurate models were not known in the graphics community. Consequently, many of the illumination and shading models traditionally used in computer graphics include a multitude of kludges, ''hacks,'' and simplifications that have no firm grounding in theory, but that work well in practice. The first part of this chapter covers these simple models, which are still in common use because they can produce attractive and useful results with minimal computation.

We begin, in Section 16.1, with a discussion of simple illumination models that take into account an individual point on a surface and the light sources directly illuminating it. We first develop illumination models for monochromatic surfaces and lights, and then show how the computations can be generalized to handle the color systems discussed in Chapter 13. Section 16.2 describes the most common shading models that are used with these illumination models. In Section 16.3, we expand these models to simulate textured surfaces.

Modeling refraction, reflection, and shadows requires additional computation that is very similar to, and often is integrated with, hidden-surface elimination. Indeed, these effects occur because some of the ''hidden surfaces'' are not really hidden at all—they are seen through, reflected from, or cast shadows on the surface being shaded! Sections 16.4 through 16.6 discuss how to model these effects. We next introduce, in Section 16.7, illumination models that more accurately characterize how an individual surface interacts with the light sources directly illuminating it. This is followed by coverage of additional ways to generate more realistic images, in Section 16.8 through 16.10.

Sections 16.11 through 16.13 describe *global illumination models* that attempt to take into account the interchange of light between all surfaces: recursive ray tracing and radiosity methods. Recursive ray tracing extends the visible-surface ray-tracing algorithm introduced in the previous chapter to interleave the determination of visibility, illumination, and shading at each pixel. Radiosity methods model the energy equilibrium in a system of surfaces; they determine the illumination of a set of sample points in the environment in a view-independent fashion before visible-surface determination is performed from the desired viewpoint. More detailed treatments of many of the illumination and shading models covered here may be found in [GLAS89; HALL89].

Finally, in Section 16.14, we look at several different graphics pipelines that integrate the rasterization techniques discussed in this and the previous chapters. We examine some ways to implement these capabilities to produce systems that are both efficient and extensible.

16.1 ILLUMINATION MODELS

16.1.1 Ambient Light

Perhaps the simplest illumination model possible is that used implicitly in this book's earliest chapters: Each object is displayed using an intensity intrinsic to it. We can think of this model, which has no external light source, as describing a rather unrealistic world of nonreflective, self-luminous objects. Each object appears as a monochromatic silhouette, unless its individual parts, such as the polygons of a polyhedron, are given different shades when the object is created. Color Plate II.28 demonstrates this effect.

An illumination model can be expressed by an *illumination equation* in variables associated with the point on the object being shaded. The illumination equation that expresses this simple model is

$$I = k_i, \tag{16.1}$$

where I is the resulting intensity and the coefficient k_i is the object's intrinsic intensity. Since this illumination equation contains no terms that depend on the position of the point being shaded, we can evaluate it once for each object. The process of evaluating the illumination equation at one or more points on an object is often referred to as *lighting* the object.

Now imagine, instead of self-luminosity, that there is a diffuse, nondirectional source of light, the product of multiple reflections of light from the many surfaces present in the environment. This is known as *ambient* light. If we assume that ambient light impinges equally on all surfaces from all directions, then our illumination equation becomes

$$I = I_a k_a. \qquad (16.2)$$

I_a is the intensity of the ambient light, assumed to be constant for all objects. The amount of ambient light reflected from an object's surface is determined by k_a, the *ambient-reflection coefficient*, which ranges from 0 to 1. The ambient-reflection coefficient is a *material property*. Along with the other material properties that we will discuss, it may be thought of as characterizing the material from which the surface is made. Like some of the other properties, the ambient-reflection coefficient is an empirical convenience and does not correspond directly to any physical property of real materials. Furthermore, ambient light by itself is not of much interest. As we see later, it is used to account for all the complex ways in which light can reach an object that are not otherwise addressed by the illumination equation. Color Plate II.28 also demonstrates illumination by ambient light.

16.1.2 Diffuse Reflection

Although objects illuminated by ambient light are more or less brightly lit in direct proportion to the ambient intensity, they are still uniformly illuminated across their surfaces. Now consider illuminating an object by a *point light source*, whose rays emanate uniformly in all directions from a single point. The object's brightness varies from one part to another, depending on the direction of and distance to the light source.

Lambertian reflection. Dull, matte surfaces, such as chalk, exhibit *diffuse reflection*, also known as *Lambertian reflection*. These surfaces appear equally bright from all viewing angles because they reflect light with equal intensity in all directions. For a given surface, the brightness depends only on the angle θ between the direction $\overline{L}$ to the light source and the surface normal $\overline{N}$ of Fig. 16.1. Let us examine why this occurs. There are two factors at work here. First, Fig. 16.2 shows that a beam that intercepts a surface covers an area whose size is inversely proportional to the cosine of the angle θ that the beam makes with $\overline{N}$. If the

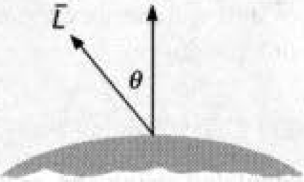


Fig. 16.1 Diffuse reflection.

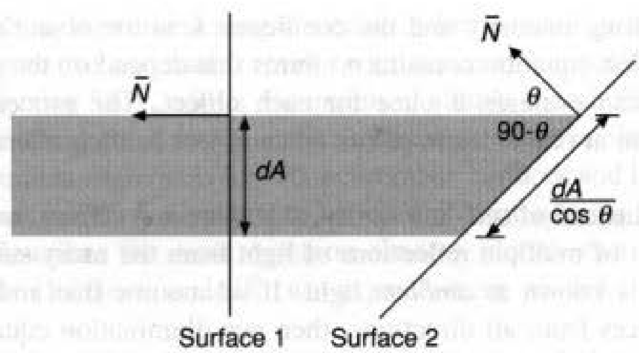


Fig. 16.2 Beam (shown in 2D cross-section) of infinitesimal cross-sectional area dA at angle of incidence θ intercepts area of dA / cos θ.

beam has an infinitesimally small cross-sectional differential area dA, then the beam intercepts an area dA / cos θ on the surface. Thus, for an incident light beam, the amount of light energy that falls on dA is proportional to cos θ. This is true for any surface, independent of its material.

Second, we must consider the amount of light seen by the viewer. Lambertian surfaces have the property, often known as Lambert's law, that the amount of light reflected from a unit differential area dA toward the viewer is directly proportional to the cosine of the angle between the direction to the viewer and $\overline{N}$. Since the amount of surface area seen is inversely proportional to the cosine of this angle, these two factors cancel out. For example, as the viewing angle increases, the viewer sees more surface area, but the amount of light reflected at that angle per unit area of surface is proportionally less. Thus, for Lambertian surfaces, the amount of light seen by the viewer is independent of the viewer's direction and is proportional only to cos θ, the angle of incidence of the light.

The diffuse illumination equation is

$$I = I_p k_d \cos\theta. \tag{16.3}$$

I_p is the point light source's intensity; the material's *diffuse-reflection coefficient* k_d is a constant between 0 and 1 and varies from one material to another. The angle θ must be between 0° and 90° if the light source is to have any direct effect on the point being shaded. This means that we are treating the surface as *self-occluding*, so that light cast from behind a point on the surface does not illuminate it. Rather than include a max(cos θ, 0) term explicitly here and in the following equations, we assume that θ lies within the legal range. When we want to light self-occluding surfaces, we can use abs(cos θ) to invert their surface normals. This causes both sides of the surface to be treated alike, as if the surface were lit by two opposing lights.

Assuming that the vectors $\overline{N}$ and $\overline{L}$ have been normalized (see Appendix), we can rewrite Eq. (16.3) by using the dot product:

$$I = I_p k_d \, (\overline{N} \cdot \overline{L}). \tag{16.4}$$

The surface normal $\overline{N}$ can be calculated using the methods discussed in Chapter 11. If polygon normals are precomputed and transformed with the same matrix used for the polygon vertices, it is important that nonrigid modeling transformations, such as shears or

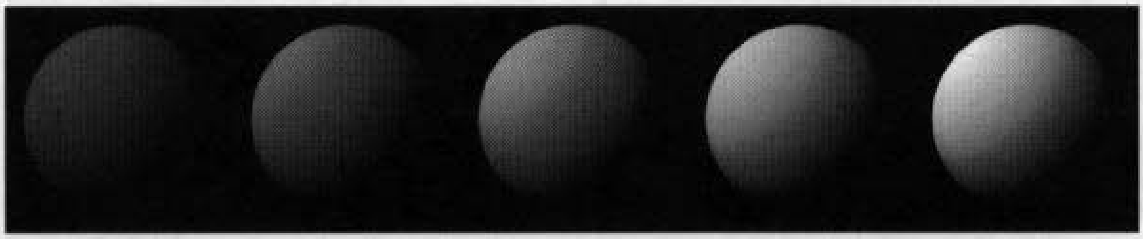


Fig. 16.3 Spheres shaded using a diffuse-reflection model (Eq. 16.4). For all spheres, $I_p = 1.0$. From left to right, $k_d = 0.4, 0.55, 0.7, 0.85, 1.0$. (By David Kurlander, Columbia University.)

differential scaling, not be performed; these transformations do not preserve angles and may cause some normals to be no longer perpendicular to their polygons. The proper method to transform normals when objects undergo arbitrary transformations is described in Section 5.6. In any case, the illumination equation must be evaluated in the WC system (or in any coordinate system isometric to it), since both the normalizing and perspective transformations will modify θ.

If a point light source is sufficiently distant from the objects being shaded, it makes essentially the same angle with all surfaces sharing the same surface normal. In this case, the light is called a *directional light source,* and $\overline{L}$ is a constant for the light source.

Figure 16.3 shows a series of pictures of a sphere illuminated by a single point source. The shading model calculated the intensity at each pixel at which the sphere was visible using the illumination model of Eq. (16.4). Objects illuminated in this way look harsh, as when a flashlight illuminates an object in an otherwise dark room. Therefore, an ambient term is commonly added to yield a more realistic illumination equation:

$$I = I_a k_a + I_p k_d (\overline{N} \cdot \overline{L}). \tag{16.5}$$

Equation (16.5) was used to produce Fig. 16.4.

Light-source attenuation. If the projections of two parallel surfaces of identical material, lit from the eye, overlap in an image, Eq. (16.5) will not distinguish where one surface leaves off and the other begins, no matter how different are their distances from the light source. To do this, we introduce a light-source attenuation factor, f_{att}, yielding

$$I = I_a k_a + f_{att} I_p k_d (\overline{N} \cdot \overline{L}). \tag{16.6}$$

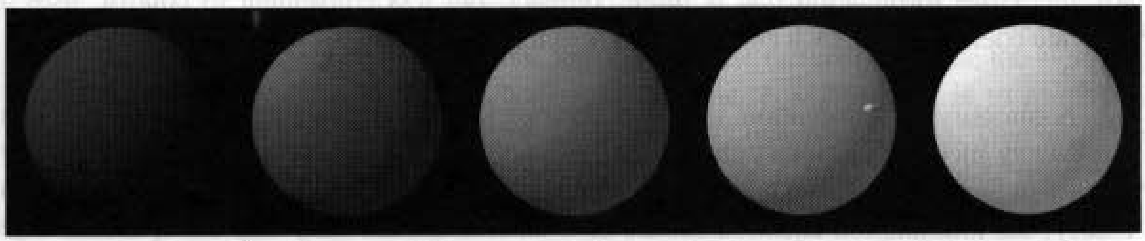


Fig. 16.4 Spheres shaded using ambient and diffuse reflection (Eq. 16.5). For all spheres, $I_a = I_p = 1.0$, $k_d = 0.4$. From left to right, $k_a = 0.0, 0.15, 0.30, 0.45, 0.60$. (By David Kurlander, Columbia University.)

An obvious choice for f_{att} takes into account the fact that the energy from a point light source that reaches a given part of a surface falls off as the inverse square of d_L, the distance the light travels from the point source to the surface. In this case,

$$f_{\text{att}} = \frac{1}{d_L^2}. \tag{16.7}$$

In practice, however, this often does not work well. If the light is far away, $1 / d_L^2$ does not vary much; if it is very close, it varies widely, giving considerably different shades to surfaces with the same angle θ between $\overline{N}$ and $\overline{L}$. Although this behavior is correct for a point light source, the objects we see in real life typically are not illuminated by point sources and are not shaded using the simplified illumination models of computer graphics. To complicate matters, early graphics researchers often used a single point light source positioned right at the viewpoint. They expected f_{att} to approximate some of the effects of atmospheric attenuation between the viewer and the object (see Section 16.1.3), as well as the energy density falloff from the light to the object. A useful compromise, which allows a richer range of effects than simple square-law attenuation, is

$$f_{\text{att}} = \min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1\right). \tag{16.8}$$

Here c_1, c_2, and c_3 are user-defined constants associated with the light source. The constant c_1 keeps the denominator from becoming too small when the light is close, and the expression is clamped to a maximum of 1 to ensure that it always attenuates. Figure 16.5 uses this illumination model with different constants to show a range of effects.

Colored lights and surfaces. So far, we have described monochromatic lights and surfaces. Colored lights and surfaces are commonly treated by writing separate equations for each component of the color model. We represent an object's *diffuse color* by one value of O_d for each component. For example, the triple (O_{dR}, O_{dG}, O_{dB}) defines an object's diffuse red, green, and blue components in the RGB color system. In this case, the illuminating light's three primary components, I_{pR}, I_{pG}, and I_{pB}, are reflected in proportion to $k_d O_{dR}$, $k_d O_{dG}$, and $k_d O_{dB}$, respectively. Therefore, for the red component,

$$I_R = I_{aR} k_a O_{dR} + f_{\text{att}} I_{pR} k_d O_{dR} \, (\overline{N} \cdot \overline{L}). \tag{16.9}$$

Similar equations are used for I_G and I_B, the green and blue components. The use of a single coefficient to scale an expression in each of the equations allows the user to control the amount of ambient or diffuse reflection, without altering the proportions of its components. An alternative formulation that is more compact, but less convenient to control, uses a separate coefficient for each component; for example, substituting k_{aR} for $k_a O_{dR}$ and k_{dR} for $k_d O_{dR}$.

A simplifying assumption is made here that a three-component color model can completely model the interaction of light with objects. This assumption is wrong, as we discuss in Section 16.9, but it is easy to implement and often yields acceptable pictures. In theory, the illumination equation should be evaluated continuously over the spectral range being modeled; in practice, it is evaluated for some number of discrete spectral samples.

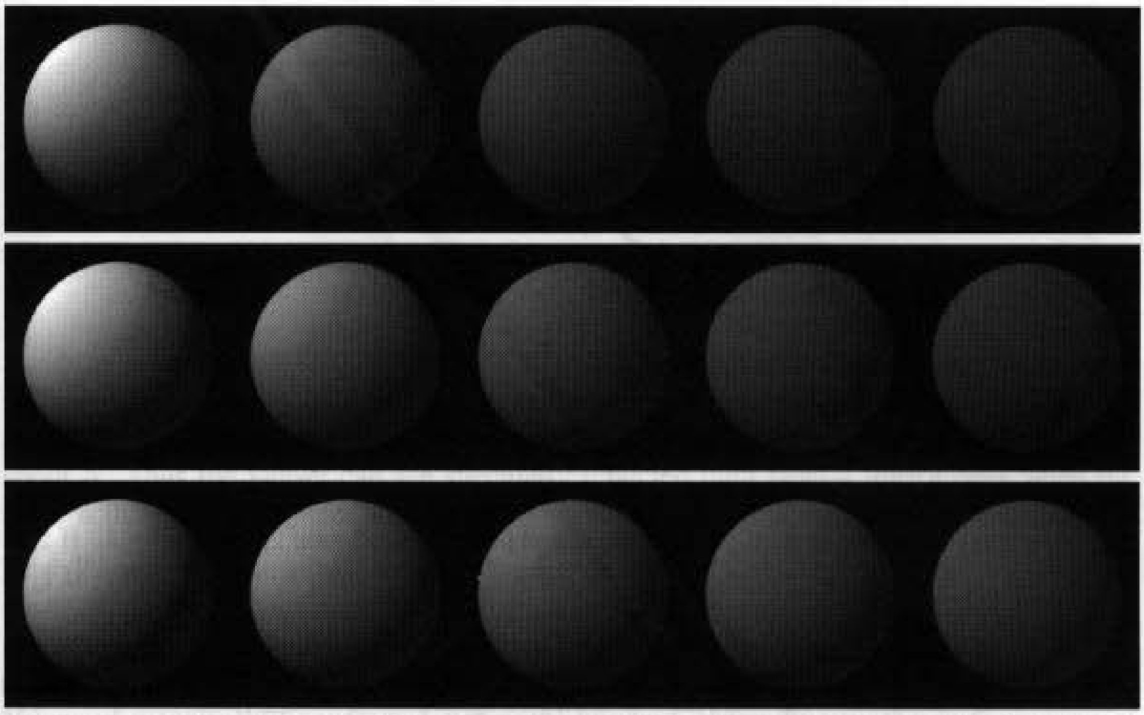


Fig. 16.5 Spheres shaded using ambient and diffuse reflection with a light-source-attenuation term (Eqs. 16.6 and 16.8). For all spheres, $I_a = I_p = 1.0$, $k_a = 0.1$, $k_d = 0.9$. From left to right, sphere's distance from light source is 1.0, 1.375, 1.75, 2.125, 2.5. Top row: $c_1 = c_2 = 0.0$, $c_3 = 1.0$ ($1/d_L^2$). Middle row: $c_1 = c_2 = 0.25$, $c_3 = 0.5$. Bottom row: $c_1 = 0.0$, $c_2 = 1.0$, $c_3 = 0.0$ ($1/d_L$). (By David Kurlander, Columbia University.)

Rather than restrict ourselves to a particular color model, we explicitly indicate those terms in an illumination equation that are wavelength-dependent by subscripting them with a λ. Thus, Eq. (16.9) becomes

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} k_d O_{d\lambda} (\bar{N} \cdot \bar{L}). \qquad (16.10)$$

16.1.3 Atmospheric Attenuation

To simulate the atmospheric attenuation from the object to the viewer, many systems provide *depth cueing*. In this technique, which originated with vector-graphics hardware, more distant objects are rendered with lower intensity than are closer ones. The PHIGS+ standard recommends a depth-cueing approach that also makes it possible to approximate the shift in colors caused by the intervening atmosphere. Front and back depth-cue reference planes are defined in NPC; each of these planes is associated with a scale factor, s_f and s_b, respectively, that ranges between 0 and 1. The scale factors determine the blending of the original intensity with that of a depth-cue color, $I_{dc\lambda}$. The goal is to modify a previously computed I_λ to yield the depth-cued value I'_λ that is displayed. Given z_o, the object's z coordinate, a scale factor s_o is derived that will be used to interpolate between I_λ and $I_{dc\lambda}$, to determine

$$I'_\lambda = s_o I_\lambda + (1 - s_o) I_{dc\lambda}. \qquad (16.11)$$

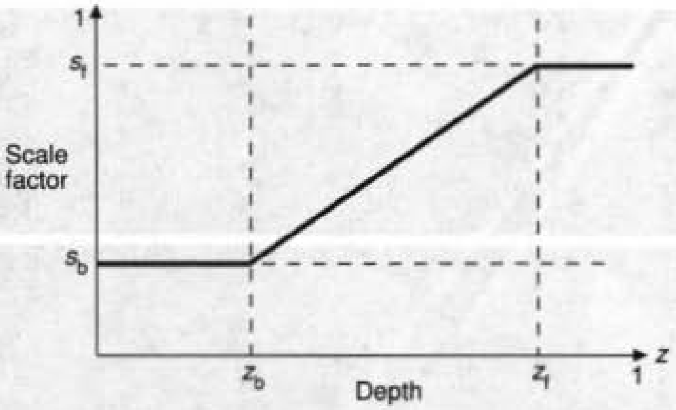


Fig. 16.6 Computing the scale factor for atmospheric attenuation.

If z_o is in front of the front depth-cue reference plane's z coordinate z_f, then $s_o = s_f$. If z_o is behind the back depth-cue reference plane's z coordinate z_b, then $s_o = s_b$. Finally, if z_o is between the two planes, then

$$s_o = s_b + \frac{(z_o - z_b)(s_f - s_b)}{z_f - z_b}. \tag{16.12}$$

The relationship between s_o and z_o is shown in Fig. 16.6. Figure 16.7 shows spheres shaded with depth cueing. To avoid complicating the equations, we ignore depth cueing as we develop the illumination model further. More realistic ways to model atmospheric effects are discussed in Section 20.8.2.

16.1.4 Specular Reflection

Specular reflection can be observed on any shiny surface. Illuminate an apple with a bright white light: The highlight is caused by specular reflection, whereas the light reflected from the rest of the apple is the result of diffuse reflection. Also note that, at the highlight, the apple appears to be not red, but white, the color of the incident light. Objects such as waxed apples or shiny plastics have a transparent surface; plastics, for example, are typically composed of pigment particles embedded in a transparent material. Light specularly reflected from the colorless surface has much the same color as that of the light source.

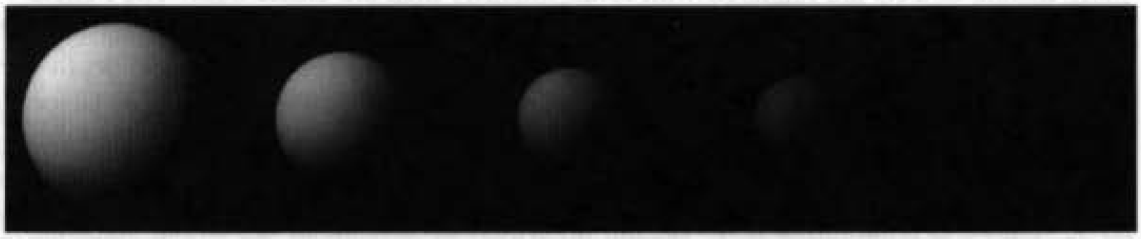


Fig. 16.7 Spheres shaded using depth cueing (Eqs. 16.5, 16.11, and 16.12). Distance from light is constant. For all spheres, $I_a = I_p = 1.0$, $k_a = 0.1$, $k_d = 0.9$, $z_f = 1.0$, $z_b = 0.0$, $s_f = 1.0$, $s_b = 0.1$, radius = 0.09. From left to right, z at front of sphere is 1.0, 0.77, 0.55, 0.32, 0.09. (By David Kurlander, Columbia University.)

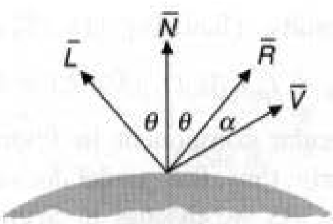


Fig. 16.8 Specular reflection.

Now move your head and notice how the highlight also moves. It does so because shiny surfaces reflect light unequally in different directions; on a perfectly shiny surface, such as a perfect mirror, light is reflected *only* in the direction of reflection $\overline{R}$, which is $\overline{L}$ mirrored about $\overline{N}$. Thus the viewer can see specularly reflected light from a mirror only when the angle α in Fig. 16.8 is zero; α is the angle between $\overline{R}$ and the direction to the viewpoint $\overline{V}$.

The Phong illumination model. Phong Bui-Tuong [BUIT75] developed a popular illumination model for nonperfect reflectors, such as the apple. It assumes that maximum specular reflectance occurs when α is zero and falls off sharply as α increases. This rapid falloff is approximated by $\cos^n \alpha$, where n is the material's *specular-reflection exponent*. Values of n typically vary from 1 to several hundred, depending on the surface material being simulated. A value of 1 provides a broad, gentle falloff, whereas higher values simulate a sharp, focused highlight (Fig. 16.9). For a perfect reflector, n would be infinite. As before, we treat a negative value of $\cos \alpha$ as zero. Phong's illumination model is based on earlier work by researchers such as Warnock [WARN69], who used a $\cos^n \theta$ term to model specular reflection with the light at the viewpoint. Phong, however, was the first to account for viewers and lights at arbitrary positions.

The amount of incident light specularly reflected depends on the angle of incidence θ. If $W(\theta)$ is the fraction of specularly reflected light, then Phong's model is

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} \left[k_d O_{d\lambda} \cos \theta + W(\theta) \cos^n \alpha \right]. \tag{16.13}$$

If the direction of reflection $\overline{R}$, and the viewpoint direction $\overline{V}$ are normalized, then $\cos \alpha = \overline{R} \cdot \overline{V}$. In addition, $W(\theta)$ is typically set to a constant k_s, the material's *specular-reflection coefficient*, which ranges between 0 and 1. The value of k_s is selected experimentally to

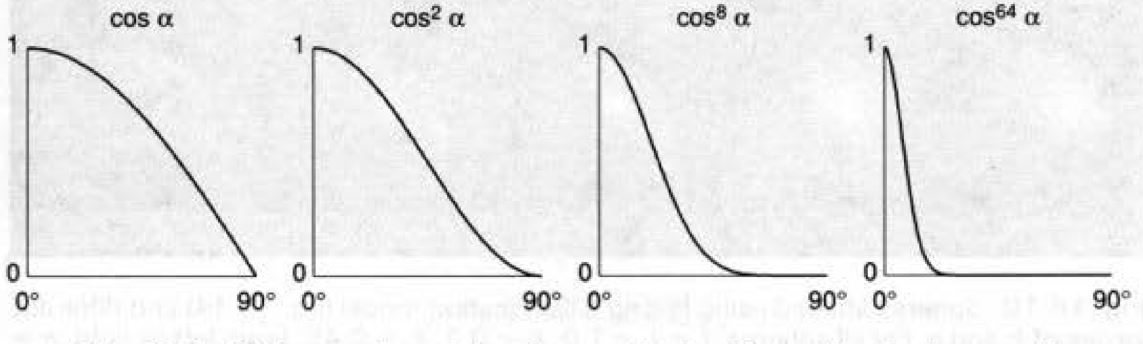


Fig. 16.9 Different values of $\cos^n \alpha$ used in the Phong illumination model.

produce aesthetically pleasing results. Then, Eq. (16.13) can be rewritten as

$$I_\lambda = I_{a\lambda}k_aO_{d\lambda} + f_{att}I_{p\lambda}[k_dO_{d\lambda}(\overline{N}\cdot\overline{L}) + k_s\,(\overline{R}\cdot\overline{V})^n]. \tag{16.14}$$

Note that the color of the specular component in Phong's illumination model is *not* dependent on any material property; thus, this model does a good job of modeling specular reflections from plastic surfaces. As we discuss in Section 16.7, specular reflection is affected by the properties of the surface itself, and, in general, may have a different color than diffuse reflection when the surface is a composite of several materials. We can accommodate this effect to a first approximation by modifying Eq. (16.14) to yield

$$I_\lambda = I_{a\lambda}k_aO_{d\lambda} + f_{att}I_{p\lambda}[k_dO_{d\lambda}(\overline{N}\cdot\overline{L}) + k_sO_{s\lambda}\,(\overline{R}\cdot\overline{V})^n], \tag{16.15}$$

where $O_{s\lambda}$ is the object's *specular color*. Figure 16.10 shows a sphere illuminated using Eq. (16.14) with different values of k_s and n.

Calculating the reflection vector. Calculating $\overline{R}$ requires mirroring $\overline{L}$ about $\overline{N}$. As shown in Fig. 16.11, this can be accomplished with some simple geometry. Since $\overline{N}$ and $\overline{L}$ are normalized, the projection of $\overline{L}$ onto $\overline{N}$ is $\overline{N}\cos\theta$. Note that $\overline{R} = \overline{N}\cos\theta + \overline{S}$, where $|\overline{S}|$ is $\sin\theta$. But, by vector subtraction and congruent triangles, $\overline{S}$ is just $\overline{N}\cos\theta - \overline{L}$. Therefore, $\overline{R} = 2\,\overline{N}\cos\theta - \overline{L}$. Substituting $\overline{N}\cdot\overline{L}$ for $\cos\theta$ and $\overline{R}\cdot\overline{V}$ for $\cos\alpha$ yields

$$\overline{R} = 2\overline{N}\,(\overline{N}\cdot\overline{L}) - \overline{L}, \tag{16.16}$$

$$\overline{R}\cdot\overline{V} = (2\overline{N}\,(\overline{N}\cdot\overline{L}) - \overline{L})\cdot\overline{V}. \tag{16.17}$$

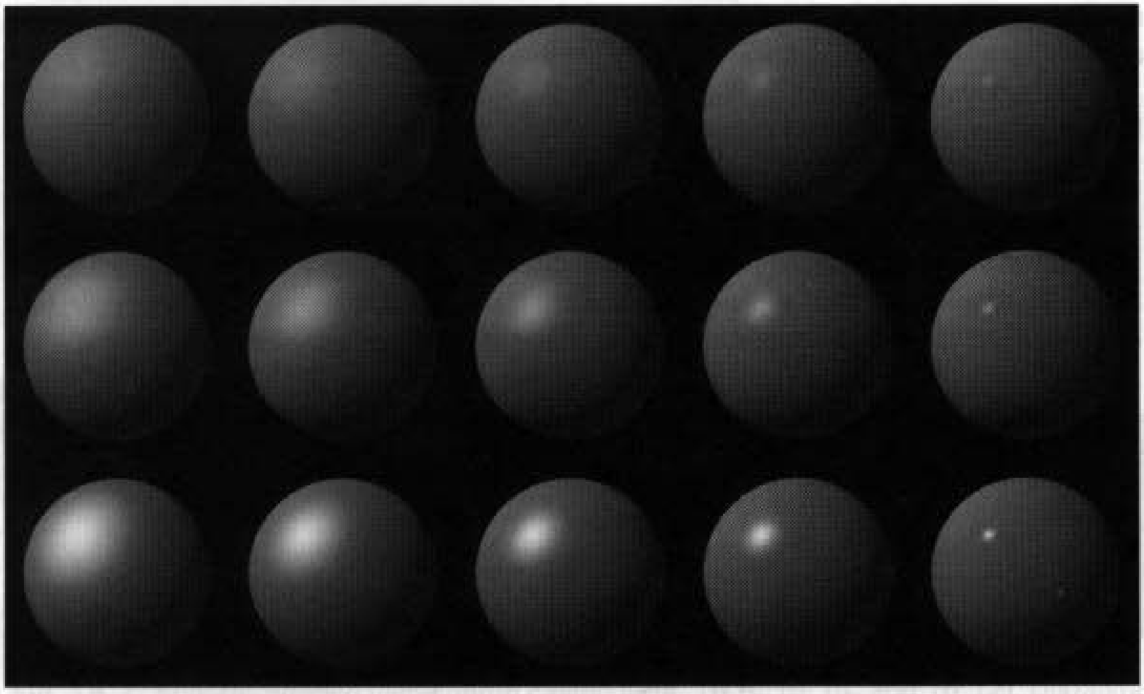


Fig. 16.10 Spheres shaded using Phong's illumination model (Eq. 16.14) and different values of k_s and n. For all spheres, $I_a = I_p = 1.0$, $k_a = 0.1$, $k_d = 0.45$. From left to right, $n =$ 3.0, 5.0, 10.0, 27.0, 200.0. From top to bottom, $k_s = 0.1, 0.25, 0.5$. (By David Kurlander, Columbia University.)

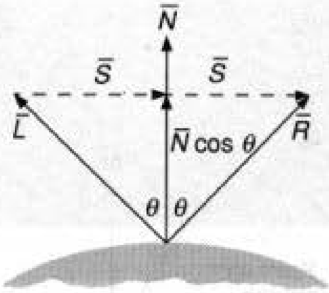


Fig. 16.11 Calculating the reflection vector.

If the light source is at infinity, $\overline{N} \cdot \overline{L}$ is constant for a given polygon, whereas $\overline{R} \cdot \overline{V}$ varies across the polygon. For curved surfaces or for a light source not at infinity, both $\overline{N} \cdot \overline{L}$ and $\overline{R} \cdot \overline{V}$ vary across the surface.

The halfway vector. An alternative formulation of Phong's illumination model uses the *halfway vector* $\overline{H}$, so called because its direction is halfway between the directions of the light source and the viewer, as shown in Fig. 16.12. $\overline{H}$ is also known as the direction of maximum highlights. If the surface were oriented so that its normal were in the same direction as $\overline{H}$, the viewer would see the brightest specular highlight, since $\overline{R}$ and $\overline{V}$ would also point in the same direction. The new specular-reflection term can be expressed as $(\overline{N} \cdot \overline{H})^n$, where $\overline{H} = (\overline{L} + \overline{V}) \,/\, |\overline{L} + \overline{V}|$. When the light source and the viewer are both at infinity, then the use of $\overline{N} \cdot \overline{H}$ offers a computational advantage, since $\overline{H}$ is constant. Note that β, the angle between $\overline{N}$ and $\overline{H}$, is not equal to α, the angle between $\overline{R}$ and $\overline{V}$, so the same specular exponent n produces different results in the two formulations (see Exercise 16.1). Although using a $\cos^n$ term allows the generation of recognizably glossy surfaces, you should remember that it is based on empirical observation, not on a theoretical model of the specular-reflection process.

16.1.5 Improving the Point-Light-Source Model

Real light sources do not radiate equally in all directions. Warn [WARN83] has developed easily implemented lighting controls that can be added to any illumination equation to model some of the directionality of the lights used by photographers. In Phong's model, a point light source has only an intensity and a position. In Warn's model, a light L is modeled by a point on a hypothetical specular reflecting surface, as shown in Fig. 16.13. This surface is illuminated by a point light source L' in the direction $\overline{L}'$. Assume that $\overline{L}'$ is normal to the hypothetical reflecting surface. Then, we can use the Phong illumination

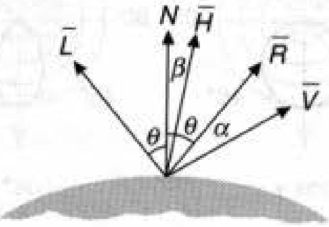


Fig. 16.12 $\overline{H}$, the halfway vector, is halfway between the direction of the light source and the viewer.

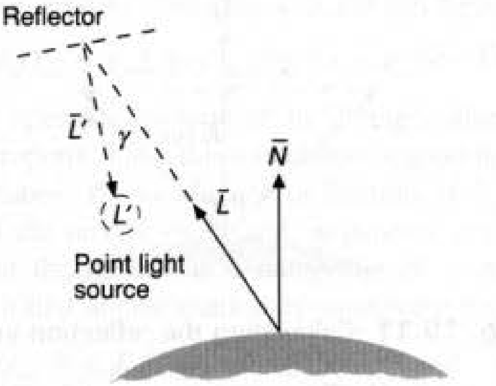


Fig. 16.13 Warn's lighting model. A light is modeled as the specular reflection from a single point illuminated by a point light source.

equation to determine the intensity of L at a point on the object in terms of the angle γ between $\overline{L}$ and $\overline{L}'$. If we further assume that the reflector reflects only specular light and has a specular coefficient of 1, then the light's intensity at a point on the object is

$$I_{L'\lambda} \cos^p \gamma, \tag{16.18}$$

where $I_{L'\lambda}$ is the intensity of the hypothetical point light source, p is the reflector's specular exponent, and γ is the angle between $-\overline{L}$ and the hypothetical surface's normal, $\overline{L}'$, which is the direction to L'. Equation (16.18) models a symmetric directed light source whose axis of symmetry is $\overline{L}'$, the direction in which the light may be thought of as pointing. Using dot products, we can write Eq.(16.18) as

$$I_{L'\lambda} (-\overline{L} \cdot \overline{L}')^p. \tag{16.19}$$

Once again, we treat a negative dot product as zero. Equation (16.19) can thus be substituted for the light-source intensity $I_{p\lambda}$ in the formulation of Eq. (16.15) or any other illumination equation. Contrast the intensity distribution of the uniformly radiating point source with the $\cos^p$ distribution of the Warn light source in Fig. 16.14. Each distribution is plotted in cross-section, showing intensity as a function of angular direction around the light's axis in polar coordinates. $\overline{L}'$ is shown as an arrow. These plots are called *goniometric diagrams*. The larger the value of p, the more the light is concentrated along $\overline{L}'$. Thus, a

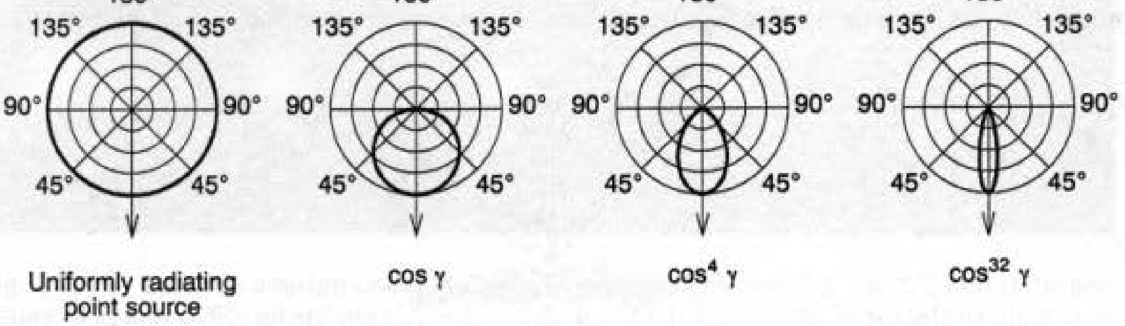


Fig. 16.14 Intensity distributions for uniformly radiating point source and Warn light source with different values of p.

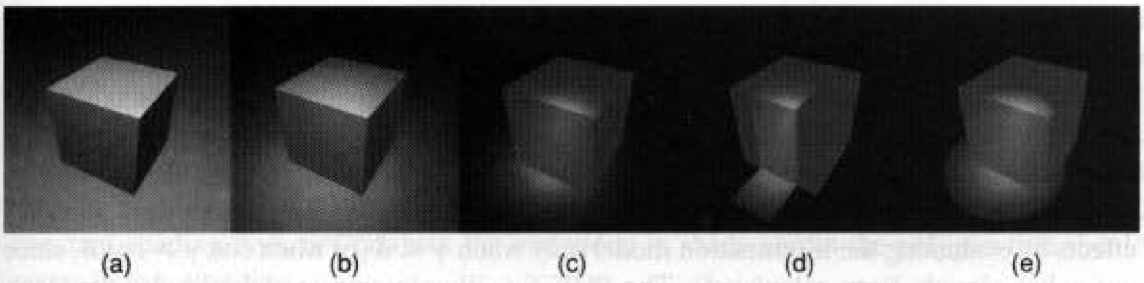


Fig. 16.15 Cube and plane illuminated using Warn lighting controls. (a) Uniformly radiating point source (or $p = 0$). (b) $p = 4$. (c) $p = 32$. (d) Flaps. (e) Cone with $\delta = 18°$. (By David Kurlander, Columbia University.)

large value of p can simulate a highly directional spotlight, whereas a small value of p can simulate a more diffuse floodlight. If p is 0, then the light acts like a uniformly radiating point source. Figure 16.15(a–c) shows the effects of different values of p. Verbeck [VERB84] and Nishita et al. [NISH85b] have modeled point light sources with more complex irregular intensity and spectral distributions. In general, however, once we determine a point light source's intensity as seen from a particular direction, this value can be used in any illumination equation.

To restrict a light's effects to a limited area of the scene, Warn implemented *flaps* and *cones*. Flaps, modeled loosely after the "barn doors" found on professional photographic lights, confine the effects of the light to a designated range in x, y, and z world coordinates. Each light has six flaps, corresponding to user-specified minimum and maximum values in each coordinate. Each flap also has a flag indicating whether it is on or off. When a point's shade is determined, the illumination model is evaluated for a light only if the point's coordinates are within the range specified by the minimum and maximum coordinates of those flaps that are on. For example, if $\overline{L}'$ is parallel to the y axis, then the x and z flaps can sharply restrict the light's effects, much like the photographic light's barn doors. Figure 16.16(a) shows the use of the x flaps in this situation. The y flaps can also be used here to

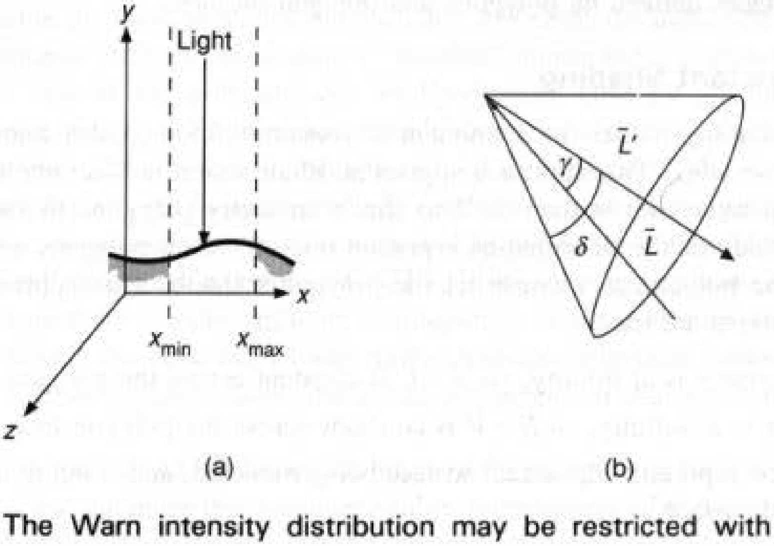


Fig. 16.16 The Warn intensity distribution may be restricted with (a) flaps and (b) cones.

restrict the light in a way that has no physical counterpart, allowing only objects within a specified range of distances from the light to be illuminated. In Fig. 16.15(d) the cube is aligned with the coordinate system, so two pairs of flaps can produce the effects shown.

Warn makes it possible to create a sharply delineated spotlight through the use of a cone whose apex is at the light source and whose axis lies along $\overline{L}'$. As shown in Fig. 16.16(b), a cone with a generating angle of δ may be used to restrict the light source's effects by evaluating the illumination model only when $\gamma < \delta$ (or when $\cos \gamma > \cos \delta$, since $\cos \gamma$ has already been calculated). The PHIGS+ illumination model includes the Warn $\cos^p \gamma$ term and cone angle δ. Figure 16.15(e) demonstrates the use of a cone to restrict the light of Fig. 16.15(c). Color Plate II.17 shows a car rendered with Warn's lighting controls.

16.1.6 Multiple Light Sources

If there are m light sources, then the terms for each light source are summed:

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \le i \le m} f_{att_i} I_{p\lambda_i} \left[k_d O_{d\lambda} (\overline{N} \cdot \overline{L}_i) + k_s O_{s\lambda} (\overline{R}_i \cdot \overline{V})^n \right]. \qquad (16.20)$$

The summation harbors a new possibility for error in that I_λ can now exceed the maximum displayable pixel value. (Although this can also happen for a single light, we can easily avoid it by an appropriate choice of f_{att} and the material coefficients.) Several approaches can be used to avoid overflow. The simplest is to clamp each I_λ individually to its maximum value. Another approach considers all of a pixel's I_λ values together. If at least one is too big, each is divided by the largest to maintain the hue and saturation at the expense of the value. If all the pixel values can be computed before display, image-processing transformations can be applied to the entire picture to bring the values within the desired range. Hall [HALL89] discusses the tradeoffs of these and other techniques.

16.2 SHADING MODELS FOR POLYGONS

It should be clear that we can shade any surface by calculating the surface normal at each visible point and applying the desired illumination model at that point. Unfortunately, this brute-force shading model is expensive. In this section, we describe more efficient shading models for surfaces defined by polygons and polygon meshes.

16.2.1 Constant Shading

The simplest shading model for a polygon is *constant shading*, also known as *faceted shading* or *flat shading*. This approach applies an illumination model once to determine a single intensity value that is then used to shade an entire polygon. In essence, we are sampling the value of the illumination equation once for each polygon, and holding the value across the polygon to reconstruct the polygon's shade. This approach is valid if several assumptions are true:

1. The light source is at infinity, so $\overline{N} \cdot \overline{L}$ is constant across the polygon face
2. The viewer is at infinity, so $\overline{N} \cdot \overline{V}$ is constant across the polygon face
3. The polygon represents the actual surface being modeled, and is not an approximation to a curved surface.

If a visible-surface algorithm is used that outputs a list of polygons, such as one of the list-priority algorithms, constant shading can take advantage of the ubiquitous single-color 2D polygon primitive.

If either of the first two assumptions is wrong, then, if we are to use constant shading, we need some method to determine a single value for each of $\overline{L}$ and $\overline{V}$. For example, values may be calculated for the center of the polygon, or for the polygon's first vertex. Of course, constant shading does not produce the variations in shade across the polygon that should occur in this situation.

16.2.2 Interpolated Shading

As an alternative to evaluating the illumination equation at each point on the polygon, Wylie, Romney, Evans, and Erdahl [WYLI67] pioneered the use of *interpolated shading*, in which shading information is linearly interpolated across a triangle from values determined for its vertices. Gouraud [GOUR71] generalized this technique to arbitrary polygons. This is particularly easy for a scan-line algorithm that already interpolates the z value across a span from interpolated z values computed for the span's endpoints. For increased efficiency, a difference equation may be used, like that developed in Section 15.4 to determine the z value at each pixel. Although z interpolation is physically correct (assuming that the polygon is planar), note that interpolated shading is not, since it only approximates evaluating the illumination model at each point on the polygon.

Our final assumption, that the polygon accurately represents the surface being modeled, is most often the one that is incorrect, which has a much more substantial effect on the resulting image than does the failure of the other two assumptions. Many objects are curved, rather than polyhedral, yet representing them as a polygon mesh allows the use of efficient polygon visible-surface algorithms. We discuss next how to render a polygon mesh so that it looks as much as possible like a curved surface.

16.2.3 Polygon Mesh Shading

Suppose that we wish to approximate a curved surface by a polygonal mesh. If each polygonal facet in the mesh is shaded individually, it is easily distinguished from neighbors whose orientation is different, producing a "faceted" appearance, as shown in Color Plate II.29. This is true if the polygons are rendered using constant shading, interpolated shading, or even per-pixel illumination calculations, because two adjacent polygons of different orientation have different intensities along their borders. The simple solution of using a finer mesh turns out to be surprisingly ineffective, because the perceived difference in shading between adjacent facets is accentuated by the Mach band effect (discovered by Mach in 1865 and described in detail in [RATL72]), which exaggerates the intensity change at any edge where there is a discontinuity in magnitude or slope of intensity. At the border between two facets, the dark facet looks darker and the light facet looks lighter. Figure 16.17 shows, for two separate cases, the actual and perceived changes in intensity along a surface.

Mach banding is caused by *lateral inhibition* of the receptors in the eye. The more light a receptor receives, the more that receptor inhibits the response of the receptors adjacent to

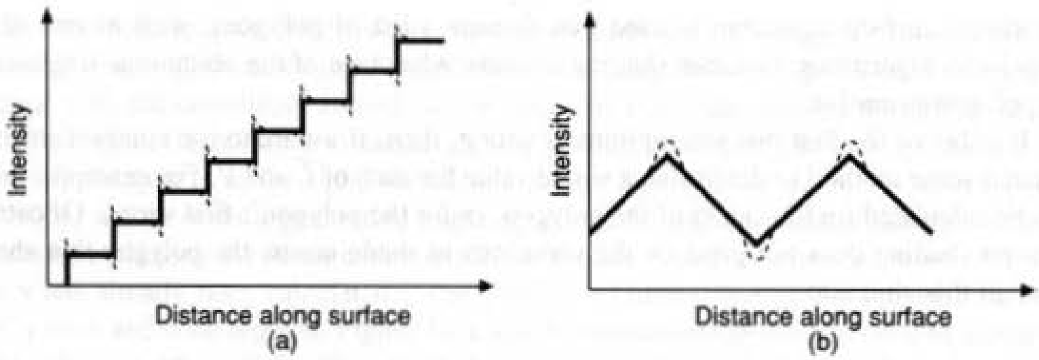


Fig. 16.17 Actual and perceived intensities in the Mach band effect. Dashed lines are perceived intensity; solid lines are actual intensity.

it. The response of a receptor to light is inhibited by its adjacent receptors in inverse relation to the distance to the adjacent receptor. Receptors directly on the brighter side of an intensity change have a stronger response than do those on the brighter side that are farther from the edge, because they receive less inhibition from their neighbors on the darker side. Similarly, receptors immediately to the darker side of an intensity change have a weaker response than do those farther into the darker area, because they receive more inhibition from their neighbors on the brighter side. The Mach band effect is quite evident in Color Plate II.29, especially between adjacent polygons that are close in color.

The polygon-shading models we have described determine the shade of each polygon individually. Two basic shading models for polygon meshes take advantage of the information provided by adjacent polygons to simulate a smooth surface. In order of increasing complexity (and realistic effect), they are known as Gouraud shading and Phong shading, after the researchers who developed them. Current 3D graphics workstations typically support one or both of these approaches through a combination of hardware and firmware.

16.2.4 Gouraud Shading

Gouraud shading [GOUR71], also called *intensity interpolation shading* or *color interpolation shading*, eliminates intensity discontinuities. Color Plate II.30 uses Gouraud shading. Although most of the Mach banding of Color Plate II.29 is no longer visible in Color Plate II.30, the bright ridges on objects such as the torus and cone are Mach bands caused by a rapid, although not discontinuous, change in the slope of the intensity curve; Gouraud shading does not completely eliminate such intensity changes.

Gouraud shading extends the concept of interpolated shading applied to individual polygons by interpolating polygon vertex illumination values that take into account the surface being approximated. The Gouraud shading process requires that the normal be known for each vertex of the polygonal mesh. Gouraud was able to compute these *vertex normals* directly from an analytical description of the surface. Alternatively, if the vertex normals are not stored with the mesh and cannot be determined directly from the actual surface, then, Gouraud suggested, we can approximate them by averaging the surface

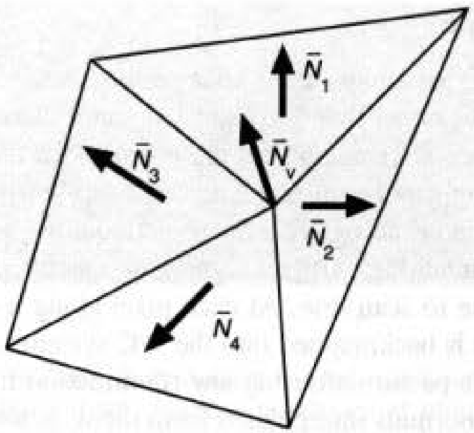


Fig. 16.18 Normalized polygon surface normals may be averaged to obtain vertex normals. Averaged normal $\overline{N}_v$ is $\sum_{1 \le i \le n} \overline{N}_i \,/\, |\sum_{1 \le i \le n} \overline{N}_i|$.

normals of all polygonal facets sharing each vertex (Fig. 16.18). If an edge is meant to be visible (as at the joint between a plane's wing and body), then we find two vertex normals, one for each side of the edge, by averaging the normals of polygons on each side of the edge separately. Normals were not averaged across the teapot's patch cracks in Color Plate II.30. (See caption to Color Plate II.21.)

The next step in Gouraud shading is to find *vertex intensities* by using the vertex normals with any desired illumination model. Finally, each polygon is shaded by linear interpolation of vertex intensities along each edge and then between edges along each scan line (Fig. 16.19) in the same way that we described interpolating z values in Section 15.4. The term *Gouraud shading* is often generalized to refer to intensity interpolation shading of even a single polygon in isolation, or to the interpolation of arbitrary colors associated with polygon vertices.

The interpolation along edges can easily be integrated with the scan-line visible-surface algorithm of Section 15.6. With each edge, we store for each color component the starting intensity and the change of intensity for each unit change in y. A visible span on a scan line is filled in by interpolating the intensity values of the two edges bounding the span. As in all linear-interpolation algorithms, a difference equation may be used for increased efficiency.

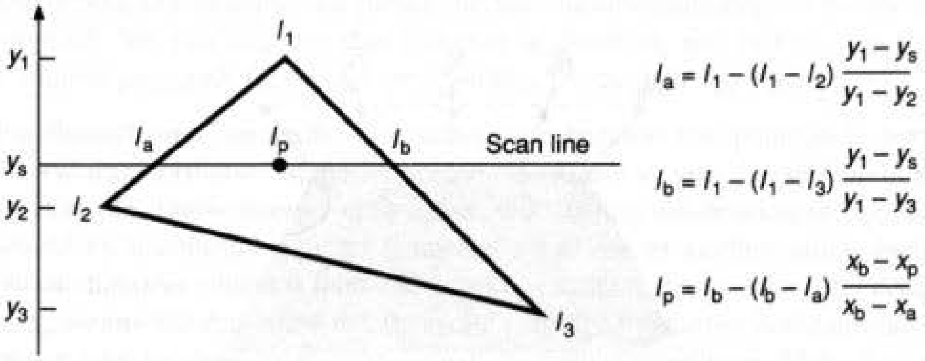

y
y_1
y_s
y_2
y_3
I_1
I_a
I_p
I_b
Scan line
I_2
I_3
$I_a = I_1 - (I_1 - I_2)\frac{y_1 - y_s}{y_1 - y_2}$
$I_b = I_1 - (I_1 - I_3)\frac{y_1 - y_s}{y_1 - y_3}$
$I_p = I_b - (I_b - I_a)\frac{x_b - x_p}{x_b - x_a}$

Fig. 16.19 Intensity interpolation along polygon edges and scan lines.

