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.

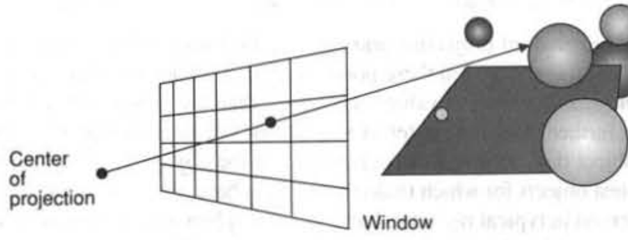

Fig. 15.55 A ray is fired from the center of projection through each pixel to which the window maps, to determine the closest object intersected.

```
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

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

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

$$\Delta x = x_1 - x_0, \qquad \Delta y = y_1 - y_0, \qquad \Delta z = z_1 - z_0. \quad (15.10)$$

Thus,

$$x = x_0 + t\,\Delta x, \qquad y = y_0 + t\,\Delta y, \qquad z = z_0 + t\,\Delta z. \quad (15.11)$$

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

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

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

$$x^2 - 2ax + a^2 + y^2 - 2by + b^2 + z^2 - 2cz + c^2 = r^2, \quad (15.13)$$

$$(x_0 + t\Delta x)^2 - 2a(x_0 + t\Delta x) + a^2 + (y_0 + t\Delta y)^2 - 2b(y_0 + t\Delta y) + b^2 \quad (15.14)$$
$$+ (z_0 + t\Delta z)^2 - 2c(z_0 + t\Delta z) + c^2 = r^2,$$

$$x_0^2 + 2x_0\Delta xt + \Delta x^2t^2 - 2ax_0 - 2a\Delta xt + a^2 \quad (15.15)$$
$$+ y_0^2 + 2y_0\,\Delta yt + \Delta y^2t^2 - 2by_0 - 2b\Delta yt + b^2$$
$$+ z_0^2 + 2z_0\,\Delta zt + \Delta z^2t^2 - 2cz_0 - 2c\Delta zt + c^2 = r^2.$$

Collecting terms gives

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

$$(\Delta x^2 + \Delta y^2 + \Delta z^2)t^2 + 2t[\Delta x\,(x_0 - a) + \Delta y(y_0 - b) + \Delta z(z_0 - c)] \quad (15.17)$$
$$+ (x_0 - a)^2 + (y_0 - b)^2 + (z_0 - c)^2 - r^2 = 0.$$

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

$$Ax + By + Cz + D = 0, \quad (15.18)$$

substitution from Eq. (15.11) yields

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

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

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

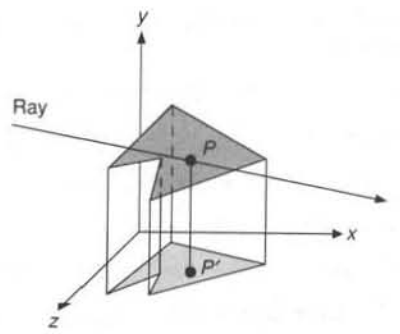

Fig. 15.57 Determining whether a ray intersects a polygon. The polygon and the ray's point of intersection p with the polygon's plane are projected onto one of the three planes defining the coordinate system. Projected point p' is tested for containment within the projected polygon.

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

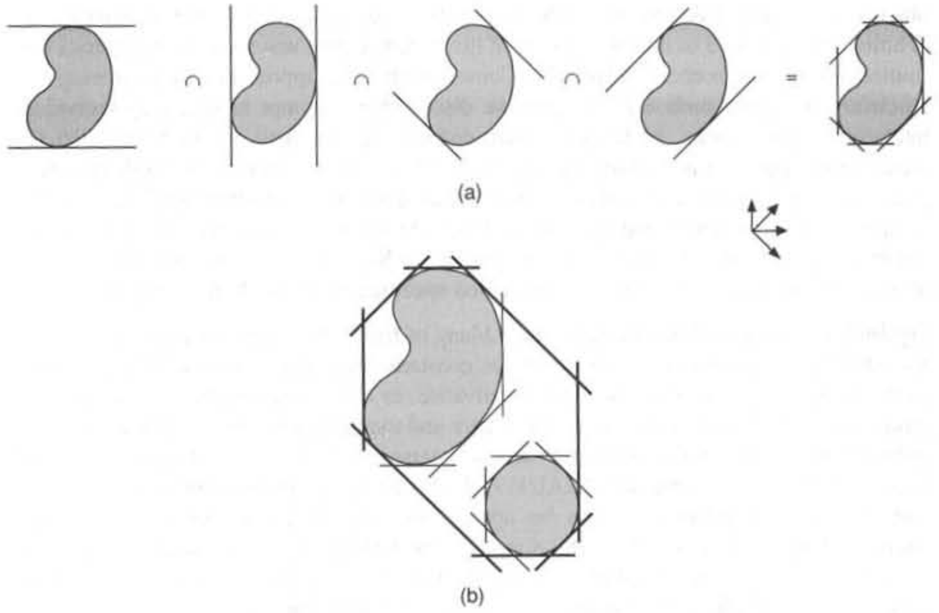

Fig. 15.58 Bounds formed intersection of slabs. (a) Object bounded by a fixed set of parameterized slabs. (b) The bounding volume of two bounding volumes.

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