16.2.5 Phong Shading

Phong shading [BUIT75], also known as *normal-vector interpolation shading*, interpolates the surface normal vector $\overline{N}$, rather than the intensity. Interpolation occurs across a polygon span on a scan line, between starting and ending normals for the span. These normals are themselves interpolated along polygon edges from vertex normals that are computed, if necessary, just as in Gouraud shading. The interpolation along edges can again be done by means of incremental calculations, with all three components of the normal vector being incremented from scan line to scan line. At each pixel along a scan line, the interpolated normal is normalized, and is backmapped into the WC system or one isometric to it, and a new intensity calculation is performed using any illumination model. Figure 16.20 shows two edge normals and the normals interpolated from them, before and after normalization.

Color Plates II.31 and II.32 were generated using Gouraud shading and Phong shading respectively, and an illumination equation with a specular-reflectance term. Phong shading yields substantial improvements over Gouraud shading when such illumination models are used, because highlights are reproduced more faithfully, as shown in Fig. 16.21. Consider what happens if n in the Phong $\cos^n \alpha$ illumination term is large and one vertex has a very small α, but each of its adjacent vertices has a large α. The intensity associated with the vertex that has a small α will be appropriate for a highlight, whereas the other vertices will have nonhighlight intensities. If Gouraud shading is used, then the intensity across the polygon is linearly interpolated between the highlight intensity and the lower intensities of the adjacent vertices, spreading the highlight over the polygon (Fig. 16.21a). Contrast this with the sharp drop from the highlight intensity that is computed if linearly interpolated normals are used to compute the $\cos^n \alpha$ term at each pixel (Fig. 16.21b). Furthermore, if a highlight fails to fall at a vertex, then Gouraud shading may miss it entirely (Fig. 16.21c), since no interior point can be brighter than the brightest vertex from which it is interpolated. In contrast, Phong shading allows highlights to be located in a polygon's interior (Fig. 16.21d). Compare the highlights on the ball in Color Plates II.31 and II.32.

Even with an illumination model that does not take into account specular reflectance, the results of normal-vector interpolation are in general superior to intensity interpolation, because an approximation to the normal is used at each point. This reduces Mach-band problems in most cases, but greatly increases the cost of shading in a straightforward implementation, since the interpolated normal must be normalized every time it is used in an illumination model. Duff [DUFF79] has developed a combination of difference

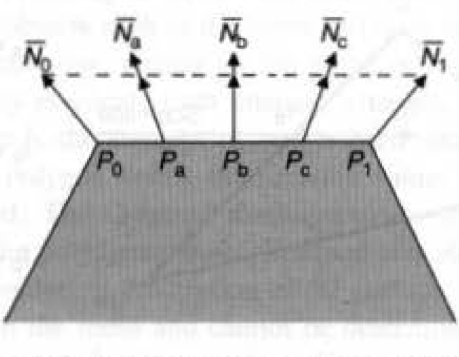


Fig. 16.20 Normal vector interpolation. (After [BUIT75].)

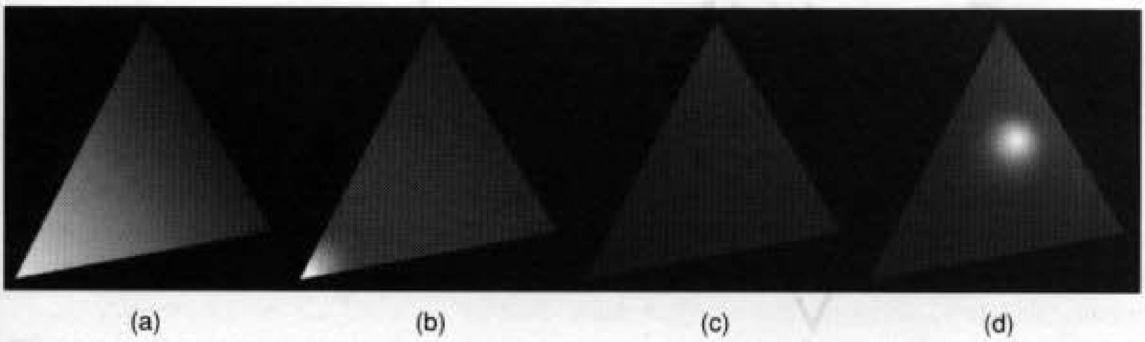


Fig. 16.21 A specular-reflection illumination model used with Gouraud shading and Phong shading. Highlight falls at left vertex: (a) Gouraud shading, (b) Phong shading. Highlight falls in polygon interior: (c) Gouraud shading, (d) Phong shading. (By David Kurlander, Columbia University.)

equations and table lookup to speed up the calculation. Bishop and Weimer [BISH86] provide an excellent approximation of Phong shading by using a Taylor series expansion that offers even greater increases in shading speed.

Another shading model, intermediate in complexity between Gouraud and Phong shading, involves the linear interpolation of the dot products used in the illumination models. As in Phong shading, the illumination model is evaluated at each pixel, but the interpolated dot products are used to avoid the expense of computing and normalizing any of the direction vectors. This model can produce more satisfactory effects than Gouraud shading when used with specular-reflection illumination models, since the specular term is calculated separately and has power-law, rather than linear, falloff. As in Gouraud shading, however, highlights are missed if they do not fall at a vertex, since no intensity value computed for a set of interpolated dot products can exceed those computed for the set of dot products at either end of the span.

16.2.6 Problems with Interpolated Shading

There are many problems common to all these interpolated-shading models, several of which we list here.

Polygonal silhouette. No matter how good an approximation an interpolated shading model offers to the actual shading of a curved surface, the silhouette edge of the mesh is still clearly polygonal. We can improve this situation by breaking the surface into a greater number of smaller polygons, but at a corresponding increase in expense.

Perspective distortion. Anomalies are introduced because interpolation is performed after perspective transformation in the 3D screen-coordinate system, rather than in the WC system. For example, linear interpolation causes the shading information in Fig. 16.19 to be incremented by a constant amount from one scan line to another along each edge. Consider what happens when vertex 1 is more distant than vertex 2. Perspective foreshortening means that the difference from one scan line to another in the untransformed z value along an edge increases in the direction of the farther coordinate. Thus, if $y_s = (y_1 + y_2) / 2$, then $I_s = (I_1 + I_2) / 2$, but z_s will not equal $(z_1 + z_2) / 2$. This problem can also be

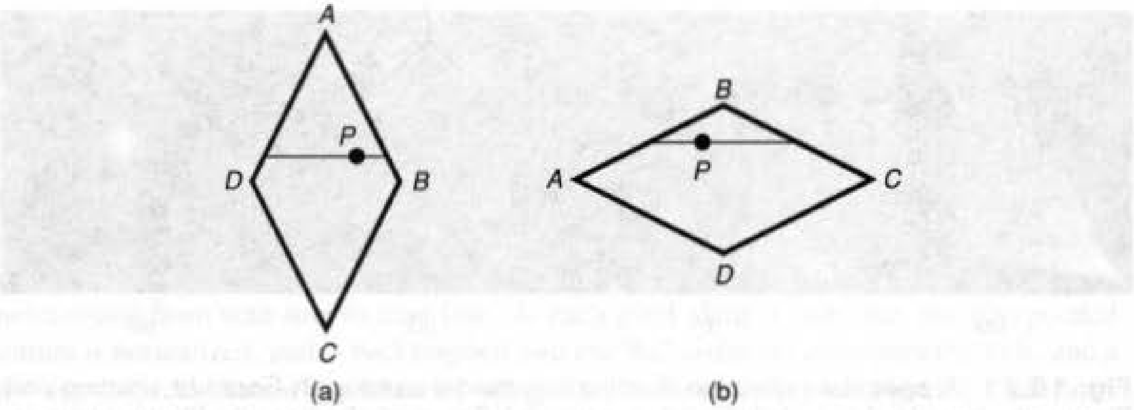


Fig. 16.22 Interpolated values derived for point *P* on the same polygon at different orientations differ from (a) to (b). *P* interpolates *A, B, D* in (a) and *A, B, C* in (b).

reduced by using a larger number of smaller polygons. Decreasing the size of the polygons increases the number of points at which the information to be interpolated is sampled, and therefore increases the accuracy of the shading.

Orientation dependence. The results of interpolated-shading models are not independent of the projected polygon's orientation. Since values are interpolated between vertices and across horizontal scan lines, the results may differ when the polygon is rotated (see Fig. 16.22). This effect is particularly obvious when the orientation changes slowly between successive frames of an animation. A similar problem can also occur in visible-surface determination when the *z* value at each point is interpolated from the *z* values assigned to each vertex. Both problems can be solved by decomposing polygons into triangles (see Exercise 16.2). Alternatively, Duff [DUFF79] suggests rotation-independent, but expensive, interpolation methods that solve this problem without the need for decomposition.

Problems at shared vertices. Shading discontinuities can occur when two adjacent polygons fail to share a vertex that lies along their common edge. Consider the three polygons of Fig. 16.23, in which vertex *C* is shared by the two polygons on the right, but not by the large polygon on the left. The shading information determined directly at *C* for the polygons at the right will typically not be the same as the information interpolated at that point from the values at *A* and *B* for the polygon at the left. As a result, there will be a discontinuity in the shading. The discontinuity can be eliminated by inserting in the

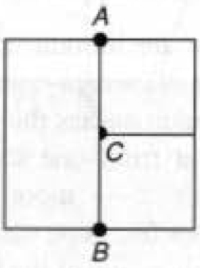


Fig. 16.23 Vertex *C* is shared by the two polygons on the right, but not by the larger rectangular polygon on the left.

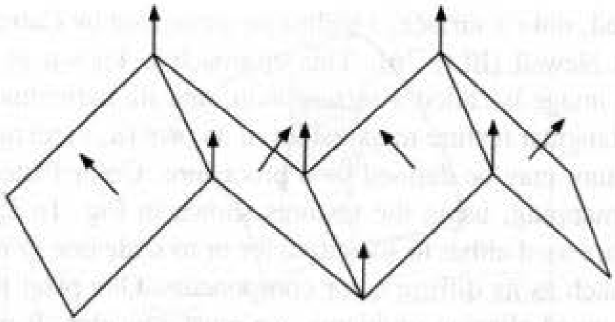


Fig. 16.24 Problems with computing vertex normals. Vertex normals are all parallel.

polygon on the left an extra vertex that shares *C*'s shading information. We can preprocess a static polygonal database in order to eliminate this problem; alternatively, if polygons will be split on the fly (e.g., using the BSP-tree visible-surface algorithm), then extra bookkeeping can be done to introduce a new vertex in an edge that shares an edge that is split.

Unrepresentative vertex normals. Computed vertex normals may not adequately represent the surface's geometry. For example, if we compute vertex normals by averaging the normals of the surfaces sharing a vertex, all of the vertex normals of Fig. 16.24 will be parallel to one another, resulting in little or no variation in shade if the light source is distant. Subdividing the polygons further before vertex normal computation will solve this problem.

Although these problems have prompted much work on rendering algorithms that handle curved surfaces directly, polygons are sufficiently faster (and easier) to process that they still form the core of most rendering systems.

16.3 SURFACE DETAIL

Applying any of the shading models we have described so far to planar or bicubic surfaces produces smooth, uniform surfaces—in marked contrast to most of the surfaces we see and feel. We discuss next a variety of methods developed to simulate this missing surface detail.

16.3.1 Surface-Detail Polygons

The simplest approach adds gross detail through the use of *surface-detail polygons* to show features (such as doors, windows, and lettering) on a base polygon (such as the side of a building). Each surface-detail polygon is coplanar with its base polygon, and is flagged so that it does not need to be compared with other polygons during visible-surface determination. When the base polygon is shaded, its surface-detail polygons and their material properties take precedence for those parts of the base polygon that they cover.

16.3.2 Texture Mapping

As detail becomes finer and more intricate, explicit modeling with polygons or other geometric primitives becomes less practical. An alternative is to map an image, either digitized or synthesized, onto a surface, a technique pioneered by Catmull [CATM74b] and refined by Blinn and Newell [BLIN76]. This approach is known as *texture mapping* or *pattern mapping*; the image is called a *texture map*, and its individual elements are often called *texels*. The rectangular texture map resides in its own (u, v) texture coordinate space. Alternatively, the texture may be defined by a procedure. Color Plate II.35 shows several examples of texture mapping, using the textures shown in Fig. 16.25. At each rendered pixel, selected texels are used either to substitute for or to scale one or more of the surface's material properties, such as its diffuse color components. One pixel is often covered by a number of texels. To avoid aliasing problems, we must consider all relevant texels.

As shown in Fig. 16.26, texture mapping can be accomplished in two steps. A simple approach starts by mapping the four corners of the pixel onto the surface. For a bicubic patch, this mapping naturally defines a set of points in the surface's (s, t) coordinate space. Next, the pixel's corner points in the surface's (s, t) coordinate space are mapped into the texture's (u, v) coordinate space. The four (u, v) points in the texture map define a quadrilateral that approximates the more complex shape into which the pixel may actually map due to surface curvature. We compute a value for the pixel by summing all texels that lie within the quadrilateral, weighting each by the fraction of the texel that lies within the quadrilateral. If a transformed point in (u, v) space falls outside of the texture map, the texture map may be thought of as being replicated, like the patterns of Section 2.1.3.

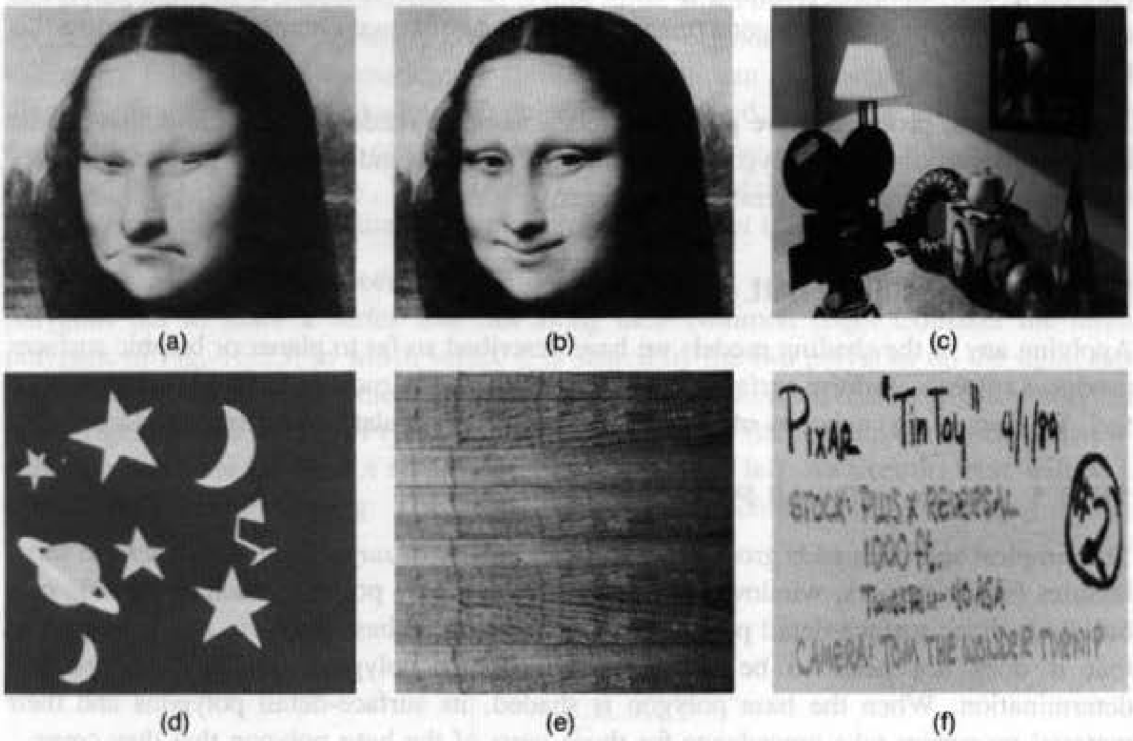


Fig. 16.25 Textures used to create Color Plate II.35. (a) Frowning Mona. (b) Smiling Mona. (c) Painting. (d) Wizard's cap. (e) Floor. (f) Film label. (Copyright © 1990, Pixar. Images rendered by Thomas Williams and H. B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

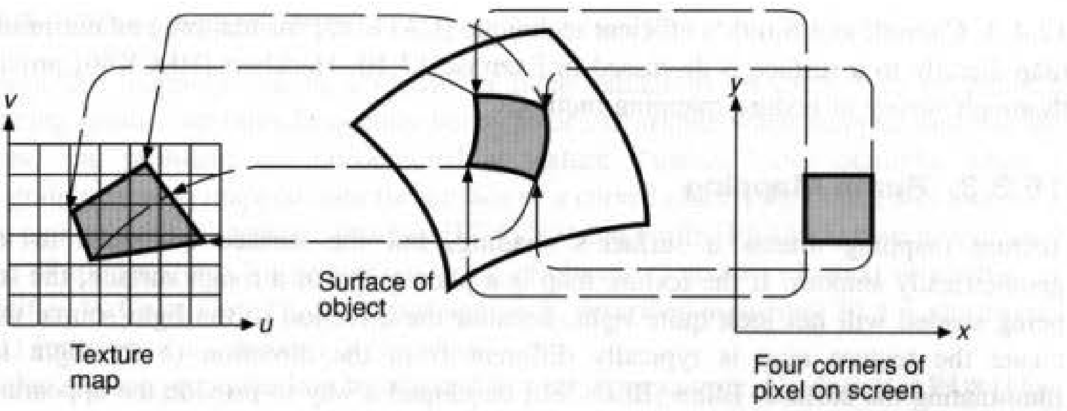


Fig. 16.26 Texture mapping from pixel to the surface to the texture map.

Rather than always use the identity mapping between (s, t) and (u, v), we can define a correspondence between the four corners of the 0-to-1 (s, t) rectangle and a quadrilateral in (u, v). When the surface is a polygon, it is common to assign texture map coordinates directly to its vertices. Since, as we have seen, linearly interpolating values across arbitrary polygons is orientation-dependent, polygons may be decomposed into triangles first. Even after triangulation, however, linear interpolation will cause distortion in the case of perspective projection. This distortion will be more noticeable than that caused when interpolating other shading information, since texture features will not be correctly foreshortened. We can obtain an approximate solution to this problem by decomposing polygons into smaller ones, or an exact solution, at greater cost, by performing the perspective division while interpolating.

The approach just described assumes square pixel geometry and simple box filtering. It also fails to take into account pixels that map to only part of a surface. Feibush, Levoy, and Cook [FEIB80] address these problems for texture-mapping polygons. Think of the square pixel in Fig. 16.26 as the bounding rectangle of the support of an arbitrary filter centered at a pixel. Pixels and texels can then be treated as points. In effect, all texels that lie within the mapped intersection of the transformed bounding rectangle and polygon are selected, and these texels' coordinates are transformed into the coordinate system of the bounding rectangle. Each texel's transformed coordinates are used to index into a filter table to determine the texel's weighting, and the weighted average of the texel intensities is computed. This weighted average must in turn be weighted by the percentage contribution that the polygon makes to the pixel's intensity. The process is repeated for each polygon whose projection intersects the pixel, and the values are summed. Section 17.4.2 discusses this algorithm in more detail.

The Feibush, Levoy, and Cook algorithm can be quite inefficient. Consider mapping a checkerboard pattern onto an infinite ground plane. An extremely large number of texels may have to be weighted and summed just to texture a single distant ground-plane pixel. One solution to this problem is to prefilter the texture and to store the results in a way that is space-efficient and that allows quick determination of the weighted average of texels mapping to a pixel. Algorithms by Williams [WILL83], Crow [CROW84], Glassner [GLAS86], and Heckbert [HECK86a] that take this approach are discussed in Section

17.4.3. Catmull and Smith's efficient technique [CATM80] for mapping an entire texture map directly to a surface is discussed in Exercise 17.10. Heckbert [HECK86] provides a thorough survey of texture-mapping methods.

16.3.3 Bump Mapping

Texture mapping affects a surface's shading, but the surface continues to appear geometrically smooth. If the texture map is a photograph of a rough surface, the surface being shaded will not look quite right, because the direction to the light source used to create the texture map is typically different from the direction to the light source illuminating the surface. Blinn [BLIN78b] developed a way to provide the appearance of modified surface geometry that avoids explicit geometrical modeling. His approach involves perturbing the surface normal before it is used in the illumination model, just as slight roughness in a surface would perturb the surface normal. This method is known as *bump mapping*, and is based on texture mapping.

A *bump map* is an array of displacements, each of which can be used to simulate displacing a point on a surface a little above or below that point's actual position. Let us represent a point on a surface by a vector $\overline{P}$, where $\overline{P} = [x(s, t), y(s, t), z(s, t)]$. We call the partial derivatives of the surface at $\overline{P}$ with respect to the surface's s and t parameterization axes, $\overline{P}_s$ and $\overline{P}_t$. Since each is tangent to the surface, their cross-product forms the (unnormalized) surface normal at $\overline{P}$. Thus,

$$\overline{N} = \overline{P}_s \times \overline{P}_t. \tag{16.21}$$

We can displace point $\overline{P}$ by adding to it the normalized normal scaled by a selected bump-map value B. The new point is

$$\overline{P}' = \overline{P} + \frac{B\overline{N}}{|\overline{N}|}. \tag{16.22}$$

Blinn shows that a good approximation to the new (unnormalized) normal $\overline{N}'$ is

$$\overline{N}' = \overline{N} + \frac{B_u(\overline{N} \times \overline{P}_t) - B_v\,(\overline{N} \times \overline{P}_s)}{|\overline{N}|}, \tag{16.23}$$

where B_u and B_v are the partial derivatives of the selected bump-map entry B with respect to the bump-map parameterization axes, u and v. $\overline{N}'$ is then normalized and substituted for the surface normal in the illumination equation. Note that only the partial derivatives of the bump map are used in Eq. (16.23), not its values. Bilinear interpolation can be used to derive bump-map values for specified (u, v) positions, and finite differences can be used to compute B_u and B_v.

The results of bump mapping can be quite convincing. Viewers often fail to notice that an object's texture does not affect its silhouette edges. Color Plates III.3 and III.4 show two examples of bump mapping. Unlike texture mapping, aliasing cannot be dealt with by filtering values from the bump map, since these values do not correspond linearly to intensities; filtering the bump map just smooths out the bumps. Instead, subpixel intensities may be computed and filtered for each pixel, or some prefiltering may be performed on the bump map to improve gross aliasing.

16.3.4 Other Approaches

Although 2D mapping can be effective in many situations, it often fails to produce convincing results. Textures frequently betray their 2D origins when mapped onto curved surfaces, and problems are encountered at texture "seams." For example, when a wood-grain texture is mapped onto the surface of a curved object, the object will look as if it were painted with the texture. Peachey [PEAC85] and Perlin [PERL85] have investigated the use of solid textures for proper rendering of objects "carved" of wood or marble, as exemplified by Color Plate IV.21. In this approach, described in Section 20.8.3, the texture is a 3D function of its position in the object.

Other surface properties can be mapped as well. For example, Gardner [GARD84] has used transparency mapping to make impressionistic trees and clouds from otherwise simple shapes, as described in Section 20.8.2. Color Plate IV.24 shows the application of a complex functional transparency texture to objects formed from groups of quadric surfaces. Cook has implemented *displacement mapping*, in which the actual surface is displaced, instead of only the surface normals [COOK84a]; this process, which must be carried out before visible-surface determination, was used to modify the surfaces of the cone and torus in Color Plate II.36. Using fractals to create richly detailed geometry from an initial simple geometric description is discussed in Section 20.3.

So far, we have made the tacit assumption that the process of shading a point on an object is unaffected by the rest of that object or by any other object. But an object might in fact be shadowed by another object between it and a light source; might transmit light, allowing another object to be seen through it; or might reflect other objects, allowing another object to be seen because of it. In the following sections, we describe how to model these effects.

16.4 SHADOWS

Visible-surface algorithms determine which surfaces can be seen from the viewpoint; shadow algorithms determine which surfaces can be "seen" from the light source. Thus, visible-surface algorithms and shadow algorithms are essentially the same. The surfaces that are visible from the light source are not in shadow; those that are not visible from the light source are in shadow. When there are multiple light sources, a surface must be classified relative to each of them.

Here, we consider shadow algorithms for point light sources; extended light sources are discussed in Sections 16.8, 16.12, and 16.13. Visibility from a point light source is, like visibility from the viewpoint, all or nothing. When a point on a surface cannot be seen from a light source, then the illumination calculation must be adjusted to take it into account. The addition of shadows to the illumination equation yields

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \le i \le m} S_i f_{att_i} I_{p\lambda_i} \left[k_d O_{d\lambda} (\overline{N} \cdot \overline{L}_i) + k_s O_{s\lambda} (\overline{R}_i \cdot \overline{V})^n \right], \qquad (16.24)$$

where

$$S_i = \begin{cases} 0, & \text{if light } i \text{ is blocked at this point;} \\ 1, & \text{if light } i \text{ is not blocked at this point.} \end{cases}$$

Note that areas in the shadow of all point light sources are still illuminated by the ambient light.

Although computing shadows requires computing visibility from the light source, as we have pointed out, it is also possible to generate "fake" shadows without performing any visibility tests. These can be created efficiently by transforming each object into its polygonal projection from a point light source onto a designated ground plane, without clipping the transformed polygon to the surface that it shadows or checking for whether it is blocked by intervening surfaces [BLIN88]. These shadows are then treated as surface-detail polygons. For the general case, in which these fake shadows are not adequate, various approaches to shadow generation are possible. We could perform all shadow processing first, interleave it with visible-surface processing in a variety of ways, or even do it after visible-surface processing has been performed. Here we examine algorithms that follow each of these approaches, building on the classification of shadow algorithms presented in [CROW77a]. To simplify the explanations, we shall assume that all objects are polygons unless otherwise specified.

16.4.1 Scan-Line Generation of Shadows

One of the oldest methods for generating shadows is to augment a scan-line algorithm to interleave shadow and visible-surface processing [APPEL68; BOUK70b]. Using the light source as a center of projection, the edges of polygons that might potentially cast shadows are projected onto the polygons intersecting the current scan line. When the scan crosses one of these shadow edges, the colors of the image pixels are modified accordingly.

A brute-force implementation of this algorithm must compute all $n(n-1)$ projections of every polygon on every other polygon. Bouknight and Kelley [BOUK70b] instead use a clever preprocessing step in which all polygons are projected onto a sphere surrounding the light source, with the light source as center of projection. Pairs of projections whose extents do not overlap can be eliminated, and a number of other special cases can be identified to limit the number of polygon pairs that need be considered by the rest of the algorithm. The authors then compute the projection from the light source of each polygon onto the plane of each of those polygons that they have determined it could shadow, as shown in Fig. 16.27. Each of these shadowing polygon projections has associated information about the polygons casting and potentially receiving the shadow. While the scan-line algorithm's regular scan keeps track of which regular polygon edges are being crossed, a separate, parallel shadow scan keeps track of which shadowing polygon projection edges are crossed, and thus which shadowing polygon projections the shadow scan is currently "in." When the shade for a span is computed, it is in shadow if the shadow scan is "in" one of the shadow projections cast on the polygon's plane. Thus span *bc* in Fig. 16.27(a) is in shadow, while spans *ab* and *cd* are not. Note that the algorithm does not need to clip the shadowing polygon projections analytically to the polygons being shadowed.

16.4.2 A Two-Pass Object-Precision Shadow Algorithm

Atherton, Weiler, and Greenberg have developed an algorithm that performs shadow determination before visible-surface determination [ATHE78]. They process the object description by using the same algorithm twice, once for the viewpoint, and once for the

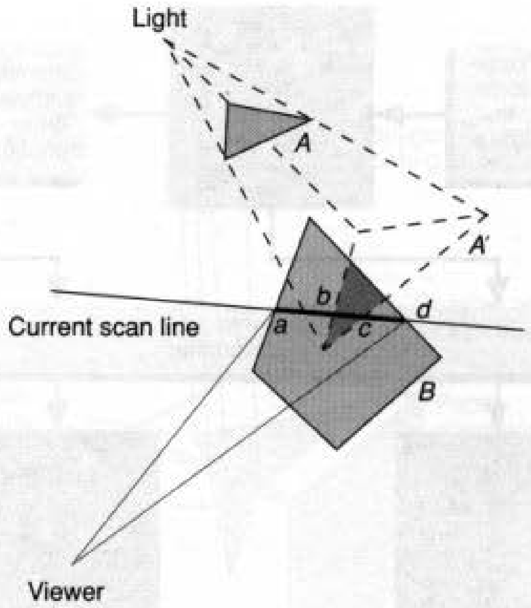
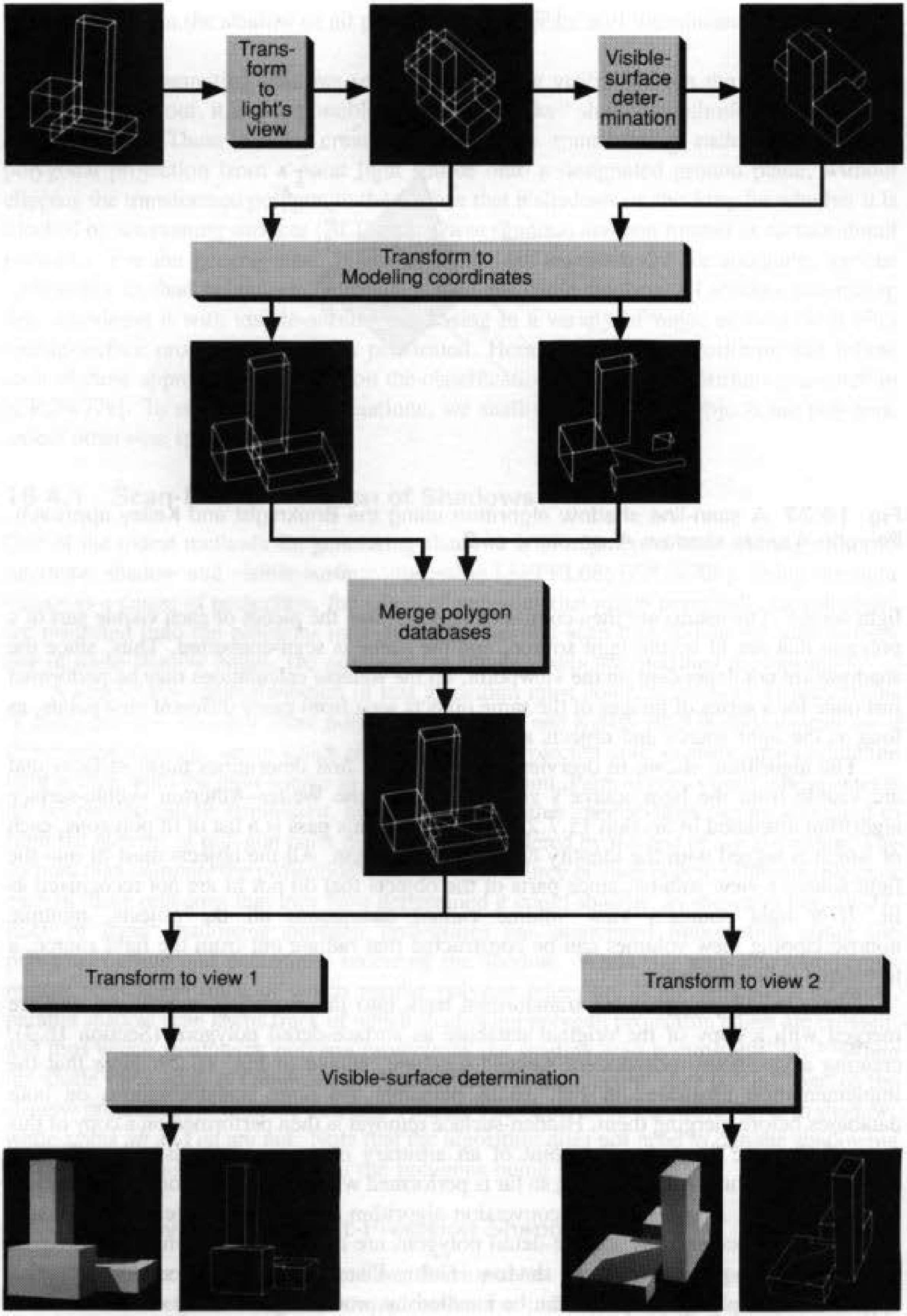


Fig. 16.27 A scan-line shadow algorithm using the Bouknight and Kelley approach. Polygon *A* casts shadow *A'* on plane of *B*.

light source. The results are then combined to determine the pieces of each visible part of a polygon that are lit by the light source, and the scene is scan-converted. Thus, since the shadows are not dependent on the viewpoint, all the shadow calculations may be performed just once for a series of images of the same objects seen from many different viewpoints, as long as the light source and objects are fixed.

The algorithm, shown in overview in Fig. 16.28, first determines those surfaces that are visible from the light source's viewpoint, using the Weiler–Atherton visible-surface algorithm discussed in Section 15.7.2. The output of this pass is a list of lit polygons, each of which is tagged with the identity of its parent polygon. All the objects must fit into the light source's view volume, since parts of the objects that do not fit are not recognized as lit. If a light source's view volume cannot encompass all the objects, multiple nonoverlapping view volumes can be constructed that radiate out from the light source, a technique called *sectoring*.

Next, the lit polygons are transformed back into the modeling coordinates and are merged with a copy of the original database as surface-detail polygons (Section 16.3), creating a viewpoint-independent merged database, shown in Fig. 16.29. Note that the implementation illustrated in Fig. 16.28 performs the same transformations on both databases before merging them. Hidden-surface removal is then performed on a copy of this merged database from the viewpoint of an arbitrary observer, again using the Weiler–Atherton algorithm. All processing so far is performed with object precision and results in a list of polygons. A polygon scan-conversion algorithm is then used to render the image. Visible surfaces covered by surface-detail polygons are rendered as lit, whereas uncovered visible surfaces are rendered in shadow. Color Plate III.5 was generated using this approach. Multiple light sources can be handled by processing the merged database from the viewpoint of each new light source, merging the results of each pass.


Trans-
form
to
light's
view
Visible-
surface
deter-
mination
Transform to
Modeling coordinates
Merge polygon
databases
Transform to view 1
Transform to view 2
Visible-surface determination

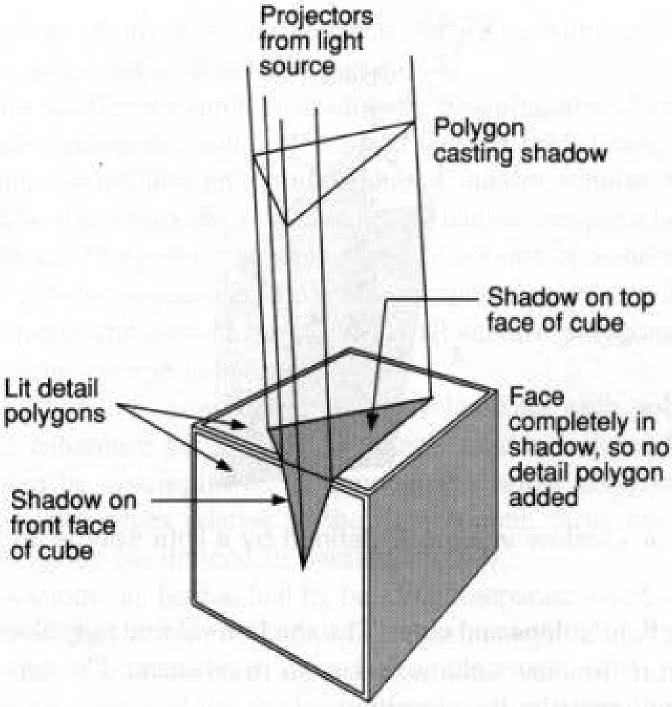

Projectors from light source
Polygon casting shadow
Shadow on top face of cube
Lit detail polygons
Face completely in shadow, so no detail polygon added
Shadow on front face of cube

Fig. 16.29 Lit surface-detail polygons.

16.4.3 Shadow Volumes

Crow [CROW77a] describes how to generate shadows by creating for each object a *shadow volume* that the object blocks from the light source. A shadow volume is defined by the light source and an object and is bounded by a set of invisible *shadow polygons*. As shown in Fig. 16.30, there is one quadrilateral shadow polygon for each silhouette edge of the object relative to the light source. Three sides of a shadow polygon are defined by a silhouette edge of the object and the two lines emanating from the light source and passing through that edge's endpoints. Each shadow polygon has a normal that points out of the shadow volume. Shadow volumes are generated only for polygons facing the light. In the implementation described by Bergeron [BERG86a], the shadow volume—and hence each of its shadow polygons—is capped on one end by the original object polygon and on the other end by a scaled copy of the object polygon whose normal has been inverted. This scaled copy is located at a distance from the light beyond which its attenuated energy density is assumed to be negligible. We can think of this distance as the light's *sphere of influence*. Any point outside of the sphere of influence is effectively in shadow and does not require any additional shadow processing. In fact, there is no need to generate a shadow volume for any object wholly outside the sphere of influence. We can generalize this approach to apply to nonuniformly radiating sources by considering a *region of influence*, for example by culling

Fig. 16.28 Shadow creation and display in the Atherton, Weiler, and Greenberg algorithm. (Images by Peter Atherton, Kevin Weiler, Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1978.)

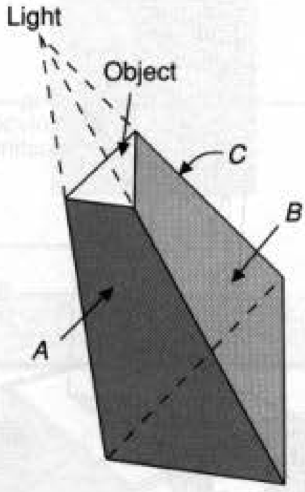


Fig. 16.30 A shadow volume is defined by a light source and an object.

objects outside of a light's flaps and cone. The shadow volume may also be further clipped to the view volume if the view volume is known in advance. The cap polygons are also treated as shadow polygons by the algorithm.

Shadow polygons are not rendered themselves, but are used to determine whether the other objects are in shadow. Relative to the observer, a front-facing shadow polygon (polygon *A* or *B* in Fig. 16.30) causes those objects behind it to be shadowed; a back-facing shadow polygon (polygon *C*) cancels the effect of a front-facing one. Consider a vector from the viewpoint *V* to a point on an object. The point is in shadow if the vector intersects more front-facing than back-facing shadow polygons. Thus, points *A* and *C* in Fig. 16.31(a) are in shadow. This is the only case in which a point is shadowed when *V* is not shadowed; therefore, point *B* is lit. If *V* is in shadow, there is one additional case in which a point is shadowed: when all the back-facing shadow polygons for the object polygons shadowing the eye have not yet been encountered. Thus, points *A*, *B*, and *C* in Fig. 16.31(b) are in shadow,

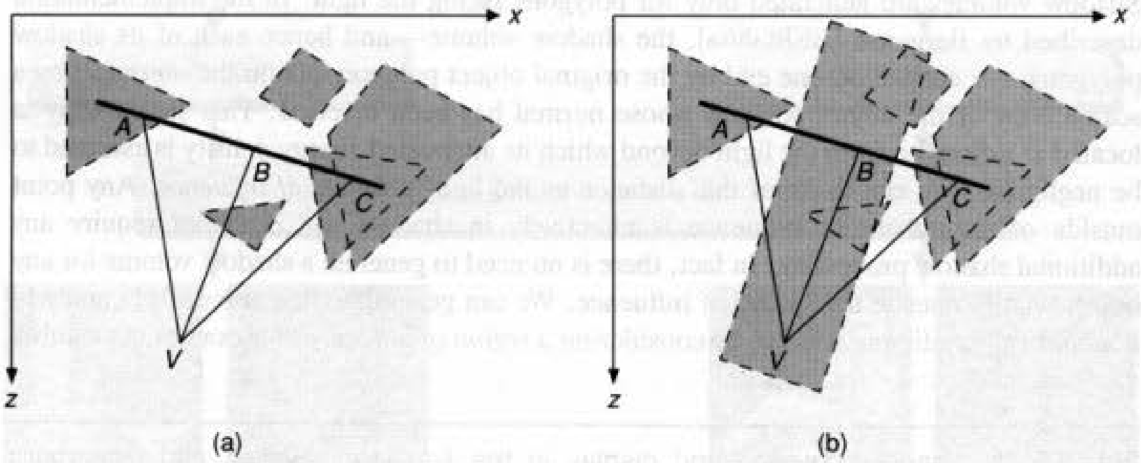


Fig. 16.31 Determining whether a point is in shadow for a viewer at *V*. Dashed lines define shadow volumes (shaded in gray). (a) *V* is not in shadow. Points *A* and *C* are shadowed; point *B* is lit. (b) *V* is in shadow. Points *A*, *B*, and *C* are shadowed.

even though the vector from V to B intersects the same number of front-facing and back-facing shadow polygons as it does in part (a).

We can compute whether a point is in shadow by assigning to each front-facing (relative to the viewer) shadow polygon a value of $+1$ and to each back-facing shadow polygon a value of -1. A counter is initially set to the number of shadow volumes that contain the eye and is incremented by the values associated with all shadow polygons between the eye and the point on the object. The point is in shadow if the counter is positive at the point. The number of shadow volumes containing the eye is computed only once for each viewpoint, by taking the negative of the sum of the values of all shadow polygons intercepted by an arbitrary projector from the eye to infinity.

Although is is possible to compute a shadow volume for each polygon, we can take advantage of object coherence by computing a single shadow volume for each connected polyhedron. This can be accomplished by generating shadow polygons from only those edges that are silhouette edges relative to the light source; these are the contour edges relative to the light source (as defined in Section 15.3.2).

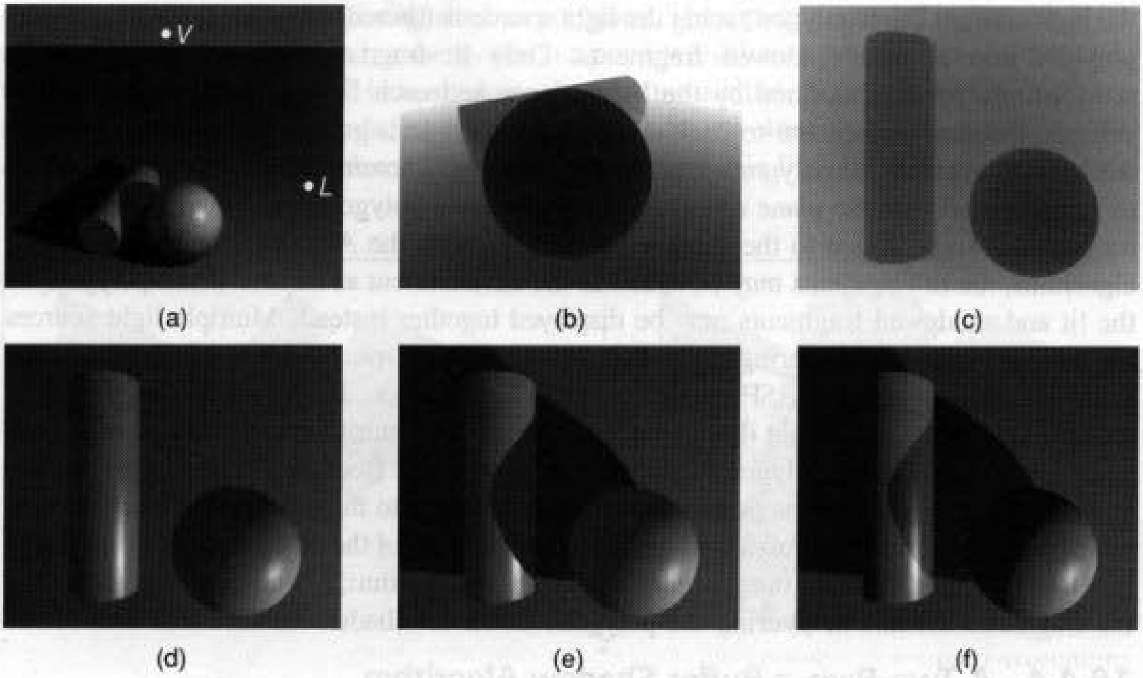
Multiple light sources can be handled by building a separate set of shadow volumes for each light source, marking the volume's shadow polygons with their light source identifier, and keeping a separate counter for each light source. Brotman and Badler [BROT84] have implemented a z-buffer version of the shadow-volume algorithm, and Bergeron [BERG86a] discusses a scan-line implementation that efficiently handles arbitrary polyhedral objects containing nonplanar polygons.

Chin and Feiner [CHIN89] describe an object-precision algorithm that builds a single shadow volume for a polygonal environment, using the BSP-tree solid modeling representation discussed in Section 12.6.4. Polygons are processed in front-to-back order relative to the light source. Each polygon facing the light source is filtered down the tree, dividing the polygon into lit and shadowed fragments. Only lit fragments cast shadows, so the semi-infinite pyramid defined by the light source and each lit fragment is added to the volume. Because of the front-to-back order, every polygon is guaranteed not to lie between the light source and the polygons processed previously. Therefore, since no polygon needs to be compared with the plane of a previously processed polygon, the polygons themselves do not need to be added to the shadow volume. As with the Atherton–Weiler–Greenberg algorithm, the lit fragments may be added to the environment as surface-detail polygons or the lit and shadowed fragments may be displayed together instead. Multiple light sources are accommodated by filtering the polygon fragments of one shadow-volume BSP tree down the shadow-volume BSP tree of the next light source. Each fragment is tagged to indicate the light sources that illuminate it, allowing the resulting fragmented environment to be displayed with any polygon visible-surface algorithm. Because of the shadow volume representation, lights may be positioned anywhere relative to the objects; thus, sectoring is not necessary. Several optimizations and a parallel version of the algorithm are discussed in [CHIN90]. Color Plate III.6(a) is rendered with the algorithm; Color Plate III.6(b) shows the fragments created in filtering the polygons down the shadow-volume BSP tree.

16.4.4 A Two-Pass z-Buffer Shadow Algorithm

Williams [WILL78] developed a shadow-generation method based on two passes through a z-buffer algorithm, one for the viewer and one for the light source. His algorithm, unlike the two-pass algorithm of Section 16.4.2, determines whether a surface is shadowed by using image-precision calculations. Figure 16.32(a) shows an overview of an environment lit by a light at L; a shadowless image from viewpoint V is shown in Fig. 16.32(d). The algorithm begins by calculating and storing just the z-buffer for the image from the viewpoint of the light (Fig. 16.32b). In Fig. 16.32(b), increasing intensities represent increasing distance. Next, the z-buffer (Fig. 16.32c) and the image (Fig. 16.32e) are calculated from the viewpoint of the observer using a z-buffer algorithm with the following modification. Whenever a pixel is determined to be visible, its object-precision coordinates in the observer's view (x_0, y_0, z_0) are transformed into coordinates in the light source's view (x'_0, y'_0, z'_0). The transformed coordinates x'_0 and y'_0 are used to select the value z_L in the light source's z-buffer to be compared with the transformed value z'_0. If z_L is closer to the light than is z'_0, then there is something blocking the light from the point, and the pixel is shaded as being in shadow; otherwise the point is visible from the light and it is shaded as lit. In analogy to texture mapping, we can think of the light's z-buffer as a *shadow map*. Multiple light sources can be accommodated by use of a separate shadow map for each light source.

Like the regular z-buffer visible-surface algorithm, this algorithm requires that each rendered pixel be shaded. Here, this means that shadow calculations must be performed for the pixel, even if it is ultimately painted over by closer objects. Williams has suggested a variation on his algorithm that exploits the ease with which the z-buffer algorithm can interleave visible-surface determination with illumination and shading, and eliminates shadow calculations for obscured objects. Rather than computing just the shadow map first, the modified algorithm also computes the regular shaded image from the observer's point of view (Fig. 16.32d), along with its z-buffer (all these computations can use conventional z-buffer–based hardware). Shadows are then added using a postprocess that is linear in the



(a) (b) (c)

(d) (e) (f)

Fig. 16.32 z-buffer shadow-generation method. (a) Overview. (b) Light's z-buffer. (c) Observer's z-buffer. (d) Observer's image. (e) Observer's image with shadows. (f) Observer's image with post-processed shadows. (By David Kurlander, Columbia University.)

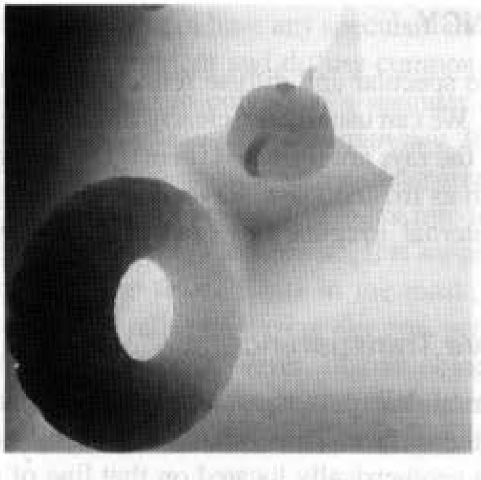


Fig. 16.33 Shadow map used to create Color Plate II.36. (Copyright © 1990, Pixar. Shadow map rendered by Thomas Williams and H. B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

number of pixels in the image to produce Fig. 16.32(f). The same transformation and comparison operation as before are performed for each pixel in the observer's image. If z_L is closer to the light than is z'_0, then the pixel's previously computed shade in the observer's image is darkened. Although this approach is significantly more efficient than is the original algorithm, it results in artifacts; most noticeably, shadowed objects will have (darkened) specular highlights, even though there should be no specular highlights on an object that is shielded from the light source. In addition, the z_0 to be transformed is at object precision in the first version, but at the typically lower z-buffer precision here. (See Exercise 16.15.)

Unlike the other shadow algorithms discussed so far, Williams's algorithm makes it especially easy to generate shadows for any object that can be scan-converted, including curved surfaces. Because all operations are performed in image precision, however, allowance must be made for the limited numerical precision. For example, the transformation from z_0 to z'_0 should also move z'_0 a little closer to the light source, to avoid having a visible point cast a shadow on itself. Like the z-buffer visible-surface algorithm from which it is constructed, this shadow algorithm is prone to aliasing. Williams describes how filtering and dithering can reduce the effects of aliasing. Reeves, Salesin, and Cook [REEV87] demonstrate improvements using *percentage closer filtering*. Each z'_0 is compared with values in a region of the shadow map, and the percentage of closer values determines the amount of shadowing. This improved algorithm was used to render Color Plates D, F, and II.36. Figure 16.33 shows the shadow map used to create Color Plate II.36.

16.4.5 Global Illumination Shadow Algorithms

Ray-tracing and radiosity algorithms have been used to generate some of the most impressive pictures of shadows in complex environments. Simple ray tracing has been used to model shadows from point light sources, whereas more advanced versions allow extended light sources. Both are discussed in Section 16.12. Radiosity methods, discussed in Section 16.13, model light sources as light-emitting surfaces that may have the same geometry as any other surface; thus, they implicitly support extended light sources.

16.5 TRANSPARENCY

Much as surfaces can have specular and diffuse reflection, those that transmit light can be transparent or translucent. We can usually see clearly through *transparent* materials, such as glass, although in general the rays are refracted (bent). Diffuse transmission occurs through *translucent* materials, such as frosted glass. Rays passing through translucent materials are jumbled by surface or internal irregularities, and thus objects seen through translucent materials are blurred.

16.5.1 Nonrefractive Transparency

The simplest approach to modeling transparency ignores refraction, so light rays are not bent as they pass through the surface. Thus, whatever is visible on the line of sight through a transparent surface is also geometrically located on that line of sight. Although refractionless transparency is not realistic, it can often be a more useful effect than refraction. For example, it can provide a distortionless view through a surface, as depicted in Color Plate III.7. As we have noted before, total photographic realism is not always the objective in making pictures.

Two different methods have been commonly used to approximate the way in which the colors of two objects are combined when one object is seen through the other. We shall refer to these as *interpolated* and *filtered* transparency.

Interpolated transparency. Consider what happens when transparent polygon 1 is between the viewer and opaque polygon 2, as shown in Fig. 16.34. *Interpolated transparency* determines the shade of a pixel in the intersection of two polygons' projections by linearly interpolating the individual shades calculated for the two polygons:

$$I_\lambda = (1 - k_{t_1})I_{\lambda 1} + k_{t_1} I_{\lambda 2}. \tag{16.25}$$

The *transmission coefficient* k_{t_1} measures the *transparency* of polygon 1, and ranges between 0 and 1. When k_{t_1} is 0, the polygon is opaque and transmits no light; when k_{t_1} is 1, the polygon is perfectly transparent and contributes nothing to the intensity I_λ; The value $1 - k_{t_1}$ is called the polygon's *opacity*. Interpolated transparency may be thought of as modeling a polygon that consists of a fine mesh of opaque material through which other objects may be seen; k_{t_1} is the fraction of the mesh's surface that can be seen through. A totally transparent

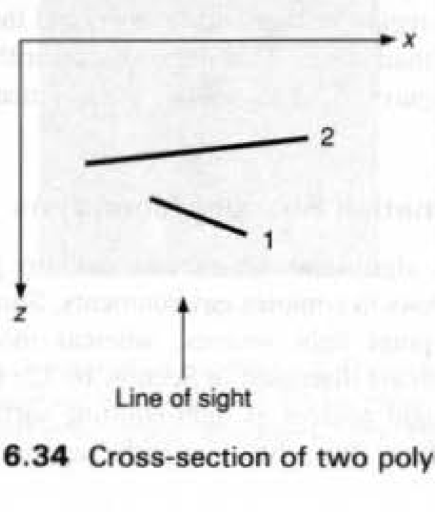


Fig. 16.34 Cross-section of two polygons.

polygon that is processed this way will not have any specular reflection. For a more realistic effect, we can interpolate only the ambient and diffuse components of polygon 1 with the full shade of polygon 2, and then add in polygon 1's specular component [KAY79b].