```
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 ith bounding volume if it intersects the root can be approximated by A_i / A_r, where A_i is the surface area of the ith 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

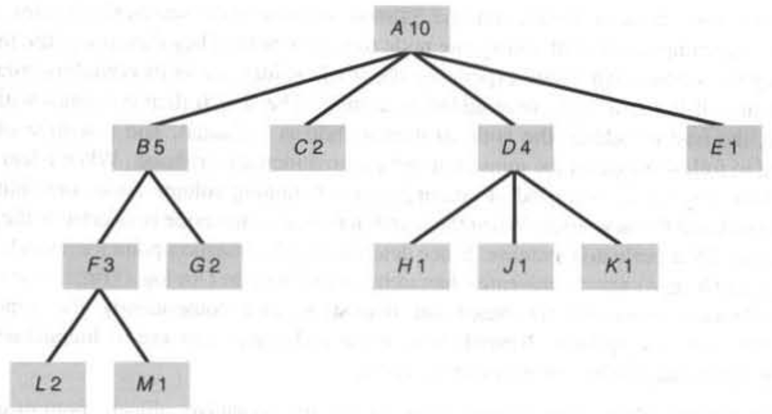

Fig. 15.60 Estimating the cost of a hierarchy. Letter is node name; number is node surface area.

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 $k (A_{\text{new}} - A_{\text{old}}) + A_{\text{new}}$, 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 $2A_{\text{new}}$. In both cases, the incremental cost to the new child's grandparent and older ancestors must also be computed as $k(A_{\text{new}} - A_{\text{old}})$, 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

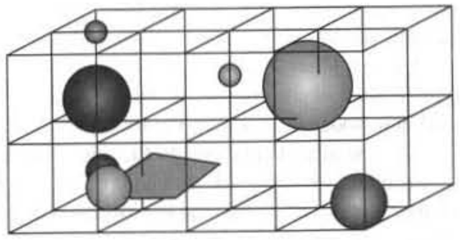

Fig. 15.61 The scene is partitioned into a regular grid of equal-sized volumes.

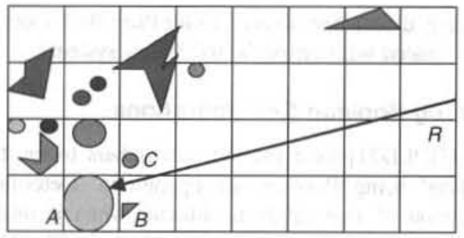

Fig. 15.62 Spatial partitioning. Ray R needs to be intersected with only objects A, B, and C, since the other partitions through which it passes are empty.

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

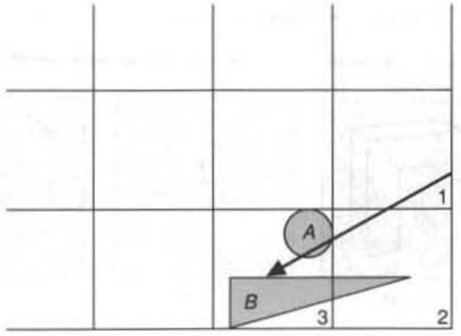

Fig. 15.63 An object may be intersected in a different voxel than the current one.