Another approach, often called *screen-door transparency,* literally implements a mesh by rendering only some of the pixels associated with a transparent object's projection. The low-order bits of a pixel's (x, y) address are used to index into a transparency bit mask. If the indexed bit is 1, then the pixel is written; otherwise, it is suppressed, and the next closest polygon at that pixel is visible. The fewer 1 bits in the mask, the more transparent the object's appearance. This approach relies on having our eyes perform spatial integration to produce interpolated transparency. Note, however, that an object fully obscures any other object drawn with the same transparency mask and that other undesirable interactions between masks are difficult to avoid.

Filtered transparency. *Filtered transparency* treats a polygon as a transparent filter that selectively passes different wavelengths; it can be modeled by

$$I_\lambda = I_{\lambda 1} + k_{t1} O_{t\lambda} I_{\lambda 2}, \tag{16.26}$$

where $O_{t\lambda}$ is polygon 1's *transparency color*. A colored filter may be modeled by choosing a different value of $O_{t\lambda}$ for each λ (but see Section 16.9). In either interpolated or filtered transparency, if additional transparent polygons are in front of these polygons, then the calculation is invoked recursively for polygons in back-to-front order, each time using the previously computed I_λ as $I_{\lambda 2}$.

Implementing transparency. Several visible-surface algorithms can be readily adapted to incorporate transparency, including scan-line and list-priority algorithms. In list-priority algorithms, the color of a pixel about to be covered by a transparent polygon is read back and used in the illumination model while the polygon is being scan-converted.

Most z-buffer–based systems support screen-door transparency because it allows transparent objects to be intermingled with opaque objects and to be drawn in any order. Adding transparency effects that use Eqs. (16.25) or (16.26) to the z-buffer algorithm is more difficult, because polygons are rendered in the order in which they are encountered. Imagine rendering several overlapping transparent polygons, followed by an opaque one. We would like to slip the opaque polygon behind the appropriate transparent cnes. Unfortunately, the z-buffer does not store the information needed to determine which transparent polygons are in front of the opaque polygon, or even the polygons' relative order. One simple, although incorrect, approach is to render transparent polygons last, combining their colors with those already in the frame buffer, but not modifying the z-buffer; when two transparent polygons overlap, however, their relative depth is not taken into account.

Mammen [MAMM89] describes how to render transparent objects properly in back-to-front order in a z-buffer-based system through the use of multiple rendering passes and additional memory. First, all the opaque objects are rendered using a conventional z-buffer. Then, transparent objects are processed into a separate set of buffers that contain, for each pixel, a transparency value and a flag bit, in addition to the pixel's color and z value. Each flag bit is initialized to off and each z value is set to the closest possible value. If a transparent object's z value at a pixel is closer than the value in the opaque z-buffer, but is more distant than that in the transparent z-buffer, then the color, z value, and transparency are saved in the transparent buffers, and the flag bit is set. After all objects have been processed, the transparent object buffers contain information for the most distant transparent object at each pixel whose flag bit is set. Information for flagged pixels is then blended with that in the original frame buffer and z-buffer. A flagged pixel's transparency z-value replaces that in the opaque z-buffer and the flag bit is reset. This process is repeated to render successively closer objects at each pixel. Color Plate III.7 was made using this algorithm.

Kay and Greenberg [KAY79b] have implemented a useful approximation to the increased attenuation that occurs near the silhouette edges of thin curved surfaces, where light passes through more material. They define k_t in terms of a nonlinear function of the z component of the surface normal after perspective transformation,

$$k_t = k_{t_{min}} + (k_{t_{max}} - k_{t_{min}})(1 - (1 - z_N)^m), \tag{16.27}$$

where $k_{t_{min}}$ and $k_{t_{max}}$ are the object's minimum and maximum transparencies, z_N is the z component of the normalized surface normal at the point for which k_t is being computed, and m is a power factor (typically 2 or 3). A higher m models a thinner surface. This new value of k_t may be used as k_{t_1} in either Eq. (16.25) or (16.26).

16.5.2 Refractive Transparency

Refractive transparency is significantly more difficult to model than is nonrefractive transparency, because the geometrical and optical lines of sight are different. If refraction is considered in Fig. 16.35, object A is visible through the transparent object along the line of sight shown; if refraction is ignored, object B is visible. The relationship between the angle of incidence θ_i and the angle of refraction θ_t is given by Snell's law

$$\frac{\sin \theta_i}{\sin \theta_t} = \frac{\eta_{t\lambda}}{\eta_{i\lambda}}, \tag{16.28}$$

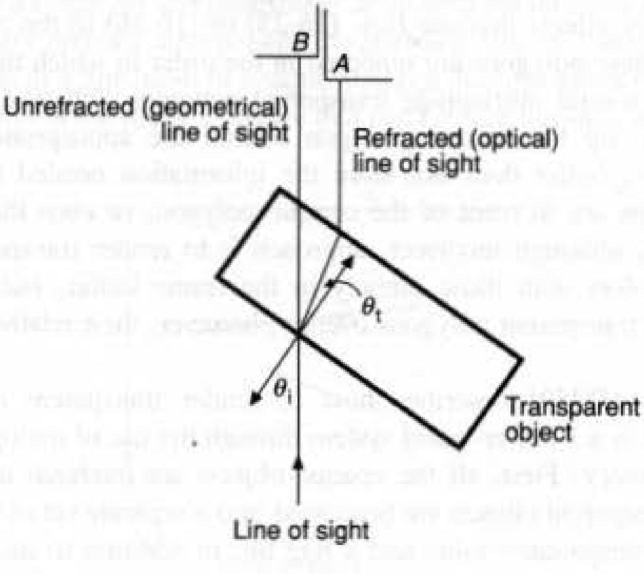

B
A
Unrefracted (geometrical) line of sight
Refracted (optical) line of sight
θ_t
θ_i
Transparent object
Line of sight

Fig. 16.35 Refraction.

where $\eta_{i\lambda}$ and $\eta_{t\lambda}$ are the *indices of refraction* of the materials through which the light passes. A material's index of refraction is the ratio of the speed of light in a vacuum to the speed of light in the material. It varies with the wavelength of the light and even with temperature. A vacuum has an index of refraction of 1.0, as does the atmosphere to close approximation; all materials have higher values. The index of refraction's wavelength-dependence is evident in many instances of refraction as *dispersion*—the familiar, but difficult to model, phenomenon of refracted light being spread into its spectrum [THOM86; MUSG89].

Calculating the refraction vector. The unit vector in the direction of refraction, $\overline{T}$, can be calculated as

$$\overline{T} = \sin \theta_t \, \overline{M} - \cos \theta_t \, \overline{N}, \tag{16.29}$$

where $\overline{M}$ is a unit vector perpendicular to $\overline{N}$ in the plane of the incident ray $\overline{I}$ and $\overline{N}$ [HECK84] (Fig. 16.36). Recalling the use of $\overline{S}$ in calculating the reflection vector $\overline{R}$ in Section 16.1.4, we see that $\overline{M} = (\overline{N} \cos \theta_i - \overline{I}) \,/\, \sin \theta_i$. By substitution,

$$\overline{T} = \frac{\sin \theta_t}{\sin \theta_i}(\overline{N} \cos \theta_i - \overline{I}) - \cos \theta_t \, \overline{N}. \tag{16.30}$$

If we let $\eta_{r\lambda} = \eta_{i\lambda} \,/\, \eta_{t\lambda} = \sin \theta_t \,/\, \sin \theta_i$, then after rearranging terms

$$\overline{T} = (\eta_{r\lambda} \cos \theta_i - \cos \theta_t) \, \overline{N} - \eta_{r\lambda} \, \overline{I}. \tag{16.31}$$

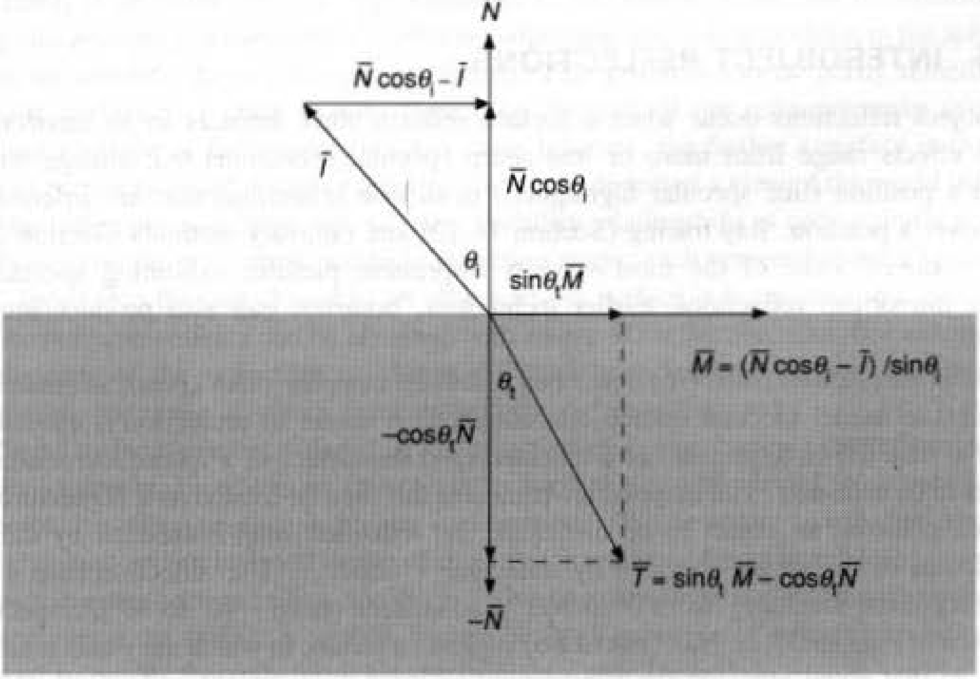

$\overline{N}$
$\overline{N}\cos\theta_i - \overline{I}$
$\overline{I}$
$\overline{N}\cos\theta_i$
θ_i
$\sin\theta_t\overline{M}$
$\overline{M} = (\overline{N}\cos\theta_i - \overline{I})\,/\sin\theta_i$
θ_t
$-\cos\theta_t\overline{N}$
$\overline{T} = \sin\theta_t\,\overline{M} - \cos\theta_t\overline{N}$
$-\overline{N}$

Fig. 16.36 Calculating the refraction vector.

Note that $\cos\theta_i$ is $\overline{N} \cdot \overline{I}$, and $\cos\theta_t$ can be computed as

$$\cos\theta_t = \sqrt{1 - \sin^2\theta_t} = \sqrt{1 - \eta_{r\lambda}^2 \sin^2\theta_i} = \sqrt{1 - \eta_{r\lambda}^2(1 - (\overline{N} \cdot \overline{I})^2)}. \quad (16.32)$$

Thus,

$$\overline{T} = \left(\eta_{r\lambda}(\overline{N} \cdot \overline{I}) - \sqrt{1 - \eta_{r\lambda}^2(1 - (\overline{N} \cdot \overline{I})^2)} \right)\overline{N} - \eta_{r\lambda}\overline{I}. \quad (16.33)$$

Total internal reflection. When light passes from one medium into another whose index of refraction is lower, the angle θ_t of the transmitted ray is greater than the angle θ_i. If θ_i becomes sufficiently large, then θ_t exceeds 90° and the ray is reflected from the interface between the media, rather than being transmitted. This phenomenon is known as *total internal reflection*, and the smallest θ_i at which it occurs is called the *critical angle*. You can observe total internal reflection easily by looking through the front of a filled fish tank and trying to see your hand through a side wall. When the viewing angle is greater than the critical angle, the only visible parts of your hand are those pressed firmly against the tank, with no intervening layer of air (which has a lower index of refraction than glass or water). The critical angle is the value of θ_i at which $\sin\theta_t$ is 1. If $\sin\theta_t$ is set to 1 in Eq. (16.28), we can see that the critical angle is $\sin^{-1}(\eta_{t\lambda} / \eta_{i\lambda})$. Total internal reflection occurs when the square root in Eq. (16.33) is imaginary.

Section 16.12 discusses the use of Snell's law in modeling refractive transparency with ray tracing; translucency is treated in Sections 16.12.4 and 16.13. An approximation of refraction can also be incorporated into renderers that proceed in back-to-front order [KAY79b].

16.6 INTEROBJECT REFLECTIONS

Interobject reflections occur when a surface reflects other surfaces in its environment. These effects range from more or less sharp specular reflections that change with the viewer's position (like specular highlights), to diffuse reflections that are insensitive to the viewer's position. Ray tracing (Section 16.12) and radiosity methods (Section 16.13) have produced some of the most visually impressive pictures exhibiting specular and diffuse interobject reflections; earlier techniques, however, can also produce attractive results.

Blinn and Newell [BLIN76] developed *reflection mapping* (also known as *environment mapping*) to model specular interobject reflection. A center of projection is chosen from which to map the environment to be reflected onto the surface of a sphere surrounding the objects to be rendered. The mapped environment can then be treated as a 2D texture map. At each point on an object to be displayed, the reflection map is indexed by the polar coordinates of the vector obtained by reflecting $\overline{V}$ about $\overline{N}$. The reflection map's *x* and *y* axes represent longitude (from 0° to 360°) and latitude (from −90° to 90°), respectively, as shown in Fig. 16.37(a). Hall [HALL86] suggests a variant in which the *y* axis is sin (latitude), so that equal areas on the sphere map to equal areas on the reflection map (Fig.

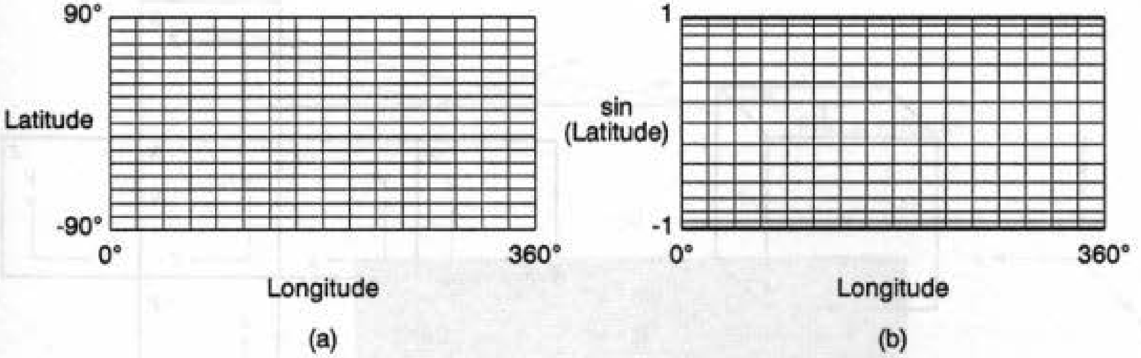
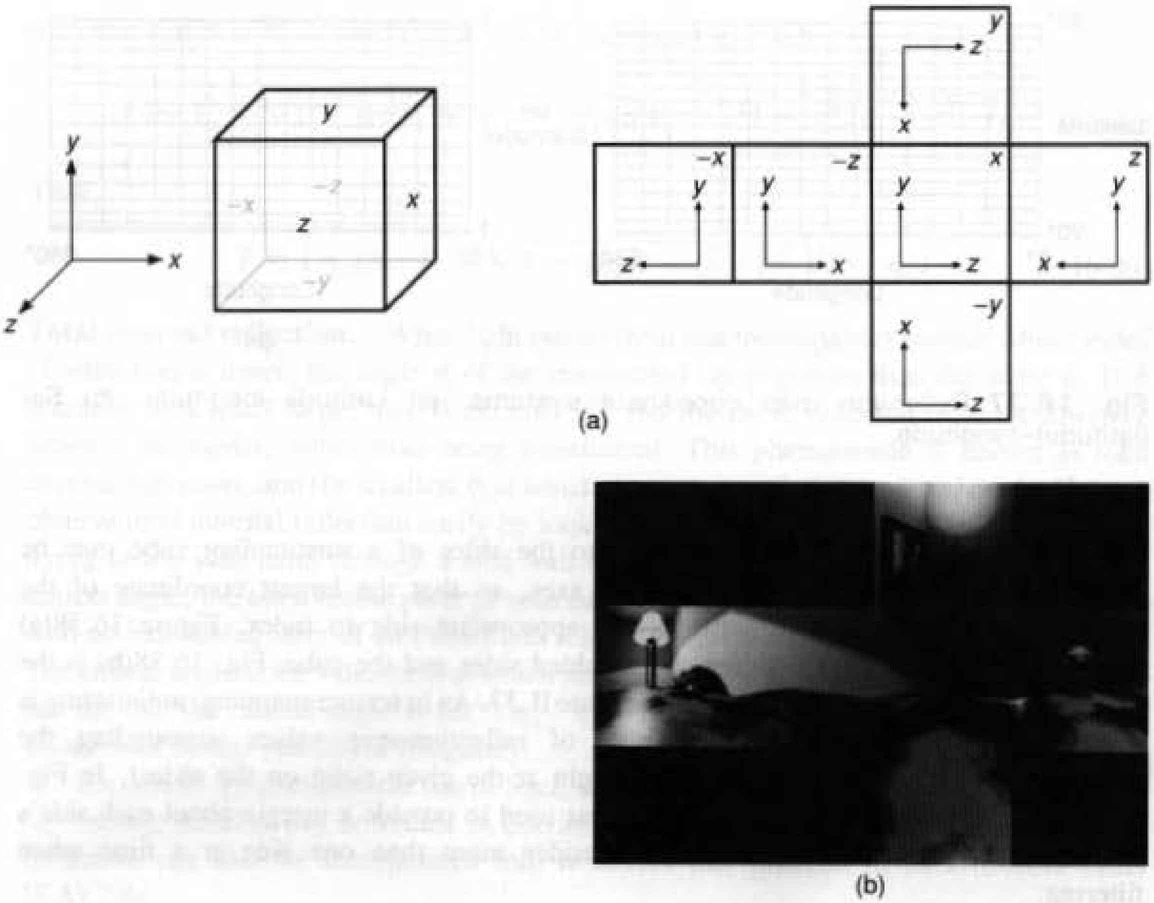


Fig. 16.37 Reflection map coordinate systems. (a) Latitude–longitude. (b) Sin (latitude)–longitude.

16.37b). Alternatively, six projections onto the sides of a surrounding cube may be used. The cube is aligned with the WC axes, so that the largest coordinate of the normalized reflection vector indicates the appropriate side to index. Figure 16.38(a) shows the correspondence between the unfolded sides and the cube; Fig. 16.38(b) is the reflection map used for the teapot in Color Plate II.37. As in texture mapping, antialiasing is accomplished by filtering some number of reflection-map values surrounding the indexed value to determine the reflected light at the given point on the object. In Fig. 16.38(b) an angle slightly wider than 90° was used to provide a margin about each side's borders that helps avoid the need to consider more than one side at a time when filtering.

Although reflection mapping can be used to produce a number of useful effects [GREE86], it provides only an approximation to the correct reflection information. By taking into account just the surface's reflection direction and not its position in the sphere, it models an infinitely large environment sphere. This problem can be partly remedied by using the surface's position to help determine the part of the reflection map to index, modeling a sphere of finite size. In either case, however, the farther a surface is from the center of projection used to create the map, the more distorted a view of the world it shows, since the reflection map takes into account visibility relationships at only a single point. A useful compromise is to create multiple reflection maps, each centered about a key object, and to index into the one closest to an object whose surface is being mapped. Simple but effective reflection effects can be obtained with even a 1D reflection map. For example, the y component of the reflection of $\overline{V}$ may be used to index into an array of intensities representing the range of colors from ground through horizon to sky.

Planar surfaces present difficulties for reflection mapping, because the reflection angle changes so slowly. If reflections from a planar surface are to be viewed from only a single viewpoint, however, another technique can be used. The viewpoint is reflected about the surface's plane and an inverted image of the scene is rendered from the reflected viewpoint, as shown in cross-section in Fig. 16.38(c). This image can then be merged with the original image wherever the surface is visible. Figure 16.38(d) was created with this technique and was used to render the reflections on the floor in Color Plate II.37.



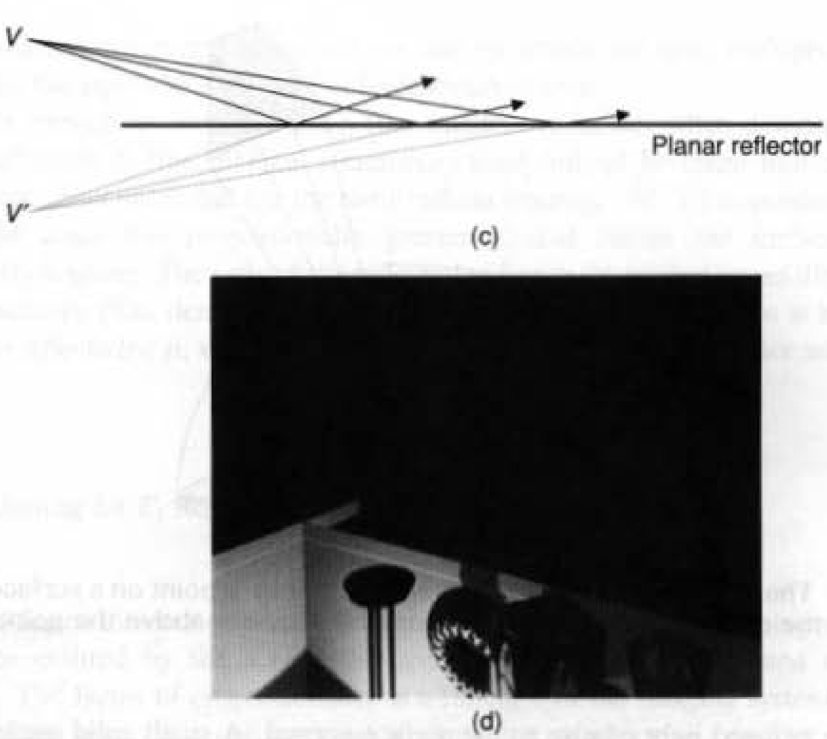
(a)

(b)

16.7 PHYSICALLY BASED ILLUMINATION MODELS

The illumination models discussed in the previous sections are largely the result of a common-sense, practical approach to graphics. Although the equations used approximate some of the ways light interacts with objects, they do not have a physical basis. In this section, we discuss physically based illumination models, relying in part on the work of Cook and Torrance [COOK82].

Thus far, we have used the word *intensity* without defining it, referring informally to the intensity of a light source, of a point on a surface, or of a pixel. It is time now to formalize our terms by introducing the radiometric terminology used in the study of thermal radiation, which is the basis for our understanding of how light interacts with objects [NICO77; SPARR78; SIEG81; IES87]. We begin with *flux*, which is the rate at which light energy is emitted and is measured in watts (W). To refer to the amount of flux emitted in or received from a given direction, we need the concept of a *solid angle*, which is the angle at the apex of a cone. Solid angle is measured in terms of the area on a sphere intercepted by a cone whose apex is at the sphere's center. A *steradian* (sr) is the solid angle of such a cone



(c)

(d)

Fig. 16.38 Reflection maps. (a) Cube reflection map layout. (b) Reflection map for teapot in Color Plate II.37. (c) Geometry of planar reflection. (d) Reflected image merged with floor in Color Plate II.37. (Copyright © 1990, Pixar. Images in (b) and (d) rendered by Thomas Williams and H. B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

that intercepts an area equal to the square of the sphere's radius r. If a point is on a surface, we are concerned with the hemisphere above it. Since the area of a sphere is $4\pi r^2$, there are $4\pi r^2 / 2r^2 = 2\pi$ sr in a hemisphere. Imagine projecting an object's shape onto a hemisphere centered about a point on the surface that serves as the center of projection. The solid angle ω subtended by the object is the area on the hemisphere occupied by the projection, divided by the square of the hemisphere's radius (the division eliminates dependence on the size of the hemisphere). Thus, for convenience, we often speak of solid angle in terms of the area projected on a unit sphere or hemisphere, as shown in Fig. 16.39.

Radiant intensity is the flux radiated into a unit solid angle in a particular direction and is measured in W / sr. When we used the word *intensity* in reference to a point source, we were referring to its radiant intensity.

Radiance is the radiant intensity per unit foreshortened surface area, and is measured in W / (sr · m^2). *Foreshortened surface area,* also known as *projected surface area,* refers to the projection of the surface onto the plane perpendicular to the direction of radiation. The foreshortened surface area is found by multiplying the surface area by cos θ_r, where θ_r is the

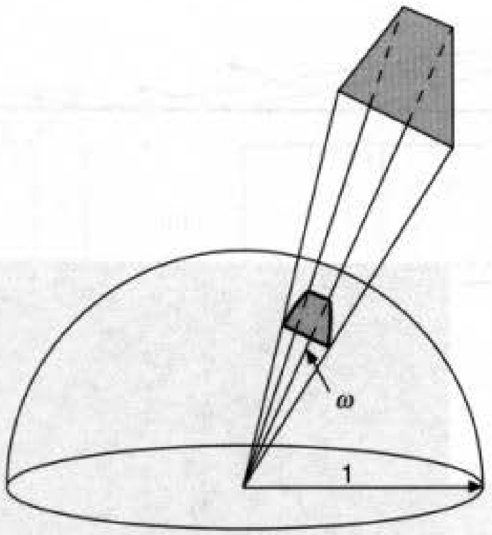


Fig. 16.39 The solid angle subtended by an object from a point on a surface is the area covered by the object's projection onto a unit hemisphere above the point.

angle of the radiated light relative to the surface normal. A small solid angle $d\omega$ may be approximated as the object's foreshortened surface area divided by the square of the distance from the object to the point at which the solid angle is being computed. When we used the word *intensity* in reference to a surface, we were referring to its radiance. Finally, *irradiance*, also known as *flux density*, is the incident flux per (unforeshortened) unit surface area and is measured in W / m^2.

In graphics, we are interested in the relationship between the light incident on a surface and the light reflected from and transmitted through that surface. Consider Fig. 16.40. The irradiance of the incident light is

$$E_i = I_i(\overline{N} \cdot \overline{L})\, d\omega_i, \tag{16.34}$$

where I_i is the incident light's radiance, and $\overline{N} \cdot \overline{L}$ is $\cos \theta_i$. Since irradiance is expressed per

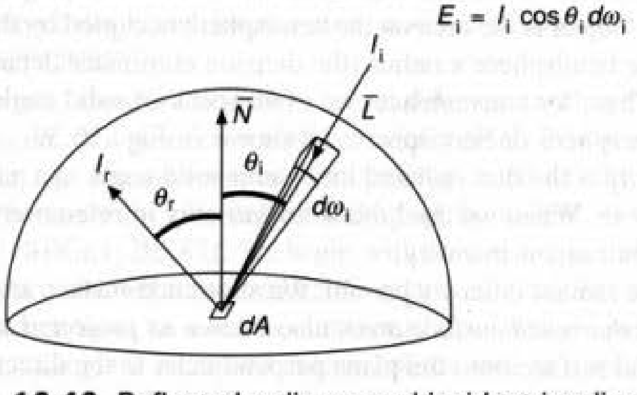


Fig. 16.40 Reflected radiance and incident irradiance.

unit area, whereas radiance is expressed per unit foreshortened area, multiplying by $\overline{N} \cdot \overline{L}$ converts it to the equivalent per unit unforeshortened area.

It is not enough to consider just I_i (the incident radiance) when determining I_r (the reflected radiance); E_i (the incident irradiance) must instead be taken into account. For example, an incident beam that has the same radiant intensity (W / sr) as another beam but a greater solid angle has proportionally greater E_i and causes the surface to appear proportionally brighter. The ratio of the reflected radiance (intensity) in one direction to the incident irradiance (flux density) responsible for it from another direction is known as the *bidirectional reflectivity*, ρ, which is a function of the directions of incidence and reflection,

$$\rho = \frac{I_r}{E_i}. \tag{16.35}$$

Thus, substituting for E_i from Eq. (16.34), we get

$$I_r = \rho I_i(\overline{N} \cdot \overline{L})d\omega_i. \tag{16.36}$$

The irradiance incident on a pixel in the image (the *image irradiance*) is proportional to the radiance emitted by the scene (the *scene radiance*) that is focused on the pixel [HORN79]. The factor of proportionality is a function of the imaging system being used.

As we have seen, it is conventional in computer graphics to consider bidirectional reflectivity as composed of diffuse and specular components. Therefore,

$$\rho = k_d\rho_d + k_s\rho_s, \tag{16.37}$$

where ρ_d and ρ_s are respectively the diffuse and specular bidirectional reflectivities, and k_d and k_s are respectively the diffuse and specular reflection coefficients introduced earlier in this chapter; $k_d + k_s = 1$. It is important to note that Eq. (16.37) is a useful approximation that is not applicable to all surfaces. For example, the lunar surface has a ρ that peaks in the direction of incidence [SIEG81]. During a full moon, when the sun, earth, and moon are nearly in line, this accounts for why the moon appears as a disk of roughly uniform intensity. If the moon had a Lambertian surface, it would, instead, reflect more light at its center than at its sides.

In addition to the effects of direct light-source illumination, we need to take into account illumination by light reflected from other surfaces. The lighting models discussed so far have assumed that this ambient light is equally incident from all directions, is independent of viewer position, and is not blocked by any nearby objects. Later in this chapter we shall discuss how to lift these restrictions. For now we retain them, modeling the ambient term as $\rho_a I_a$, where ρ_a is the nondirectional ambient reflectivity, and I_a is the incident ambient intensity.

The resulting illumination equation for n light sources is

$$I_r = \rho_a I_a + \sum_{1 \le j \le n} I_{ij}(\overline{N} \cdot \overline{L}_j)\, d\omega_{ij}(k_d\rho_d + k_s\rho_s). \tag{16.38}$$

To reinterpret the illumination model of Eq. (16.15) in terms of Eq. (16.38), we expressed diffuse reflectivity as the object's diffuse color, and specular reflectivity using the product of the object's specular color and a $\cos^n \alpha$ term. We have already noted some of the inadequacies of that specular reflectivity formulation. Now we shall examine how to replace it.

16.7.1 Improving the Surface Model

The Torrance–Sparrow model [TORR66; TORR67], developed by applied physicists, is a physically based model of a reflecting surface. Blinn was the first to adapt the Torrance–Sparrow model to computer graphics, giving the mathematical details and comparing it to the Phong model in [BLIN77a]; Cook and Torrance [COOK82] were the first to approximate the spectral composition of reflected light in an implementation of the model.

In the Torrance–Sparrow model, the surface is assumed to be an isotropic collection of planar microscopic facets, each a perfectly smooth reflector. The geometry and distribution of these *microfacets* and the direction of the light (assumed to emanate from an infinitely distant source, so that all rays are parallel) determine the intensity and direction of specular reflection as a function of I_p (the point light source intensity), $\overline{N}$, $\overline{L}$, and $\overline{V}$. Experimental measurements show a very good correspondence between the actual reflection and the reflection predicted by this model [TORR67].

For the specular component of the bidirectional reflectivity, Cook and Torrance use

$$\rho_s = \frac{F_\lambda}{\pi} \frac{DG}{(\overline{N} \cdot \overline{V})(\overline{N} \cdot \overline{L})}, \tag{16.39}$$

where D is a distribution function of the microfacet orientations, G is the *geometrical attenuation factor*, which represents the masking and shadowing effects of the microfacets on each other, and F_λ is the Fresnel term computed by Fresnel's equation (described later), which, for specular reflection, relates incident light to reflected light for the smooth surface of each microfacet. The π in the denominator is intended to account for surface roughness (but see [JOY88, pp. 227–230] for an overview of how the equation is derived). The $\overline{N} \cdot \overline{V}$ term makes the equation proportional to the surface area (and hence to the number of microfacets) that the viewer sees in a unit piece of foreshortened surface area, whereas the $\overline{N} \cdot \overline{L}$ term makes the equation proportional to the surface area that the light sees in a unit piece of foreshortened surface area.

16.7.2 The Microfacet Distribution Function

Since the microfacets are considered to be perfect specular reflectors, the model considers only those microfacets whose normals lie along the halfway vector $\overline{H}$, introduced in Section 16.1.4. Only a fraction D of the microfacets have this orientation. Torrance and Sparrow assumed a Gaussian distribution function for D in their original work. Blinn used the Trowbridge and Reitz distribution [TROW75], and Cook and Torrance used the Beckmann distribution function [BECK63]. Cook and Torrance point out the Beckmann distribution has a good theoretical basis and has no arbitrary constants, unlike the distributions used by Torrance and Sparrow, and by Blinn. The Beckmann distribution function for rough surfaces is

$$D = \frac{1}{4m^2 cos^4\beta} e^{-[(\tan \beta)/m]^2}, \tag{16.40}$$

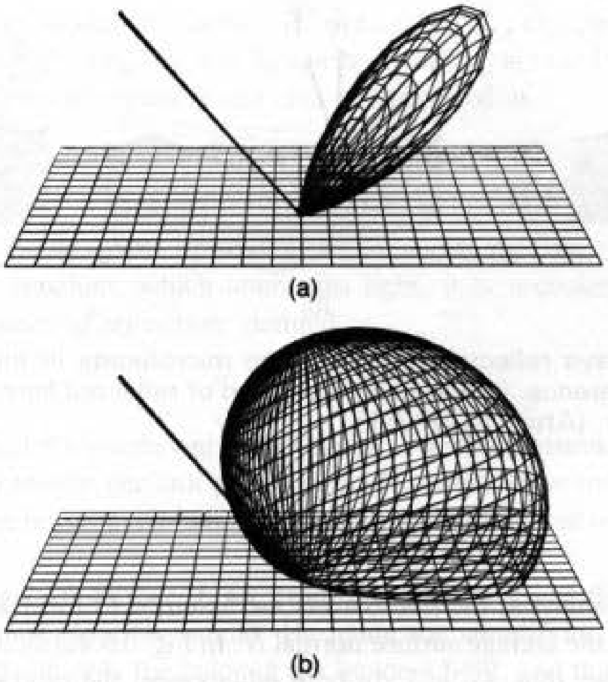


Fig. 16.41 Beckmann microfacet distribution function for (a) $m = 0.2$ and (b) $m = 0.6$. (From [COOK82] with permission. By Robert Cook, Program of Computer Graphics, Cornell University.)

where β is the angle between $\overline{N}$ and $\overline{H}$, and m is the root-mean-square slope of the microfacets.[1] When m is small, the microfacet slopes vary only slightly from the surface normal and, as we would expect, the reflection is highly directional (Fig. 16.41a). When m is large, the microfacet slopes are steep and the resulting rough surface spreads out the light it reflects (Fig. 16.41b). To model surfaces that have multiple scales of roughness, Cook and Torrance use a weighted sum of distribution functions,

$$D = \sum_{1 \le j \le n} w_j D(m_j), \tag{16.41}$$

where the sum of the weights w_j is 1.

16.7.3 The Geometrical Attenuation Factor

The model takes into account that some microfacets may shadow others. Torrance and Sparrow and Blinn discuss the calculation of G, considering three different situations. Figure 16.42(a) shows a microfacet whose incident light is totally reflected. Figure 16.42(b) shows a microfacet that is fully exposed to the rays, but has some reflected rays that are shielded by other microfacets. These shielded rays ultimately contribute to the diffuse reflection. Figure 16.42(c) shows a microfacet that is partially shielded from the light. The geometric attenuation factor G ranges from 0 (total shadowing) to 1 (no shadowing).

[1] Hall [HALL89] mentions that [COOK82] is missing the 4 in the denominator.

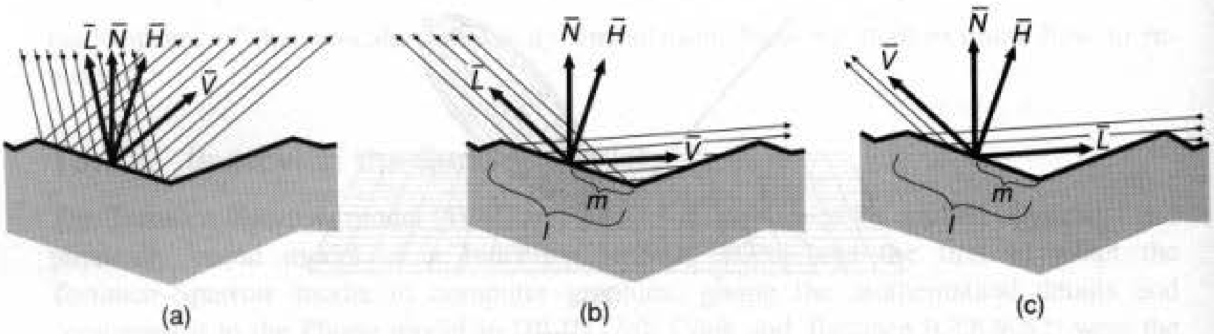


Fig. 16.42 Light rays reflecting from surface microfacets in the Torrance-Sparrow model. (a) No interference. (b) Partial interception of reflected light. (c) Partial interception of incident light. (After [BLIN77a].)

To simplify the analysis, the microfacets are assumed to form V-shaped grooves that are symmetric about the average surface normal $\overline{N}$. In Fig. 16.42(a), all the incident light is reflected to the viewer; thus, G_a is 1. In both other cases, the ratio of intercepted light is given by m / l, where l is the total area of the facet and m is the area whose reflected light is blocked (Fig. 16.42b) or that is itself blocked (Fig 16.42c) from the light. Therefore, $G_b = G_c = 1 - m / l$. Blinn gives a trigonometric derivation of the proportion of reflected light in Fig. 16.42(b) as

$$G_b = \frac{2(\overline{N} \cdot \overline{H})(\overline{N} \cdot \overline{V})}{(\overline{V} \cdot \overline{H})}. \qquad (16.42)$$

The ratio for the case in Fig. 16.42(c) follows by noticing that it is the same as the case in part (b), except that $\overline{L}$ and $\overline{V}$ trade places:

$$G_c = \frac{2(\overline{N} \cdot \overline{H})(\overline{N} \cdot \overline{L})}{(\overline{V} \cdot \overline{H})}. \qquad (16.43)$$

The denominator does not need to change because $(\overline{V} \cdot \overline{H}) = (\overline{L} \cdot \overline{H})$ by definition of $\overline{H}$ as the halfway vector between $\overline{V}$ and $\overline{L}$.

G is the minimum of the three values:

$$G = \min\left\{1, \frac{2(\overline{N} \cdot \overline{H})(\overline{N} \cdot \overline{V})}{(\overline{V} \cdot \overline{H})}, \frac{2(\overline{N} \cdot \overline{H})(\overline{N} \cdot \overline{L})}{(\overline{V} \cdot \overline{H})}\right\}. \qquad (16.44)$$

16.7.4 The Fresnel Term

The Fresnel equation for unpolarized light specifies the ratio of reflected light from a dielectric (nonconducting) surface as

$$F_\lambda = \frac{1}{2}\left(\frac{\tan^2(\theta_i - \theta_t)}{\tan^2(\theta_i + \theta_t)} + \frac{\sin^2(\theta_i - \theta_t)}{\sin^2(\theta_i + \theta_t)}\right) = \frac{1}{2}\,\frac{\sin^2(\theta_i - \theta_t)}{\sin^2(\theta_i + \theta_t)}\left(1 + \frac{\cos^2(\theta_i + \theta_t)}{\cos^2(\theta_i - \theta_t)}\right), \qquad (16.45)$$

where θ_i is the angle of incidence relative to $\overline{H}$ (i.e., $\cos^{-1}(\overline{L} \cdot \overline{H})$), and, as before, θ_t is the angle of refraction; $\sin \theta_t = (\eta_{i\lambda} / \eta_{t\lambda}) \sin \theta_i$, where $\eta_{i\lambda}$ and $\eta_{t\lambda}$ are the indices of refraction of the two media. The Fresnel equation can also be expressed as

$$F_\lambda = \frac{1}{2} \frac{(g - c)^2}{(g + c)^2} \left(1 + \frac{[c(g + c) - 1]^2}{[c(g - c) + 1]^2}\right), \tag{16.46}$$

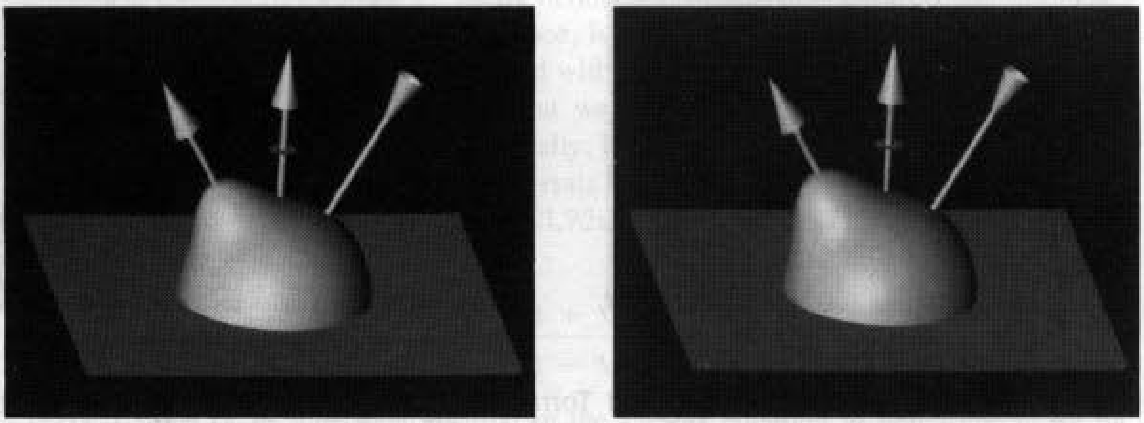
where $c = \cos \theta_i = \overline{L} \cdot \overline{H}$, $g^2 = \eta_\lambda^2 + c^2 - 1$, and $\eta_\lambda = \eta_{t\lambda} / \eta_{i\lambda}$.

In a conducting medium, which attenuates light, it is necessary to refer to $\overline{\eta}_\lambda$, the material's *complex index of refraction,* defined as

$$\overline{\eta}_\lambda = \eta_\lambda - i\kappa_\lambda, \tag{16.47}$$

where κ_λ is the material's *coefficient of extinction,* which measures the amount that the material attenuates intensity per unit path length. To simplify the reflection computations for conductors, κ_λ can be assumed to be zero and a single equivalent real value for η_λ can be determined.

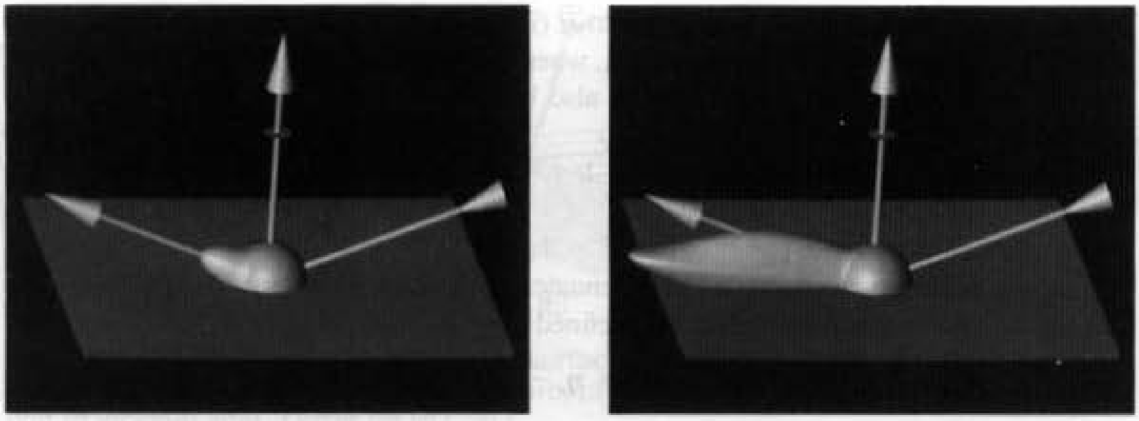
Blinn created Figs. 16.43 and 16.44, comparing the effects of the Phong illumination model and the Torrance–Sparrow model. He made the simplifying assumptions that the specular term depends on only the color of the incident light, and that the viewer and light source are both at infinity. Figures 16.43 and 16.44 show the reflected illumination from a surface for angles of incidence of 30° and 70°, respectively. In each figure, the vertical arrow represents the surface normal, the incoming arrow represents the direction of light rays, and the outgoing arrow represents the direction of reflection for a perfect reflector. The rounded part of each figure is the diffuse reflection, whereas the bump is the specular reflection. For the 30° case in Fig. 16.43, the models produce nearly similar results, but for the 70° case in Fig. 16.44, the Torrance–Sparrow model has much higher specular reflectance and the peak occurs at an angle greater than the angle of incidence. This so-called *off-specular peak* is observed in actual environments. Figure 16.45, also by Blinn, shows a marked difference in the visual effect of the two models as the light source moves away from the viewpoint to the side and then to the rear of a metallic sphere.



(a) Phong model

(b) Torrance-Sparrow model

Fig. 16.43 Comparison of Phong and Torrance–Sparrow illumination models for light at a 30° angle of incidence. (By J. Blinn [BLIN77a], courtesy of the University of Utah.)



(a) Phong model

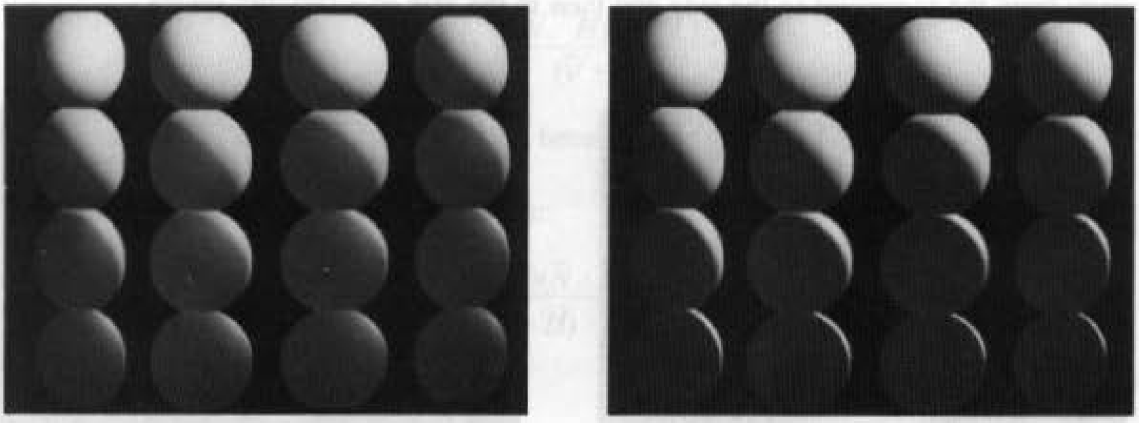
(b) Torrance-Sparrow model

Fig. 16.44 Comparison of Phong and Torrance–Sparrow illumination models for light at a 70° angle of incidence. (By J. Blinn [BLIN77a], courtesy of the University of Utah.)

The specular-reflection color shift. Like Blinn, Cook and Torrance use the Torrance–Sparrow surface model to determine the specular term. Unlike Blinn, however, they make the color of the specular reflection a function of the interaction between the material and the incident light, depending both on the light's wavelength and on its angle of incidence. This is correct because the Fresnel equation, Eq. (16.45), is responsible for a shift in the specular reflection color based on the angle the incident light makes with the microfacet normal $\overline{H}$, as shown in Fig. 16.46.

When the incident light is in the same direction as $\overline{H}$, then $\theta_i = 0$, so $c = 1$ and $g = \eta_\lambda$. Substituting these values in Eq. (16.46) yields the Fresnel term for $\theta_i = 0$,

$$F_{\lambda 0} = \left(\frac{\eta_\lambda - 1}{\eta_\lambda + 1}\right)^2. \qquad (16.48)$$



(a) Phong model

(b) Torrance-Sparrow model

Fig. 16.45 Comparison of Phong and Torrance–Sparrow illumination models for a metallic sphere illuminated by a light source from different directions. Differences are most apparent for back-lit cases (bottom rows). (By J. Blinn [BLIN77a], courtesy of the University of Utah.)

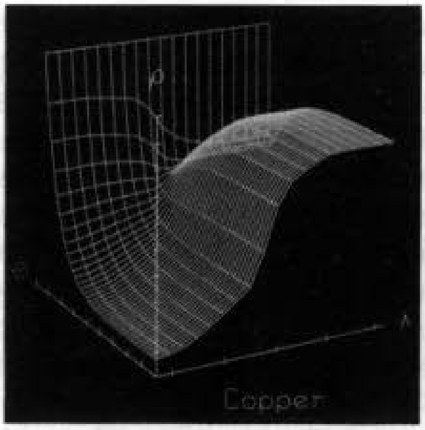


Fig. 16.46 Fresnel term for a copper mirror as a function of wavelength and angle of incidence. (By Robert Cook, Program of Computer Graphics, Cornell University.)

When the incident light grazes the surface of the microfacet, then $\theta_i = \pi / 2$, so $c = 0$. Substituting in Eq. (16.46) yields the Fresnel term for $\theta_i = \pi / 2$,

$$F_{\lambda\pi/2} = 1. \tag{16.49}$$

Thus, if light is normal to the surface, then $F_{\lambda 0}$, and hence the specular reflectance ρ_s, are functions of the surface's index of refraction, which in turn varies with the wavelength. When the light grazes the surface (and when the viewer is looking 180° opposite from the light, since only microfacets with normals of $\overline{H}$ are considered), $F_{\lambda\pi/2}$, and hence the specular reflection, are both 1. Specular reflectance depends on η_λ for all θ_i except $\pi / 2$. For metals, essentially all reflection occurs at the surface and is specular. Only at extreme glancing angles is the specular reflection not influenced by the object's material. Notice how this differs from the Phong specular-reflection term, which was always unaffected by the object color. Cook and Torrance point out that the monochromatic Phong specular term is a good model of plastic, which is colored by pigment particles embedded in a transparent substrate. The presence of these particles causes the diffuse reflection, resulting from light bouncing around in the substrate, to be a function of the pigment and light colors. Specular reflection occurs from the transparent surface, however, and therefore is unaffected by the pigment color. This is why objects rendered with a Phong illumination model look plastic.

If the indices of refraction at different wavelengths are known, they may be used directly in the Fresnel equation. More typically, however, they are not known. Reflectance, however, has been measured for many materials with $\theta_i = 0$ at a variety of wavelengths, as recorded in sources such as [TOUL70; TOUL72a; TOUL72b]. In this case, each η_λ may be determined from Eq. (16.48), as

$$\eta_\lambda = \frac{1 + \sqrt{F_{\lambda 0}}}{1 - \sqrt{F_{\lambda 0}}}. \tag{16.50}$$

A derived value of η_λ may then be used in the Fresnel equation to determine F_λ for an arbitrary θ_i. Rather than perform this calculation for each value of λ, Cook and Torrance simplify the computationally expensive color shift calculation by calculating $F_{\mathrm{avg}\theta_i}$ for η_{avg}, the average index of refraction for average normal reflectance. They use the value computed for $F_{\text{avg}\theta_i}$ to interpolate between the color of the material at $\theta_i = 90°$ and the color of the material at $\theta_i = 0°$ for each component of the color model. Because $F_{\lambda\pi/2}$ is always 1, the color of the material at $\theta_i = 90°$ is the color of the light source. Thus, when the light grazes the surface at a 90° angle of incidence, the color of the reflected light is that of the incident light. Using the RGB system, we call the red component of the material at $\theta_i = 0°$, Red_0, and the red component of the light, $\text{Red}_{\pi/2}$. Red_0 is calculated by integrating the product of F_0, the spectrum of the incident light, and the color-matching curves of Fig. 13.22, and applying the inverse of matrix M of Eq. (13.24) to the result. $\text{Red}_{\pi/2}$ is obtained by applying the inverse of M to Eq. (13.18). The approximation computes the color of the material at θ_i as

$$\text{Red}_{\theta i} = \text{Red}_0 + (\text{Red}_{\pi/2} - \text{Red}_0)\,\frac{\max(0,\ F_{\text{avg}\theta_i} - F_{\text{avg}0})}{F_{\text{avg}\pi/2} - F_{\text{avg}0}}. \qquad (16.51)$$

$\text{Red}_{\theta i}$ is then used in place of F_λ in Eq. (16.39). Because the approximation takes the light's spectrum into account, Eq. (16.38) must be modified to multiply the specular term by a wavelength-independent intensity scale factor, instead of by the light's spectrum. Hall [HALL89] suggests an alternative approximation that interpolates a value for $F_{\lambda\theta_i}$, given $F_{\lambda0}$. Since $F_{\lambda\pi/2}$ is always 1 (as is $F_{\text{avg}\pi/2}$ in Eq. 16.51),

$$F_{\lambda\theta_i} = F_{\lambda0} + (1 - F_{\lambda0})\,\frac{\max\,(0,\ F_{\text{avg}\theta_i} - F_{\text{avg}0})}{1 - F_{\text{avg}0}}. \qquad (16.52)$$

Color Plate III.8 shows two copper vases rendered with the Cook–Torrance model, both of which use the bidirectional reflectance of copper for the diffuse term. The first models the specular term using the reflectance of a vinyl mirror and represents results similar to those obtained with the original Phong illumination model of Eq. (16.14). The second models the specular term with the reflectance of a copper mirror. Note how accounting for the dependence of the specular highlight color on both angle of incidence and surface material produces a more convincing image of a metallic surface.

In general, the ambient, diffuse, and specular components are the color of the material for both dielectrics and conductors. Composite objects, such as plastics, typically have diffuse and specular components that are different colors. Metals typically show little diffuse reflection and have a specular component color that ranges between that of the metal and that of the light source as θ_i approaches 90°. This observation suggests a rough approximation to the Cook–Torrance model that uses Eq. (16.15) with $O_{s\lambda}$ chosen by interpolating from a look-up table based on θ_i.

Further work. Kajiya [KAJI85] has generalized the Cook–Torrance illumination model to derive *anisotropic illumination models* whose reflective properties are not symmetric about the surface normal. These more accurately model the way that light is reflected by hair or burnished metal—surfaces whose microfeatures are preferentially oriented. To do this, Kajiya extends bump mapping to perturb not just the surface normal, but also the tangent and a binormal formed by the cross-product of the tangent and normal. Together these form a coordinate system that determines the orientation of the surface relative to the anisotropic illumination model. The surface shown in Fig. 16.47 is mapped to an anisotropic texture that represents a cross-weave of threads. Cabral, Max and Springmeyer [CABR87] have developed a method for determining G and ρ for a surface finish specified by an arbitrary bump map by computing the shadowing and masking effects of the bumps.

The Fresnel equation used by Blinn and by Cook and Torrance is correct only for unpolarized light. The polarization state of light changes, however, when light is reflected from a surface, and a surface's ρ is a function of the polarization state of the light incident on it. Wolff and Kurlander [WOLF90] have extended the Cook–Torrance model to take this into account and have generated images that evidence two effects that are most visible after two or more interobject reflections. First, dielectrics have an angle of incidence, known as the *Brewster angle*, at which incident light is completely polarized when reflected, or is not reflected at all if it is inappropriately polarized. If interobject specular reflection of initially unpolarized light occurs between two dielectric surfaces such that the angle of incidence of each reflection is equal to the Brewster angle, and the plane defined by $\overline{N}$ and $\overline{L}$ on one surface is perpendicular to that of the other, then no light at all will be specularly reflected from the second object; we can produce noticeable, but less dramatic, attenuation if the angles and orientations are varied. Second, colored conductors (metals such as copper or gold) tend to polarize light at different wavelengths differently. Therefore, when a colored conductor is reflected from a dielectric surface, the reflection will have a color slightly different from that when polarization is not taken into account.

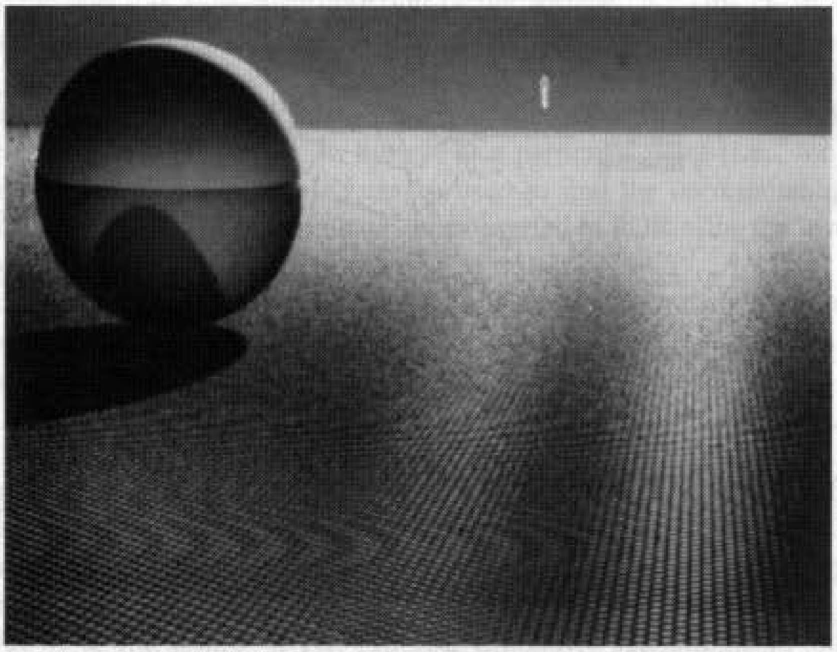


Fig. 16.47 Anisotropic texture. (By J. Kajiya [KAJI85], California Institute of Technology.)

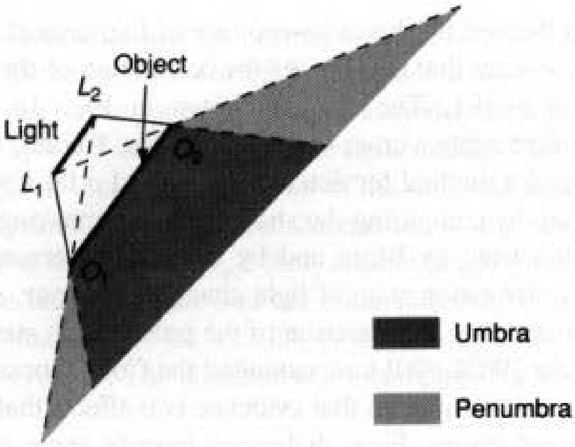


Fig. 16.48 Umbra and penumbra.

16.8 EXTENDED LIGHT SOURCES

The light sources discussed thus far have been point lights. In contrast, *extended* or *distributed* light sources actually have area and consequently cast "soft" shadows containing areas only partially blocked from the source, as shown in Fig. 16.48. The part of a light source's shadow that is totally blocked from the light source is the shadow's *umbra*;

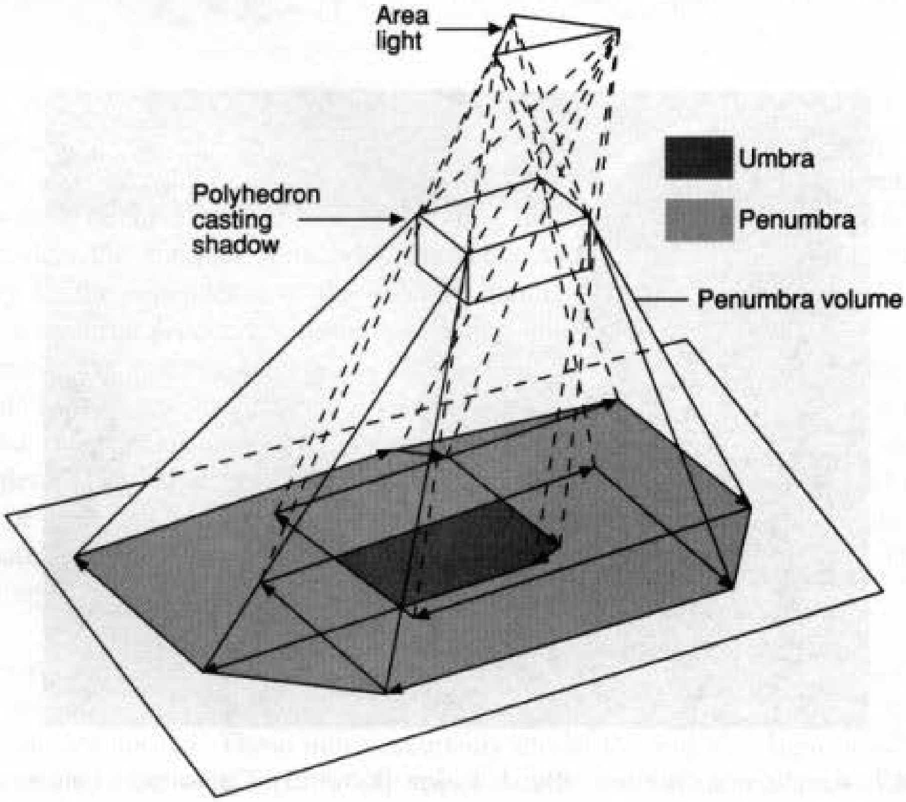


Fig. 16.49 Determining the penumbra and umbra volumes. (After [NISH85a].)

that part of the shadow that is only partially shielded from the source is the shadow's *penumbra*. All of a point light source's shadow is umbra. An obvious approach to modeling an extended light source is to approximate it with closely clustered point light sources [VERB84; BROT84]. The process is computationally expensive, however, and the results are less than satisfactory unless there are very many point sources and no specular reflection.

Nishita and Nakamae [NISH83; NISH85a; NISH85b] have developed an extension of shadow volumes for modeling linear, convex polygonal, and convex polyhedral light sources with Lambertian intensity distributions. Their method employs an object-precision algorithm for determining shadow volumes for convex polyhedral objects. As shown in Fig. 16.49, the shadow volumes defined by each vertex of the source and the polyhedron are determined. The penumbra volume is then the smallest convex polyhedron containing these shadow volumes (their convex hull), and the umbra volume is the intersection of the shadow volumes. The volumes are then intersected with the faces of the objects to determine the areas in each penumbra or umbra volume. Any point lying within an umbra volume defined by a light source and any other polyhedron is not affected by the light source. Determining the color of a point in a penumbra involves computing those parts of the light source visible from the point on the object. A BSP-tree–based approach is discussed in [CHIN90].

Excellent simulations of extended light sources have been achieved by using variations on ray-tracing (Section 16.12) and radiosity methods (Section 16.13).

16.9 SPECTRAL SAMPLING

As light is reflected from a surface, the spectral energy distribution P_λ of the incident light is modified by the spectral reflectance function of the surface, ρ_λ. This curve specifies the percentage of light of each wavelength λ that is reflected. Therefore, the spectral energy distribution of the reflected light is $P_\lambda\rho_\lambda$. Similarly, light passing through a material is modified by the spectral transmittance function of the material, τ_λ. Once a final spectral energy distribution for light falling on the eye at a particular point (i.e., for a particular pixel) is determined, then Eq. (13.18) can be used to find the corresponding CIE XYZ specification for the light, and this result can be converted to RGB using the inverse of M of Eq. (13.24).

It is tempting to assume that ρ_λ and τ_λ can be replaced by equivalent tristimulus RGB or XYZ color specifications. If this could be done, then the product $P_\lambda\rho_\lambda$ could be replaced by a sum of the tristimulus values for each of P_λ and ρ_λ, of the form

$$R\rho_R + G\rho_G + B\rho_B. \tag{16.53}$$

This is clearly incorrect. Consider the P_λ and ρ_λ shown in Fig. 16.50. P_λ is uniform except for a narrow interval around 600 nanometers, whereas ρ_λ reflects light only at the same narrow interval. The product $P_\lambda\rho_\lambda$ is thus zero everywhere, and the surface appears black. On the other hand, the XYZ components of P_λ will be nonzero (this can be seen by applying the inverse of M to the results of Eq. (13.18). Similarly, substituting ρ_λ for P_λ in Eq. (13.18) and applying the inverse of M gives nonzero values for ρ_R, ρ_G, ρ_B, so the product (Eq. 16.53) is also nonzero! This is the wrong answer, and demonstrates that ρ_λ and τ_λ cannot in general be replaced by an equivalent tristimulus color specification. As Hall

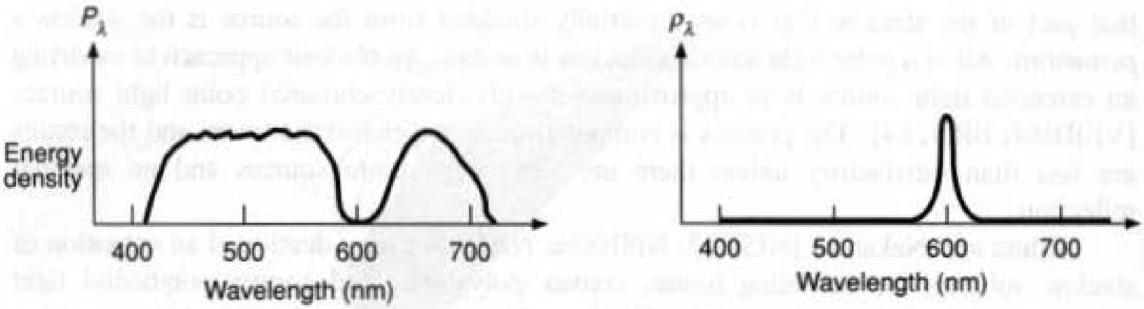


Fig. 16.50 P_λ and ρ_λ whose product is 0 everywhere.

[HALL83] points out, this approach fails because it is actually point sampling in color space and consequently is prone to all the problems that arise from undersampling.

A more accurate representation of a light source's spectral distribution or a surface's reflectivity or transmissivity can be obtained by representing either one as a curve that interpolates a sufficiently large set of samples spaced across the visible spectrum. Color Plate III.9 shows images, generated with different color models, of two overlapping filters lit by a D6500 light source. The filters do not pass a common band of wavelengths, so no transmitted light should be visible where they overlap. Color Plate III.9(a), the control picture, was generated using spectra maintained at 1-nm intervals from 360 to 830 nanometers. Color Plates III.9(b) and III.9(c) were generated with three samples for the primaries of the CIE and RGB color spaces, respectively. Color Plate III.9(d) approximates the spectra of Color Plate III.9(a) with nine spectral values. Note how much more closely Color Plate III.9(d) matches the control picture than do the others.

16.10 IMPROVING THE CAMERA MODEL

Thus far, we have modeled the image produced by a pinhole camera: each object, regardless of its position in the environment, is projected sharply and without distortion in the image. Real cameras (and eyes) have lenses that introduce a variety of distortion and focusing effects.

Depth of field. Objects appear to be more or less in focus depending on their distance from the lens, an effect known as *depth of field*. A lens has a focal length F corresponding to the distance from the lens at which a perfectly focused image of an object converges. If a point is out of focus, its image converges on a plane that is closer or farther than F. An out-of-focus point projects on a plane at F as a circle known as the *circle of confusion*.

Potmesil and Chakravarty [POTM82] have developed a postprocessing technique for simulating some of the effects of depth of field and other properties of real lenses, demonstrated in Color Plate II.38. Their system first produces images using a conventional pinhole-lens renderer that generates not only the intensity at each point, but also that point's z value. Each sampled point is then turned into a circle of confusion with a size and intensity distribution determined by its z value and the lens and aperture being used. The intensity of each pixel in the output image is calculated as a weighted average of the intensities in the circles of confusion that overlap the pixel. Since the image is initially computed from a single point at the center of projection, the results of this technique are still only an approximation. A real lens's focusing effect causes light rays that would not pass through the pinhole to strike the lens and to converge to form the image. These rays see a slightly different view of the scene, including, for example, parts of surfaces that are not visible to the rays passing through the pinhole. This information is lost in the images created with the Potmesil and Chakravarty model.

Motion blur. *Motion blur* is the streaked or blurred appearance that moving objects have because a camera's shutter is open for a finite amount of time. To achieve this effect, we need to solve the visible-surface problem over time, as well as over space, to determine which objects are visible at a given pixel and when they are visible. Korein and Badler [KORE83] describe two contrasting approaches: an analytic algorithm that uses continuous functions to model the changes that objects undergo over time, and a simple image-precision approach that relies on temporal supersampling. In the temporal-supersampling method, a separate image is rendered for each point in time to be sampled. The motion-blurred image is created by taking a weighted sum of the images, in essence convolving them with a temporal filter. For example, if each of n images is weighted by $1/n$, this corresponds to temporal box filtering. The more closely the samples are spaced, the better the results. Temporal supersampling, like spatial supersampling, suffers from aliasing: Unless samples are spaced sufficiently close together in time, the final image will appear to be a set of discrete multiple exposures. Potmesil and Chakravarty [POTM83] have extended their depth-of-field work to model the ways in which actual camera shutters move. As we shall see in Section 16.12, the stochastic sampling techniques used in distributed ray tracing offer a uniform framework for integrating lens effects, motion blur, and spatial antialiasing demonstrated in Color Plates II.39 and III.16.

16.11 GLOBAL ILLUMINATION ALGORITHMS

An illumination model computes the color at a point in terms of light directly emitted by light sources and of light that reaches the point after reflection from and transmission through its own and other surfaces. This indirectly reflected and transmitted light is often called *global illumination*. In contrast, *local illumination* is light that comes directly from the light sources to the point being shaded. Thus far, we have modeled global illumination by an ambient illumination term that was held constant for all points on all objects. It did not depend on the positions of the object or the viewer, or on the presence or absence of nearby objects that could block the ambient light. In addition, we have seen some limited global illumination effects made possible by shadows, transparency, and reflection maps.

Much of the light in real-world environments does not come from direct light sources. Two different classes of algorithms have been used to generate pictures that emphasize the contributions of global illumination. Section 16.12 discusses extensions to the visible-surface ray-tracing algorithm that interleave visible-surface determination and shading to depict shadows, reflection, and refraction. Thus, global specular reflection and transmission supplement the local specular, diffuse, and ambient illumination computed for a surface. In contrast, the radiosity methods discussed in Section 16.13 completely separate shading and visible-surface determination. They model all an environment's interactions with light sources first in a view-independent stage, and then compute one or more images for the desired viewpoints using conventional visible-surface and interpolative shading algorithms.

The distinction between view-dependent algorithms, such as ray tracing, and view-independent ones, such as radiosity, is an important one. *View-dependent* algorithms discretize the view plane to determine points at which to evaluate the illumination equation, given the viewer's direction. In contrast, *view-independent* algorithms discretize the environment, and process it in order to provide enough information to evaluate the illumination equation at any point and from any viewing direction. View-dependent algorithms are well-suited for handling specular phenomena that are highly dependent on the viewer's position, but may perform extra work when modeling diffuse phenomena that change little over large areas of an image, or between images made from different viewpoints. On the other hand, view-independent algorithms model diffuse phenomena efficiently, but require overwhelming amounts of storage to capture enough information about specular phenomena.

Ultimately, all these approaches attempt to solve what Kajiya [KAJI86] has referred to as the *rendering equation*, which expresses the light being transferred from one point to another in terms of the intensity of the light emitted from the first point to the second and the intensity of light emitted from all other points that reaches the first and is reflected from the first to the second. The light transferred from each of these other points to the first is, in turn, expressed recursively by the rendering equation. Kajiya presents the rendering equation as

$$I\,(x, x') = g(x, x')\left[\varepsilon(x, x') + \int_S \rho(x, x', x'')\, I(x', x'')dx''\right], \qquad (16.54)$$

where x, x', and x'' are points in the environment. $I(x, x')$ is related to the intensity passing from x' to x. $g(x, x')$ is a geometry term that is 0 when x and x' are occluded from each other, and $1 / r^2$ when they are visible to each other, where r is the distance between them. $\varepsilon(x, x')$ is related to the intensity of light that is emitted from x' to x. The initial evaluation of $g(x, x')\varepsilon(x, x')$ for x at the viewpoint accomplishes visible-surface determination in the sphere about x. The integral is over all points on all surfaces S. $\rho(x, x', x'')$ is related to the intensity of the light reflected (including both specular and diffuse reflection) from x'' to x from the surface at x'. Thus, the rendering equation states that the light from x' that reaches x consists of light emitted by x' itself and light scattered by x' to x from all other surfaces, which themselves emit light and recursively scatter light from other surfaces.

As we shall see, how successful an approach is at solving the rendering equation depends in large part on how it handles the remaining terms and the recursion, on what combinations of diffuse and specular reflectivity it supports, and on how well the visibility relationships between surfaces are modeled.

16.12 RECURSIVE RAY TRACING

In this section, we extend the basic ray-tracing algorithm of Section 15.10 to handle shadows, reflection, and refraction. This simple algorithm determined the color of a pixel at the closest intersection of an eye ray with an object, by using any of the illumination models described previously. To calculate shadows, we fire an additional ray from the point of intersection to each of the light sources. This is shown for a single light source in Fig.

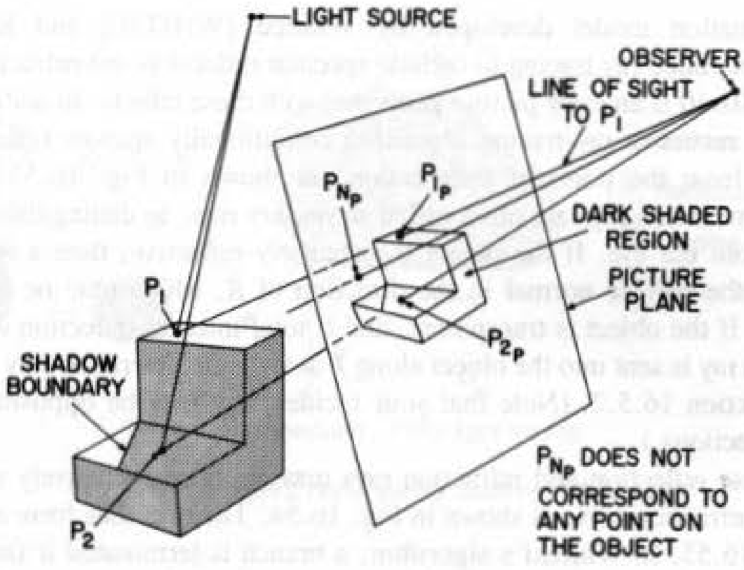


Fig. 16.51 Determining whether a point on an object is in shadow. (Courtesy of Arthur Appel, IBM T.J. Watson Research Center.)

16.51, which is reproduced from a paper by Appel [APPE68]—the first paper published on ray tracing for computer graphics. If one of these *shadow rays* intersects any object along the way, then the object is in shadow at that point and the shading algorithm ignores the contribution of the shadow ray's light source. Figure 16.52 shows two pictures that Appel rendered with this algorithm, using a pen plotter. He simulated a halftone pattern by placing a different size "+" at each pixel in the grid, depending on the pixel's intensity. To compensate for the grid's coarseness, he drew edges of visible surfaces and of shadows using a visible-line algorithm.

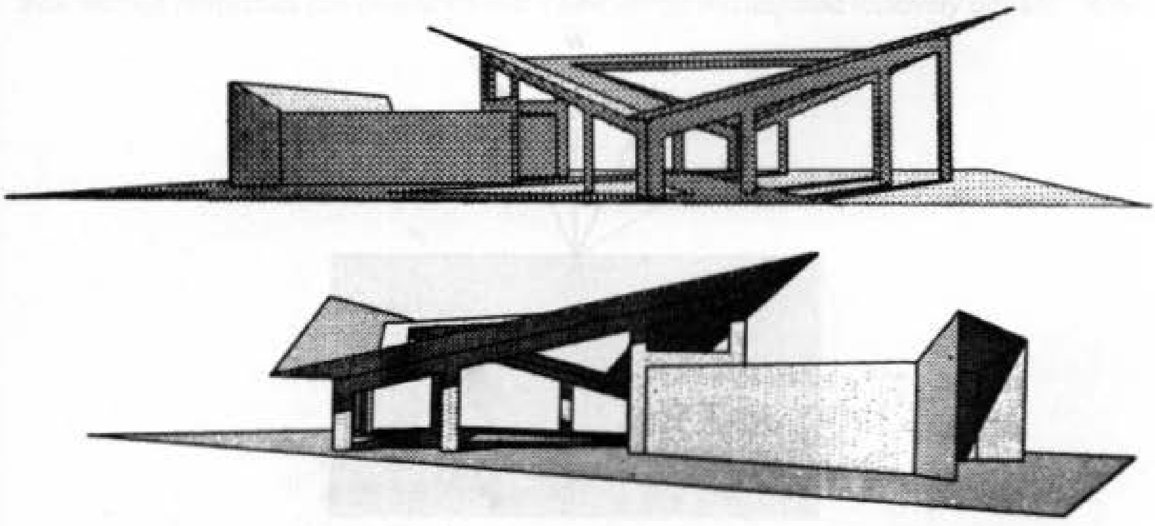


Fig. 16.52 Early pictures rendered with ray tracing. (Courtesy of Arthur Appel, IBM T.J. Watson Research Center.)

The illumination model developed by Whitted [WHIT80] and Kay [KAY79a] fundamentally extended ray tracing to include specular reflection and refractive transparency. Color Plate III.10 is an early picture generated with these effects. In addition to shadow rays, Whitted's recursive ray-tracing algorithm conditionally spawns *reflection rays* and *refraction rays* from the point of intersection, as shown in Fig. 16.53. The shadow, reflection, and refraction rays are often called *secondary rays*, to distinguish them from the *primary rays* from the eye. If the object is specularly reflective, then a reflection ray is reflected about the surface normal in the direction of $\overline{R}$, which may be computed as in Section 16.1.4. If the object is transparent, and if total internal reflection does not occur, then a refraction ray is sent into the object along $\overline{T}$ at an angle determined by Snell's law, as described in Section 16.5.2. (Note that your incident ray may be oppositely oriented to those in these sections.)

Each of these reflection and refraction rays may, in turn, recursively spawn shadow, reflection, and refraction rays, as shown in Fig. 16.54. The rays thus form a *ray tree*, such as that of Fig. 16.55. In Whitted's algorithm, a branch is terminated if the reflected and refracted rays fail to intersect an object, if some user-specified maximum depth is reached or if the system runs out of storage. The tree is evaluated bottom-up, and each node's intensity is computed as a function of its children's intensities. Color Plate III.11(a) and (b) were made with a recursive ray-tracing algorithm.

We can represent Whitted's illumination equation as

$$I_\lambda = I_{a\lambda}k_aO_{d\lambda} + \sum_{1\le i\le m} S_i f_{att_i} I_{p\lambda_i}[k_dO_{d\lambda}(\overline{N}\cdot\overline{L}_i) + k_s(\overline{N}\cdot\overline{H}_i)^n] + k_sI_{r\lambda} + k_tI_{t\lambda}, \qquad (16.55)$$

where $I_{r\lambda}$ is the intensity of the reflected ray, k_t is the *transmission coefficient* ranging between 0 and 1, and $I_{t\lambda}$ is the intensity of the refracted transmitted ray. Values for $I_{r\lambda}$ and $I_{t\lambda}$ are determined by recursively evaluating Eq. (16.55) at the closest surface that the reflected and transmitted rays intersect. To approximate attenuation with distance, Whitted multiplied the I_λ calculated for each ray by the inverse of the distance traveled by the ray. Rather than treating S_i as a delta function, as in Eq. (16.24), he also made it a continuous function

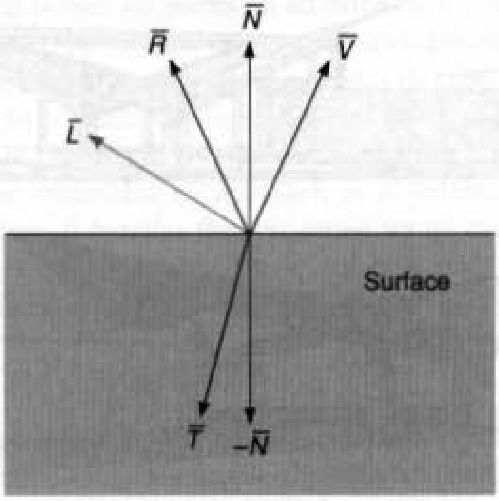


Fig. 16.53 Reflection, refraction, and shadow rays are spawned from a point of intersection.

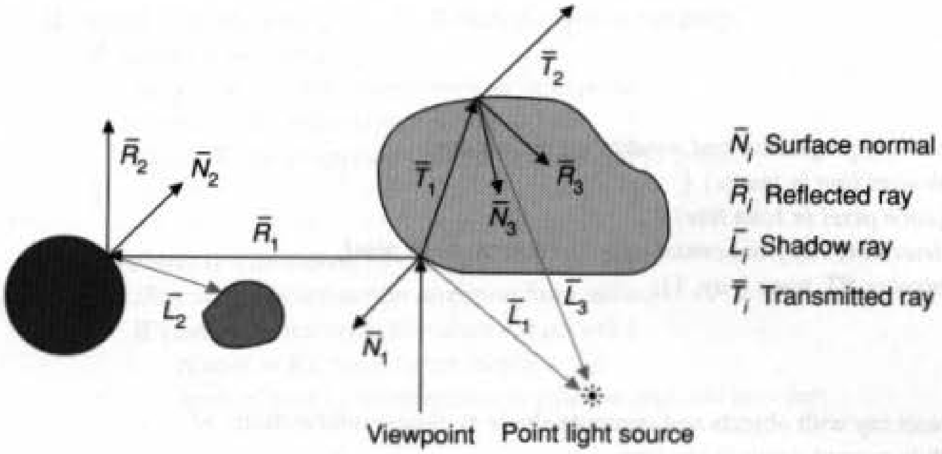


Fig. 16.54 Rays recursively spawn other rays.

of the k_t of the objects intersected by the shadow ray, so that a transparent object obscures less light than an opaque one at those points it shadows.

Figure 16.56 shows pseudocode for a simple recursive ray tracer. RT_trace determines the closest intersection the ray makes with an object and calls RT_shade to determine the shade at that point. First, RT_shade determines the intersection's ambient color. Next, a shadow ray is spawned to each light on the side of the surface being shaded to determine its contribution to the color. An opaque object blocks the light totally, whereas a transparent one scales the light's contribution. If we are not too deep in the ray tree, then recursive calls are made to RT_trace to handle reflection rays for reflective objects and refraction rays for transparent objects. Since the indices of refraction of two media are needed to determine the direction of the refraction ray, the index of refraction of the material in which a ray is traveling can be included with each ray. RT_trace retains the ray tree only long enough to determine the current pixel's color. If the ray trees for an entire image can be preserved, then surface properties can be altered and a new image recomputed relatively quickly, at the

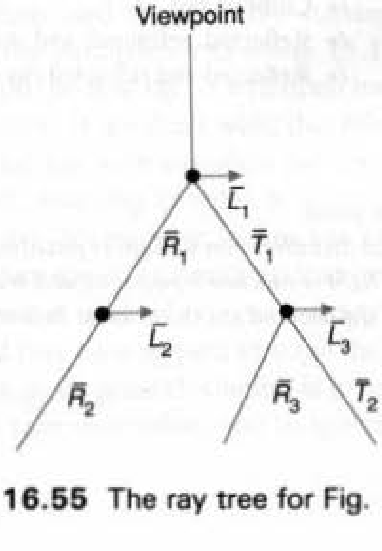


Fig. 16.55 The ray tree for Fig. 16.54.

```
select center of projection and window on view plane;
for (each scan line in image) {
    for (each pixel in scan line) {
        determine ray from center of projection through pixel;
        pixel = RT_trace (ray, 1);
    }
}

/* Intersect ray with objects and compute shade at closest intersection.  */
/* Depth is current depth in ray tree.  */

RT_color RT_trace (RT_ray ray, int depth)
{
    determine closest intersection of ray with an object;

    if (object hit) {
        compute normal at intersection;
        return RT_shade (closest object hit, ray, intersection, normal, depth);
    } else
        return BACKGROUND_VALUE;
}  /* RT_trace */

/* Compute shade at point on object, tracing rays for shadows, reflection and refraction.  */
RT_color RT_shade (
    RT_object object,          /* Object intersected  */
    RT_ray ray,                /* Incident ray  */
    RT_point point,            /* Point of intersection to shade  */
    RT_normal normal,          /* Normal at point  */
    int depth)                 /* Depth in ray tree  */
{
    RT_color color;            /* Color of ray  */
    RT_ray rRay, tRay, sRay;   /* Reflected, refracted, and shadow rays  */
    RT_color rColor, tColor;   /* Reflected and refracted ray colors  */

    color = ambient term;
    for (each light) {
        sRay = ray to light from point;
        if (dot product of normal and direction to light is positive) {
            compute how much light is blocked by opaque and transparent surfaces,
                and use to scale diffuse and specular terms before adding them to color;
        }
    }
```

Fig. 16.56 *(Cont'd.)*

```
    if (depth < maxDepth) {        /* Return if depth is too deep. */
        if (object is reflective) {
            rRay = ray in reflection direction from point;
            rColor = RT_trace (rRay, depth + 1);
            scale rColor by specular coefficient and add to color;
        }

        if (object is transparent) {
            tRay = ray in refraction direction from point;
            if (total internal reflection does not occur) {
                tColor = RT_trace (tRay, depth + 1);
                scale tColor by transmission coefficient and add to color;
            }
        }
    }
    return color;                  /* Return color of ray */
} /* RT_shade */
```

Fig. 16.56 Pseudocode for simple recursive ray tracing without antialiasing.

cost of only reevaluating the trees. Sequin and Smyrl [SEQU89] present techniques that minimize the time and space needed to process and store ray trees.

Figure 16.54 shows a basic problem with how ray tracing models refraction: The shadow ray $\overline{L}_3$ is not refracted on its path to the light. In fact, if we were to simply refract $\overline{L}_3$ from its current direction at the point where it exits the large object, it would not end at the light source. In addition, when the paths of rays that are refracted are determined, a single index of refraction is used for each ray. Later, we discuss some ways to address these failings.

Ray tracing is particularly prone to problems caused by limited numerical precision. These show up when we compute the objects that intersect with the secondary rays. After the *x*, *y*, and *z* coordinates of the intersection point on an object visible to an eye ray have been computed, they are then used to define the starting point of the secondary ray for which we must determine the parameter *t* (Section 15.10.1). If the object that was just intersected is intersected with the new ray, it will often have a small, nonzero *t*, because of numerical-precision limitations. If not dealt with, this false intersection can result in visual problems. For example, if the ray were a shadow ray, then the object would be considered as blocking light from itself, resulting in splotchy pieces of incorrectly "self-shadowed" surface. A simple way to solve this problem for shadow rays is to treat as a special case the object from which a secondary ray is spawned, so that intersection tests are not performed on it. Of course, this does not work if objects are supported that really could obscure themselves or if transmitted rays have to pass through the object and be reflected from the inside of the same object. A more general solution is to compute abs(*t*) for an intersection, to compare it with a small tolerance value, and to ignore it if it is below the tolerance.

The paper Whitted presented at *SIGGRAPH '79* [WHIT80], and the movies he made using the algorithm described there, started a renaissance of interest in ray tracing. Recursive ray tracing makes possible a host of impressive effects—such as shadows, specular reflection, and refractive transparency—that were difficult or impossible to obtain previously. In addition, a simple ray tracer is quite easy to implement. Consequently, much effort has been directed toward improving both the algorithm's efficiency and its image quality. We provide a brief overview of these issues here, and discuss several parallel hardware implementations that take advantage of the algorithm's intrinsic parallelism in Section 18.11.2. For more detail, see [GLAS89].

16.12.1 Efficiency Considerations for Recursive Ray Tracing

Section 15.10.2 discussed how to use extents, hierarchies, and spatial partitioning to limit the number of ray–object intersections to be calculated. These general efficiency techniques are even more important here than in visible-surface ray tracing for several reasons. First, a quick glance at Fig. 16.55 reveals that the number of rays that must be processed can grow exponentially with the depth to which rays are traced. Since each ray may spawn a reflection ray and a refraction ray, in the worst case, the ray tree will be a complete binary tree with $2^n - 1$ rays, where the tree depth is n. In addition, each reflection or refraction ray that intersects with an object spawns one shadow ray for each of the m light sources. Thus, there are potentially $m(2^n - 1)$ shadow rays for each ray tree. To make matters worse, since rays can come from any direction, traditional efficiency ploys, such as clipping objects to the view volume and culling back-facing surfaces relative to the eye, cannot be used in recursive ray tracing. Objects that would otherwise be invisible, including back faces, may be reflected from or refracted through visible surfaces.

Item buffers. One way of speeding up ray tracing is simply not to use it at all when determining those objects directly visible to the eye. Weghorst, Hooper, and Greenberg [WEGH84] describe how to create an *item buffer* by applying a less costly visible-surface algorithm, such as the *z*-buffer algorithm, to the scene, using the same viewing specification. Instead of determining the shade at each pixel, however, they record in the item buffer pixel the identity of the closest object. Then, only this object needs to be processed by the ray tracer to determine the eye ray's exact intersection for this pixel, so that further rays may be spawned.

Reflection maps. Tracing rays can be avoided in other situations, too. Hall [HALL86] shows how to combine ray tracing with the reflection maps discussed in Section 16.6. The basic idea is to do less work for the secondary rays than for primary rays. Those objects that are not directly visible in an image are divided into two groups on the basis of an estimation of their indirect visibility. Ray tracing is used to determine the global lighting contributions of the more visible ones, whereas indexing into a suitably prepared reflection map handles the others. One way to estimate the extent to which an object is indirectly visible is to measure the solid angle subtended by the directly visible objects as seen from the centroid of the indirectly visible object. If the solid angle is greater than some threshold, then the object will be included in the environment to be traced by reflection rays (this environment includes the directly visible objects); otherwise, the object will be represented only in the reflection map. When ray tracing is performed, if a reflection ray does not intersect one of the objects in the reflection-ray environment, then the ray is used to index into the reflection map. Hall also points out that reflected and refracted images are often extremely distorted. Therefore, good results may be achieved by intersecting the reflection and refraction rays with object definitions less detailed than those used for the eye rays.

Adaptive tree-depth control. Although ray tracing is often used to depict highly specular objects, most of an image's area is usually not filled with such objects. Consequently, a high recursion level often results in unnecessary processing for large parts of the picture. Hall [HALL83] introduced the use of *adaptive tree-depth control*, in which a ray is not cast if its contribution to the pixel's intensity is estimated to be below some preset threshold. This is accomplished by approximating a ray's maximum contribution by calculating its intensity with the assumption that the ray's child rays have intensities of 1. This allows the ray's contribution to its parent to be estimated. As the ray tree is built, the maximum contribution of a ray is multiplied by those of its ancestors to derive the ray's maximum contribution to the pixel. For example, suppose that $\overline{R}_1$ and $\overline{R}_2$ in Fig. 16.55 are spawned at surfaces with k_s values of .1 and .05, respectively. At the first intersection, we estimate the maximum contribution to the pixel of $\overline{R}_1$ to be .1. At the second intersection, we estimate the maximum contribution to the pixel of $\overline{R}_2$ to be .05 × .1 = .005. If this is below our threshold, we may decide not to cast $\overline{R}_2$. Although adaptive tree-depth control has been shown to work well for many images, it is easy to design cases in which it will fail. Although one uncast ray may have an imperceptible effect on a pixel's shade, a pixel may receive a significant amount of light from a large number of individually insignificant rays.

Light buffers. We noted that *m* shadow rays are spawned for each reflection or refraction ray that hits an object. Shadow rays are special, however, in that each is fired toward one of a relatively small set of objects. Haines and Greenberg [HAIN86] have introduced the notion of a *light buffer* to increase the speed with which shadow rays are processed. A light buffer is a cube centered about a light source and aligned with the world-coordinate axes, as shown in Fig. 16.57(a). Each side is tiled with a regular grid of squares, and each square is associated with a depth-sorted list of surfaces that can be seen through it from the light. The

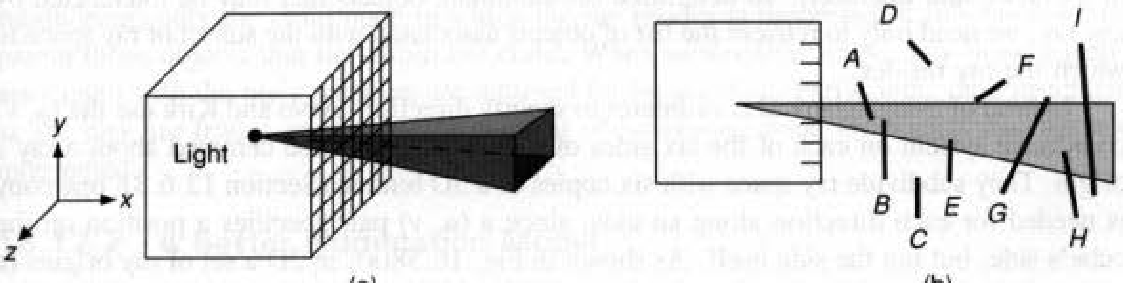


Fig. 16.57 Light buffer centered about a light source. (a) Volume defined by one square. (b) Cross-section shows volume of square in 2D with intersected surfaces. Square's depth-sorted list is initially *A, B, E, G, H,* and *I*.

lists are filled by scan converting the objects in the scene onto each face of the light buffer with the center of projection at the light. The scan-conversion algorithm places an object on the list of every square covered by that object's projection, no matter how small the overlap. Figure 16.57(b) shows a square's set of intersecting surfaces. A shadow ray is processed by determining the light-buffer square through which it passes. The ray needs to be tested only for intersection against that square's ordered list of surfaces. Thus, a light buffer implements a kind of 3D spatial partitioning of the 3D view of its light.

A number of efficiency speedups are possible. For example, any object whose projection onto the light buffer totally covers a square (e.g., *G* in Fig. 16.57b) can be associated with a special "full-occlusion" record in the square's list; any object more distant from the light than a fully occluding object is purged from the list. In this case, *H* and *I* are purged. Whenever the ray from an intersection point is tested against a square's list, the test can be terminated immediately if there is a full-occlusion record closer to the light than the intersection point. In addition, back-face culling may be used to avoid adding any faces to a list that are both part of a closed opaque solid and back facing relative to the light. Color Plate III.12 was made using the light-buffer technique.

Haines and Greenberg also mention an interesting use of object coherence to determine shadows that can be applied even without using light buffers. A pointer is associated with each light and is initialized to null. When an object is found to be shadowed from a light by some opaque surface, the light's pointer is set to the shadowing surface. The next time a shadow ray is cast for the light, it is first intersected with the object pointed at by the light's pointer. If the ray hits it, then intersection testing is finished; otherwise, the pointer is set to null and testing continues.

Ray classification. The spatial-partitioning approaches discussed in Section 15.10.2 make it possible to determine which objects lie in a given region of 3D space. Arvo and Kirk [ARVO87] have extended this concept to partition rays by the objects that they intersect, a technique called *ray classification*. A ray may be specified by its position in 5D *ray space*, determined by the 3D position of its origin in space and its 2D direction in spherical coordinates. A point in ray space defines a single ray, whereas a finite subset of ray space defines a family of rays or *beam*. Ray classification adaptively partitions ray space into subsets, each of which is associated with a list of objects that it contains (i.e,. that one of its rays could intersect). To determine the candidate objects that may be intersected by any ray, we need only to retrieve the list of objects associated with the subset of ray space in which the ray resides.

Instead of using spherical coordinates to specify direction, Arvo and Kirk use the (u, v) coordinate system on each of the six sides of an axis-aligned cube centered about a ray's origin. They subdivide ray space with six copies of a 5D bintree (Section 12.6.3); one copy is needed for each direction along an axis, since a (u, v) pair specifies a position on the cube's side, but not the side itself. As shown in Fig. 16.58(a), in 2D a set of ray origins (a rectangle in (x, y)), combined with a set of ray directions (an interval in u on one of four sides), define a partially bounded polygon that defines the beam. In 3D, shown in Fig. 16.58(b), a set of ray origins (a rectangular parallepiped in (x, y, z)), combined with a set of ray directions (a rectangle in (u, v)), defines a partially bounded polyhedral volume that defines the beam. Objects (or their extents) may be intersected with this volume (e.g., using

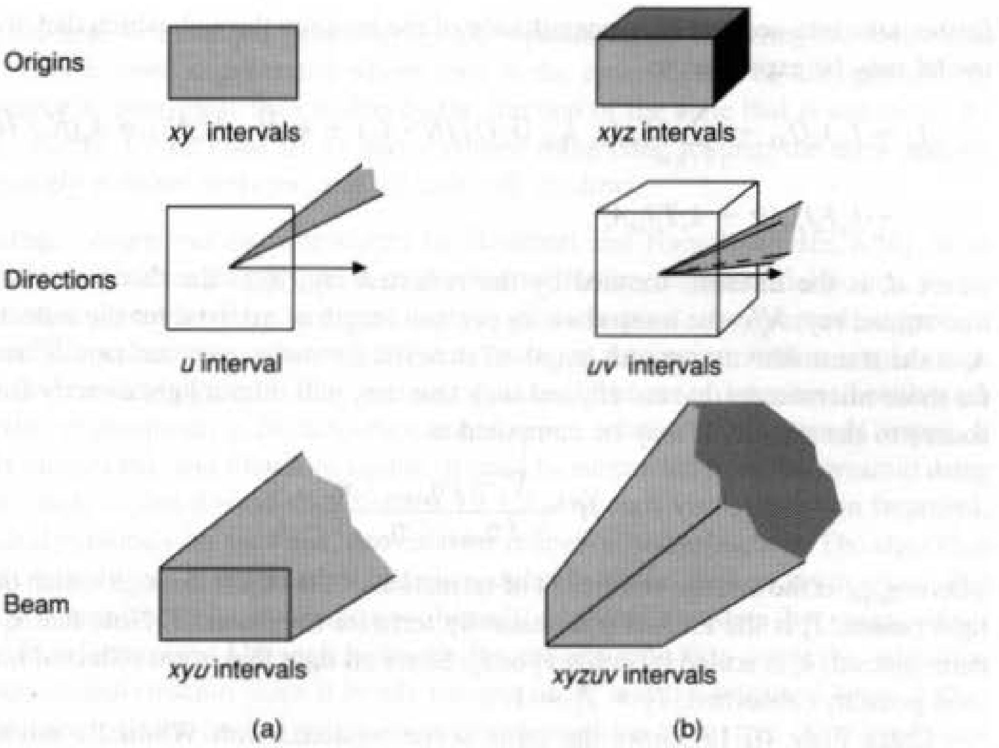


Fig. 16.58 Ray classification: (a) 2-space beams; (b) 3-space beams. (After [ARVO87].)

the BSP-tree-based approach of Section 12.6.4) to determine whether they are contained in it. Initially, each unpartitioned bintree represents positions in all of the environment's 3-space extent, and directions that pass through any part of the bintree's designated side. Each bintree is initially associated with all the objects in the environment.

Arvo and Kirk use *lazy evaluation* to build the trees at the same time that they trace rays. When a ray is traced, its direction selects the root of the bintree whose (u, v) coordinate system it will intersect. If more than some threshold number of objects exist at the node, the bintree is subdivided equally along each of its five axes. With each subdivision, only the child node in which the ray resides is processed; it inherits from its parent those objects that lie within the child. When subdivision stops, only those objects associated with the ray's partition are returned for intersection. Partitioning data structures as the rays are traced minimizes the amount of processing done for objects that are not intersected.

16.12.2 A Better Illumination Model

It is possible to make a number of extensions to Eq. (16.55). Hall [HALL83] has developed a model in which the specular light expressions are scaled by a wavelength-dependent Fresnel reflectivity term. An additional term for transmitted local light is added to take into account the contribution of transmitted light directly emitted by the light sources, and is also scaled by the Fresnel transmissivity term. The global reflected and refracted rays further take into account the transmittance of the medium through which they travel. Hall's model may be expressed as

$$I_\lambda = I_{a\lambda}k_a O_{d\lambda} + \sum_{1 \le i \le m} S_i f_{att_i} I_{p\lambda_i} [k_d O_{d\lambda}(\overline{N} \cdot \overline{L}_i) + k_s(\overline{N} \cdot \overline{H}_i)^n F_\lambda + k_s(\overline{N} \cdot \overline{H'}_i)^n T_\lambda]$$

$$+ k_s F_\lambda I_{r\lambda} A_r^{d_r} + k_s T_\lambda I_{t\lambda} A_t^{d_t}, \tag{16.56}$$

where d_r is the distance traveled by the reflected ray, d_t is the distance traveled by the transmitted ray, A_r is the transmissivity per unit length of material for the reflected ray, and A_t is the transmissivity per unit length of material for the transmitted ray. $\overline{H}'$ is the normal for those microfacets that are aligned such that they will refract light directly from the light source to the viewer. It may be computed as

$$\overline{H}' = \frac{\overline{V} - (\eta_{avg\theta_t} / \eta_{avg\theta_i})\overline{L}}{(\eta_{avg\theta_t} / \eta_{avg\theta_i}) - 1}, \tag{16.57}$$

where $\eta_{avg\theta_t}$ is the average coefficient of refraction of the object through which the ray to the light passes. T_λ is the Fresnel transmissivity term for the material. Note that k_t is not used here; instead, k_s is scaled by either F_λ or T_λ. Since all light that is not reflected is transmitted (and possibly) absorbed, $F_\lambda + T_\lambda = 1$.

Color Plate III.13 shows the same scene rendered with Whitted's model and with Hall's model. The color of the glass sphere in part (b) clearly shows the filtering effects resulting from the use of the transmissivity terms.

16.12.3 Area-Sampling Variations

One of conventional ray tracing's biggest drawbacks is that this technique point samples on a regular grid. Whitted [WHIT80] suggested that unweighted area sampling could be accomplished by replacing each linear eye ray with a pyramid defined by the eye and the four corners of a pixel. These pyramids would be intersected with the objects in the environment, and sections of them would be recursively refracted and reflected by the objects that they intersect. A pure implementation of this proposal would be exceedingly complex, however, since it would have to calculate exact intersections with occluding objects, and to determine how the resulting pyramid fragments are modified as they are recursively reflected from and refracted by curved surfaces. Nevertheless, it has inspired several interesting algorithms that accomplish antialiasing, and at the same time decrease rendering time by taking advantage of coherence.

Cone tracing. *Cone tracing*, developed by Amanatides [AMAN84], generalizes the linear rays into cones. One cone is fired from the eye through each pixel, with an angle wide enough to encompass the pixel. The cone is intersected with objects in its path by calculating approximate fractional blockage values for a small set of those objects closest to the cone's origin. Refraction and reflection cones are determined from the optical laws of spherical mirrors and lenses as a function of the surface curvature of the object intersected and the area of intersection. The effects of scattering on reflection and refraction are simulated by further broadening the angles of the new reflection and refraction cones. The soft-edged shadows of extended light sources are reproduced by modeling the sources as spheres. A shadow cone is generated whose base is the cross-section of the light source. The light source's intensity is then scaled by the fraction of the cone that is unblocked by intervening objects. Color Plate III.14 was rendered using cone tracing; the three spheres have successively rougher surfaces, and all cast soft shadows.

Beam tracing. *Beam tracing,* introduced by Heckbert and Hanrahan [HECK84], is an object-precision algorithm for polygonal environments that traces pyramidal beams, rather than linear rays. Instead of tracing beams through each pixel, as Whitted suggested, Heckbert and Hanrahan take advantage of coherence by beginning with a single beam defined by the viewing pyramid. The viewing pyramid's beam is intersected with each polygon in the environment, in front-to-back sorted order, relative to the pyramid's apex. If a polygon is intersected, and therefore visible, it must be subtracted from the pyramid using an algorithm such as that described in Section 19.1.4. For each visible polygon fragment, two polyhedral pyramids are spawned, one each for reflection and refraction. The algorithm proceeds recursively, with termination criteria similar to those used in ray tracing. The environment is transformed into each new beam's coordinate system by means of an appropriate transformation. Although beam tracing models reflection correctly, refraction is not a linear transformation since it bends straight lines, and the refracted beam is thus only approximated. Beam tracing produces an object-precision beam tree of polygons that may be recursively rendered using a polygon scan-conversion algorithm. Each polygon is rendered using a local illumination equation, and then its reflected and refracted child polygons are rendered on top and are averaged with it, taking into account the parent's specular and transmissive properties. Beam tracing takes advantage of coherence to provide impressive speedups over conventional ray tracing at the expense of a more complex algorithm, limited object geometry, and incorrectly modeled refraction. Color Plate III.15 was rendered using this algorithm.

Beam tracing can accommodate shadows by using a variant of the Atherton–Weiler–Greenberg shadow algorithm (Section 16.4.2). Beams are traced from the point of view of each light source to determine all surfaces directly visible from the light sources, and the resulting polygons are added to the data structure as lit detail polygons that affect only the shading. This produces shadows similar to those obtained with conventional ray tracing.

Pencil tracing. Shinya, Takahashi, and Naito [SHIN87] have implemented an approach called *pencil tracing* that solves some of the problems of cone tracing and beam tracing. A *pencil* is a bundle of rays consisting of a central *axial* ray, surrounded by a set of nearby *paraxial* rays. Each paraxial ray is represented by a 4D vector that represents its relationship to the axial ray. Two dimensions express the paraxial ray's intersection with a plane perpendicular to the axial ray; the other two dimensions express the paraxial ray's direction. In many cases, only an axial ray and solid angle suffice to represent a pencil. If pencils of sufficiently small solid angle are used, then reflection and refraction can be approximated well by a linear transformation expressed as a 4×4 matrix. Shinya, Takahashi, and Naito have developed error-estimation techniques for determining an appropriate solid angle for a pencil. Conventional rays must be traced where a pencil would intersect the edge of an object, however, since the paraxial transformations are not valid in these cases.