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

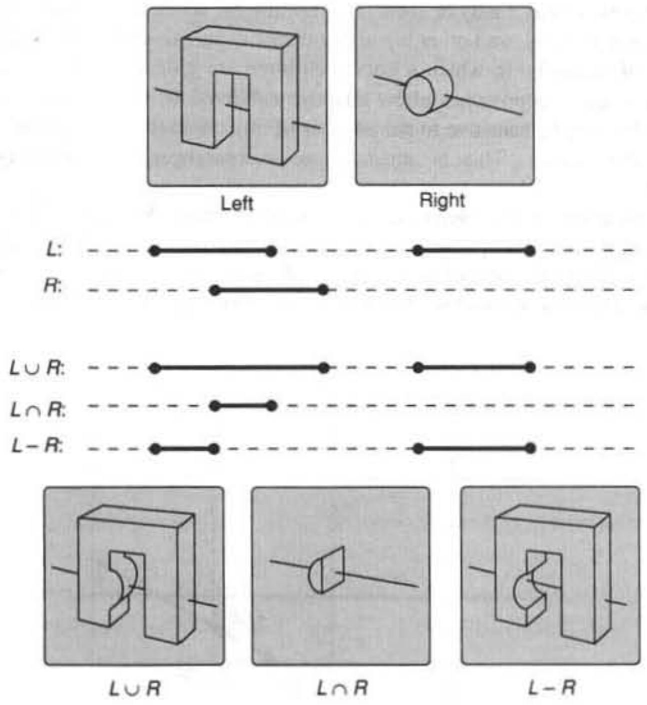

Fig. 15.64 Combining ray–object intersection span lists. (Adapted from [ROTH82] with permission.)

```
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

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

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

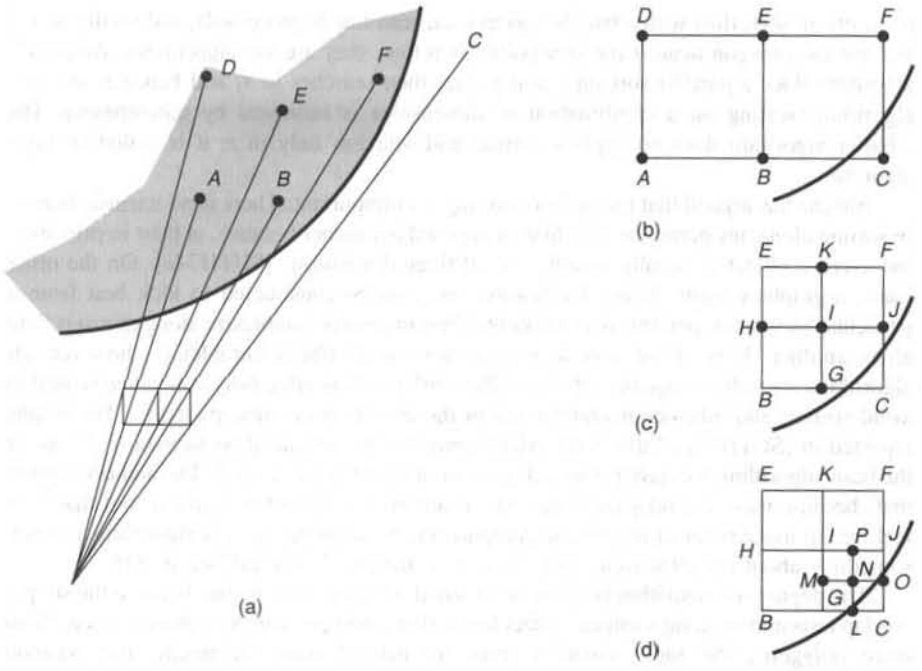

Fig. 15.66 Adaptive supersampling. (a) Two pixels and the rays fired through their corners. (b) The left pixel is not subdivided. (c) The right pixel is subdivided. (d) The lower-right subpixel is subdivided.

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