16.12.4 Distributed Ray Tracing

The approaches we have just discussed avoid the aliasing problems of regular point sampling by casting solid beams rather than infinitesimal rays. In contrast, *distributed ray tracing*, developed by Cook, Porter, and Carpenter [COOK84b], is based on a stochastic approach to supersampling that trades off the objectionable artifacts of aliasing for the less offensive artifacts of noise [COOK86]. As we shall see, the ability to perform antialiased spatial sampling can also be exploited to sample a variety of other aspects of the scene and its objects to produce effects such as motion blur, depth of field, extended light sources, and specular reflection from rough surfaces. The word *distributed* in this technique's name refers to the fact that rays are stochastically distributed to sample the quantities that produce these effects. The basic concepts have also been applied to other algorithms besides ray tracing [COOK87].

Stochastic sampling. As explained in Section 14.10, aliasing results when a signal is sampled with regularly spaced samples below the Nyquist rate. This is true even if we supersample and filter to compute the value of a pixel. If the samples are not regularly spaced, however, the sharply defined frequency spectrum of the aliases is replaced by noise, an artifact that viewers find much less objectionable than the individually recognizable frequency components of regular aliasing, such as staircasing.

It is not enough, however, merely to replace ray tracing's regular grid of eye rays with an equal number of rays passing through random points on the image plane, since purely random samples cluster together in some areas and leave others unsampled. Cook [COOK86] suggests the desirability of a minimum-distance Poisson distribution in which no pair of random samples is closer than some minimum distance. Calculating such a distribution is expensive, however, and even if one is created in advance, along with filters to determine each sample's contributions to neighboring pixels, a very large look-up table will be required to store the information. Instead, a satisfactory approximation to the minimum-distance Poisson distribution is obtained by displacing by a small random distance the position of each element of a regularly spaced sample grid. This technique is called *jittering*. In sampling the 2D image plane, each sample in a regular grid is jittered by two uncorrelated random quantities, one each for x and y, both generated with a sufficiently small variance that the samples do not overlap (Fig. 16.59). Figure 16.60 shows a minimum-distance Poisson distribution and a jittered regular distribution. In fact, if the

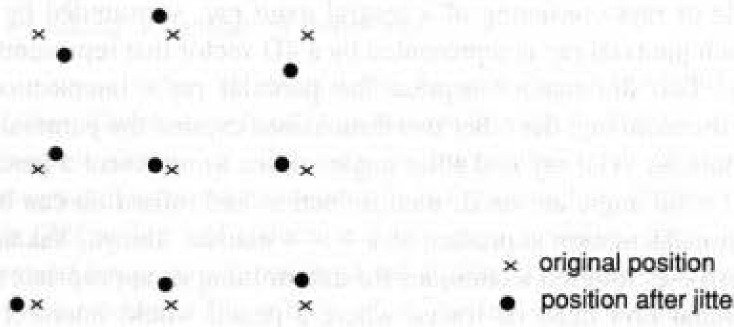


Fig. 16.59 Jittered sampling. Each sample in a regular 2D grid is jittered by two small uncorrelated random quantities. × = original position; ● = position after jitter.

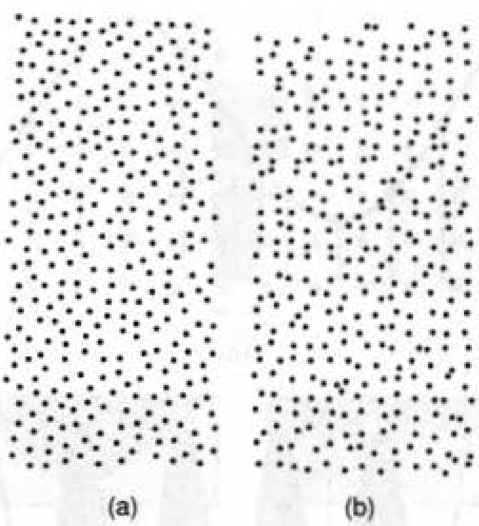


Fig. 16.60 (a) A minimum distance Poisson distribution. (b) A jittered regular distribution. (Courtesy of Mark A. Z. Dippé and Earl Wold, University of California, Berkeley.)

amount of jitter is small compared to the filter width, then the filter can be precomputed, taking into account the positions of the unjittered samples. Cook, Porter, and Carpenter found that a 4 × 4 subpixel grid is adequate for most situations. Poisson and jittered sampling are analyzed in [DIPP85], and strategies for performing adaptive stochastic sampling, including the statistical analysis of samples to determine whether to place new ones, are discussed in [DIPP85; LEE85b; KAJI86; MITC87].

Figure 16.61 compares the use of regularly spaced samples with and without added jitter to sample frequencies at rates above and below the Nyquist rate. In Fig. 16.61(a), sampling above the Nyquist rate, the shape of the signal is captured well, but with some added noise. In Fig. 16.61(b), the sampled amplitude is totally random if there is an integral number of cycles in the sampled range. If there is a fractional number of cycles in the range, then some parts of the waveform have a better chance of being sampled than do others, and thus a combination of aliasing and noise will result. The higher the frequency, the greater the proportion of noise to aliasing. Figure 16.62 demonstrates how a comb of regularly spaced triangles, each $(n + 1)/n$ pixels wide, produces an aliased image when it is sampled by regularly spaced sample points, and produces a noisy image when the sample points are jittered.

Sampling other dimensions. As long as the extra rays needed for spatial antialiasing have been cast, this same basic technique of stochastic sampling can also be used to distribute the rays to sample other aspects of the environment. Motion blur is produced by distributing rays in time. Depth of field is modeled by distributing the rays over the area of the camera lens. The blurred specular reflections and translucent refraction of rough surfaces are simulated by distributing the rays according to the specular reflection and transmission functions. Soft shadows are obtained by distributing the shadow rays over the solid angle subtended by an extended light source as seen from the point being shaded. In all cases, distributed ray tracing uses stochastic sampling to perturb the same rays that would be cast to accomplish spatial antialiasing alone.

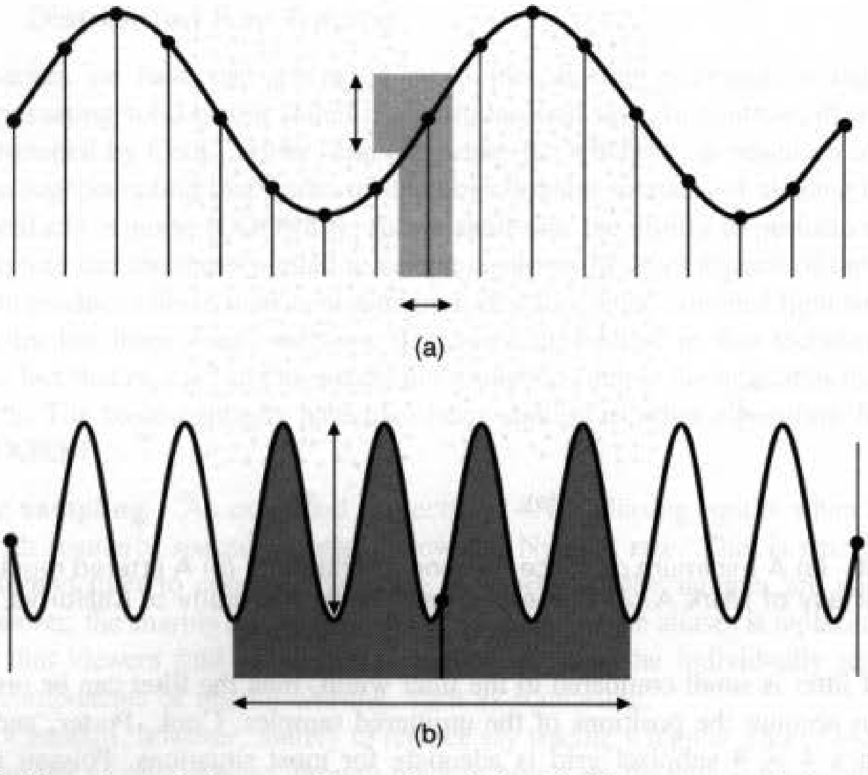


Fig. 16.61 Use of regularly spaced samples with added jitter to sample frequencies (a) above and (b) below the Nyquist rate. The nominal position of each sample is shown as a dot (●). Horizontal arrows indicate the range of jittered sample positions. Vertical arrows indicate how much sampled values can vary. (After [COOK86].)

Sampling in a nonspatial dimension is accomplished by associating each of the pixel's subsampled rays with a range of the value being sampled. Jittering is then used to determine the exact sample point. The ranges may be allocated by partitioning the entire interval being sampled into the same number of subintervals as there are subpixels and randomly allocating subintervals to subpixels. Thus, subpixel *ij* of each pixel is always associated with the same range for a particular dimension. It is important to ensure that the method of allocating the ranges for each dimension does not correlate the values of any two dimensions. For example, if temporally earlier samples tended to sample the left side of an extended light source, and later samples tended to sample the right side, then obscuring the right side of the light early in the temporal interval being depicted might have no effect on the shadow cast. In the case of temporal sampling, each object being intersected must first be moved to its position at the point in time associated with the sampling ray. Cook [COOK86] suggests computing a bounding box for the object's entire path of motion, so that the bounding-box test can be performed without the expense of moving the object.

A weighted distribution in some dimension can be simulated by applying unequal weights to evenly distributed samples. Figure 16.63(a) shows such a distribution. A more attractive alternative, however, is to use *importance sampling,* in which proportionately more sample points are located at positions of higher weight. This is accomplished by dividing the weighting filter into regions of equal area and assigning the same number of equally weighted sample points to each, as shown in Fig. 16.63(b). The amount of jitter

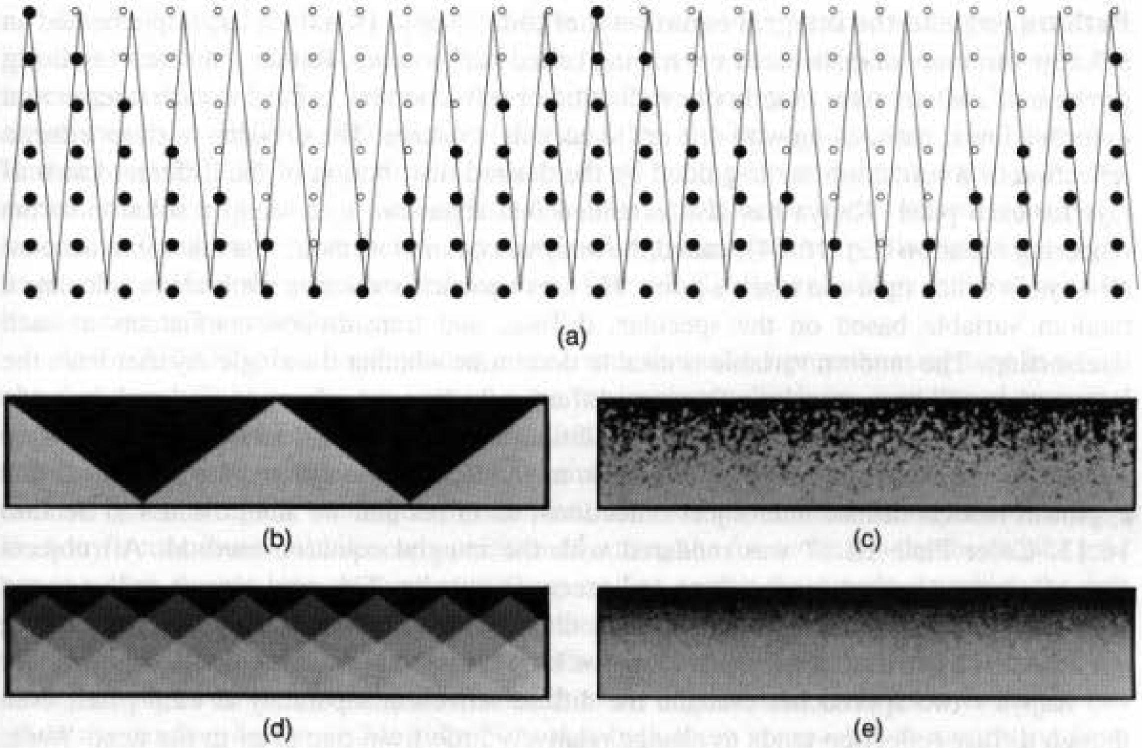


Fig. 16.62 Aliasing vs. noise. (a) A comb with regularly spaced triangles, each $(n + 1)/n$ pixels wide, sampled with one sample per pixel. o = samples that fall outside comb; ● = samples that fall inside comb. (b) A comb with 200 triangles, each 1.01 pixels wide and 50 pixels high. 1 sample/pixel, regular grid. (c) 1 sample/pixel, jittered $\pm\frac{1}{2}$ pixel. (d) 16 samples/pixel, regular grid. (e) 16 samples/pixel, jittered $\pm\frac{1}{8}$ pixel. (Images (b)–(e) by Robert Cook, Lucasfilm Ltd.)

associated with a region also varies in proportion to its size. Color Plate III.16 was created using distributed ray tracing. It shows five billiard balls with motion blur and penumbrae cast by extended light sources. Note the blurred shadows and reflections of the four moving billiard balls.

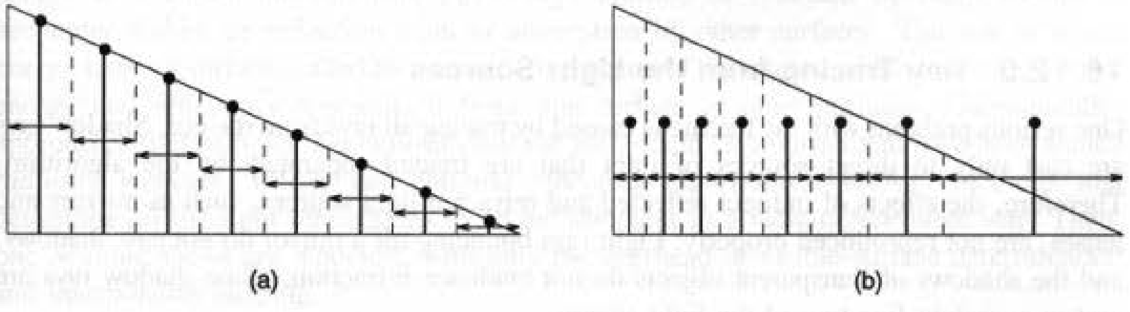


Fig. 16.63 Importance sampling is accomplished by partitioning the weighting function into regions of equal area. The horizontal axis is the dimension sampled; the vertical axis is the weighting. Dots show nominal position of samples; arrows show jitter range. (a) Evenly distributed, unequally weighted samples. (b) Importance sampling: unevenly distributed, equally weighted samples.

Path tracing and the integral equation method. Kajiya [KAJI86] has implemented an efficient variation on distributed ray tracing called *path tracing*. Rather than each ray being grown into a binary tree, exactly one reflection or refraction ray is fired at each intersection to form a linear path, along with one ray to each light source. The decision to shoot either a reflection or a refraction ray is guided by the desired distribution of the different kinds of rays for each pixel. Kajiya has also extended this algorithm to develop a solution to the rendering equation (Eq. 16.54), called the *integral equation method*, that takes into account all ways in which light can reach a point. He uses variance-reduction methods to calculate a random variable based on the specular, diffuse, and transmission coefficients at each intersection. The random variable is used to determine whether the single ray cast from the intersection will be a specular reflection, diffuse reflection, or refraction ray, and the ray's direction is then chosen by sampling. In addition, a shadow ray is cast to a point on a light source, also chosen using variance-reduction methods. Because diffuse rays are traced, this approach models diffuse interobject reflections, an effect that we shall discuss in Section 16.13. Color Plate III.17 was rendered with the integral equation method. All objects shown are gray, except for the floor and green glass balls. The gray objects reflect green light focused by the balls and reddish light diffusely reflected from the floor, phenomena not modeled by conventional ray tracing (or by path tracing).

Kajiya's two approaches evaluate the diffuse reflection separately at each pixel, even though diffuse reflection tends to change relatively little from one pixel to the next. Ward, Rubinstein, and Clear [WARD88] have supplemented a ray tracer with a recursive diffuse reflection stage in which rays are used to trace some number of diffuse bounces from a surface to others illuminating it. Rather than computing the diffuse reflection for each pixel separately, they instead cache all of the values computed. When a pixel is processed, they use an estimate of the illuminance gradient at each cached diffuse reflection value "nearby" the pixel's intersection point to estimate the error associated with using that value. If the error is considered acceptable, then a weighted average of these cached values is used to compute a new value for the pixel; otherwise, a new diffuse calculation is made by tracing rays that sample the hemisphere, and its value is cached. The cached values can then be reused in computing other views of the same scene. Color Plate III.18 shows a series of images rendered using this technique, with different numbers of diffuse bounces. Color Plate III.19 contrasts a photograph of a conference room with a rendered image.

16.12.5 Ray Tracing from the Light Sources

One serious problem with ray tracing is caused by tracing all rays from the eye. Shadow rays are cast only to direct sources of light that are treated separately by the algorithm. Therefore, the effects of indirect reflected and refracted light sources, such as mirrors and lenses, are not reproduced properly: Light rays bouncing off a mirror do not cast shadows, and the shadows of transparent objects do not evidence refraction, since shadow rays are cast in a straight line toward the light source.

It might seem that we would need only to run a conventional ray tracer "backward" from the light sources to the eye to achieve these effects. This concept has been called *backward ray tracing*, to indicate that it runs in the reverse direction from regular ray tracing, but it is also known as *forward ray tracing* to stress that it follows the actual path from the lights to the eye. We call it *ray tracing from the light sources* to avoid confusion!

Done naively, ray tracing from the light sources results in new problems, since an insufficient number of rays ever would strike the image plane, let alone pass through the focusing lens or pinhole. Instead, ray tracing from the light sources can be used to supplement the lighting information obtained by regular ray tracing. Heckbert and Hanrahan [HECK84] suggest an elaboration of their proposed beam-tracing shadow method (Section 16.12.3) to accomplish this. If a light's beam tree is traced recursively, successive levels of the tree below the first level represent indirectly illuminated polygon fragments. Adding these to the database as surface-detail polygons allows indirect specular illumination to be modeled.

Arvo [ARVO86] has implemented a ray tracer that uses a preprocessing step in which rays from each light source are sent into the environment. Each ray is assigned an initial quota of energy, some of which is deposited at each intersection it makes with a diffusely reflecting object. He compensates for the relative sparseness of ray intersections by mapping each surface to a regular rectangular grid of counters that accumulate the deposited energy. Each ray's contribution is bilinearly partitioned among the four counters that bound the grid box in which the ray hits. A conventional ray-tracing pass is then made, in which the first pass's interpolated contributions at each intersection are used, along with the intensities of the visible light sources, to compute the diffuse reflection. Unfortunately, if a light ray strikes an object on the invisible side of a silhouette edge as seen from the eye, the ray can affect the shading on the visible side. Note that both these approaches to ray tracing from the light sources use purely specular reflectivity geometry to propagate rays in both directions.

16.13 RADIOSITY METHODS

Although ray tracing does an excellent job of modeling specular reflection and dispersionless refractive transparency, it still makes use of a directionless ambient-lighting term to account for all other global lighting contributions. Approaches based on thermal-engineering models for the emission and reflection of radiation eliminate the need for the ambient-lighting term by providing a more accurate treatment of interobject reflections. First introduced by Goral, Torrance, Greenberg, and Battaile [GORA84] and by Nishita and Nakamae [NISH85a], these algorithms assume the conservation of light energy in a closed environment. All energy emitted or reflected by every surface is accounted for by its reflection from or absorption by other surfaces. The rate at which energy leaves a surface, called its *radiosity,* is the sum of the rates at which the surface emits energy and reflects or transmits it from that surface or other surfaces. Consequently, approaches that compute the radiosities of the surfaces in an environment have been named *radiosity methods*. Unlike conventional rendering algorithms, radiosity methods first determine all the light interactions in an environment in a view-independent way. Then, one or more views are rendered, with only the overhead of visible-surface determination and interpolative shading.

16.13.1 The Radiosity Equation

In the shading algorithms considered previously, light sources have always been treated separately from the surfaces they illuminate. In contrast, radiosity methods allow any surface to emit light; thus, all light sources are modeled inherently as having area. Imagine breaking up the environment into a finite number n of discrete patches, each of which is assumed to be of finite size, emitting and reflecting light uniformly over its entire area. If we consider each patch to be an opaque Lambertian diffuse emitter and reflector, then, for surface i,

$$B_i = E_i + \rho_i \sum_{1 \le j \le n} B_j F_{j-i} \frac{A_j}{A_i}. \tag{16.58}$$

B_i and B_j are the radiosities of patches i and j, measured in energy/unit time/unit area (i.e., W / m^2). E_i is the rate at which light is emitted from patch i and has the same units as radiosity. ρ_i is patch i's reflectivity and is dimensionless. F_{j-i} is the dimensionless *form factor* or *configuration factor*, which specifies the fraction of energy leaving the entirety of patch j that arrives at the entirety of patch i, taking into account the shape and relative orientation of both patches and the presence of any obstructing patches. A_i and A_j are the areas of patches i and j.

Equation (16.58) states that the energy leaving a unit area of surface is the sum of the light emitted plus the light reflected. The reflected light is computed by scaling the sum of the incident light by the reflectivity. The incident light is in turn the sum of the light leaving the entirety of each patch in the environment scaled by the fraction of that light reaching a unit area of the receiving patch. $B_j F_{j-i}$ is the amount of light leaving a unit area of A_j that reaches all of A_i. Therefore, it is necessary to multiply by the area ratio A_j / A_i to determine the light leaving all of A_j that reaches a unit area of A_i.

Conveniently, a simple reciprocity relationship holds between form factors in diffuse environments,

$$A_i F_{i-j} = A_j F_{j-i}. \tag{16.59}$$

Thus, Eq. (16.58) can be simplified, yielding

$$B_i = E_i + \rho_i \sum_{1 \le j \le n} B_j F_{i-j}. \tag{16.60}$$

Rearranging terms,

$$B_i - \rho_i \sum_{1 \le j \le n} B_j F_{i-j} = E_i. \tag{16.61}$$

Therefore, the interaction of light among the patches in the environment can be stated as a set of simultaneous equations:

$$\begin{bmatrix} 1 - \rho_1 F_{1-1} & -\rho_1 F_{1-2} & \cdots & -\rho_1 F_{1-n} \\ -\rho_2 F_{2-1} & 1 - \rho_2 F_{2-2} & \cdots & -\rho_2 F_{2-n} \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ -\rho_n F_{n-1} & -\rho_n F_{n-2} & \cdots & 1 - \rho_n F_{n-n} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \cdot \\ \cdot \\ \cdot \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \cdot \\ \cdot \\ \cdot \\ E_n \end{bmatrix}. \tag{16.62}$$

Note that a patch's contribution to its own reflected energy must be taken into account (e.g., it may be concave); so, in the general case, each term along the diagonal is not merely 1. Equation (16.62) must be solved for each band of wavelengths considered in the lighting model, since ρ_i and E_i are wavelength-dependent. The form factors, however, are

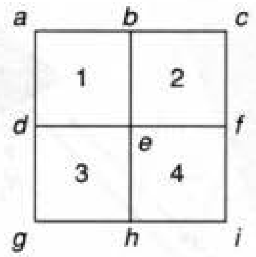


Fig. 16.64 Computing vertex radiosities from patch radiosities.

independent of wavelength and are solely a function of geometry, and thus do not need to be recomputed if the lighting or surface reflectivity changes.

Equation (16.62) may be solved using Gauss–Seidel iteration [PRES88], yielding a radiosity for each patch. The patches can then be rendered from any desired viewpoint with a conventional visible-surface algorithm; the set of radiosities computed for the wavelength bands of each patch are that patch's intensities. Instead of using faceted shading, we can compute vertex radiosities from the patch radiosities to allow intensity interpolation shading.

Cohen and Greenberg [COHE85] suggest the following approach for determining vertex radiosities. If a vertex is interior to a surface, it is assigned the average of the radiosities of the patches that share it. If it is on the edge, then the nearest interior vertex v is found. The radiosity of the edge vertex when averaged with B_v should be the average of the radiosities of the patches that share the edge vertex. Consider the patches in Fig. 16.64. The radiosity for interior vertex e is $B_e = (B_1 + B_2 + B_3 + B_4) / 4$. The radiosity for edge vertex b is computed by finding its nearest interior vertex, e, and noting that b is shared by patches 1 and 2. Thus, to determine B_b, we use the preceding definition; $(B_b + B_e) / 2 = (B_1 + B_2) / 2$. Solving for B_b, we get $B_b = B_1 + B_2 - B_e$. The interior vertex closest to a is also e, and a is part of patch 1 alone. Thus, since $(B_a + B_e) / 2 = B_1$, we get $B_a = 2B_1 - B_e$. Radiosities for the other vertices are computed similarly.

The first radiosity method was implemented by Goral et al. [GORA84], who used contour integrals to compute exact form factors for convex environments with no occluded surfaces, as shown in Color Plate III.20. Note the correct "color-bleeding" effects due to diffuse reflection between adjacent surfaces, visible in both the model and the rendered image: diffuse surfaces are tinged with the colors of other diffuse surfaces that they reflect. For radiosity methods to become practical, however, ways to compute form factors between occluded surfaces had first to be developed.

16.13.2 Computing Form Factors

Cohen and Greenberg [COHE85] adapted an image-precision visible-surface algorithm to approximate form factors for occluded surfaces efficiently. Consider the two patches shown in Fig. 16.65. The form factor from differential area dA_i to differential area dA_j is

$$dF_{di-dj} = \frac{\cos \theta_i \cos \theta_j}{\pi r^2} H_{ij} \, dA_j. \tag{16.63}$$

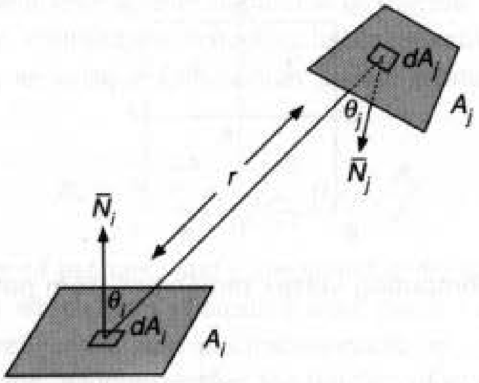


Fig. 16.65 Computing the form factor between a patch and a differential area.

For the ray between differential areas dA_i and dA_j in Fig. 16.65, θ_i is the angle that the ray makes with A_i's normal, θ_j is the angle that it makes with A_j's normal, and r is the ray's length. H_{ij} is either 1 or 0, depending on whether or not dA_j is visible from dA_i. To determine F_{di-j}, the form factor from differential area dA_i to finite area A_j, we need to integrate over the area of patch j. Thus,

$$F_{di-j} = \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij}\, dA_j. \qquad (16.64)$$

Finally, the form factor from A_i to A_j is the area average of Eq. (16.64) over patch i:

$$F_{i-j} = \frac{1}{A_i} \int_{A_i}\int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij}\, dA_j\, dA_i. \qquad (16.65)$$

If we assume that the center point on a patch typifies the patch's other points, then F_{i-j} can be approximated by F_{di-j} computed for dA_i at patch i's center.

Nusselt has shown [SIEG81] that computing F_{di-j} is equivalent to projecting those parts of A_j that are visible from dA_i onto a unit hemisphere centered about dA_i, projecting this projected area orthographically down onto the hemisphere's unit circle base, and dividing by the area of the circle (Fig. 16.66). Projecting onto the unit hemisphere accounts for $\cos\theta_j / r^2$ in Eq. (16.64), projecting down onto the base corresponds to a multiplication by $\cos\theta_i$, and dividing by the area of the unit circle accounts for the π in the denominator.

Rather than analytically projecting each A_j onto a hemisphere, Cohen and Greenberg developed an efficient image-precision algorithm that projects onto the upper half of a cube centered about dA_i, with the cube's top parallel to the surface (Fig. 16.67). Each face of this *hemicube* is divided into a number of equal-sized square cells. (Resolutions used in pictures included in this book range from 50 by 50 to several hundred on a face.) All the other patches are clipped to the view-volume frusta defined by the center of the cube and each of its upper five faces, and then each of the clipped patches is projected onto the appropriate face of the hemicube. An item-buffer algorithm (Section 16.12.1) is used that records the

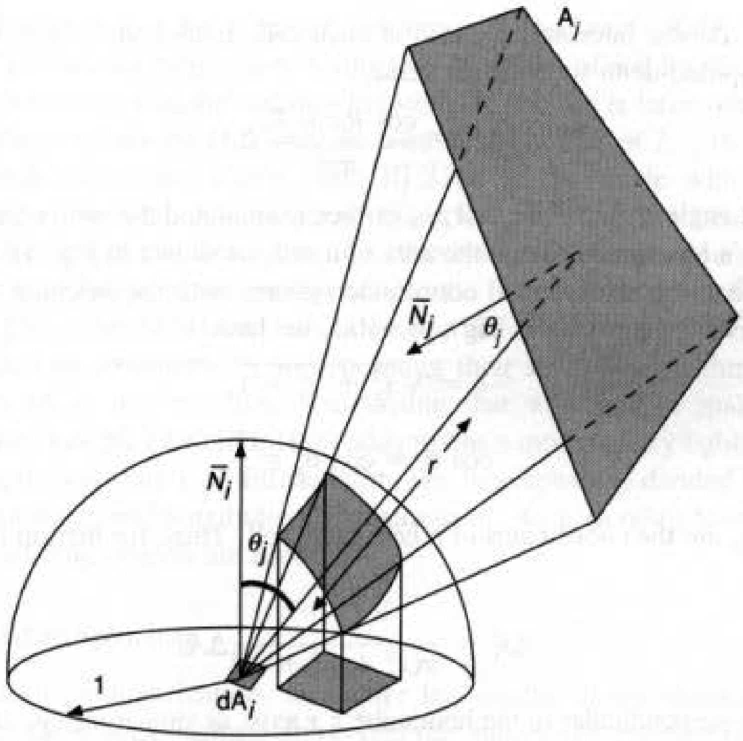


Fig. 16.66 Determining the form factor between a differential area and a patch using Nusselt's method. The ratio of the area projected onto the hemisphere's base to the area of the entire base is the form factor. (After [SIEG81].)

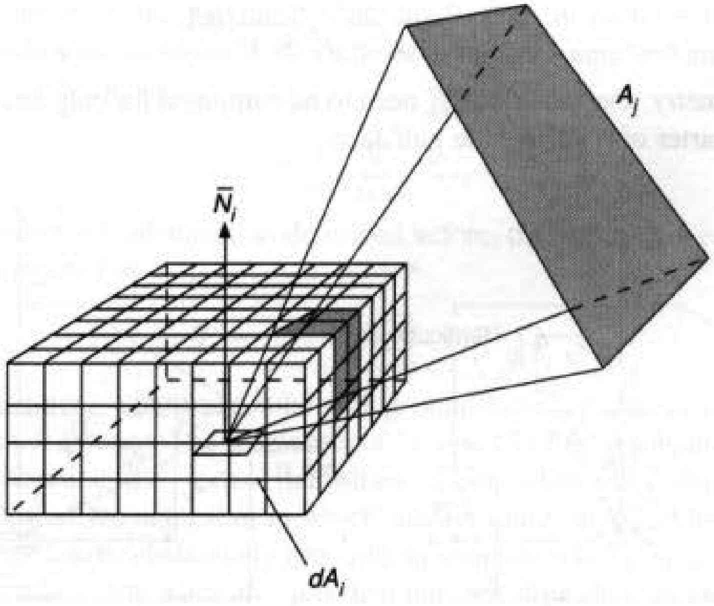


Fig. 16.67 The hemicube is the upper half of a cube centered about the patch. (After [COHE85].)

identity of the closest intersecting patch at each cell. Each hemicube cell p is associated with a precomputed delta form factor value,

$$\Delta F_p = \frac{\cos\theta_i \cos\theta_p}{\pi r^2}\Delta A, \tag{16.66}$$

where θ_p is the angle between the cell p's surface normal and the vector between dA_i and p, r is this vector's length, and ΔA is the area of a cell, as shown in Fig. 16.68. Assume that the hemicube has its own (x, y, z) coordinate system, with the origin at the center of the bottom face. For the top face in Fig. 16.68(a), we have

$$r = \sqrt{x_p^2 + y_p^2 + 1}, \tag{16.67}$$

$$\cos\theta_i = \cos\theta_p = \frac{1}{r},$$

where x_p and y_p are the coordinates of a hemicube cell. Thus, for the top face, Eq. (16.66) simplifies to

$$\Delta F_p = \frac{1}{\pi(x_p^2 + y_p^2 + 1)^2}\Delta A. \tag{16.68}$$

For a side face perpendicular to the hemicube's x axis, as shown in Fig. 16.68(b), we have

$$r = \sqrt{y_p^2 + z_p^2 + 1}, \tag{16.69}$$

$$\cos\theta_i = \frac{z_p}{r}, \quad \cos\theta_p = \frac{1}{r}.$$

Here, Eq. (16.66) simplifies to

$$\Delta F_p = \frac{z_p}{\pi(y_p^2 + z_p^2 + 1)^2}\Delta A. \tag{16.70}$$

Because of symmetry, the values of ΔF_p need to be computed for only one-eighth of the top face and one-quarter of a single side half face.

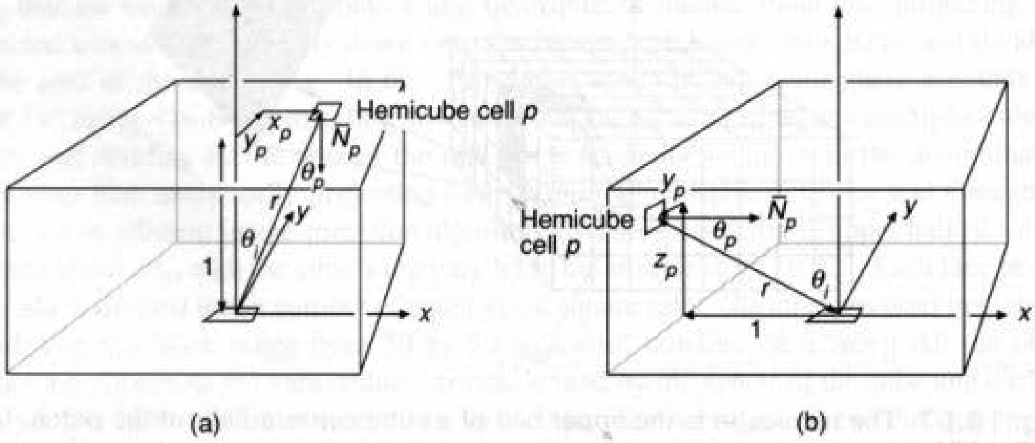


Fig. 16.68 Delta form factors. (a) The top face. (b) A side face. (After [COHE85].)

We can approximate F_{di-j} for any patch j by summing the values of ΔF_p associated with each cell p in A_j's hemicube projections. (Note that the values of ΔF_p for all the hemicube's cells sum to 1.) Assuming that the distance between the patches is large relative to the size of the patches, these values for F_{di-j} may be used as the values of F_{i-j} in Eq. (16.62) to compute the patch radiosities. Color Plate III.21(a–b) was made with the hemicube algorithm. Because much of the computation performed using the hemicube involves computing item buffers, it can take advantage of existing z-buffer hardware. On the other hand, because it uses image-precision operations, the hemicube is prone to aliasing.

Nishita and Nakamae [NISH85a] have adopted a different approach to computing form factors in occluded environments by incorporating their shadow algorithm for area light sources (Section 16.8) into a radiosity algorithm that was used to make Color Plate III.22(a–b). Color Plate III.22(c) [NISH86] adds to this a model of sky light, approximated by a large hemispherical source of diffuse light. The hemisphere is divided into bands that are transversely uniform and longitudinally nonuniform. As with other luminous surfaces, the effects of occluding objects are modeled.

16.13.3 Substructuring

The finer the patch parametrization, the better the results, at the expense of increased computation time for n^2 form factors. To prevent this square-law increase in the number of form factors, Cohen, Greenberg, Immel, and Brock [COHE86] adaptively subdivide patches into subpatches at places in the patch mesh at which a high radiosity gradient is found. The subpatches created by this *substructuring* process are not treated like full-fledged patches. Whenever a patch i is subdivided into subpatches, the form factors F_{s-j} from each subpatch s to each patch j are computed using the hemicube technique, but form factors from the patches to the subpatches are not computed. After a patch has been broken into subpatches, however, the previously calculated values of each form factor from the patch to other patches are replaced by the more accurate area-weighted average of the form factors from its m subpatches:

$$F_{i-j} = \frac{1}{A_i} \sum_{1 \le s \le m} F_{s-j} A_s. \tag{16.71}$$

After patch radiosities are calculated as described before, the radiosity of each subpatch s of patch i can be computed as

$$B_s = E_i + \rho_i \sum_{1 \le j \le n} B_j F_{s-j}. \tag{16.72}$$

The algorithm iterates, adaptively subdividing subpatches at places of high radiosity gradient, until the differences reach an acceptable level. The final subpatch radiosities are then used to determine the vertex radiosities. Color Plate III.21(b), made using a nonadaptive version of the algorithm in which subdivision is specified by the user, shows that the same image takes substantially less time to compute when patches are divided into one level of subpatches, than when an equivalent number of patches are used. The adaptive version of the algorithm is initialized with a "first guess" subdivision specified by the user. Color Plate III.21(c) was created by adaptively subdividing the subpatches of Color Plate III.21(b). Note the improved shadow resolution about the table's legs.

Substructuring allows subpatch radiosities to be determined without changing the size of the matrix to be solved in Eq. (16.62). Note that a subpatch's contribution to other patches is still approximated coarsely by its patch's radiosity, but this is a second-order effect in diffuse environments. In a similar fashion, texture mapping can be implemented by computing a single average reflectivity value for a texture-mapped patch that is used for the radiosity computations [COHE86]. When each pixel in the texture-mapped surface is finally rendered, its shade is scaled by the ratio of the texture-map reflectivity value computed for the pixel and the average reflectivity used for the patch.

16.13.4 Progressive Refinement

Given the high costs of executing the radiosity algorithm described thus far, it makes sense to ask whether it is possible to approximate the algorithm's results incrementally. Can we produce a useful, although perhaps inaccurate, image early on, which can be successively refined to greater accuracy as more time is allocated? The radiosity approach described in the previous sections will not let us do this for two reasons. First, an entire Gauss–Seidel iteration must take place before an estimate of the patch radiosities becomes available. Second, form factors are calculated between all patches at the start and must be stored throughout the computation, requiring $O(n^2)$ time and space. Cohen, Chen, Wallace, and Greenberg [COHE88] have developed a progressive-refinement radiosity algorithm that addresses both of these issues.

Consider the approach described thus far. Evaluating the ith row of Eq. (16.62) provides an estimate of patch i's radiosity, B_i, expressed in Eq. (16.60), based on the estimates of the other patch radiosities. Each term of the summation in Eq. (16.60) represents patch j's effect on the radiosity of patch i:

$$B_i \text{ due to } B_j = \rho_i B_j F_{i-j}, \qquad \text{for all } j. \tag{16.73}$$

Thus, this approach *gathers* the light from the rest of the environment. In contrast, the progressive-refinement approach *shoots* the radiosity from a patch into the environment. A straightforward way to do this is to modify Eq. (16.73) to yield

$$B_j \text{ due to } B_i = \rho_j B_i F_{j-i}, \qquad \text{for all } j. \tag{16.74}$$

Given an estimate of B_i, the contribution of patch i to the rest of the environment can be determined by evaluating Eq. (16.74) for each patch j. Unfortunately, this will require knowing F_{j-i} for each j, each value of which is determined with a separate hemicube. This imposes the same overwhelmingly large space–time overhead as does the original approach. By using the reciprocity relationship of Eq. (16.59), however, we can rewrite Eq. (16.74) as

$$B_j \text{ due to } B_i = \rho_j B_i F_{i-j} \frac{A_i}{A_j}, \qquad \text{for all } j. \tag{16.75}$$

Evaluating this equation for each j requires only the form factors calculated using a single hemicube centered about patch i. If the form factors from patch i can be computed quickly (e.g., by using z-buffer hardware), then they can be discarded as soon as the radiosities shot from patch i have been computed. Thus, only a single hemicube and its form factors need to be computed and stored at a time.

As soon as a patch's radiosity has been shot, another patch is selected. A patch may be selected to shoot again after new light has been shot to it from other patches. Therefore, it is not patch i's total estimated radiosity that is shot, but rather ΔB_i, the amount of radiosity that patch i has received since the last time that it shot. The algorithm iterates until the desired tolerance is reached. Rather than choose patches in random order, it makes sense to select the patch that will make the most difference. This is the patch that has the most energy left to radiate. Since radiosity is measured per unit area, a patch i is picked for which $\Delta B_i A_i$ is the greatest. Initially, $B_i = \Delta B_i = E_i$ for all patches, which is nonzero only for light sources. The pseudocode for a single iteration is shown in Fig. 16.69.

Each execution of the pseudocode in Fig. 16.69 will cause another patch to shoot its unshot radiosity into the environment. Thus, the only surfaces that are illuminated after the first execution are those that are light sources and those that are illuminated directly by the first patch whose radiosity is shot. If a new picture is rendered at the end of each execution, the first picture will be relatively dark, and those following will get progressively brighter. To make the earlier pictures more useful, we can add an ambient term to the radiosities. With each additional pass through the loop, the ambient term will be decreased, until it disappears.

One way to estimate the ambient term uses a weighted sum of the unshot patch radiosities. First, an average diffuse reflectivity for the environment, ρ_{avg} is computed as a weighted sum of the patch diffuse reflectivities,

$$\rho_{avg} = \sum_{1 \le i \le n} \rho_i A_i \,/ \sum_{1 \le i \le n} A_i. \qquad (16.76)$$

This equation is used to compute an overall reflection factor R, intended to take into account the different reflected paths through which energy can travel from one patch to another,

$$R = 1 + \rho_{avg} + \rho_{avg}^2 + \rho_{avg}^3 + \ldots = \frac{1}{1 - \rho_{avg}}. \qquad (16.77)$$

```
select patch $i$;
calculate $F_{i-j}$ for each patch $j$;

for (each patch $j$) {
    $\Delta Radiosity = \rho_j \Delta B_i F_{i-j} A_i / A_j$;
    $\Delta B_j$ += $\Delta Radiosity$;
    $B_j$ += $\Delta Radiosity$;
}

$\Delta B_i = 0$;
```

Fig. 16.69 Pseudocode for shooting radiosity from a patch.

```
for (each patch i) {
    $\Delta B_i = E_i$;
    for (each subpatch s in i)
        $B_s = E_i$;
}

$AreaSum = \sum_{1 \le i \le n} A_i$;

$Ambient = R \sum_{1 \le i \le n} (\Delta B_i A_i) / AreaSum$;

while (not converged) {
    select patch i with greatest $\Delta B_i A_i$;
    determine $F_{i-s}$ for all subpatches s in all patches;

    /* ΔEnergy is initialized to the total energy shot. */
    $\Delta Energy = \Delta B_i A_i$;

    /* Shoot radiosity from patch i. */
    for (each patch j seen by i) {
        $Old\Delta B = \Delta B_j$;
        for (each subpatch s in j seen by i) {
            $\Delta Radiosity = \rho_j \Delta B_i F_{i-s} A_i / A_s$;
            $B_s$ += $\Delta Radiosity$;
            $\Delta B_j$ += $\Delta Radiosity\ A_s / A_j$;
        }
        /* Decrement ΔEnergy by total energy gained by patch j. */
        $\Delta Energy$ −= $(\Delta B_j - Old\Delta B)\ A_j$;
    }

    determine vertex radiosities from subpatch radiosities, using
        $B_s + \rho_j$ Ambient as radiosity of subpatch s of patch j;
    if (radiosity gradient between adjacent vertices is too high)
        subdivide offending subpatches and reshoot from patch i to them;
    $\Delta B_i = 0$;

    perform view-dependent visible-surface determination and shading;

    /* Use ΔEnergy (energy absorbed by patches hit) to determine new value of Ambient. */
    Ambient −= $R\ \Delta Energy\ /\ AreaSum$;
} /* while */
```

Fig. 16.70 Pseudocode for progressive-refinement radiosity method with ambient light and substructuring.

Each patch's unshot radiosity is weighted by the ratio of the patch's area to the environment's area, providing an approximation to the form factor from an arbitrary differential area to that patch. Thus, the estimate of the ambient term accounting for unshot radiosity is

$$Ambient = R \sum_{1 \le i \le n} (\Delta B_i A_i) \; / \sum_{1 \le i \le n} A_i. \qquad (16.78)$$

This ambient term is used to augment the patch's radiosity for display purposes only, yielding

$$B'_i = B_i + \rho_i Ambient. \qquad (16.79)$$

Figure 16.70 shows the pseudocode for the entire algorithm. Substructuring is provided by shooting radiosity from patches to subpatches to determine subpatch radiosities. Thus, hemicubes are created for patches, but not for subpatches. Adaptive subdivision is accomplished by subdividing a patch further when the radiosity gradient between adjacent subpatch vertices is found to be too high. Color Plate III.23, which is rendered using an ambient term, depicts stages in the creation of an image after 1, 2, 24, and 100 iterations.

16.13.5 Computing More Accurate Form Factors

Although the use of fast *z*-buffer hardware makes the hemicube an efficient algorithm, the technique has a number of failings [BAUM89; WALL89]:

- Recall that the identity of only one patch is stored per hemicube pixel. Therefore, a grid of patches may alias when projected onto a side of the hemicube, just as they would when processed with a *z*-buffer algorithm. This can show up as a regular pattern of patches that are not represented in the hemicube. Furthermore, a patch that is small when projected on the hemicube may be large when projected on the image plane.
- Use of the hemicube assumes that the center point of a patch is representative of the patch's visibility to other patches. If this assumption is shown to be untrue, the surface can be broken up into subpatches, but there is only a single subdivision granularity for the patch; the same patch cannot be subdivided to different levels for different patches that it views.
- Patches must be far from each other for the hemicube approach to be correct. This is a serious problem if two patches are adjacent; since all calculations are done from the center of the hemicube, the form factor will be underestimated, because the calculations do not take into account the proximity of the adjacent parts of the patches.

A progressive radiosity approach developed by Wallace, Elmquist, and Haines [WALL89] uses ray tracing to evaluate form factors, instead of the hemicube. When a source patch is to shoot its radiosity, rays are fired from each vertex in the scene to the source to compute the form factor from the source to the vertex. This is accomplished by decomposing the source patch into a number of small finite subareas, each of which is the target of a ray shot from a vertex. If the ray is not occluded, then the target is visible, and the form factor between the differential vertex and the finite area target is computed, using an analytic expression, based on some simplifying geometric assumptions. If desired, the ray intersection calculations can be performed with a resolution-independent true curved-surface database, so that the time for an individual ray test does not depend on the number of polygons. The form factor between the vertex and the entire source is computed as an area-weighted average of the form factors between each subarea and the vertex, and the result is used to compute the contribution of the source to the vertex. This approach has a number of advantages. Radiosities are computed at the vertices themselves, where they are ultimately needed for shading. Vertex normals can be used, allowing polygonal meshes that approximate curved surfaces. Nonphysical point light sources can be handled by tracing a single ray to the light source and using its illumination equation to determine the irradiance at each vertex. The number of areas into which a source is decomposed and whether rays are actually fired (i.e., if shadow testing is to be performed) can all be determined individually for each vertex. Color Plates III.24 and III.25 were created using this algorithm.

A contrasting approach to solving inaccuracies caused by the hemicube is taken by Baum, Rushmeier, and Winget [BAUM89]. They recognize that the hemicube form factors often are accurate; therefore, they have developed error-analysis tests to choose, for each patch, when to use the hemicube, when to subdivide the patch further, and when to use a more expensive, but more accurate, analytic technique for computing form factors.

16.13.6 Specular Reflection

The radiosity methods described so far treat only diffuse reflection. Therefore, all of a patch's radiosity may be treated uniformly when it is dispersed to other patches: The radiosity leaving a patch in any direction is influenced by the patch's total radiosity, not by the directions from which its incoming energy was acquired. Immel, Cohen, and Greenberg [IMME86] extended the radiosity method to model specular reflection. Rather than compute a single radiosity value for each patch, they partition the hemisphere over the patch into a finite set of solid angles, each of which establishes a direction for incoming or outgoing energy. Given the patch's bidirectional reflectivity (Section 16.7), they compute the outgoing radiosity in each direction in terms of its emittance in that direction and the incident light from each of the set of directions, weighting each direction's contribution accordingly. Finally, they render a picture from intensities that are determined at each vertex by using the direction from the vertex to the eye to interpolate among the closest directional radiosities. Although the results shown in Color Plate III.26 are promising, the approach has a tremendous overhead in both time and space, which will only increase if highly specular surfaces are modeled. One solution is to combine a radiosity method with ray tracing.

16.13.7 Combining Radiosity and Ray Tracing

Consider the tradeoffs between radiosity methods and ray tracing. Radiosity methods are well suited to diffuse reflection because a diffuse surface's bidirectional reflectivity is constant in all outgoing directions. Thus, all radiosities computed are view-independent. On the other hand, the pure radiosity method for specular surfaces described previously is not practical, because specular reflection from a surface is highly dependent on the angle with which an observer (or another surface) views the surface. Therefore, much extra information must be computed, because no information about the desired view is provided. In addition, this directional information is discretized and must be interpolated to accommodate a specific view. Not only does the interpolation make it impossible to model sharp reflections, but also the sampling performed by the discretization can result in aliasing.

In contrast, ray tracing calculates specular reflections well, since the eyepoint is known in advance. Although conventional ray tracing does not model global diffuse phenomena, some of the approaches discussed in Section 16.12.4 do. Correctly solving for the diffuse reflection from a piece of surface requires that all the surfaces with which a surface exchanges energy be taken into account; in short, it requires a radiosity method.

It makes sense to combine ray tracing and radiosity to take advantage of ray tracing's ability to model specular phenomena and of the radiosity method's ability to model diffuse interactions. Unfortunately, simply summing the pixel values computed by a diffuse radiosity method and a specular ray tracer will not suffice. For example, the diffuse radiosity method will fail to take into account the extra illumination falling on a diffuse surface from a specular surface. It is necessary to account for transfer from diffuse to diffuse, diffuse to specular, specular to diffuse, and specular to specular reflection.

Wallace, Cohen, and Greenberg [WALL87] describe a two-pass approach that combines a view-independent radiosity method, executed in the first pass, with a view-dependent ray-tracing approach, executed in the second pass. As mentioned previously, the first pass must take into account specular, as well as diffuse, reflection. If only perfect, mirrorlike specular reflection is allowed, this can be supported by reflecting each patch about the plane of a specular surface [RUSH86]. Each specular patch is thus treated as a window onto a "mirror world." The form factor from a patch to one of these mirror reflections accounts for the specular reflection from the patch that is doing the mirroring.

In the second view-dependent pass, a *reflection frustum* is erected at each point on a surface that corresponds to a pixel in the image. As shown in Fig. 16.71, the reflection frustum consists of a little z-buffer, positioned perpendicular to the reflection direction, and covering the small incoming solid angle that is most significant for the surface's

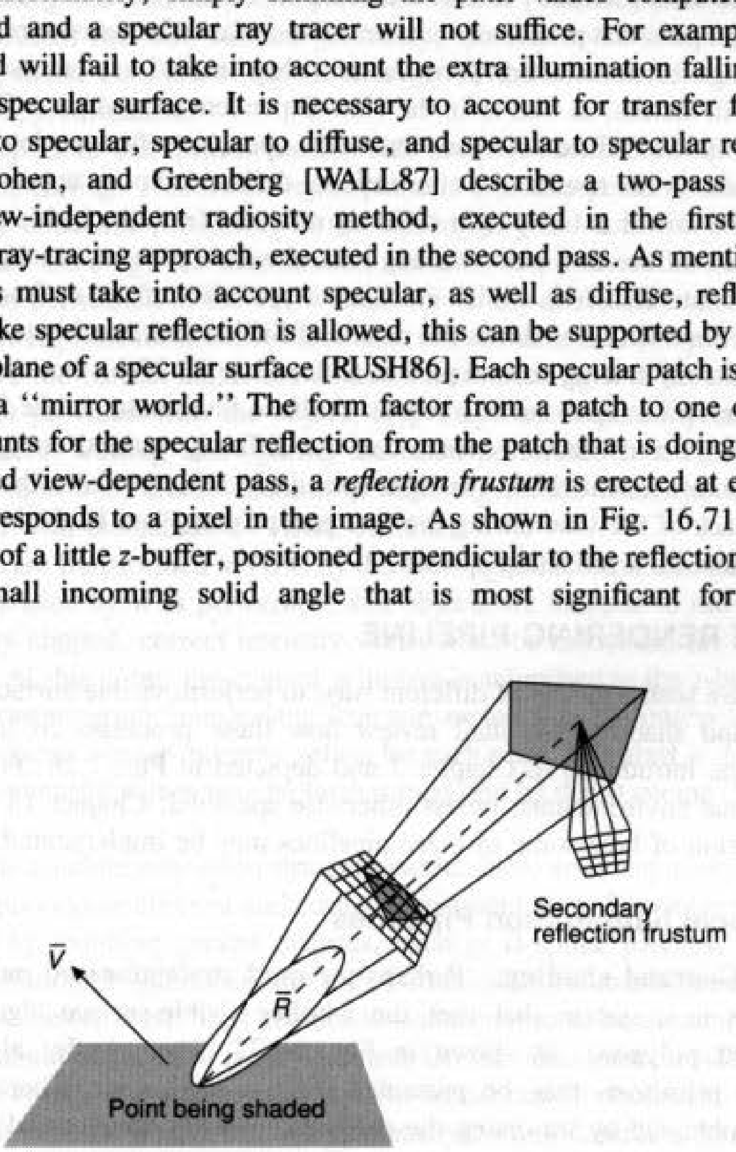


Fig. 16.71 The reflection frustum. (After [WALL87].)

bidirectional reflectivity. The patches are z-buffered onto the frustum, using Gouraud shading to interpolate the patches' first-pass diffuse intensities across their projections. A ray is traced recursively through each pixel on the frustum that sees a specular surface, spawning a new reflection frustum at each intersection. The values computed for each frustum pixel are then weighted to model the surface's ρ_s. The second pass thus uses specular transport to combine radiosities determined during the first pass. Transparency can be accommodated by erecting a transmission frustrum in the direction of refraction. The image on the cover of this book (Color Plate I.9) was created using this algorithm.

The mirror-world approach used in the first pass handles only perfect specular reflection and results in a proliferation of form factors. Shao, Peng, and Liang [SHAO88] have implemented a two-pass approach that allows Phong-like bidirectional reflectance functions in the first pass, without the need to duplicate patches.

Sillion and Puech [SILL89] extend the two-pass technique to calculate *extended form factors* in the first pass that model any number of specular reflections or refractions. Rather than proliferating mirror-reflection form factors, they instead use recursive ray tracing to compute the form factors, as well as in the view-dependent second pass. Color Plate III.27 demonstrates why the diffuse first pass must take specular reflection into account. Color Plate III.27(a) shows the results of a conventional diffuse radiosity approach. (The part of the table near the mirror is lit by light diffusely reflected from the inside of the tall lamp.) The conventional diffuse first pass was augmented with a pure ray-tracing second pass to produce Color Plate III.27(b), which includes a specular reflection from the mirror. In contrast, Color Plate III.27(c) shows the results of Sillion and Puech's two-pass approach. It shares the same ray-tracing second pass as Color Plate III.27(b), but uses extended form factors in the first pass. Each surface acts as a diffuse illuminator in the first pass, but the use of the extended form factors means that the diffusely emitted energy takes specular interreflection into account. Note the light specularly reflected from the mirror onto the table and the back of the vase during the first pass. Color Plate III.28 is a more complex example that includes a reflecting sphere.

16.14 THE RENDERING PIPELINE

Now that we have seen a variety of different ways to perform visible-surface determination, illumination, and shading, we shall review how these processes fit into the standard graphics pipeline introduced in Chapter 7 and depicted in Fig. 7.26. For simplicity, we assume polygonal environments, unless otherwise specified. Chapter 18 provides a more detailed discussion of how some of these pipelines may be implemented in hardware.

16.14.1 Local Illumination Pipelines

z-buffer and Gouraud shading. Perhaps the most straightforward modification to the pipeline occurs in a system that uses the z-buffer visible-surface algorithm to render Gouraud-shaded polygons, as shown in Fig. 16.72. The z-buffer algorithm has the advantage that primitives may be presented to it in any order. Therefore, as before, primitives are obtained by traversing the database, and are transformed by the modeling transformation into the WC system.

Primitives may have associated surface normals that were specified when the model was built. Since the lighting step will require the use of surface normals, it is important to

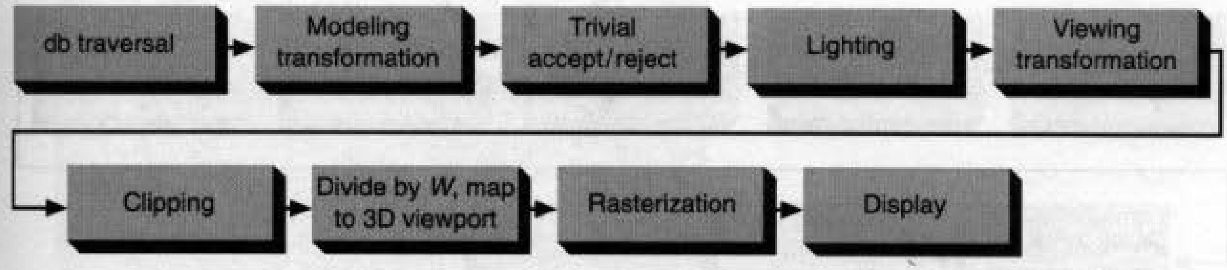


Fig. 16.72 Rendering pipeline for *z*-buffer and Gouraud shading.

remember that normals must be transformed correctly, using the methods discussed in the Appendix. Furthermore, we cannot just ignore stored normals and attempt to recompute new ones later using the correctly transformed vertices. The normals defined with the objects may represent the true surface geometry, or may specify user-defined surface blending effects, rather than just being the averages of the normals of shared faces in the polygonal mesh approximation.

Our next step is to cull primitives that fall entirely outside of the window and to perform back-face culling. This trivial-reject phase is typically performed now because we want to eliminate unneeded processing in the lighting step that follows. Now, because we are using Gouraud shading, the illumination equation is evaluated at each vertex. This operation must be performed in the WC system (or in any coordinate system isometric to it), before the viewing transformation (which may include skew and perspective transformations), to preserve the correct angle and distance from each light to the surface. If vertex normals were not provided with the object, they may be computed immediately before lighting the vertices. Culling and lighting are often performed in a lighting coordinate system that is a rigid body transformation of WC (e.g., VRC when the view orientation matrix is created with the standard PHIGS utilities).

Next objects are transformed to NPC by the viewing transformation, and clipped to the view volume. Division by W is performed, and objects are mapped to the viewport. If an object is partially clipped, correct intensity values must be calculated for vertices created during clipping. At this point, the clipped primitive is submitted to the z-buffer algorithm, which performs rasterization, interleaving scan conversion with the interpolation needed to compute the z value and color-intensity values for each pixel. If a pixel is determined to be visible, its color-intensity values may be further modified by depth cueing (Eq. 16.11), not shown here.

Although this pipeline may seem straightforward, there are many new issues that must be dealt with to provide an efficient and correct implementation. For example, consider the problems raised by handling curved surfaces, such as B-spline patches, which must be tessellated. Tessellation should occur after transformation into a coordinate system in which screen size can be determined. This enables tessellation size to be determined adaptively, and limits the amount of data that is transformed. On the other hand, tessellated primitives must be lit in a coordinate system isometric to world coordinates. Abi-Ezzi [ABIE89] addresses these issues, proposing a more efficient, yet more complex, formulation of the pipeline that incorporates feedback loops. This new pipeline uses a lighting coordinate system that is an isometric (i.e., rigid or Euclidean) transformation of WC, yet is computationally close to DC to allow tessellation decisions to be made efficiently.

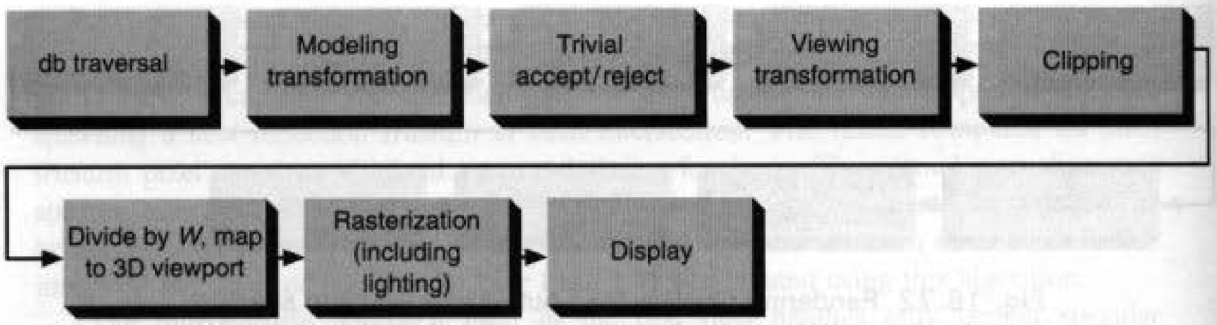


Fig. 16.73 Rendering pipeline for *z*-buffer and Phong shading.

***z*-buffer and Phong shading.** This simple pipeline must be modified if we wish to accommodate Phong shading, as shown in Fig. 16.73. Because Phong shading interpolates surface normals, rather than intensities, the vertices cannot be lit early in the pipeline. Instead, each object must be clipped (with properly interpolated normals created for each newly created vertex), transformed by the viewing transformation, and passed to the *z*-buffer algorithm. Finally, lighting is performed with the interpolated surface normals that are derived during scan conversion. Thus, each point and its normal must be backmapped into a coordinate system that is isometric to WC to evaluate the illumination equation.

List-priority algorithm and Phong shading. When a list-priority algorithm is used, primitives obtained from traversal and processed by the modeling transformation are inserted in a separate database, such as a BSP tree, as part of preliminary visible-surface determination. Figure 16.74 presents the pipeline for the BSP tree algorithm, whose preliminary visible-surface determination is view-independent. As we noted in Chapter 7, the application program and the graphics package may each keep separate databases. Here, we see that rendering can require yet another database. Since, in this case, polygons are split, correct shading information must be determined for the newly created vertices. The rendering database can now be traversed to return primitives in a correct, back-to-front order. The overhead of building this database can, of course, be applied toward the creation of multiple pictures. Therefore, we have shown it as a separate pipeline whose output is a new database. Primitives extracted from the rendering database are clipped and normalized, and are presented to the remaining stages of the pipeline. These stages are structured much like those used for the *z*-buffer pipeline, except that the only visible-surface process they need to perform is to guarantee that each polygon will correctly overwrite any

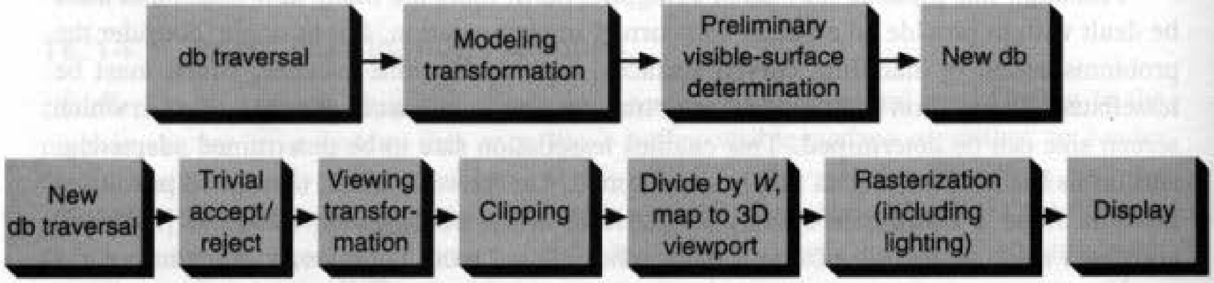


Fig. 16.74 Rendering pipeline for list-priority algorithm and Phong shading.

Plate III.1 Grass and trees. 2×10^9 objects, organized in a hierarchy of lists and grids (Section 15.10.2). Ray traced at 512×512 resolution with 16 rays/pixel, taking 16 hours on an IBM 4381/Group 12. (John Snyder and Alan Barr, © Caltech Computer Graphics Group.)

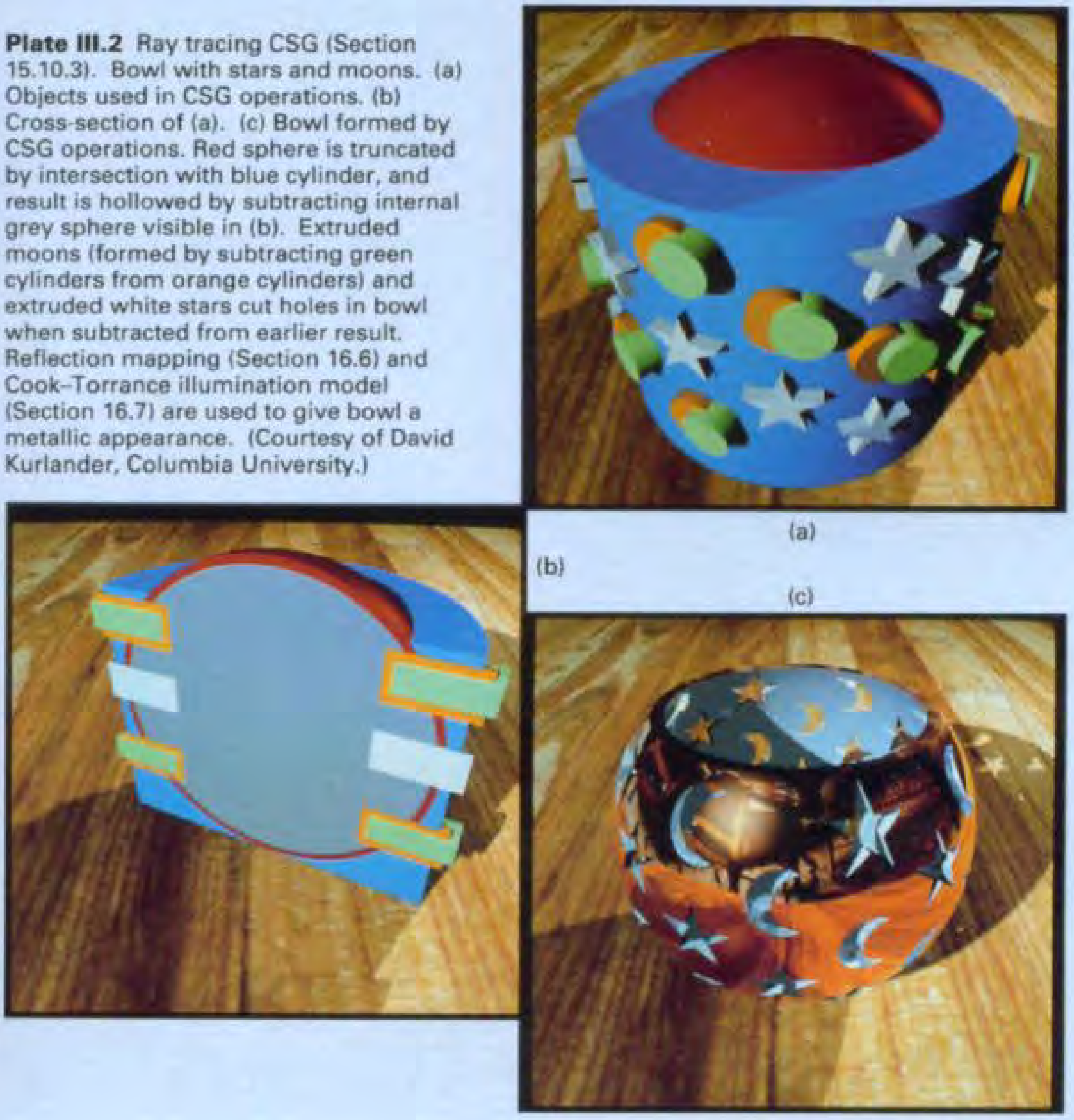


Plate III.2 Ray tracing CSG (Section 15.10.3). Bowl with stars and moons. (a) Objects used in CSG operations. (b) Cross-section of (a). (c) Bowl formed by CSG operations. Red sphere is truncated by intersection with blue cylinder, and result is hollowed by subtracting internal grey sphere visible in (b). Extruded moons (formed by subtracting green cylinders from orange cylinders) and extruded white stars cut holes in bowl when subtracted from earlier result. Reflection mapping (Section 16.6) and Cook–Torrance illumination model (Section 16.7) are used to give bowl a metallic appearance. (Courtesy of David Kurlander, Columbia University.)

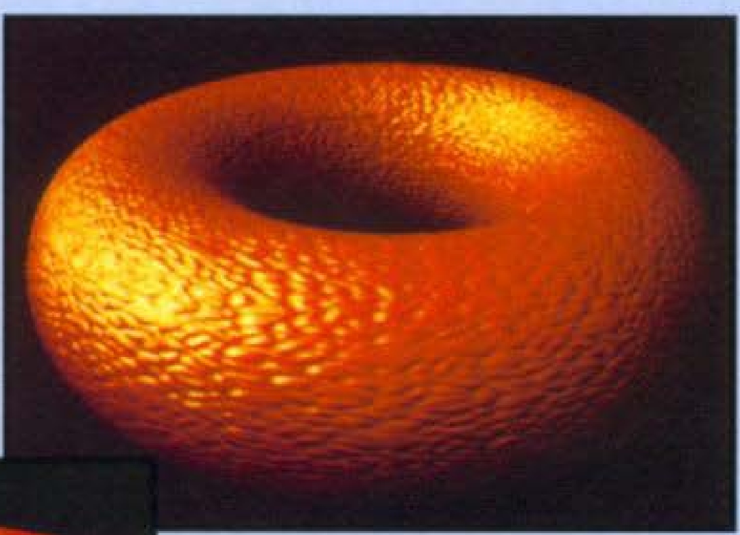


Plate III.3 A torus bump mapped with a hand-generated bump function (Section 16.3.3). (By Jim Blinn. Courtesy of University of Utah.)

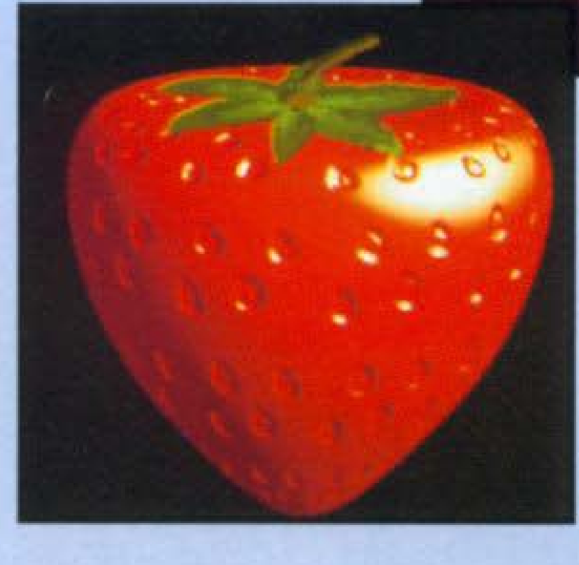
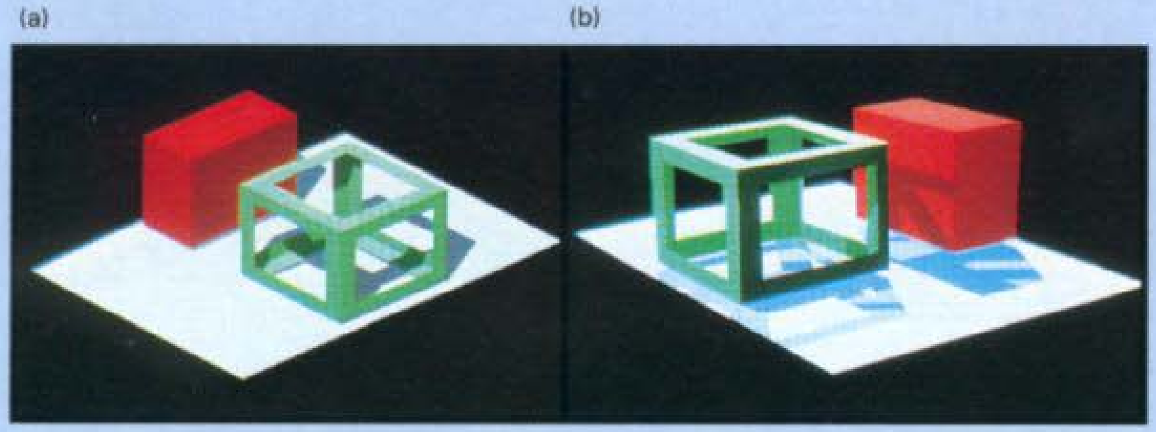
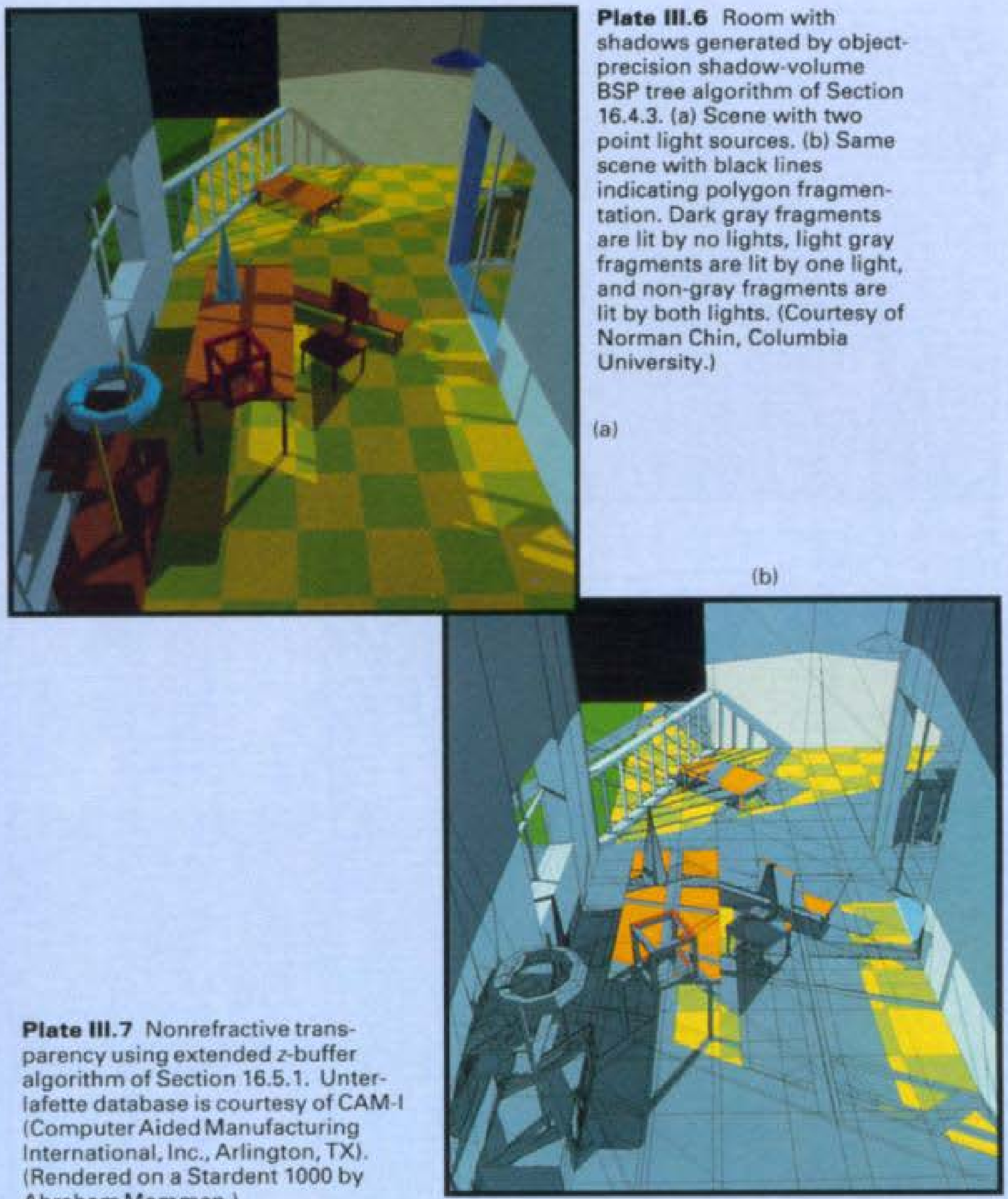


Plate III.4 A strawberry bump mapped with a hand-generated bump function (Section 16.3.3). (By Jim Blinn. Courtesy of University of Utah.)

Plate III.5 Objects with shadows generated by two-pass object-precision algorithm of Section 16.4.2. (a) One light source. (b) Two light sources. (Peter Atherton, Kevin Weiler, and Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1978.)



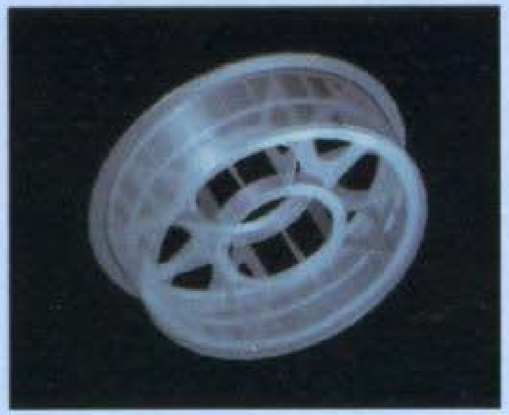


(a)

(b)

Plate III.6 Room with shadows generated by object-precision shadow-volume BSP tree algorithm of Section 16.4.3. (a) Scene with two point light sources. (b) Same scene with black lines indicating polygon fragmentation. Dark gray fragments are lit by no lights, light gray fragments are lit by one light, and non-gray fragments are lit by both lights. (Courtesy of Norman Chin, Columbia University.)

Plate III.7 Nonrefractive transparency using extended *z*-buffer algorithm of Section 16.5.1. Unterlafette database is courtesy of CAM-I (Computer Aided Manufacturing International, Inc., Arlington, TX). (Rendered on a Stardent 1000 by Abraham Mammen.)



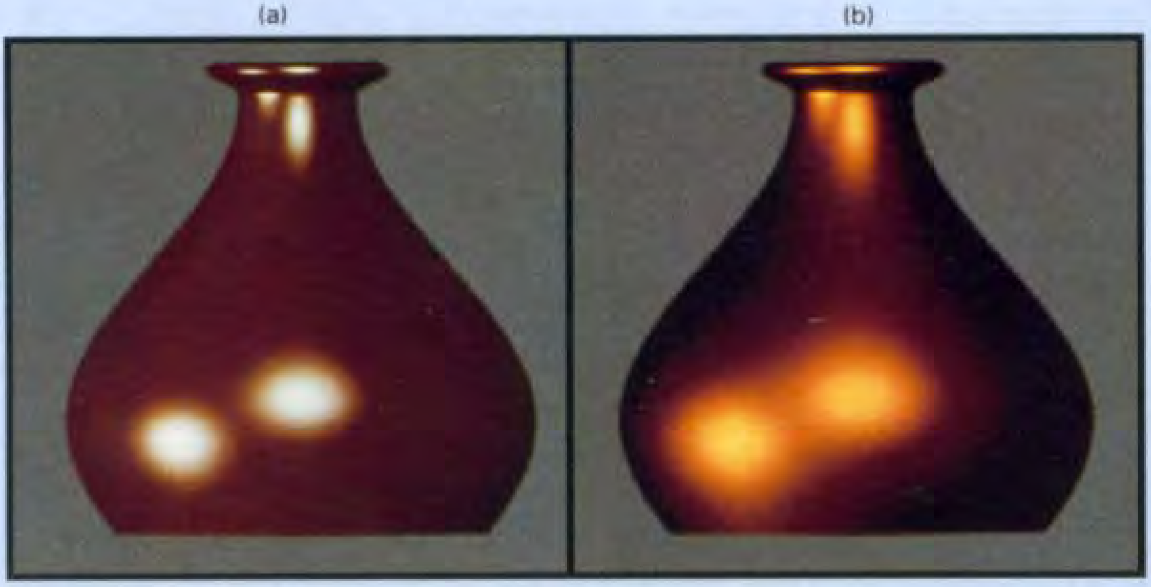


Plate III.8 Two vases rendered with the Cook-Torrance illumination model (Section 16.7). Both are lit by two lights with $I_{l_1} = I_{l_2} =$ CIE standard illuminant D6500, $d\omega_{l_1} = 0.0001$, and $d\omega_{l_2} = 0.0002$; $I_a = 0.01 I_{l_1}$; $\rho_d =$ the bidirectional reflectivity of copper for normal incidence; $\rho_a = \pi\rho_d$. (a) Copper-colored plastic: $k_s = 0.1$; $F =$ reflectivity of a vinyl mirror; $D =$ Beckmann function with $m = 0.15$; $k_d = 0.9$. (b) Copper metal: $k_s = 1.0$; $F =$ reflectivity of a copper mirror; $D =$ Beckmann function with $m_1 = 0.4$, $w_1 = 0.4$, $m_2 = 0.2$, $w_2 = 0.6$; $k_d = 0.0$. (By Robert Cook, Program of Computer Graphics, Cornell University.)

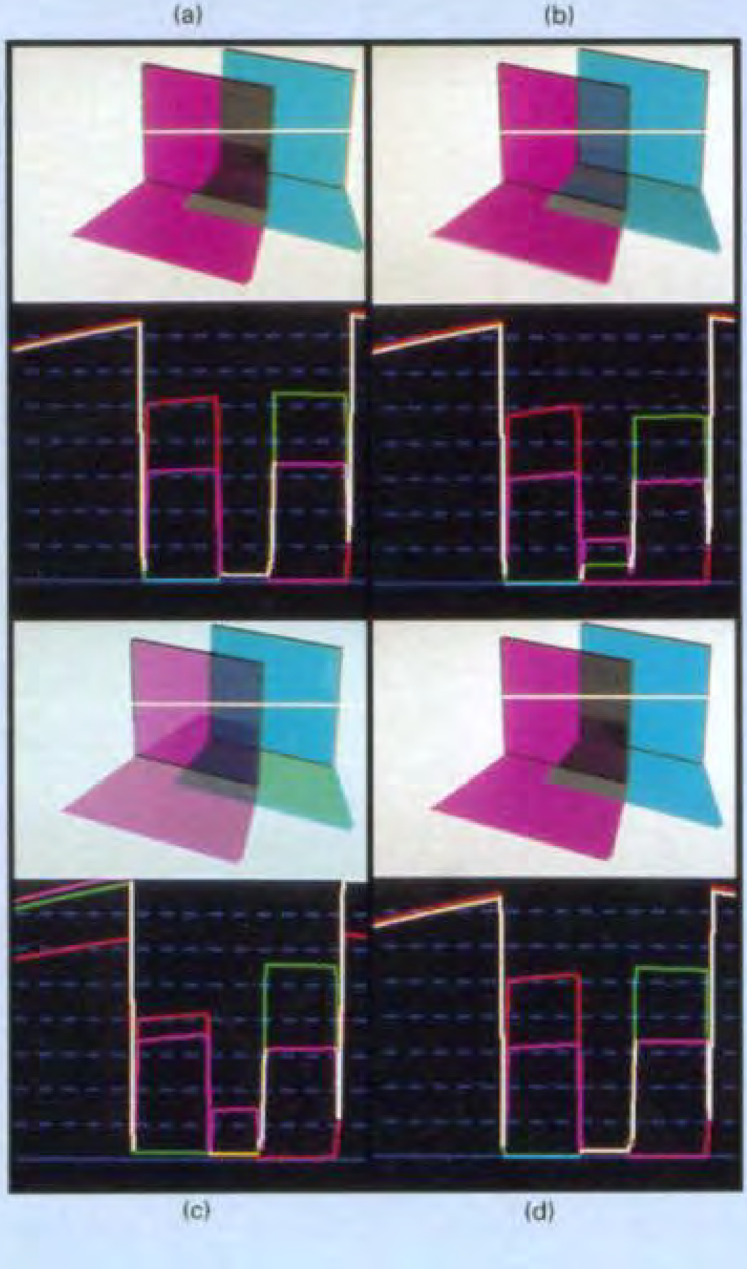


Plate III.9 Comparison of spectral sampling techniques for two overlapping filters (Section 16.9). Plots show RGB values for marked scan line with red (R), green (G), and magenta (B) lines. (a) One sample per nm from 360–830 nm. (b) 3 CIE XYZ samples. (c) 3 RGB samples. (d) 9 spectral samples. (Roy A. Hall and Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1983.)

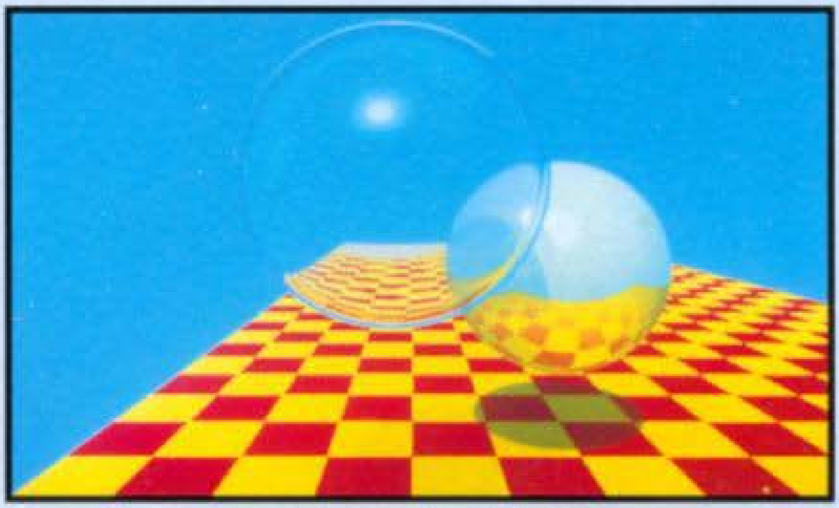
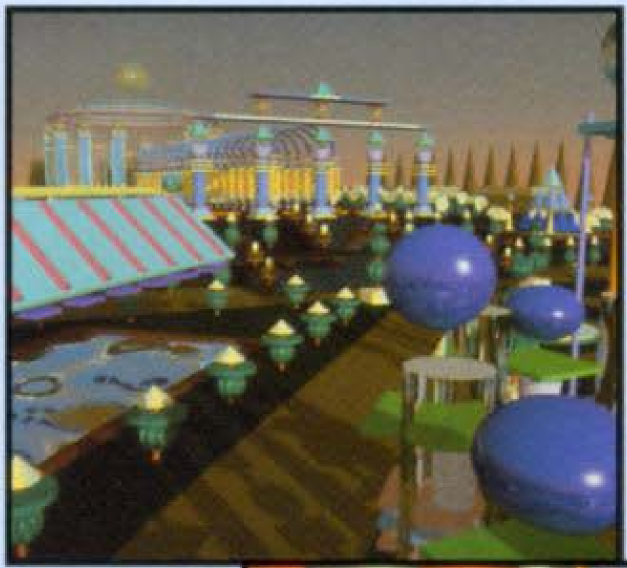


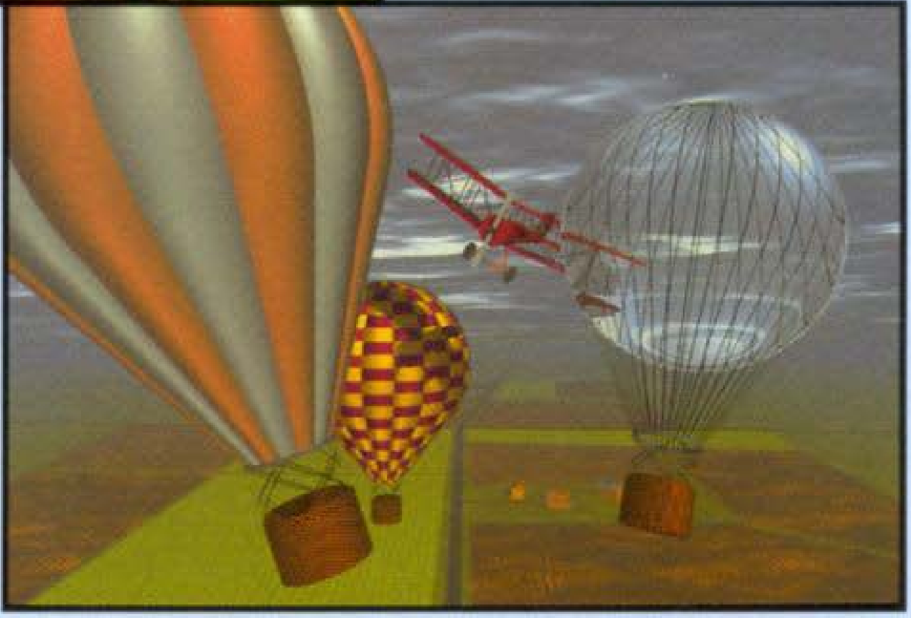
Plate III.10 Spheres and checkerboard. An early image produced with recursive ray tracing (Section 16.12). (Courtesy of Turner Whitted, Bell Laboratories.)



(a)

Plate III.11 Ray-traced images. (a) Scene from short film *Quest* (1985). (Michael Sciulli, James Arvo, and Melissa White. © Hewlett-Packard.) (b) "Haute Air." Functions were used to modify color, surface normals, and transparency at nearly every pixel. (Courtesy of David Kurlander, Columbia University, 1986.)

(b)



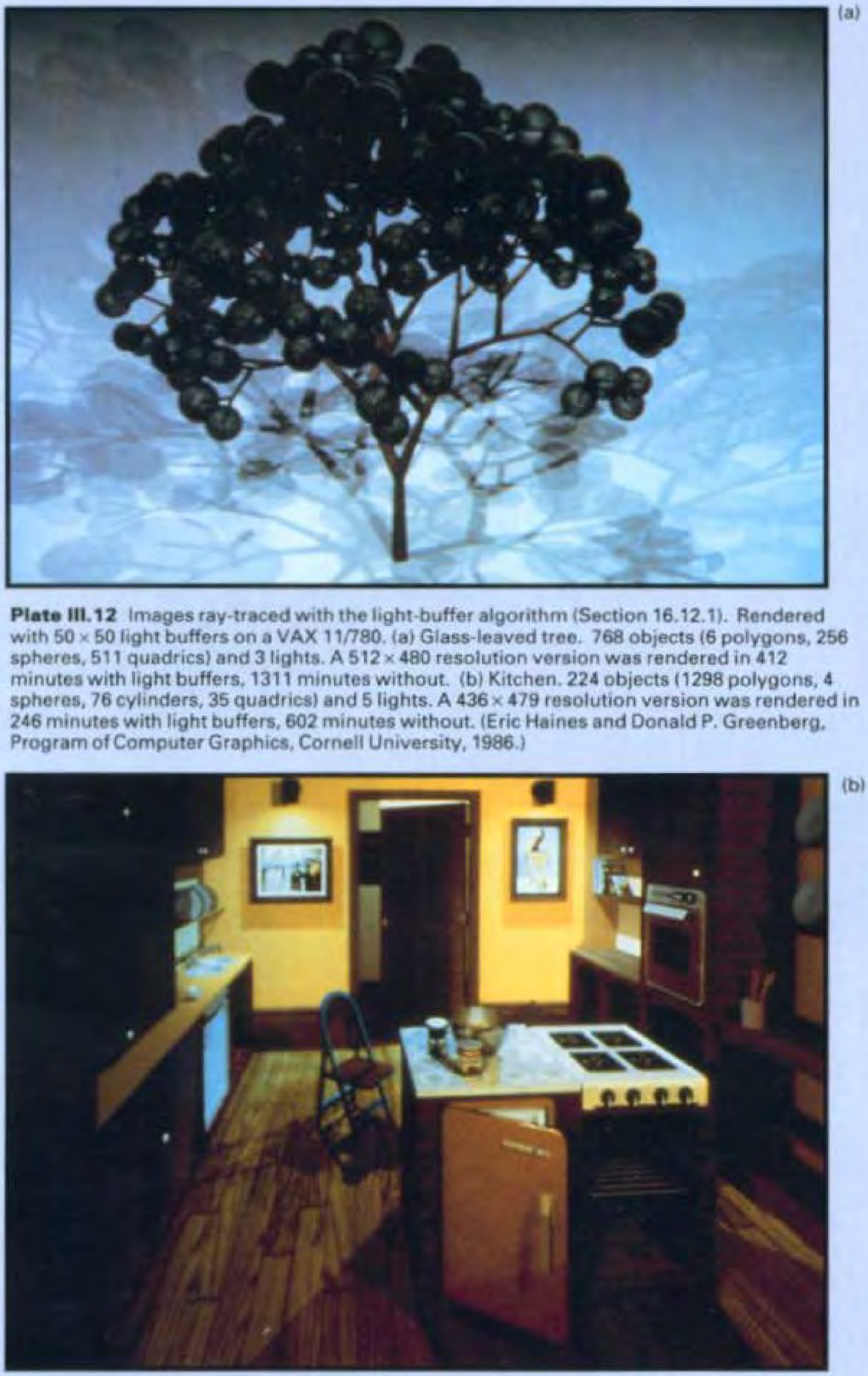
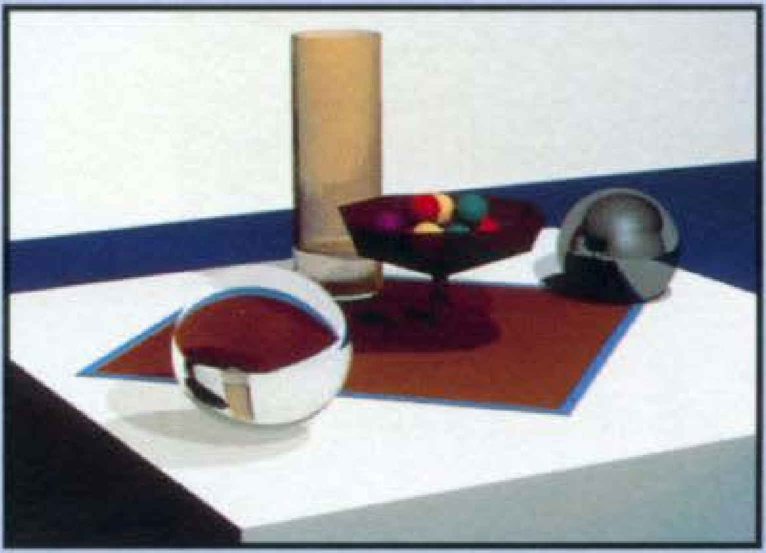


Plate III.12 Images ray-traced with the light-buffer algorithm (Section 16.12.1). Rendered with 50 × 50 light buffers on a VAX 11/780. (a) Glass-leaved tree. 768 objects (6 polygons, 256 spheres, 511 quadrics) and 3 lights. A 512 × 480 resolution version was rendered in 412 minutes with light buffers, 1311 minutes without. (b) Kitchen. 224 objects (1298 polygons, 4 spheres, 76 cylinders, 35 quadrics) and 5 lights. A 436 × 479 resolution version was rendered in 246 minutes with light buffers, 602 minutes without. (Eric Haines and Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1986.)



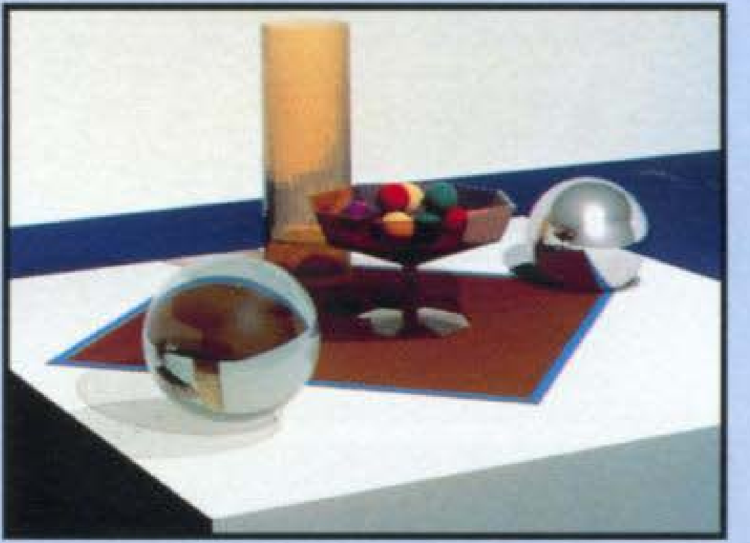


Plate III.13 Comparison of illumination models for ray tracing (Section 16.12.2). Note differences in reflectivity of the base of the dish, and the color of the transparent and reflective spheres. (a) Whitted illumination model. (b) Hall illumination model. (Roy A. Hall and Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1983.)

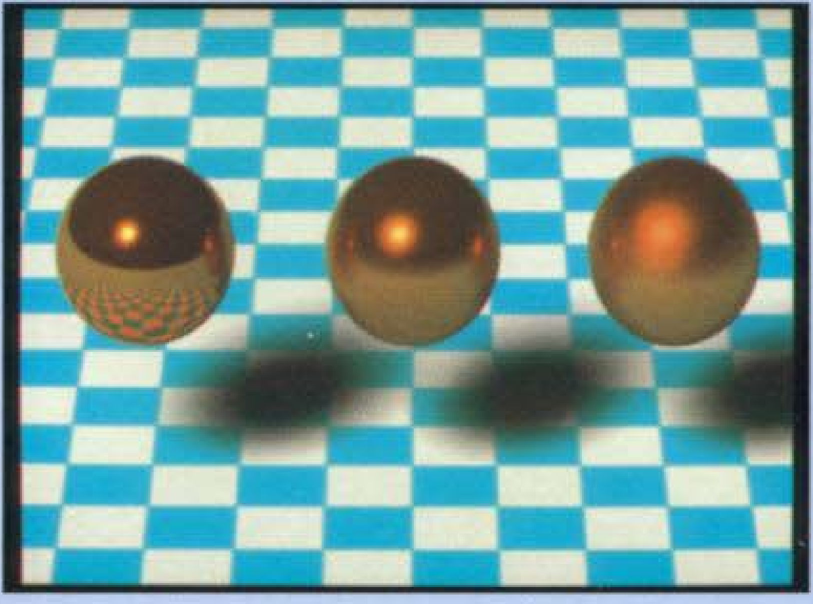


Plate III.14 Cone tracing (Section 16.12.3). Three spheres. Dull reflections are created by increasing the angular spread of reflected rays by 0.0, 0.2, and 0.4 radians, respectively, for spheres from left to right. (Courtesy of John Amanatides, University of Toronto.)

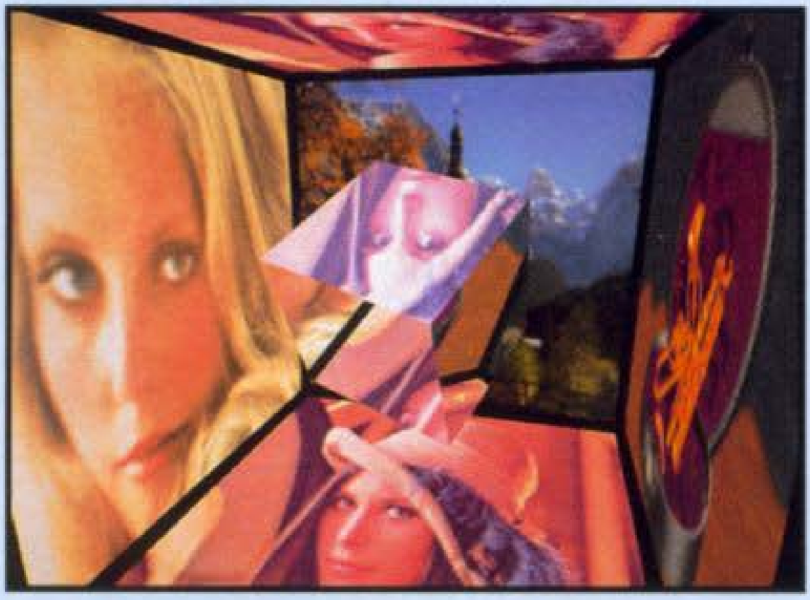
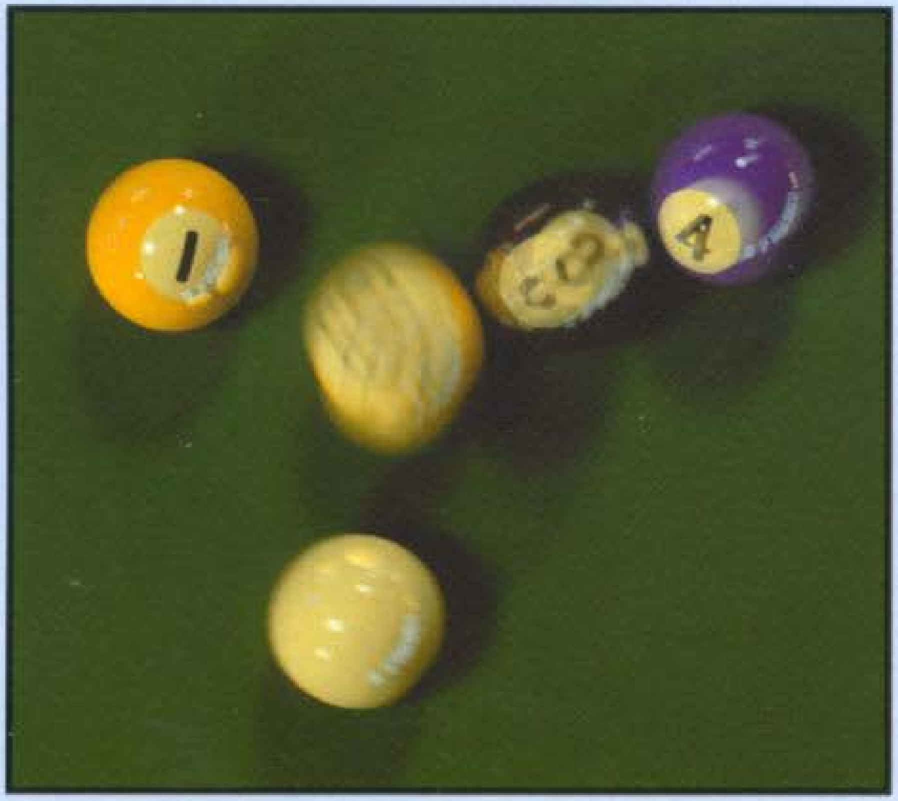


Plate III.15 Beam tracing (Section 16.12.3). A mirrored cube in a texture-mapped room. (Paul Heckbert and Pat Hanrahan, © NYIT 1984.)

Plate III.16 *1984.* Rendered using distributed ray tracing (Section 16.12.4) at 4096 × 3550 pixels with 16 samples per pixel. Note the motion-blurred reflections and shadows with penumbrae cast by extended light sources. (By Thomas Porter. © Pixar 1984. All Rights Reserved.)



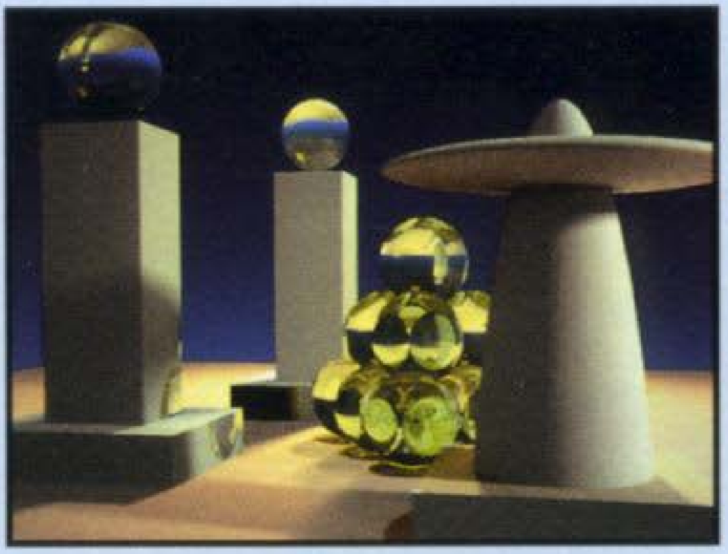


Plate III.17 Scene rendered with integral equation method (Section 16.12.4). All opaque objects are Lambertian. Note interobject reflections. Computed at 512 × 512 resolution with 40 paths/pixel in 1221 minutes on an IBM 3081. (J. Kajiya, California Institute of Technology.)

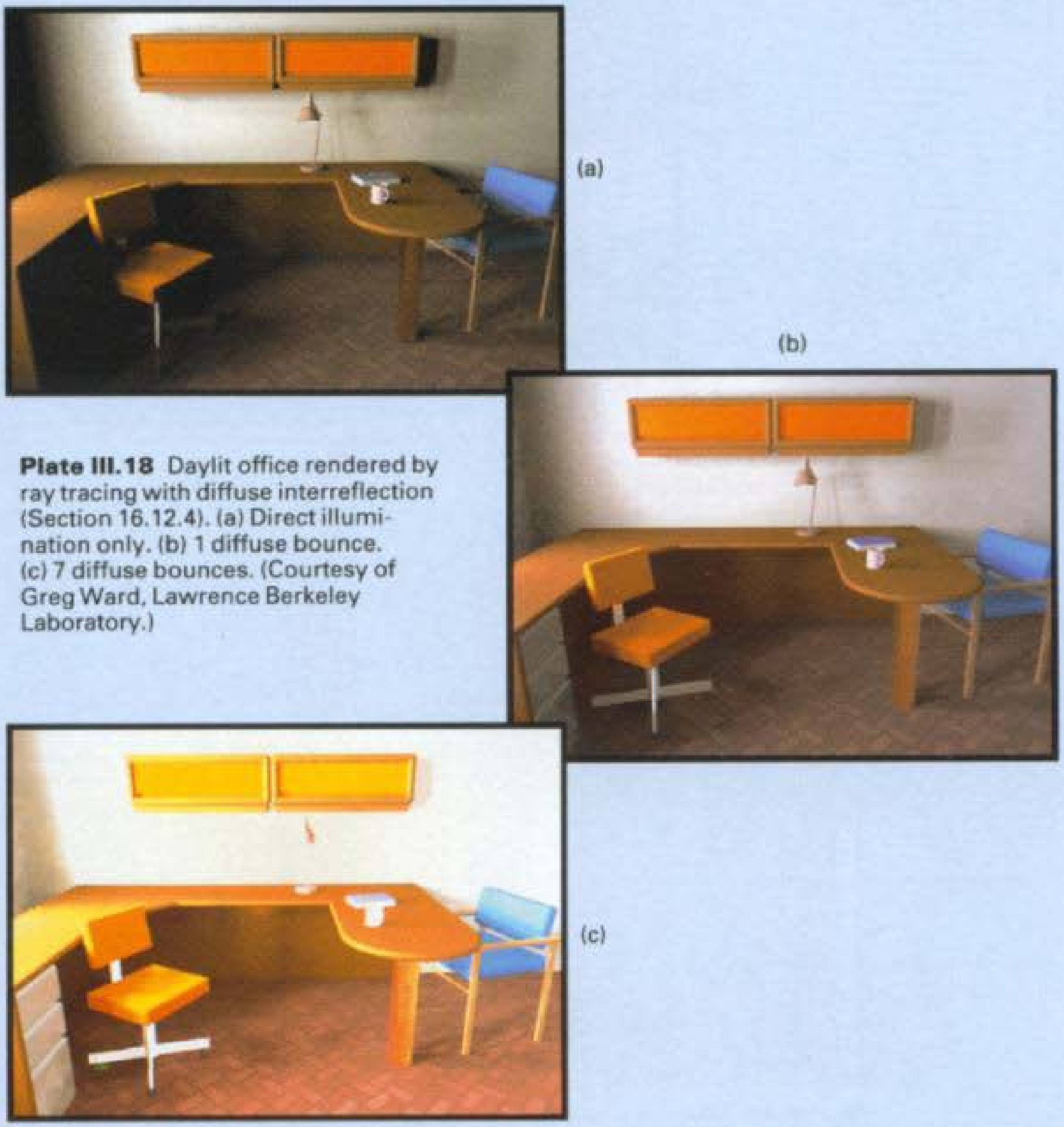
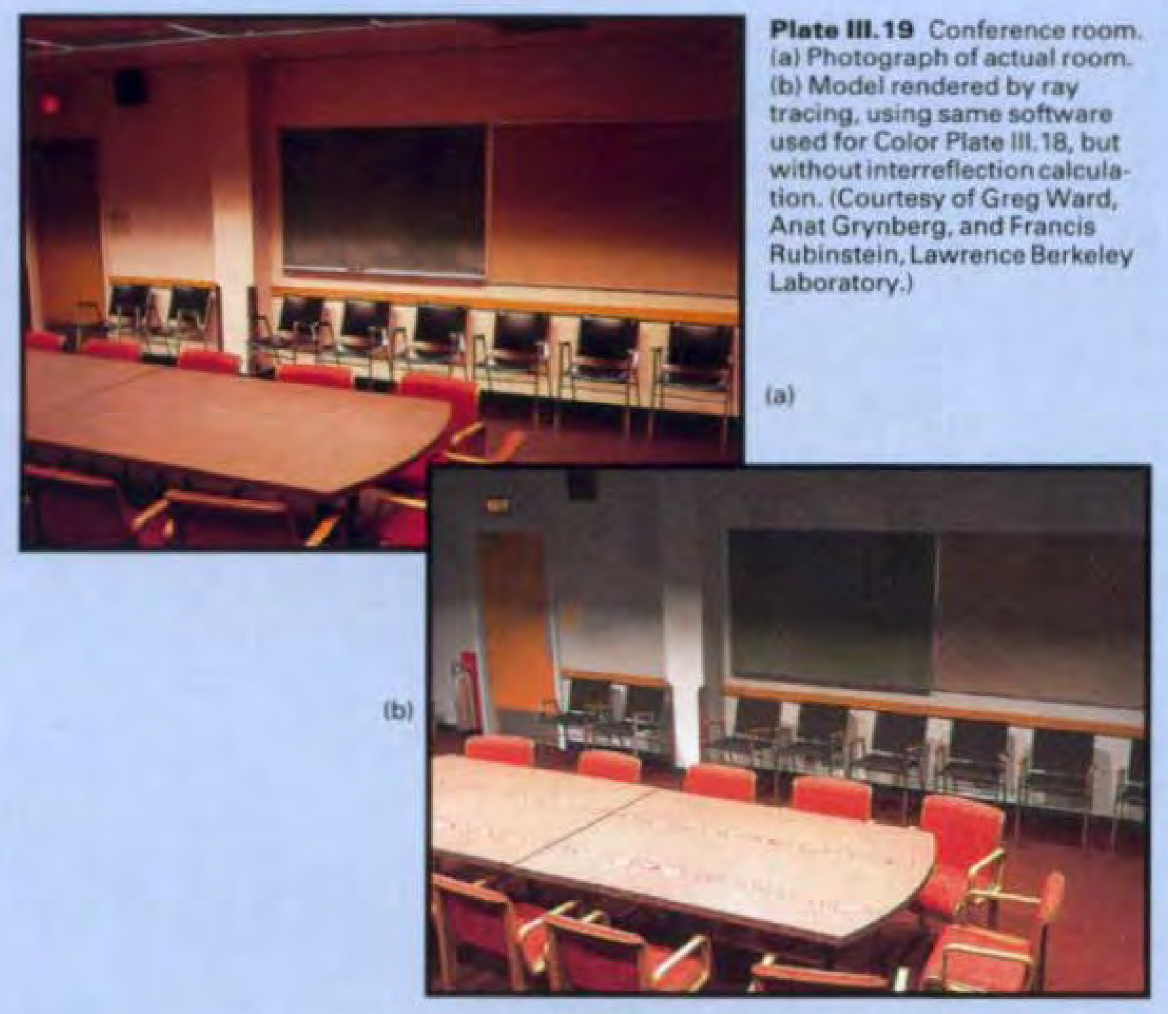


Plate III.18 Daylit office rendered by ray tracing with diffuse interreflection (Section 16.12.4). (a) Direct illumination only. (b) 1 diffuse bounce. (c) 7 diffuse bounces. (Courtesy of Greg Ward, Lawrence Berkeley Laboratory.)

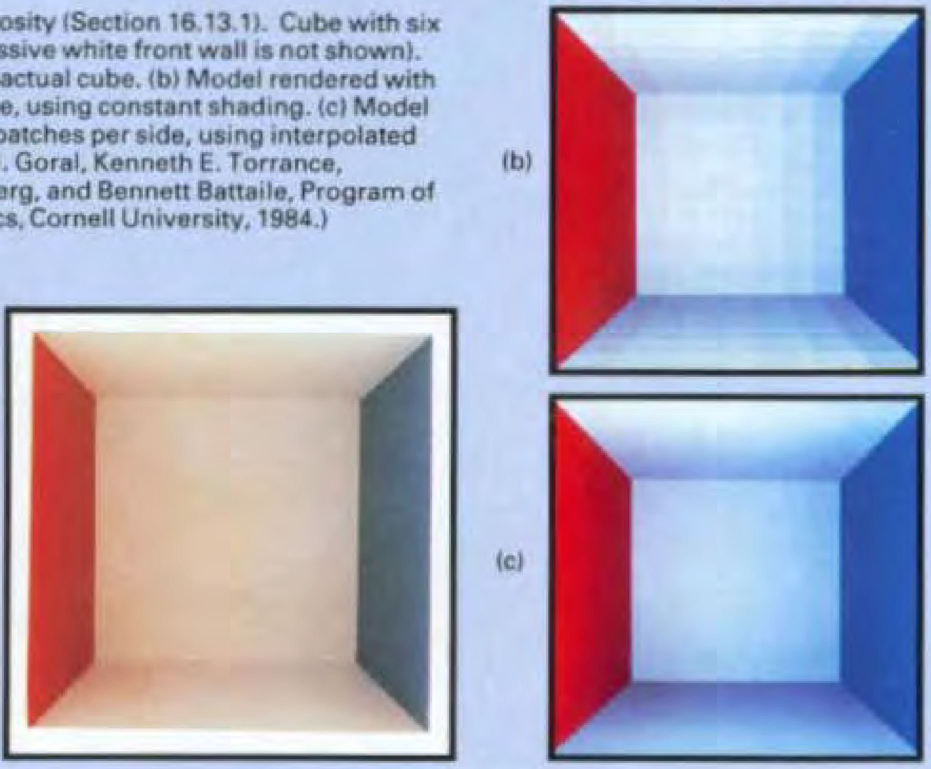


(a)

(b)

Plate III.19 Conference room. (a) Photograph of actual room. (b) Model rendered by ray tracing, using same software used for Color Plate III.18, but without interreflection calculation. (Courtesy of Greg Ward, Anat Grynberg, and Francis Rubinstein, Lawrence Berkeley Laboratory.)

Plate III.20 Radiosity (Section 16.13.1). Cube with six diffuse walls (emissive white front wall is not shown). (a) Photograph of actual cube. (b) Model rendered with 49 patches per side, using constant shading. (c) Model rendered with 49 patches per side, using interpolated shading. (Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile, Program of Computer Graphics, Cornell University, 1984.)



(b)

(a)

(c)

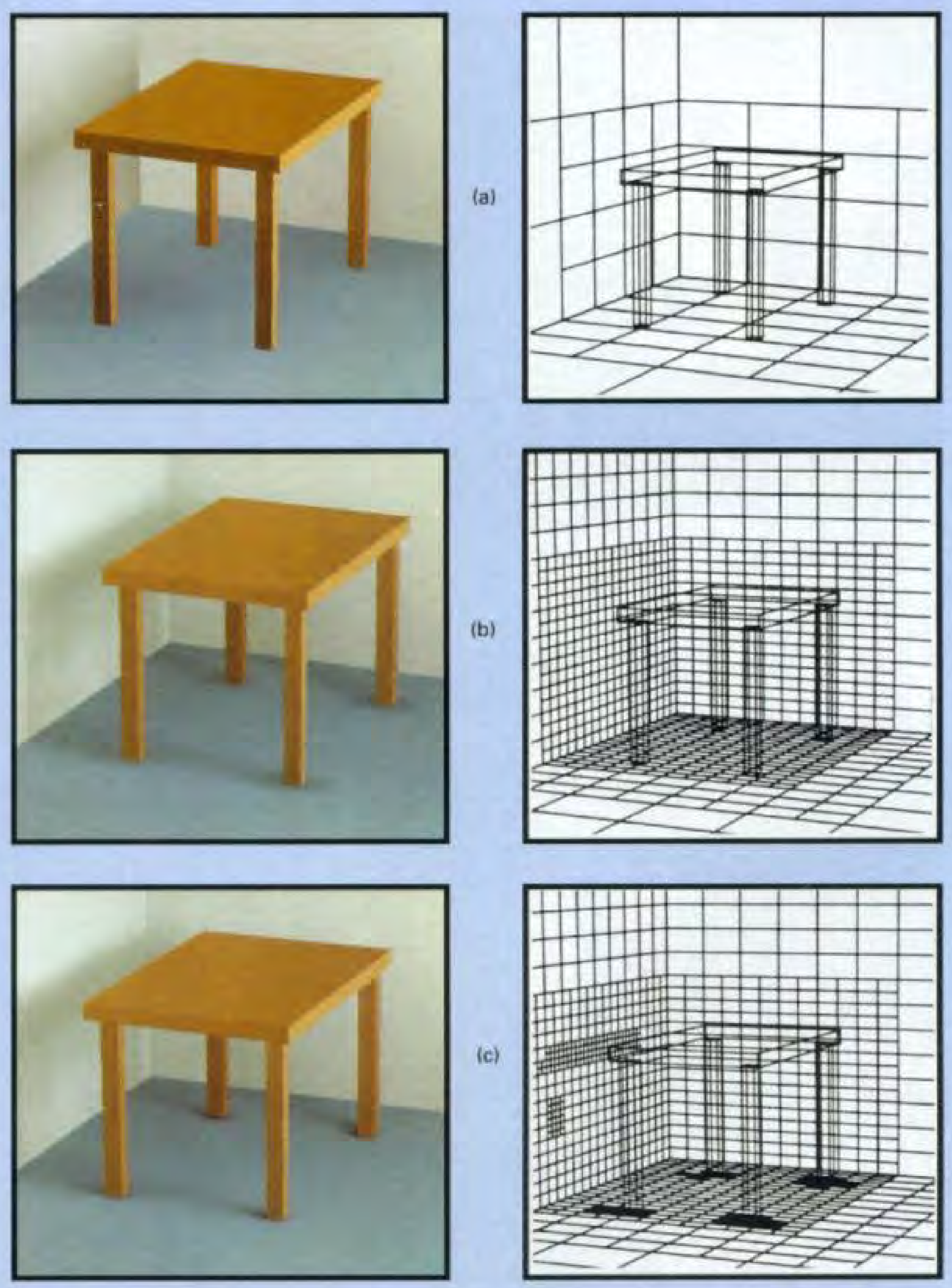


Plate III.21 Radiosity algorithm, using hemicube to compute form factors (Sections 16.13.2–3). (a) Coarse patch solution (145 patches, 10 minutes). (b) Improved solution using either more patches (1021 patches, 175 minutes) or substructuring (1021 subpatches, 62 minutes). (c) Refinement of substructured version of (b) using adaptive subdivision (1306 subpatches, 24 additional minutes).(Michael F. Cohen, Program of Computer Graphics, Cornell University, 1985.)

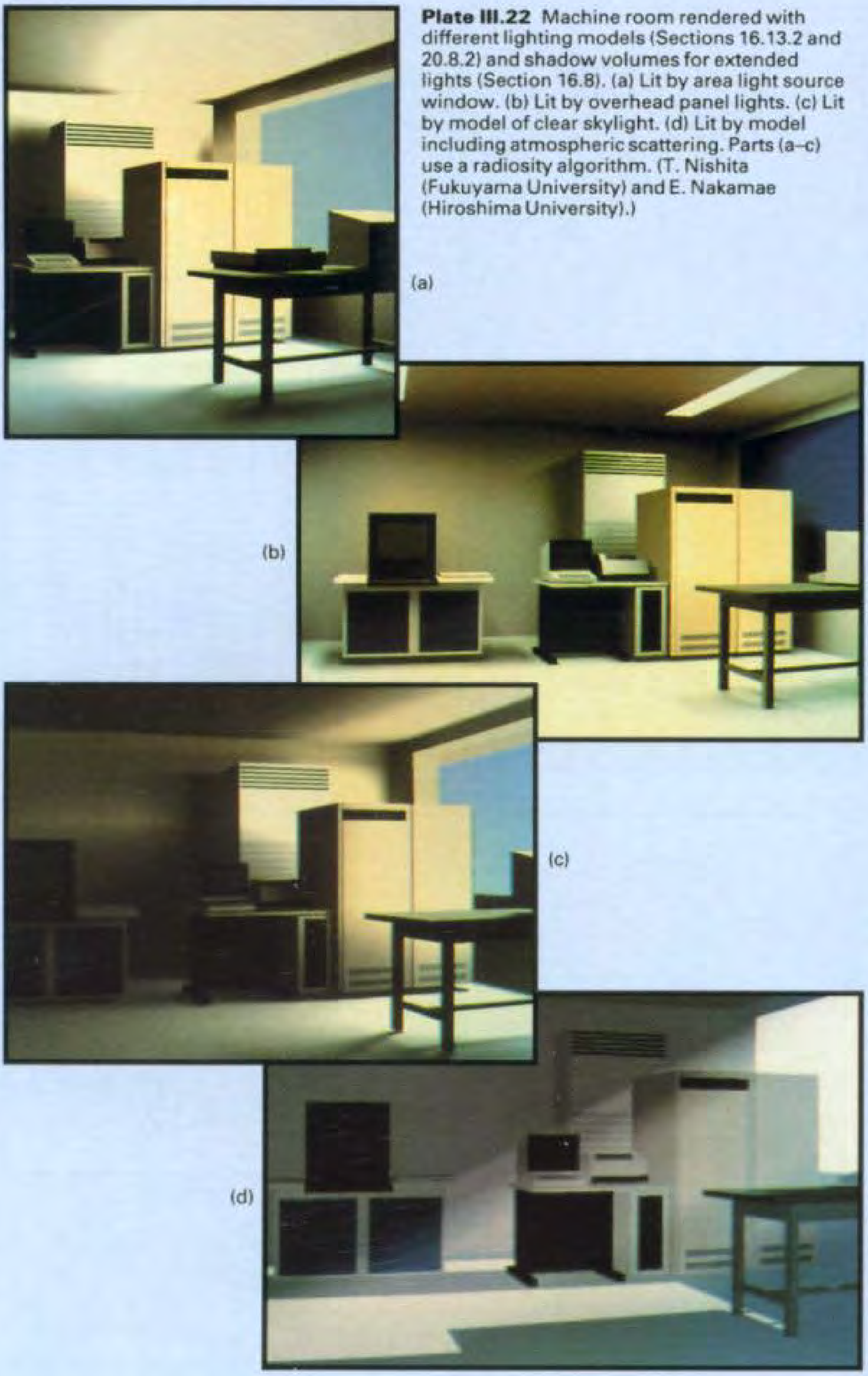
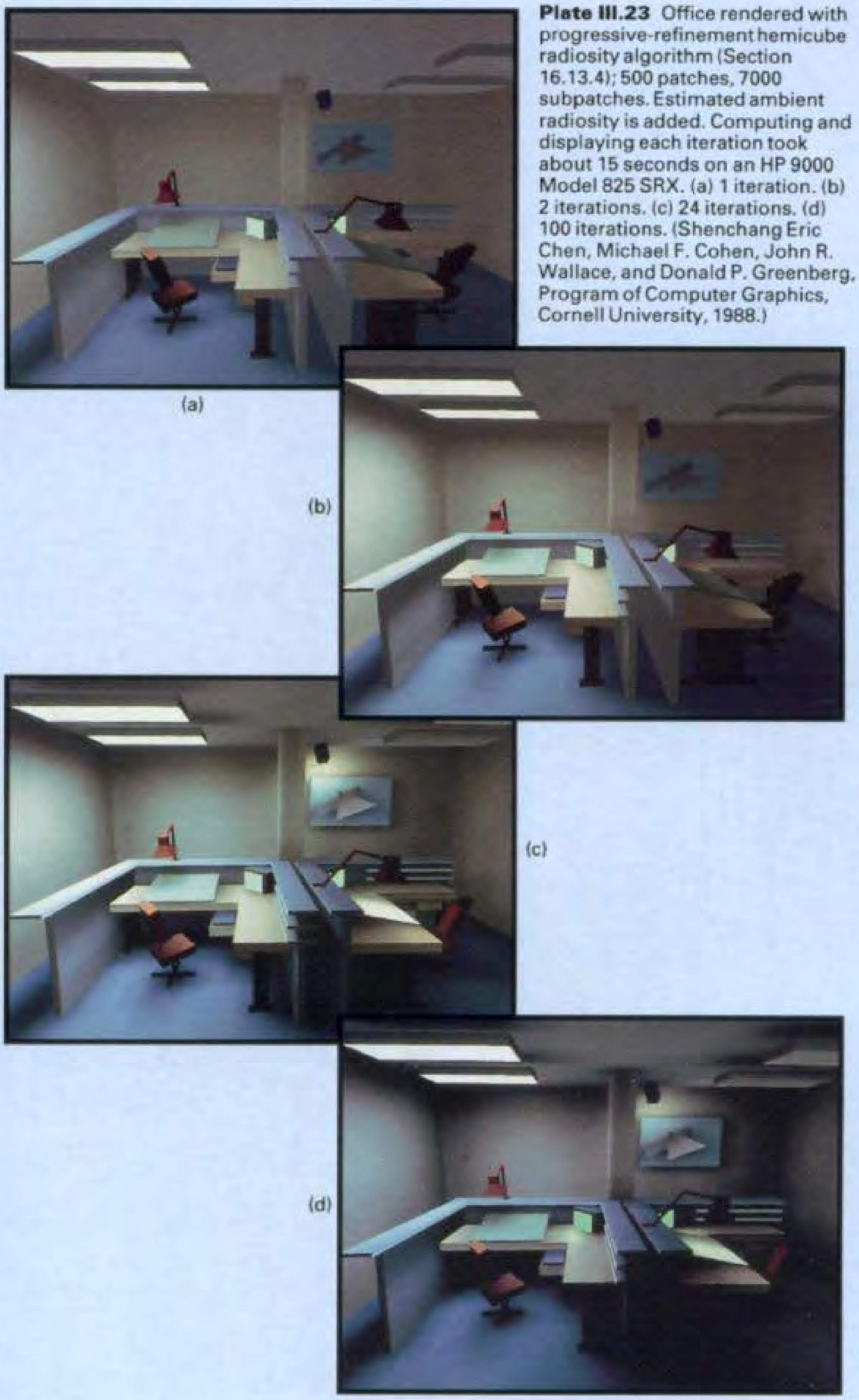


Plate III.22 Machine room rendered with different lighting models (Sections 16.13.2 and 20.8.2) and shadow volumes for extended lights (Section 16.8). (a) Lit by area light source window. (b) Lit by overhead panel lights. (c) Lit by model of clear skylight. (d) Lit by model including atmospheric scattering. Parts (a–c) use a radiosity algorithm. (T. Nishita (Fukuyama University) and E. Nakamae (Hiroshima University).)



(a)

(b)

(c)

(d)

Plate III.23 Office rendered with progressive-refinement hemicube radiosity algorithm (Section 16.13.4); 500 patches, 7000 subpatches. Estimated ambient radiosity is added. Computing and displaying each iteration took about 15 seconds on an HP 9000 Model 825 SRX. (a) 1 iteration. (b) 2 iterations. (c) 24 iterations. (d) 100 iterations. (Shenchang Eric Chen, Michael F. Cohen, John R. Wallace, and Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1988.)

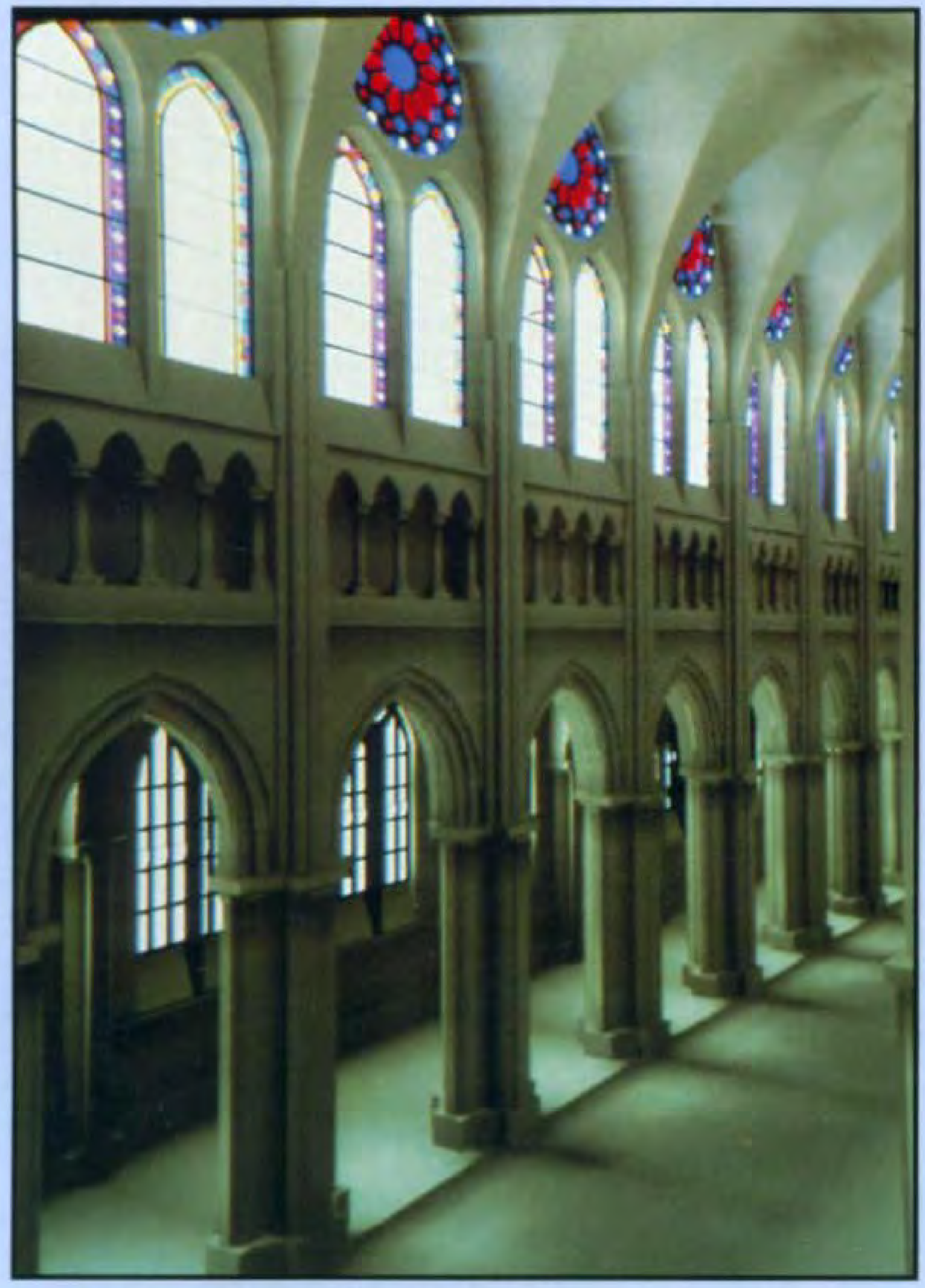


Plate III.24 Nave of Chartres cathedral rendered with progressive-refinement radiosity algorithm, using ray tracing to compute form factors (Section 16.13.5). Two bays, containing 9916 polygons, were processed and copied three more times. Sixty iterations took 59 minutes on HP 9000 Model 835 TurboSRX. (By John Wallace and John Lin, using Hewlett-Packard's Starbase Radiosity and Ray Tracing software. © 1989, Hewlett-Packard Co.)

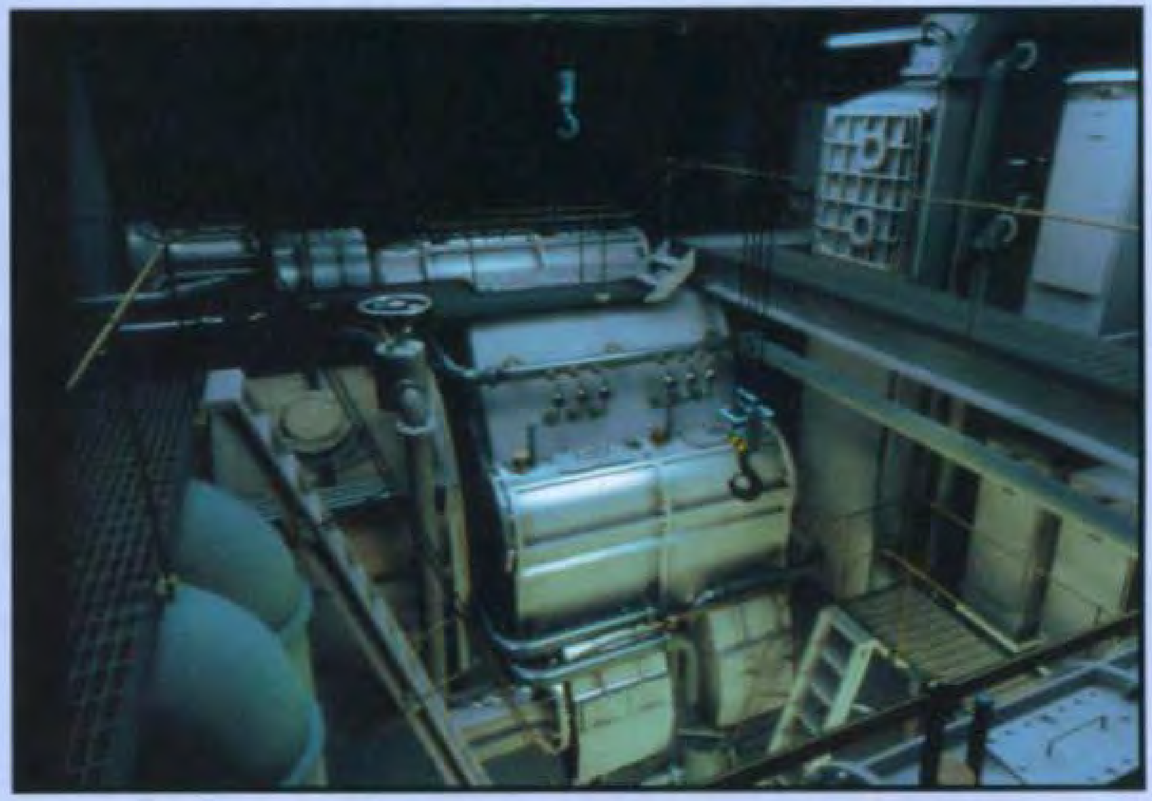
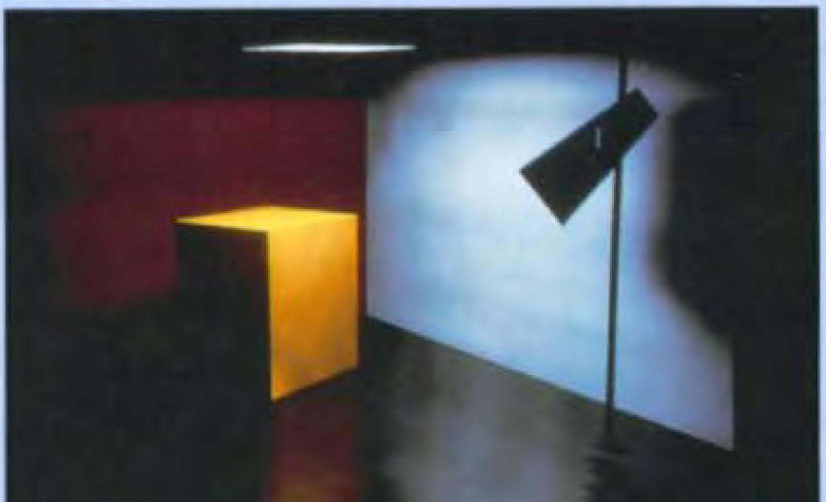


Plate III.25 Boiler room rendered with progressive-refinement radiosity algorithm, using ray tracing to compute form factors. (By John Wallace, John Lin, and Eric Haines, using Hewlett-Packard's Starbase Radiosity and Ray Tracing software. © 1989, Hewlett-Packard Company.)

Plate III.26 Specular radiosity algorithm (Section 16.13.6); 64 specular patches and 237 diffuse patches. (David S. Immel, Michael F. Cohen, Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1986.)



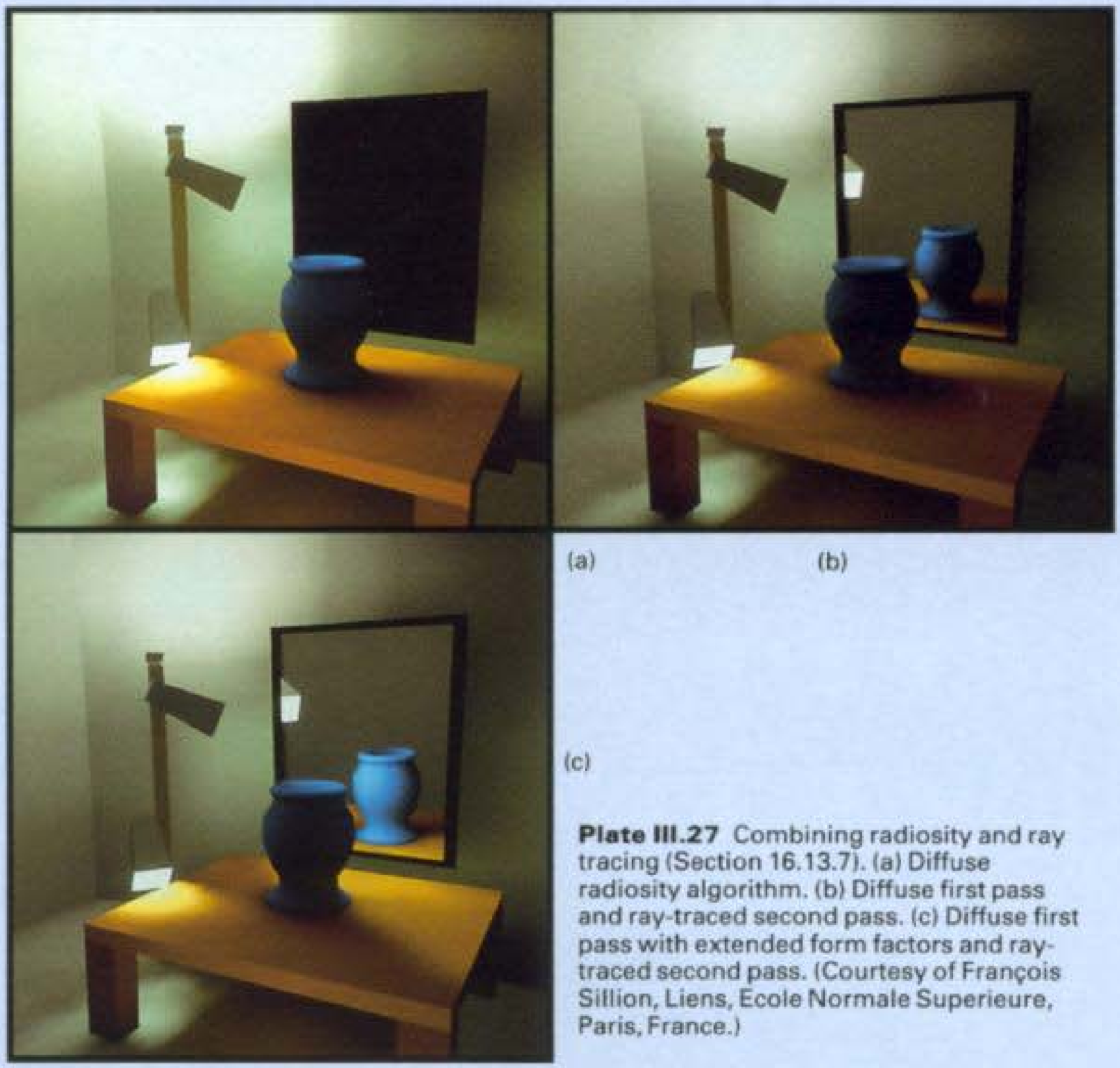
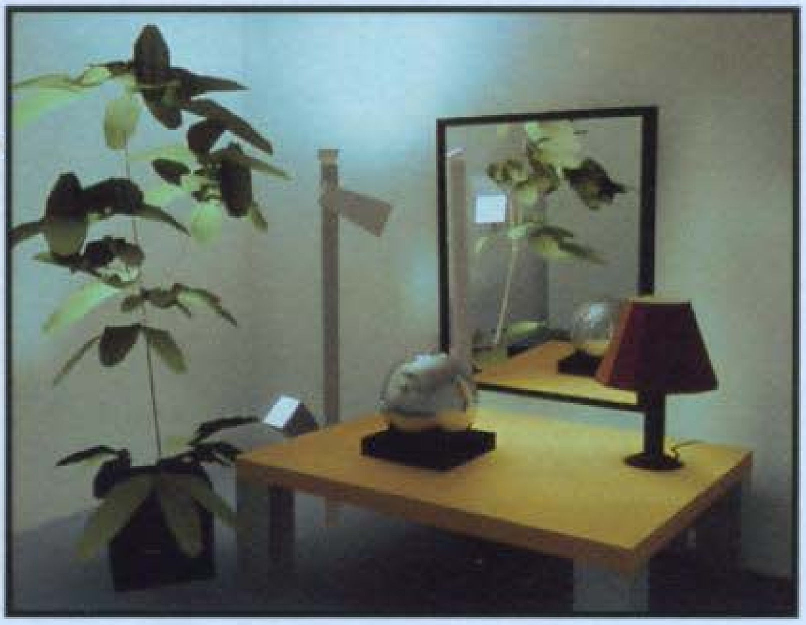


Plate III.27 Combining radiosity and ray tracing (Section 16.13.7). (a) Diffuse radiosity algorithm. (b) Diffuse first pass and ray-traced second pass. (c) Diffuse first pass with extended form factors and ray-traced second pass. (Courtesy of François Sillion, Liens, Ecole Normale Superieure, Paris, France.)

Plate III.28 Room rendered with combined radiosity and ray tracing. (Courtesy of François Sillion, Liens, Ecole Normale Superieure, Paris, France.)



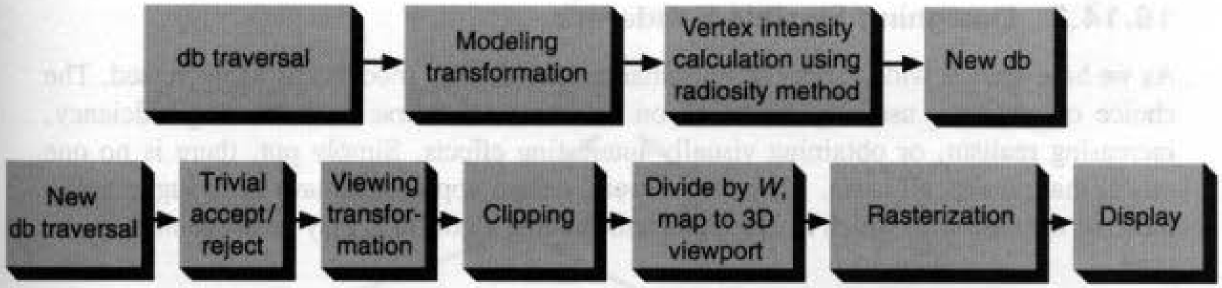


Fig. 16.75 Rendering pipeline for radiosity and Gouraud shading.

previously scan-converted polygon that it intersects. Even this simple overwrite capability is not needed if we instead use an object-precision algorithm that generates a list of fully visible primitives, such as the Weiler–Atherton algorithm.

16.14.2 Global Illumination Pipelines

Thus far, we have ignored global illumination. As we have noted before, incorporating global illumination effects requires information about the geometric relationships between the object being rendered and the other objects in the world. One approach, of which we have seen many examples, is to calculate needed information from a specific viewpoint in advance of scan conversion and to store it in tables (e.g., reflection maps and shadow maps). This eliminates the need to access the full db representation of other objects while processing the current object. In the case of shadows, which depend only on the position of the light source, and not on that of the viewer, preprocessing the environment to add surface-detail polygon shadows is another way to allow the use of an otherwise conventional pipeline.

Radiosity. The diffuse radiosity algorithms offer an interesting example of how to take advantage of the conventional pipeline to achieve global-illumination effects. These algorithms process objects and assign to them a set of view-independent vertex intensities. These objects may then be presented to a modified version of the pipeline for *z*-buffer and Gouraud shading, depicted in Fig. 16.75, that eliminates the lighting stage.

Ray tracing. Finally, we consider ray tracing, whose pipeline, shown in Fig. 16.76, is the simplest because those objects that are visible at each pixel and their illumination are determined entirely in WC. Once objects have been obtained from the database and transformed by the modeling transformation, they are loaded into the ray tracer's WC database, which is typically implemented using the techniques of Sections 15.10.2 and 16.12.1, to support efficient ray intersection calculations.

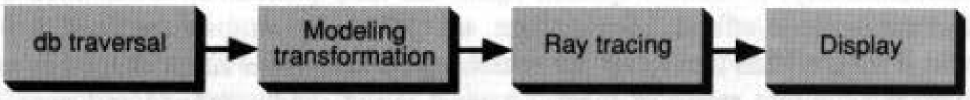


Fig. 16.76 Rendering pipeline for ray tracing.

16.14.3 Designing Flexible Renderers

As we have seen, a wide variety of illumination and shading models has been created. The choice of which to use may be based on concerns as diverse as increasing efficiency, increasing realism, or obtaining visually interesting effects. Simply put, there is no one model that pleases all users. Therefore, several design approaches have been suggested to increase the ease with which illumination and shading algorithms may be implemented and used.

Modularization. A straightforward approach is to modularize the illumination and shading model in a part of the rendering system that is often known as its *shader*. Whitted and Weimer [WHIT82] showed that, by establishing a standard mechanism for passing parameters to shaders, different shaders can be used in the same system; the decision about which shader to call can even be made at run time based on some attribute of the object. Their system performs scan conversion using a scan-line algorithm, and accumulates results as a linked list of spans for each line. Each span contains information about a set of values associated with its endpoints, including their x and z values, and additional information such as normal components, and intensities. The shader interpolates specified values across each span. (Since it typically uses the interpolated z values to perform visible-surface determination with a scan-line z-buffer, the term *shader* is being used quite loosely.)

The Doré graphics system [ARDE89] is designed to offer the programmer additional flexibility. It provides a standard way of expressing a scene database in terms of a set of objects that have methods for performing operations such as rendering, picking, or computing a bounding volume. The display list and its traversal functions form a common core that is intended to make it easy to interface to different rendering systems. A programmer can use the standard set of objects, methods, and attributes, or can design her own using the framework.

Special languages. In contrast to providing extensibility at the level of the programming language in which the system is built, it is possible to design special languages that are better suited to specific graphics tasks. Cook [COOK84a] has designed a special-purpose language in which a shader is built as a tree expression called a *shade tree*. A shade tree is a tree of nodes, each of which takes parameters from its children and produces parameters for its parent. The parameters are the terms of the illumination equation, such as the specular coefficient, or the surface normal. Some nodes, such as *diffuse*, *specular*, or *square root*, are built into the language with which shade trees are specified. Others can be defined by the user and dynamically loaded when needed. All nodes can access information about the lights. Figure 16.77 shows a shade tree for a description of copper. A shade tree thus describes a particular shading process and is associated with one or more objects through use of a separate modeling language. Different objects may have different shade trees, so that an image can be rendered in which a multiplicity of different special-purpose models are mixed. Similarly, in Cook's system, lights and their parameters are defined by light trees, and atmospheric effects, such as haze, are defined by atmosphere trees.

Perlin [PERL85] has developed the notion of a *pixel-stream editor* that takes as input and produces as output arrays of pixels. A pixel is not rigidly defined and may include

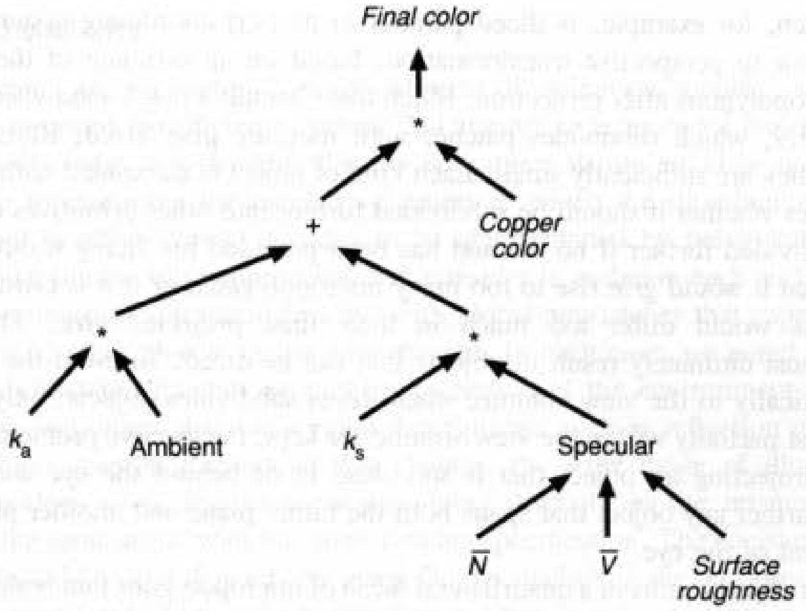


Fig. 16.77 Shade tree for copper. (After [COOK84].)

arbitrary data for a point in the image, such as the material identifier or normal vector at that point. An output pixel need not have the same structure as an input pixel. The pixel-stream editor executes a program written by the user in a high-level language oriented toward pixel manipulation. Thus, the user is encouraged to think of creating images in a series of passes, with intermediate results represented by arrays of pixels that may differ in the kind of information they encode.

The flexibility of shade trees and pixel-stream editors may be combined by designing a rendering system that allows its users to write their own shaders in a special programming language and to associate them with selected objects. This approach is taken in the RenderMan Interface [PIXA88; UPST89], a scene description specification that provides such a shading language. RenderMan defines a set of key places in the rendering process at which user-defined or system-defined shaders can be called. For example, the most common kind of shader, called a *surface shader*, returns the light reflected in a specified direction given a point on the surface, its orientation, and a set of light sources. A user-provided surface shader could implement an illumination equation totally different from those discussed so far. Other shaders include atmosphere shaders that modify the color of light passing between two points, and—in another example of how the word *shader* can be stretched—projection shaders that allow user-defined projections implementing other than parallel or linear perspective projections.

An example. Cook, Carpenter, and Catmull's Reyes Rendering Architecture [COOK87], which was used to produce Color Plates II.24–37, D, and F, provides an interesting example of how to structure a renderer. Reyes chops all objects up into *micropolygons*: small, constant-shaded quadrilaterals that are approximately ½ pixel on a side. This approach, known as *dicing*, occurs along boundaries that are natural for the object. A patch, for example, is diced parallel to its (s,t) coordinate system. Dicing is performed prior to perspective transformation, based on an estimate of the size of the resulting micropolygons after projection. Much like Catmull's patch subdivision algorithm of Section 15.9, which subdivides patches until they are pixel-sized, Reyes subdivides objects until they are sufficiently small. Each kind of object is associated with a procedure that determines whether it should be subdivided further into other primitives or diced. An object is subdivided further if no method has been provided for dicing it directly, if it is determined that it would give rise to too many micropolygons, or if it is estimated that its micropolygons would differ too much in their final projected size. This recursive subdivision must ultimately result in objects that can be diced. To avoid the need to clip objects analytically to the view volume, when Reyes subdivides objects, only those parts that are at least partially within the view volume are kept. Perspective problems that would result from projecting an object that is too close to or behind the eye are avoided by subdividing further any object that spans both the hither plane and another plane that lies slightly in front of the eye.

Dicing an object results in a quadrilateral mesh of micropolygons that is shaded in WC. Because the micropolygons are sufficiently small, each is given a single shade, avoiding all the interpolated shading problems discussed in Section 16.2.6. Since a patch is diced parallel to its (s,t) coordinate system, some of the texture-mapping approaches discussed in Chapter 17 are particularly efficient to use. Dicing and shading can both take advantage of incremental algorithms. Reyes relies on the mapping techniques discussed in this chapter for its global lighting effects.

Visible surface determination is done with a subpixel z-buffer whose subpixel centers are jittered to accomplish stochastic sampling. The closest micropolygon covering a subpixel center is visible at that subpixel. To avoid the need to store micropolygon meshes and subpixel z and intensity values for the entire image, Reyes uses spatial partitioning. The image is divided into rectangular partitions into which each object is sorted by the upper left hand corner of its extent. The partitions are then processed left to right, top to bottom. As objects are subdivided or diced, the resulting subobjects or micropolygons are placed in the partitions that they intersect. Thus, only enough z-buffer memory is needed for a single partition, and other storage needed for a partition can be freed after it is processed.

16.14.4 Progressive Refinement

One interesting modification to the pipelines that we have discussed takes advantage of the fact that the image is viewed for a finite time. Instead of attempting to render a final version of a picture all at once, we can first render the picture coarsely, and then *progressively refine* it, to improve it. For example, a first image might have no antialiasing, simpler object models, and simpler shading. As the user views an image, idle cycles may be spent improving its quality [FORR85]. If there is some metric by which to determine what to do next, then refinement can occur adaptively. Bergman, Fuchs, Grant, and Spach [BERG86b] have developed such a system that uses a variety of heuristics to determine how it should spend its time. For example, a polygon is Gouraud-shaded, rather than constant-shaded, only if the range of its vertex intensities exceeds a threshold. Ray-tracing [PAIN89] and radiosity [COHE88] algorithms are both amenable to progressive refinement.

16.15 SUMMARY

In this chapter, we encountered many different illumination models, some inspired primarily by the need for efficiency, others that attempt to account for the physics of how surfaces actually interact with light. We saw how interpolation could be used in shading models, both to minimize the number of points at which the illumination equation is evaluated, and to allow curved surfaces to be approximated by polygonal meshes. We contrasted local illumination approaches that consider in isolation each surface point and the lights illuminating each point directly, with global approaches that support refraction and reflection of other objects in the environment. In each case, we noted that there are some methods that use the full geometric description of the environment in computing global effects, and others that use simpler descriptions, such as reflection maps.

As we have stressed throughout this chapter, the wide range of illumination and shading algorithms gives rise to a corresponding diversity in the images that can be produced of the same scene with the same viewing specification. The decision about which algorithms should be used depends on many factors, including the purposes for which an image is to be rendered. Although photorealism is often sacrificed in return for efficiency, advances in algorithms and hardware will soon make real-time implementations of physically correct, global illumination models a reality. When efficiency is no longer an issue, however, we may still choose to render some images without texture, shadows, reflections, or refraction, because in some cases this will remain the best way to communicate the desired information to the viewer.

EXERCISES

16.1 (a) Describe the difference in appearance you would expect between a Phong illumination model that used $(\overline{N} \cdot \overline{H})^n$ and one that used $(\overline{R} \cdot \overline{V})^n$. (b) Show that $\alpha = 2\beta$ when all vectors of Fig. 16.12 are coplanar. (c) Show that this relationship is *not* true in general.

16.2 Prove that the results of interpolating vertex information across a polygon's edges and scan lines are independent of orientation in the case of triangles.

16.3 Suppose there are polygons A, B, and C intersecting the same projector in order of increasing distance from the viewer. Show that, in general, if polygons A and B are transparent, the color computed for a pixel in the intersection of their projections will depend on whether Eq. (16.25) is evaluated with polygons A and B treated as polygons 1 and 2 or as polygons 2 and 1.

16.4 Consider the use of texture mapping to modify or replace different material properties. List the effects you can produce by mapping properties singly or in combination. How would you apply antialiasing to them?

16.5 Although using a reflection map may appear to require precomputing the lighting for the environment, a reflection map containing object identities and surface normals could be used instead. What are the disadvantages of using this kind of map?

16.6 Explain how to simulate reflections from surfaces of different roughness using a reflection map.

16.7 What other lighting effects can you think of that would generalize Warn's flaps and cones?

16.8 Suppose that the array of patches shown in Fig. 16.64 is continued for another two rows, adding patches 5 and 6, and that the radiosity values for the patches are $B_1 = B_2 = 2$, $B_3 = B_4 = 4$, $B_5 = B_6 = 6$. Show that B_h and B_e are 5 and 3, respectively. Then show that B_b is 1. Is this a reasonable value?

Notice that it extends the linear trend from h to e. What happens as you add more rows of patches in a similar pattern? Suppose that you added a mirror image of the patches about the line ac and computed the radiosity values. Then B_b would be 2. Does this seem contradictory? Explain your answer.

16.9 Implement a simple recursive ray tracer based on the material in Sections 15.10 and 16.12.

16.10 Make your ray tracer from Exercise 16.9 more efficient by using some of the techniques discussed in Section 16.12.1.

16.11 Extend your ray tracer from Exercise 16.10 to do distributed ray tracing.

16.12 Implement a progressive-refinement radiosity algorithm, based on the pseudocode of Fig. 16.70. Use the hemicube method of computing form factors. Begin by computing only patch to patch exchange (ignoring substructuring). Leave out the ambient computation to make coding and visual debugging easier. Check your hemicube code by verifying that the delta form factors sum to (approximately) 1.

To display your images, you will need to implement a polygon visible-surface algorithm (perhaps the one used by your hemicube) or have access to an existing graphics system. Using constant-shaded polygons will improve interactivity if shaded graphics hardware is not available (and will make programming and debugging easier).

16.13 Explain why lighting must be done before clipping in the pipeline of Fig. 16.72.

16.14 Implement a testbed for experimenting with local illumination models. Store an image that contains for each pixel its visible surface's index into a table of material properties, the surface normal, the distance from the viewer, and the distance from and normalized vector to one or more light sources. Allow the user to modify the illumination equation, the intensity and color of the lights, and the surface properties. Each time a change is made, render the surface. Use Eq. (16.20) with light-source attenuation (Eq. 16.8) and depth-cueing (Eq. 16.11).

16.15 Add a shadow algorithm to a visible-surface algorithm that you have already implemented. For example, if you have built a z-buffer system, you might want to add the two-pass z-buffer shadow algorithm discussed in Section 16.4.4. (The postprocessing variant may be particularly easy to add if you have access to a graphics system that uses a hardware z-buffer. Explain how extra storage at each pixel, as described in Exercise 16.14, could be used to design a shadow postprocess that produced correct shading and proper highlights.)

16.16 Add interobject reflections to a visible-surface algorithm. Use reflection mapping for curved surfaces and the mirror approach for planar surfaces, as described in Section 16.6.

17 Image Manipulation and Storage

In this chapter, we explore methods for manipulating and storing images efficiently. We begin by considering the kinds of operations we would like to perform on images. Bear in mind that the images we are manipulating may be used either as images in their own right, or in the manufacture of some subsequent image, as in the environment mapping described in Chapter 16.

Several sorts of operations on images immediately come to mind. One is combining two images by overlaying or blending them, known as *compositing*. One application of compositing is in animation, when we wish to show a character moving around in front of a complicated background that remains unchanged. Rather than rerendering the background for each frame, we can instead render the background once and then generate many frames of the character moving about on a black background. We can then composite these individual frames as overlays to the background frame, thus producing images of a character moving about on the background. In compositing operations like this, antialiasing becomes extremely important to ensure that the outline of the character is not jagged against the background. It is also necessary to distinguish the background of an image from the content; in our example, the black background against which the character is drawn is the *background*, and the character itself is the *content*.

Often, the images to be composited are of different sizes, so we may wish to translate, scale, or rotate them before the composition. We may even wish to distort an image, so that it appears in perspective or appears to have be drawn on a rubber sheet and then stretched. Although we could make these changes by rerendering the image with an appropriate geometric transformation, this is often so difficult or time consuming as to be impractical. Indeed, it can be impossible if, say, the image has been obtained from an optical scan of a photograph, or if the original program or parameters used to create it have been lost.

We might also wish to apply various filters to an image so as to produce false colors, to blur the image, or to accentuate color or intensity discontinuities. This sort of filtering is applied to satellite photographs and to computed-tomography (CT) data, where the intensity of a point in the image reflects the density of material in the body. For example, very slight changes in intensity may indicate the boundaries between normal and cancerous cells, and we may wish to highlight these boundaries.

Images tend to be very large collections of data. A 1024 by 1024 image in which the color of each pixel is represented by a *n*-bit number takes *n*/8 MB of memory (in an 8-bit-per-byte machine). As described in Chapter 4, many graphics systems dedicate a great deal of memory to image storage (the frame buffer). If the image memory is accessible by other programs, then it may be used for output by one program, and then for input by another, or even for output by two different programs. This happens, for example, when we use a pixel-painting program to adjust individual pixels of a rendered image. This use of image memory (and the rigid structure of the memory, which constitutes a database format for diverse programs) has been called ''frame-buffer synergy'' by Blinn [BLIN85].

When an image is being stored in secondary memory, it is often convenient to compress the stored data (but not the image). Several schemes have been developed. The look-up tables (LUTs) described in Chapter 4, for example, significantly reduce the storage needed for an image, provided the image contains substantial color repetition. Of course, storing LUTs is typically done only when the frame buffers used for displaying the image support LUTs. We discuss several more sophisticated methods in Section 17.7.

Here we begin by reexamining our notion of an image. Then we describe some elementary operations on images: filtering and geometric transformations. We then discuss techniques for storing additional data with each pixel of an image, and using these data in compositing. Following this, we discuss various image storage formats; finally, we describe a few special effects that can be performed at the image level rather than in modeling or rendering.

17.1 WHAT IS AN IMAGE?

Images, as described in Chapter 14, are (at the most basic level) arrays of *values*, where a value is a collection of numbers describing the attributes of a pixel in the image (in bitmaps, e.g., the values are single binary digits). Often these numbers are fixed-point representations of a range of real numbers; for example, the integers 0 through 255 often are used to represent the numbers from 0.0 to 1.0. Often, too, these numbers represent the intensity at a point in the image (*gray scale*) or the intensity of one color component at that point. The dimensions of the array are called the *width* and *height* of the image, and the number of bits associated with each pixel in the array is called the *depth*.

We often consider an image as more than a mere array of values, however. An image is usually intended to represent an *abstract* image, which is a function of a continuous variable; each position in the abstract image has some value.[1] The images we work with

[1] What we are really talking about is a function whose domain is a rectangle in the Euclidean plane, rather than a discrete lattice of points in the plane.

(sometimes called *digital images* or *discrete images*) are functions of a discrete variable; for each $[i, j]$ pair, a value is associated with the pixel labeled $[i, j]$. As described in Chapters 3 and 14, choosing the best discrete image to represent an abstract image is difficult. In this chapter, we sometimes discuss reconstructing the abstract image in order to take new samples from it. Of course, we do not actually perform this reconstruction, since to do so we would need to generate values at infinitely many points. But we can reconstruct any individual value in the abstract image—in particular, we can reconstruct the finitely many values we want to sample.

If we create a discrete image from an abstract image by sampling (see Chapter 14), then reconstruct an abstract image from the digital image, the reconstructed abstract image and the original abstract image may or may not be the same. If the original abstract image had no high-frequency components, then the reconstructed image would be the same as the original, and the reconstruction would be said to be *faithful*. On the other hand, if the original image had components whose frequencies were too high, then the sampled image could not represent it accurately, and the reconstructed image would differ from the original.

One other aspect of images is important. Although filtering theory tells us a great deal about selecting a discrete image to represent an abstract image most accurately, much of the theory assumes that the abstract-image values at each point are real numbers and that the discrete-image values at each pixel will also be real numbers. In the case of bitmaps, however, nothing could be further from the truth: The values are binary. In more complex pixmaps, the values may be small binary numbers (e.g., 4 bits per pixel), or may range over so large a collection of numbers as to be effectively continuous. This *value discretization* leads to significant questions in image manipulation, such as how best to compress a bitmap. If 4 pixels—2 white and 2 black—are to be compressed into 1, should the compressed pixel be black or white? We discuss the consequences of value discretization when they are known and significant, but note that there is much that we do not yet understand.

17.2 FILTERING

Suppose we have an image produced without any antialiasing—for example, a drawing of a graph that was read into memory with an optical scanner that sampled the drawing at an array of points. How can we improve its appearance? The image certainly has jaggies that we would like to remove. But *every* image we can create is correct for *some* source image (where by *correct* we mean that it accurately represents a sample of the source image after low-pass filtering). If applying some mechanism alters and improves one image, applying the same mechanism to another image may damage that image. Thus, the mechanism we are about to describe should be used only on images that need smoothing. If jagged steps are present in a image that has been generated properly, then they are meant to be there, and postfiltering will only blur the image. (After all, what *should* an image of a staircase look like?)

Suppose we do want to smooth out an image to hide some jaggies. What can we do? An obvious start is to replace each pixel with the average of itself and its neighbors. This process, applied to the discrete image rather than to the abstract image, is called *postfiltering*. With postfiltering, pixels near the stair steps in the jaggies are blended so as to hide the steps; see Fig. 17.1, in which the filtering has been exaggerated. As we saw in Chapter 14, this constitutes filtering with a box filter; other, more sophisticated filters may yield better results. Before we examine other filters, let us consider the drawbacks of even this simple filtering method.

Suppose that we point sample a photograph of a picket fence. The pickets and the gaps between them are of the same width, the pickets are white, and the background is black. The pickets are spaced in the photograph so that the width of nine pickets and nine gaps covers a width of 10 pixels in the image. What will the sampled image look like? If the photograph is positioned so that the first pixel is exactly at the left-hand edge of the first picket, then the first pixel will be white, the next 5 pixels will be black, but the sixth through tenth pixels will be at pickets and hence will be white. The next 5 pixels will be black, and so on (see Fig. 17.2).

Now, what does our postfiltering do in this situation? It smoothes out the boundary between the sixth and seventh pixels, and leaves a large block of black followed by a large block of white. It cannot possibly fix all the problems implicit in the image. Clearly, postfiltering is not a good solution to the aliasing problem. In addition, since postfiltering will also blur any other edges in the image (even those that *should* be there), the resulting image will be unpleasantly fuzzy.

This problem can be partly remedied at the cost of shrinking the image: We can convert a $2n$ by $2n$ image into an n by n image by imagining that the source image is overlaid with a grid, each square of the grid enclosing 4 pixels of the source image. We can then average the 4 pixels in the square to create 1 pixel in the target image for each grid square. This amounts to postfiltering the image, then selecting alternate pixels on alternate scan lines. Note that less filtering computation is involved; we need to apply the filter to compute values for only those pixels included in the final image. That is, for only those pixels to appear in the output image, we compute a weighted average of pixels around the corresponding point in the source image. Of course, the resulting image is one-fourth the size of the original.

We can see how this works by recalling the analysis in Chapter 14. If the source image is produced by sampling at a frequency of 2ω, then any component of the original signal whose frequency is between 0 and ω will be accurately represented. For any frequency above ω, say $\omega + \phi$, the sampled signal will contain an alias at frequency $\omega - \phi$. Box filtering the sampled signal with a filter of width 2 substantially (but not completely) filters out the components of this signal with frequencies greater than $\omega/2$ (because the Fourier transform of the box filter is a sinc function, which tapers off rapidly as the frequency increases). Resampling at alternate pixels yields an effective sampling rate of ω; with this

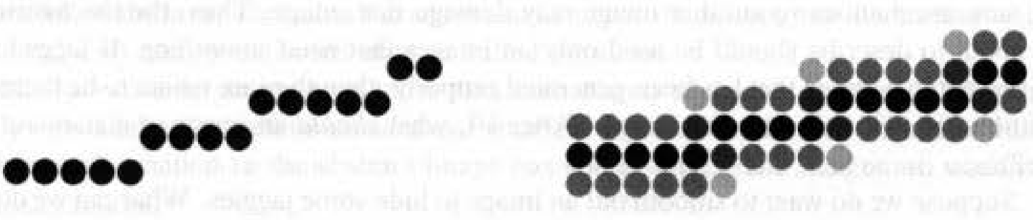


Fig. 17.1 The stair steps are smoothed by box filtering.

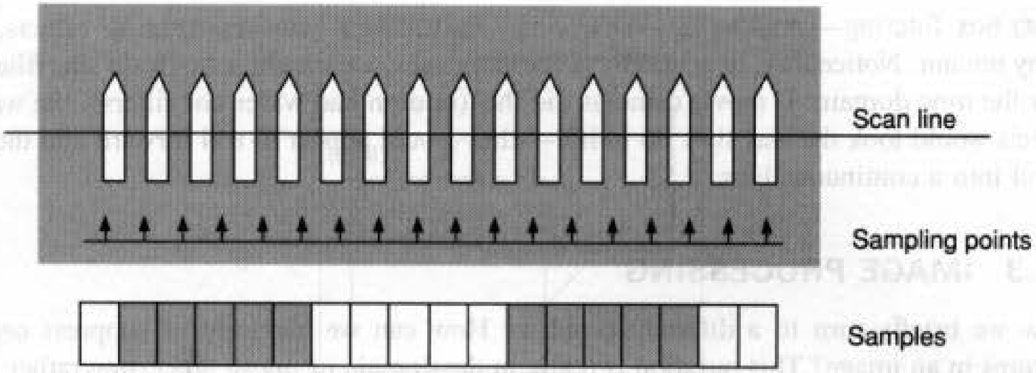


Fig. 17.2 Aliasing in a sampled image.

sampling rate, all frequencies up to $\omega/2$ can be accurately represented. But after applying the filter, these are exactly the frequencies that remain. If the original signal had components $\omega + \phi$ for which ϕ was large (i.e., greater than $\omega/2$), then the aliases of these components occur at frequencies below $\omega/2$, and hence persist in the final image. But for small values of ϕ, the aliases are filtered out, and so supersampling and postfiltering really do help reduce aliases.

Remember that the signals that represent such primitives as lines, rectangles, and any other geometric objects with clearly defined edges have components of arbitrarily high frequencies, so there is no hope of representing these correctly by any such method. At best, we can hope postfiltering will improve a bad image at the cost of fuzziness.

Other filters, such as the sinc filter, the Catmull–Rom filter, and the triangle filter, can produce better postfiltering results than can a pure box filter. The analysis of those filters given in Chapter 14 applies here as well. As a convenient rule of thumb, Whitted has suggested that postfiltering a high-resolution image produces obvious fuzziness, but that a 2048 by 2048 image can usually be postfiltered and sampled down to a 512 by 512 image with good results [WHIT85].

Now consider a temporal analogue of this problem: The spokes on a wagon wheel pass by a pixel on the screen very fast in an animation of a rolling wagon (this is the temporal analog of an object being striped with rapidly changing color; i.e., to closely spaced stripes). The frequency with which the spokes pass a point may be far greater than 30 times per second, the speed of typical video recording. Temporal aliasing is the inevitable result: The spokes appear to stand still or to turn backward. We are used to seeing this effect in movies, of course. In a movie, however, we actually see a *blurred* wagon wheel moving backward, because, in each exposure of a movie frame, the shutter is open for a brief (but not infinitesimal) time period (about one half of the time allocated to the frame; the remaining half is dedicated to moving the film forward). The shutter effectively applies a box filter to the scene in the time dimension. The result is some blurring, but aliases are still present. The blurring is due to the box filtering, and the aliases are due to the narrowness of the filter. All the box filters taken together cover only about half of the time sequence of the movie—the remainder is lost while the shutter is closed. The implication for computer graphics is clear: To get movie-quality frames for animation, we need to do (at the very least) box filtering—*prefiltering*—over time. Postfiltering removes some ill effects, but many remain. Notice that, to get really accurate images, we should actually do sinc filtering over the time domain. If movie cameras did this (or even had wider box filters), the wagon wheels would look the way they do in life—they would appear to roll forward and then to blend into a continuous blur.

17.3 IMAGE PROCESSING

Now we briefly turn to a different problem: How can we highlight or suppress certain features in an image? This question is really in the domain of *image processing* rather than computer graphics, but a few basic ideas are worth discussing. By scanning an image for rapid changes in value at adjacent points, we can do *edge detection* and *enhancement*. At places where the values of adjacent points differ sufficiently, we can push the values even further apart. If an image is *noisy*—that is, if random displacements have been added to its pixel values—then it can be *smoothed* by the filtering techniques discussed in the previous section. If the noise is sufficiently random, then filtering, which computes averages of adjacent pixels, should average out the noise, or at least filter its high-frequency components.

Another image-processing technique is *thresholding*, in which the points of an image at or near a particular value are highlighted. In a gray-scale image, this highlighting can be done by converting all pixels below some value to black, and all pixels above that value to white, producing a threshold edge between the black and white regions. The marching-cubes algorithm discussed in Chapter 20 gives a different mechanism for thresholding (in 3D): It explicitly constructs the boundary between the two regions as a surface (or a curve, in the 2D case). The components of this boundary can then be rendered into a new image with appropriate antialiasing to give a smoother indication of the threshold. For further information on this, see [GONZ87; SIG85].

17.4 GEOMETRIC TRANSFORMATIONS OF IMAGES

Suppose we wish to transform an image geometrically. Such transformations include translation, rotation, scaling, and other, nonlinear, operations. How can we do this?

Translating an image makes sense only if the image is thought of as a subimage of some larger image. Suppose we wish to move an n by k array of pixels (the *source*), whose upper-left corner is at (a, b), to a new position, with the upper-left corner at position (c, d) (the *target*). This transformation should be easy; we simply copy pixels from the source position to the target position, and (if we want) replace all source pixels that are not target pixels with the background color (see Fig. 17.3). Provided that care is taken to ensure that the copying is done in an order that prevents overwriting source pixels, when the source and destination overlap, and provided that the four numbers a, b, c, and d are all integers, this approach works fine.

But what if the starting and ending positions are not integers? Then we wish to reconstruct the abstract image for the source image, to translate it, and to sample this translated version. To do this explicitly is not feasible—we certainly do not wish to reconstruct the abstract image at *every* possible location, and then to select just a few of

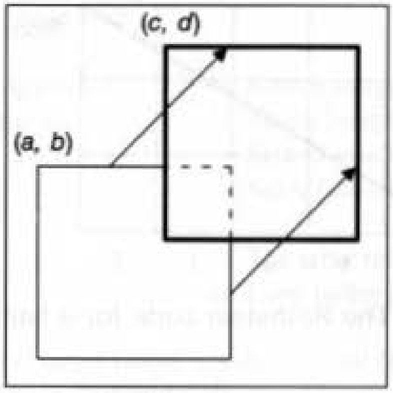


Fig. 17.3 A simple translation of an image.

these (infinitely many) points. Indeed, the same objection holds for scaling and rotation. Algorithms have been developed, however, that perform these operations in ways that are computationally correct (in various senses). Weiman has developed algorithms for performing scaling and shearing of images by rational amounts [WEIM80]. Rotations can be performed by a clever combination of these algorithms (see Exercise 17.1). Finding a similar algorithm for translating by arbitrary rational amounts is posed as Exercise 17.2.

17.4.1 Basic Geometric Transformations

Weiman posits that a gray-scale pixmap represents an abstract image in the following fashion: The abstract image is divided into squares (one square per pixel), and the average intensity of this abstract image in the square is the value assigned to the pixel. He thus assumes that he can perform a faithful reconstruction by drawing a picture consisting of gray squares whose tone is determined by the pixel values. Stretching a pixmap by a factor of p/q takes q columns of the original and stretches them to cover p columns of the target image. Performing area sampling on the result then generates the target image. Filtering theory tells us that this assumption about the nature of a sampled image and the consequent stretching algorithm are wrong in every sense: An abstract image should never be sampled while it has frequency components above the Nyquist frequency, and hence a proper reconstruction of an abstract image from a sampled one never has high-frequency components. An image in which adjacent squares have different (constant) values is a perfect example of an image with lots of high-frequency components, so this is certainly a bad reconstruction. And finally, filtering theory says that when converting such an image to a pixmap, we should use sinc filtering rather than box filtering. Nevertheless, if Weiman's hypotheses are allowed, his algorithm for performing these linear transformations is quite clever. It is also the basis for a very good bitmap-scaling algorithm (see Exercise 17.3).

Suppose we wish to scale an image by a factor p/q (where $p > q$, and p and q are integers with no common factors). The first step is to generate a Rothstein code [ROTH76] for the number p/q. This code is a binary sequence that describes a line whose slope is q/p (any scan-converted line can be used to generate a similar code). Figure 17.4 shows a line of slope $\frac{3}{5}$ with 15 tick marks on it. As the line passes from left to right through the figure, it crosses the horizontal grid lines. If a column contains such a grid-line crossing, it is marked

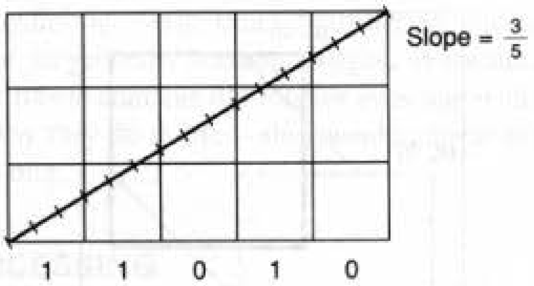


Fig. 17.4 The Rothstein code for a line of slope $\frac{3}{5}$.

with a 1; otherwise, is marked with a 0. Each column contains three tick marks; the bit associated with the column is 0 unless one of the three tick marks is at a multiple of 5, since multiples of 5 are where horizontal-line crossings occur. Thus, the interval between mark 9 and mark 12 is assigned a 1, since mark 10 lies within it. (A tick at the left side of a column is considered to be in the column, whereas ticks on the right are not.)

The Rothstein code may be viewed as a mechanism for distributing q 1s evenly among p binary digits. We can therefore use it to tell us how to distribute each of the q columns of the source image among p columns of the target image. Unfortunately, using the 1s in the Rothstein code as indicators of where to copy the source data leaves some of the target columns blank. The Rothstein code can be cyclically permuted, however, to give different mappings of source to destination.[2] Taking the average of these gives the result.

The pseudocode for this procedure is shown in Fig. 17.5.

To scale by a number smaller than 1, we simply reverse the process. The Rothstein code for q/p tells us which of the source columns should appear in the target (a 1 in the Rothstein code tells us to select that column). Again, we average over all cyclic permutations of the code.

The Rothstein code can also be used to generate a description of shearing. A 1 in the Rothstein code indicates that the corresponding column of the source pixmap should be shifted up 1 pixel. Over the course of q columns, there are p shifts, resulting in a vertical shear of amount q/p. Once again, we should cyclically permute and average the results.

17.4.2 Geometric Transformations with Filtering

Feibush, Levoy, and Cook give a somewhat more sophisticated mechanism for transforming images [FEIB80]. (Their algorithm is developed for use in an algorithm for mapping textures onto surfaces—see Chapter 16.) Before discussing the details, we note the algorithm's good and bad points. The algorithm has the advantage that it computes reasonable values for boundary pixels: If an image is rotated so that some target pixel is only partly covered by a source pixel, the algorithm recognizes that pixel as a special case and processes it accordingly. It computes values by applying a weighted filter to the source image to determine pixel values for the target image, which helps to reduce aliasing in the resulting image. But since this filter is more than 1 pixel wide, if the algorithm is used to

[2] A cyclic permutation of a binary sequence is the repeated application of logical shift operations to the sequence.

```
/* Expand the width of an image by p/q. */
void WeimanExpansion(
      const grayscalePixmap source,      /* Source image, size n × k */
      grayscalePixmap target,            /* Target image, width at least k * p/q */
      int n, int k,                      /* Size of source image */
      int p, int q)                      /* Scale factor is p/q */
{
   char roth[MAX];                       /* The array must hold at least p items. */
   int i, j, s;                          /* Loop indices */

   /* Source image is n × k, target image is to be n × ceil (k * p/q).³ */
   int targetWidth = ceil (k * p / (double) q);

   Rothstein (roth, p, q);               /* Store the Rothstein code for p/q in array roth. */
   SetToBlank (target, n, targetWidth);  /* Clear the target array. */
   for (i = 0; i < p; i++) {             /* For several passes through the algorithm. . . */
      int sourceCol = 0;
      Permute (roth);                    /* Apply cyclic permutation to Rothstein code. */
      /* For each column of the target */
      for (j = 0; j < targetWidth; j++) {
         if (roth[j] == 1) {             /* If code says to copy source column */
            for (s = 0; s < n; s++) {    /* Copy all the pixels. */
               target[s][j] += source[s][sourceCol];
            }
            sourceCol++;                 /* Go to next column. */
         }
      }
   }

   /* Divide by q to compensate for adding each source column to target q times. */
   for (i = 0; i < n; i++)
      for (j = 0; j < targetWidth; j++)
         target[i][j] = rint (target[i][j] / (double) q);
}  /* WeimanExpansion */
```

Fig. 17.5 The Weiman algorithm for expanding an image.

perform the identity transformation on an image, it blurs the values. In Section 17.5.3, we discuss this and other drawbacks of transformation algorithms.[4]

At this point, it is worthwhile to separate two aspects of image transformation. The first is computing which point in the source image is mapped to the center of the pixel in the

[3] The ceiling of a number is the smallest integer greater than or equal to the number; ceiling(1.6) = 2, ceiling(1.1) = 2, and ceiling(6.0) = 6.

[4] Feibush, Levoy and Cook note that any filter can be used, but describe the algorithm in terms of a filter of diameter 2. The algorithm generally performs better with this filter than it does with a unit-area box filter.

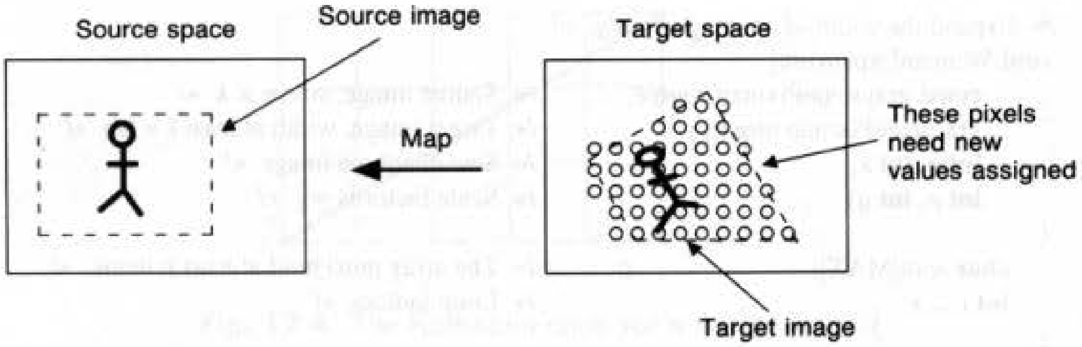


Fig. 17.6 The relationship of the source space, source image, target space, and target image in the Feibush–Levoy–Cook algorithm.

target image. The second is computing the value for the pixel in the target image. The first task is merely algebraic, in that it involves computing values (and inverse values) of a transformation. This may be done efficiently by various incremental methods. The second task also has numerous solutions, all of which involve choosing some filtering function to apply to the original image. The method described in the next few pages assumes a filtering function that is circularly symmetric and has a modest size (i.e., is nonzero only on a small part of the plane).

The algorithm starts with a source image (thought of as lying in one copy of the Euclidean plane, called the *source space*), a projective map[5] from another copy of the Euclidean plane (the *target space*) to the source space, and a polygonal region in the target space. The target image is the collection of pixels in the target space that are near the polygonal region, and it is these pixels whose values need to be assigned (see Fig. 17.6). Note that the projective map here goes from target to source, the reverse of the usual naming convention for mathematical functions.

To start, we choose a symmetric filter function that is nonzero only for (x, y) very close to $(0, 0)$ (perhaps within a 2-pixel distance). The *support* of this filter function is the set of points on which it is nonzero. We take a copy of the bounding rectangle for the support of the filter and translate it to each pixel in the target space. Whenever this rectangle intersects the target polygon, the pixel is considered to be in the target image. This translated rectangle is called the *bounding rectangle for the target pixel*, and the translated support of the filter function is called the pixel's *convolution mask* (see Fig. 17.7).

The vertices of the target polygon are transformed to source space just once, for repeated use. The resulting polygon is called the *source polygon*. The bounding rectangle of each target pixel is transformed to the source space, where it becomes a quadrilateral. A bounding rectangle for this quadrilateral is computed, then is clipped by the source polygon (because clipping a rectangle by the source polygon is much easier than clipping a general quadrilateral). The pixels in the source space that lie in this clipped quadrilateral are transformed to the target space; only those that fall within the target pixel's bounding rectangle are retained.

[5] A *projective map* is a map represented by a 3 × 3 matrix operating on the plane using homogeneous coordinates, in the manner described in Chapter 5.

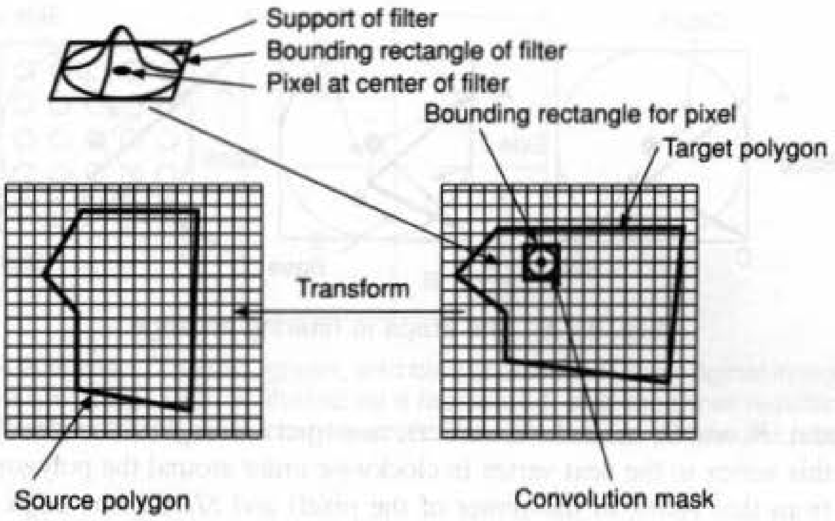

Support of filter
Bounding rectangle of filter
Pixel at center of filter
Bounding rectangle for pixel
Target polygon
Transform
Source polygon
Convolution mask

Fig. 17.7 Terms used in the Feibush–Levoy–Cook algorithm.

These transformed pixels are then averaged together by the weights given by the filter to yield a value for the target pixel. This target pixel value is correct only if the entire pixel is within the transformed image boundaries. If the image has been rotated, for example, then the transformed edges of the image may cut across pixels (more precisely, across their convolution masks).

Thus pixels are not entirely determined by the value just computed; that value only contributes to the pixel's value, in proportion to the coverage of the pixels. The contribution can be determined analytically. Figure 17.8 shows the transformed edge of the source image passing through a pixel's bounding rectangle, and within that rectangle passing through the pixel's convolution mask. To find the contribution of the computed value to the pixel's final value, we do the following:

1. Clip the image polygon against the bounding rectangle for the pixel (see Fig. 17.9). The points of intersection with the edges of the bounding rectangle were already computed in determining whether the pixel was in the target image.
2. For each vertex of the clipped polygon (in Fig. 17.9, a single triangle with vertices

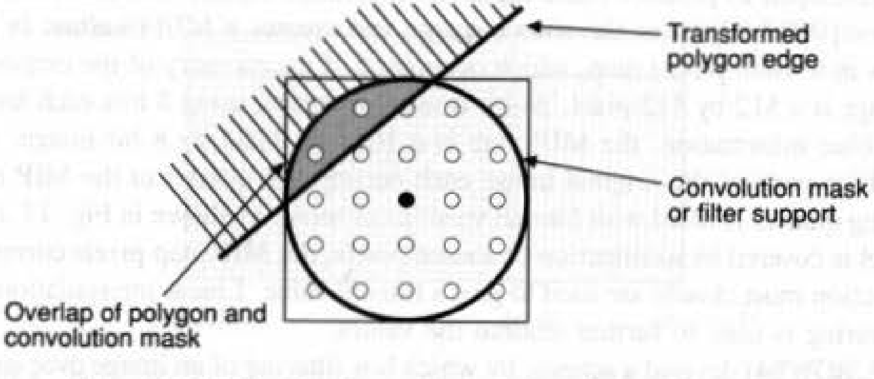

Transformed polygon edge
Convolution mask or filter support
Overlap of polygon and convolution mask

Fig. 17.8 Filtering for a pixel at the edge of the polygon.

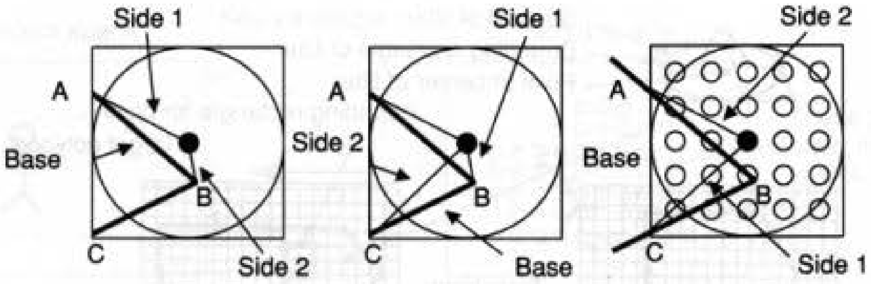


Fig. 17.9 The steps in filtering an edge.

labeled *A*, *B*, and *C*, in clockwise order), construct a triangle with sides *BASE* (the edge from this vertex to the next vertex in clockwise order around the polygon), *SIDE1* (the edge from this vertex to the center of the pixel) and *SIDE2* (the edge from the next vertex to the center of the pixel).

3. Consider the filter function as being plotted in a third dimension above the convolution mask. The weight a region contributes to the total for a pixel is proportional to the volume above that region and under the graph of the filter. So we now compute the volumes above each triangle. As Fig. 17.9 shows, some of these volumes must be added and some subtracted to create the correct total contribution for the region. The rule is that the volume is added if the cross-product of *SIDE1* and *SIDE2* points into the page; otherwise, it is subtracted (see Exercise 17.4).

Computing the volumes in step 3 is easier than it might appear, in that they can be precomputed and then extracted from a look-up table during the actual filtering process. Exercise 17.5 shows how to do this precomputation.

17.4.3 Other Pattern Mapping Techniques

The Feibush-Levoy-Cook algorithm provides excellent results for pattern mapping onto polygons, but requires computing a filtered value at each point, so that for each pixel in the image a filtering computation is performed. In a perspective picture of a plane (receding to a vanishing point) a single pixel in the final image may correspond to thousands of pixels in the source pattern, and thus require an immense filtering computation. Several techniques have been developed to produce more rapid (if sometimes slightly less accurate) filtering.

Williams [WILL83] takes the source image and creates a MIP (*multum in parvo*—many things in a small place) map, which occupies $\frac{4}{3}$ of the memory of the original. If the original image is a 512 by 512 pixel, 24-bit true color image, using 8 bits each for the red, green, and blue information, the MIP map is a 1024 by 1024 by 8 bit image. The red, green, and blue parts of the original image each occupy one quarter of the MIP map, and the remaining quarter is filled with filtered versions of these, as shown in Fig. 17.10. When a target pixel is covered by a collection of source pixels, the MIP map pixels corresponding to this collection most closely are used to give a filtered value. Linear interpolation between levels of filtering is used to further smooth the values.

Crow [CROW84] devised a scheme by which box filtering of an image over any aligned rectangle can be done rapidly. For quick pattern mapping, this suffices in many cases—a rectangular box corresponding closely to the shape of the transformed target pixel is used to

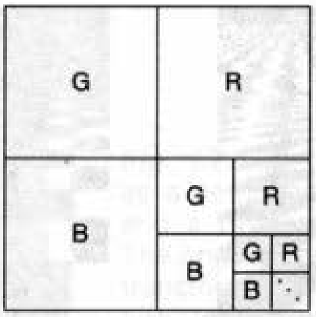


Fig. 17.10 A MIP map. The red, green, and blue channels of the original image fill three quarters of the MIP map. Each is filtered by a factor of 4, and the three resulting images fill up three quarters of the remaining quarter. The process is continued until the MIP map is filled.

compute a filtered pattern value for the pixel. The scheme is based on the algebraic identity $(x + a)(y + b) - (x + a)\,y - x\,(y + b) + xy = ab$. Interpreted geometrically, this says that the area of the small white rectangle in Fig. 17.11 can be computed by taking the area of the large rectangle and subtracting the areas of both the vertically and the horizontally shaded rectangles, and then adding back in the crosshatched rectangle (which has been subtracted twice). By taking the source image and creating a new image, whose value at pixel (x, y) is the sum of all the values in the source image in the rectangle with corners $(0, 0)$ and (x, y), we create a *summed area table*, S. We can now compute the sum of the pixels in the rectangle with corners at (x, y) and $(x + a, y + b)$, for example, by taking $S[x + a, y + b] - S[x + a, y] - S[x, y + b] + S[x, y]$.

Glassner [GLAS86] observes that if the transformed pixel is not approximately an aligned rectangle, then summed area tables may blur the result excessively. He therefore develops a system in which the excess area in the aligned bounding box for the pixel is systematically trimmed, in order to provide a more accurate estimate of the filtered source image at the point. This requires detecting the geometry of the inverse-mapped target pixel relative to its bounding box.

Heckbert [HECK86a] proposes a system using both the Feibush-Levoy-Cook method and MIP maps. He maps the target pixel's filter support (which is supposed to be circular,

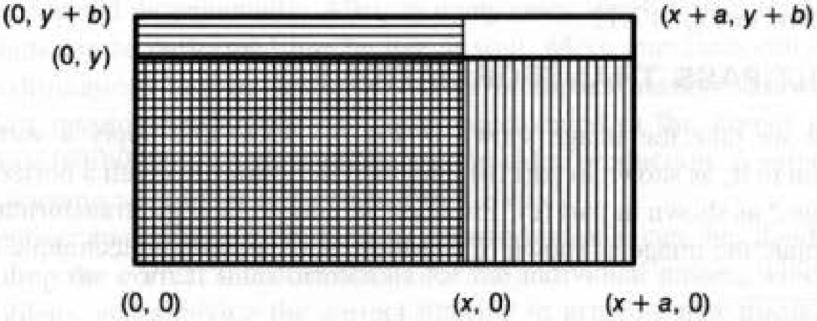


Fig. 17.11 The area of the small white rectangle in the image is computed by subtracting the horizontally and vertically shaded areas from the area of the large rectangle, and then adding back in the area of the crosshatched rectangle.

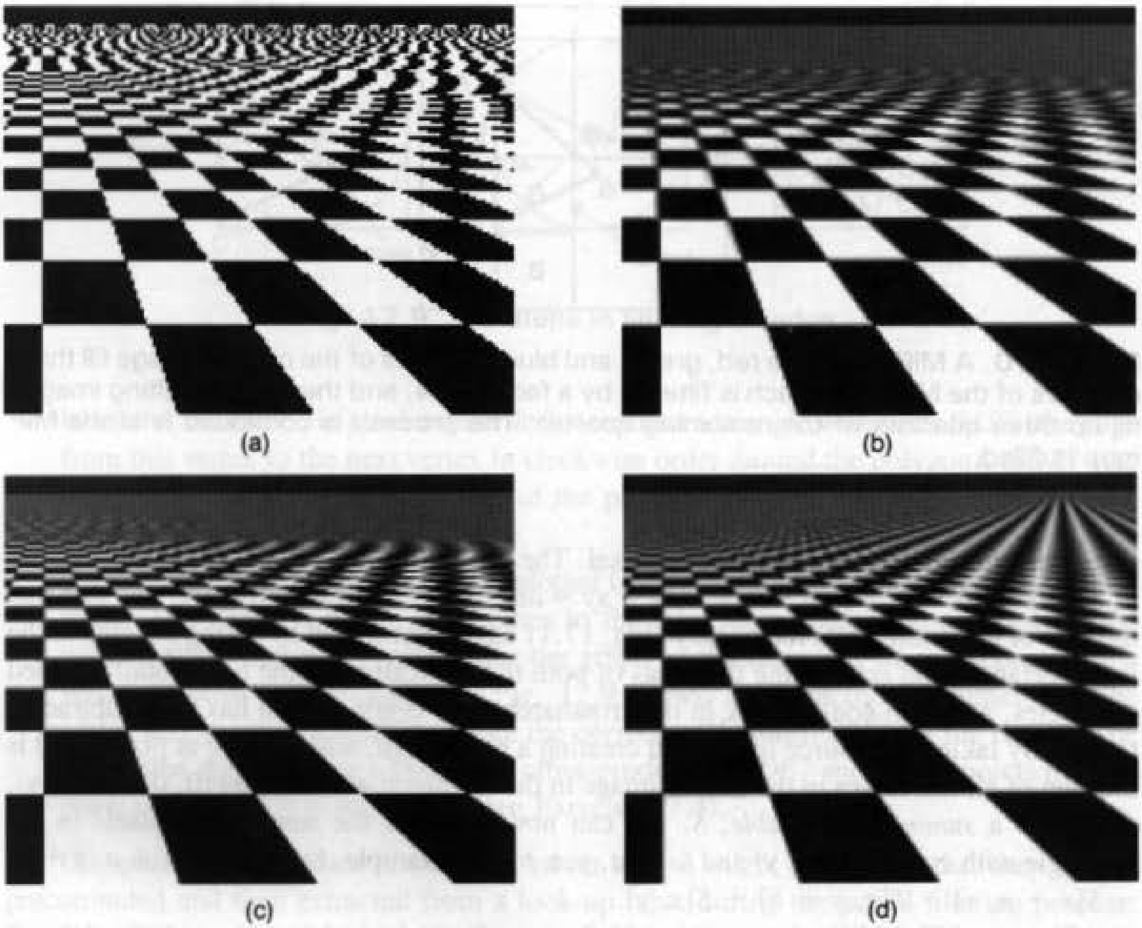


Fig. 17.12 (a) Point sampling of the source image. (b) MIP map filtering. (c) Summed area table filtering. (d) Elliptical weighted average using MIP maps and a Gaussian filter. (Courtesy of P. Heckbert.)

and is defined by a quadratic function) to an elliptical region in the source image (defined by a different quadratic). Depending on the size of this region in the source image, an appropriate level in a MIP map for the source image is selected, and the pixels within it are collected in a weighted sum over the elliptical region. This weighted sum is the value assigned to the target pixel. This combines the accuracy of the Feibush-Levoy-Cook technique with the efficiency of the MIP map system. A comparison of pattern-mapping results is shown in Fig. 17.12.

17.5 MULTIPASS TRANSFORMATIONS

Suppose that we take the image shown in Fig. 17.13(a) and apply a vertical shearing transformation to it, as shown in part (b), and then we follow this with a horizontal shearing transformation,[6] as shown in part (c). Provided we choose the right transformations, the net effect is to rotate the image as shown [CATM80]. Such a two-pass technique may be much

[6] The two shearing transformations are actually shear-and-scale transformations. The first takes a column of pixels and translates and compresses it in the vertical direction. The second does the same for a row of pixels.

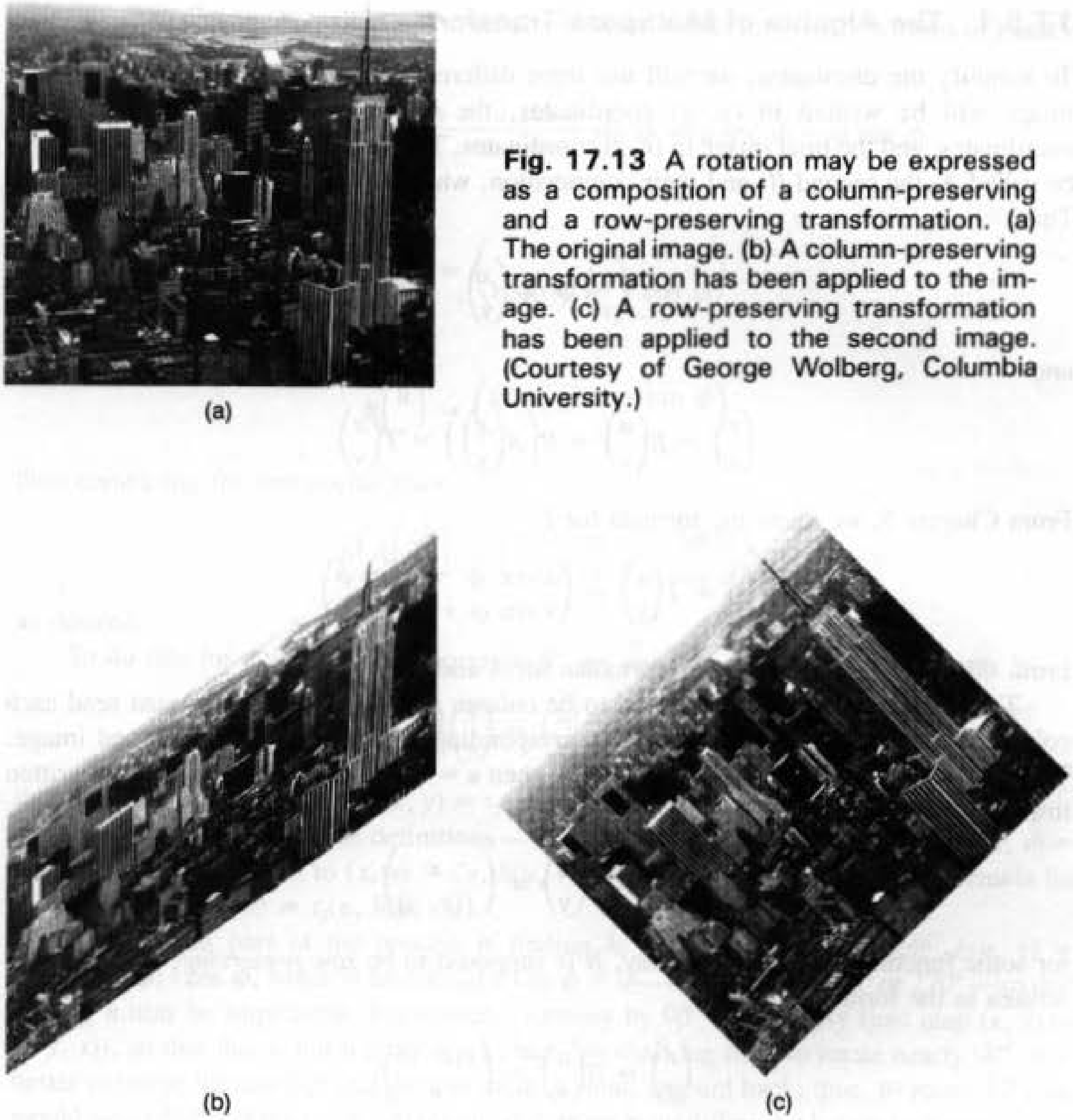


Fig. 17.13 A rotation may be expressed as a composition of a column-preserving and a row-preserving transformation. (a) The original image. (b) A column-preserving transformation has been applied to the image. (c) A row-preserving transformation has been applied to the second image. (Courtesy of George Wolberg, Columbia University.)

faster to compute than a direct application of the rotation transformation, since it operates on one vertical or horizontal line of pixels at a time, and the computations within each such line can be performed incrementally. Also, in many cases, the filtering necessary to avoid aliasing artifacts can be performed line by line as well. More important still is that a wide class of transformations can be implemented as multipass transformations [CATM80, SMIT87]. This multipass technique has been implemented in the Ampex digital optics (ADO) machine [BENN84], which is widely used in video production. A survey of this and other image warping techniques is given in [WOLB90].

Implementing two-pass (or multipass) transformations can be divided into two subtasks: finding the correct transformations for the individual passes, which is a purely algebraic problem, and applying the correct filtering to generate new pixels, which is an antialiasing problem. Since the second part will depend on the solution to the first, we begin by solving the first problem in the case of a rotation.

17.5.1 The Algebra of Multipass Transforms

To simplify the discussion, we will use three different sets of coordinates. The original image will be written in (x, y) coordinates, the vertically sheared image in (u, v) coordinates, and the final image in (r, s) coordinates. The first shearing transformation will be called A, the second B, and their composition, which is the rotation, will be called T. Thus,

$$\begin{pmatrix} u \\ v \end{pmatrix} = A\begin{pmatrix} x \\ y \end{pmatrix},$$

and

$$\begin{pmatrix} r \\ s \end{pmatrix} = B\begin{pmatrix} u \\ v \end{pmatrix} = B\left(A\begin{pmatrix} x \\ y \end{pmatrix}\right) = T\begin{pmatrix} x \\ y \end{pmatrix}.$$

From Chapter 5, we know the formula for T:

$$\begin{pmatrix} r \\ s \end{pmatrix} = T\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x\cos\phi - y\sin\phi \\ x\sin\phi + y\cos\phi \end{pmatrix}.$$

From this, we will determine the formulae for A and B.

The transformation A is supposed to be *column preserving*; that is, it must send each column of the original image into the corresponding column of the transformed image. Thus, if the pixel (x, y) is sent to (u, v) by A, then $u = x$. In other words, A must be written in the form

$$\begin{pmatrix} u \\ v \end{pmatrix} = A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ f(x, y) \end{pmatrix}$$

for some function f. In the same way, B is supposed to be *row preserving*, so B must be written in the form

$$\begin{pmatrix} r \\ s \end{pmatrix} = B\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} g(u, v) \\ v \end{pmatrix}$$

for some function g. To determine the formulae for A and B, we need to find the functions f and g.

Writing out the composite, we have

$$\begin{pmatrix} r \\ s \end{pmatrix} = B\begin{pmatrix} u \\ v \end{pmatrix} = B\left(A\begin{pmatrix} x \\ y \end{pmatrix}\right) = B\begin{pmatrix} x \\ f(x, y) \end{pmatrix} = \begin{pmatrix} g(x, f(x, y)) \\ f(x, y) \end{pmatrix}.$$

From this equation, we see that s and $f(x, y)$ are equal. Thus, the formula for s in terms of x and y gives the formula for $f(x, y)$: $f(x, y) = x\sin\phi + y\cos\phi$. Determining the formula for $g(u, v)$ is more complex. We know that, in terms of x and y, we can write $g(u, v) = x\cos\phi - y\sin\phi$. To write this in terms of u and v, we must solve for x and y in terms of u and v and substitute. Solving for x is easy, since we observed previously that $u = x$. Solving for y is slightly more difficult: $v = f(x, y) = x\sin\phi + y\cos\phi$, so $y = (v - x\sin\phi) / \cos\phi =$

$(v - u \sin \phi) / \cos \phi$. Substituting this result into the formula for $g(u, v)$ in terms of x and y, we get

$$g(u, v) = u \cos \phi - \frac{v - u \sin \phi}{\cos \phi} \sin \phi = u \sec \phi - v \tan \phi.$$

In summary, if we define

$$A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ x \sin \phi + y \cos \phi \end{pmatrix},$$

and

$$B\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} u \sec \phi - v \tan \phi \\ v \end{pmatrix},$$

then computing the composite gives

$$B\left(A\begin{pmatrix} x \\ y \end{pmatrix}\right) = \begin{pmatrix} x \cos \phi - y \sin \phi \\ x \sin \phi + y \cos \phi \end{pmatrix},$$

as desired.

To do this for a general transformation T, we must do exactly the same work. If

$$T\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} t_1(x, y) \\ t_2(x, y) \end{pmatrix},$$

then we define $u = x$ and $v = f(x, y) = t_2(x, y)$. To define $g(u, v)$, we need to solve for y in terms of u and v, using these definitions—that is, to find a function h such that $(u, v) = (x, t_2(x, y))$ is equivalent to $(x, y) = (u, h(u, v))$. When we have found h, the formula for $g(u, v)$ is just $g(u, v) = t_2(u, h(u, v))$.

The difficult part of the process is finding h. In fact, in our example, $h(u, v) = (v - u \sin \phi) / \cos \phi$, which is undefined if $\cos \phi = 0$—that is, if $\phi = 90°$ or $270°$—so that finding h may be impossible. Fortunately, rotating by 90° is very easy (just map (x, y) to $(-y, x)$), so that this is not a problem. In fact, we shall see that, to rotate nearly 90°, it is better to rotate the full 90° and then to rotate a small amount back; thus, to rotate 87°, we would rotate 90° and then −3°. Algebraically, there is no difference between the two maps; at the pixel level, however, where filtering is involved, the difference is significant.

A rotation can also be broken into three transformations so as to avoid this *bottleneck* problem [PAET86; TANA86; WOLB90]. The decomposition for a rotation by ϕ is

$$\begin{bmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{bmatrix} = \begin{bmatrix} 1 & -\tan \phi/2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \sin \phi & 1 \end{bmatrix} \begin{bmatrix} 1 & -\tan \phi/2 \\ 0 & 1 \end{bmatrix}.$$

Note that each transformation involves a computation with one multiplication and one addition. Also, when $\phi > 90°$, we can do the rotation by first rotating by 180° and then by $180° - \phi$, so that the argument of the tangent function is never greater than 45°.[7]

[7] The tangent function is well behaved for angles near 0°, but has singularities at ±90°. Evaluating it for angles near 0° is therefore preferable.

To show that the multipass technique is not limited to rotations, let us factor a different map, which distorts a square into a trapezoid. (Such maps arise in the perspective transformations described in Chapter 6.) As an example, we take

$$T\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x/(y+1) \\ y/(y+1) \end{pmatrix}.$$

Just as before, we wish to find functions

$$A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ f(x, y) \end{pmatrix} \quad \text{and} \quad B\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} g(u, v) \\ v \end{pmatrix}$$

such that $B\left(A\binom{x}{y}\right) = T\binom{x}{y}$. In this case, $v = f(x, y) = t_2(x, y) = y/(y + 1)$. We need to find $g(u, v)$ so that $g(u, v) = x/(y + 1)$. Solving the equation of f for y, we get $y = -v/(v - 1)$. Thus (recalling that $u = x$), we can write $g(u, v) = u\,/\,(-v/(v - 1) + 1) = u\,/\,(-1/(v - 1)) = u(1 - v)$. Our two passes become

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} x \\ y/(y+1) \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} r \\ s \end{pmatrix} = \begin{pmatrix} u(1-v) \\ v \end{pmatrix}.$$

You should check that the composition of these transformations is really the original transformation T.

The technique has been generalized to handle other maps by Smith and colleagues [SMIT87]. Translation, rotation, scaling, and shearing all work easily. In addition, Smith considers functions of the form

$$T(x, y) = S(m(x)\,h_1(y),\, m(x)\,h_2(y)),$$

where S is a standard computer graphics transform—that is, a transformation of the plane by translation, scaling, rotation, and perspective transformations—and $m(x)$, $h_1(y)$ and $h_2(y)$ are arbitrary. He also considers maps T whose component functions $t_1(x, y)$ and $t_2(x, y)$ are bicubic functions of x and y, under the special hypothesis that T is injective (i.e., no two (x, y) points map to the same (r, s) point).

17.5.2 Generating Transformed Images with Filtering

When we transform an image by a row-preserving (or column-preserving) transformation, the source pixels are likely not to map exactly to the target pixels. For example, the pixels in a row might all be translated by $3\frac{1}{2}$ pixels to the right. In this case, we must compute values for the target pixels by taking combinations of the source pixels. What we are doing, in effect, is considering the values of the source pixels as samples of a function on a real line (the row); the values at the target pixels will be different samples of this same function. Hence, the process is called *resampling*.

The theoretically ideal resampling process is to take, for a given target pixel, a weighted average of the source pixels whose transformed positions are near it. The weights associated with each source pixel should be sinc(kd), where d is the distance from the transformed source pixel to the target pixel and k is some constant. Unfortunately, this requires that every source pixel in a row contribute to every target pixel. As usual, we can instead work