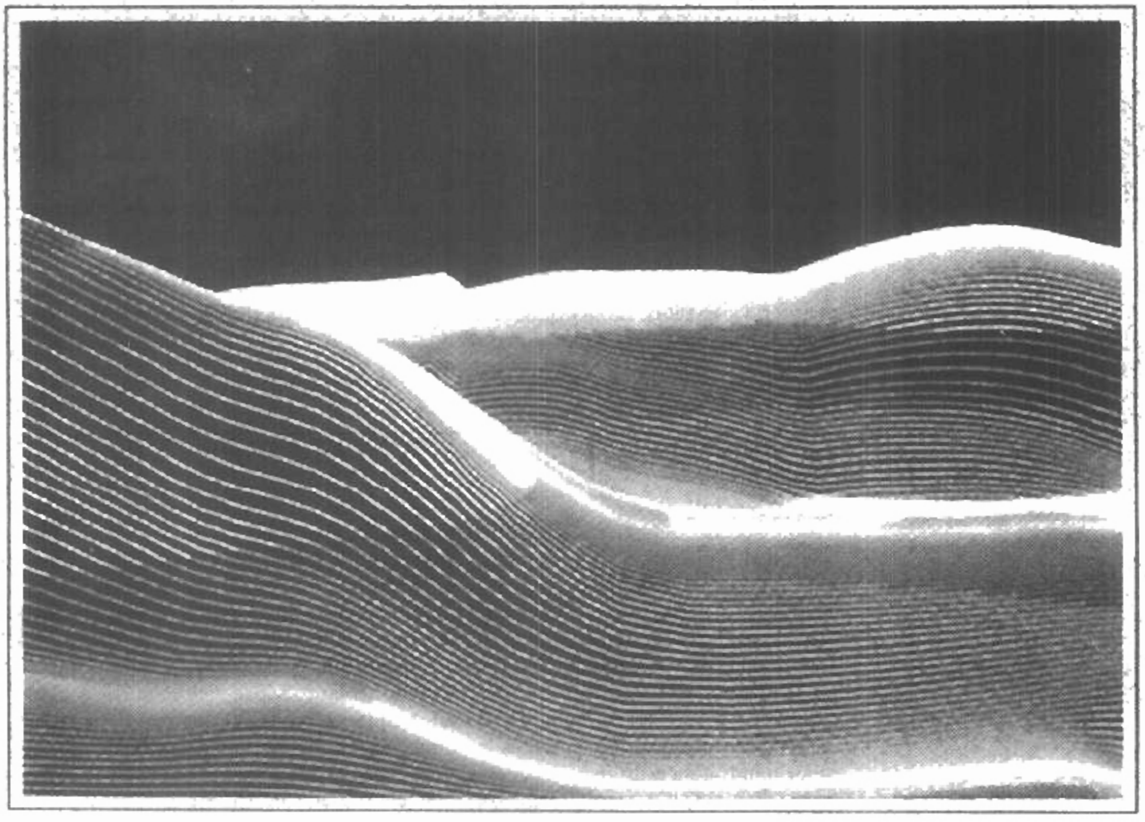
CHAPTER

13 Visible-Surface Detection Methods



A major consideration in the generation of realistic graphics displays is identifying those parts of a scene that are visible from a chosen viewing position. There are many approaches we can take to solve this problem, and numerous algorithms have been devised for efficient identification of visible objects for different types of applications. Some methods require more memory, some involve more processing time, and some apply only to special types of objects. Deciding upon a method for a particular application can depend on such factors as the complexity of the scene, type of objects to be displayed, available equipment, and whether static or animated displays are to be generated. The various algorithms are referred to as **visible-surface detection methods.** Sometimes these methods are also referred to as **hidden-surface elimination methods,** although there can be subtle differences between identifying visible surfaces and eliminating hidden surfaces. For wireframe displays, for example, we may not want to actually eliminate the hidden surfaces, but rather to display them with dashed boundaries or in some other way to retain information about their shape. In this chapter, we explore some of the most commonly used methods for detecting visible surfaces in a three-dimensional scene.

13-1 CLASSIFICATION OF VISIBLE-SURFACE DETECTION ALGORITHMS

Visible-surface detection algorithms are broadly classified according to whether they deal with object definitions directly or with their projected images. These two approaches are called **object-space methods** and **image-space methods,** respectively. An object-space method compares objects and parts of objects to each other within the scene definition to determine which surfaces, as a whole, we should label as visible. In an image-space algorithm, visibility is decided point by point at each pixel position on the projection plane. Most visible-surface algorithms use image-space methods, although object-space methods can be used effectively to locate visible surfaces in some cases. Line-display algorithms, on the other hand, generally use object-space methods to identify visible lines in wireframe displays, but many image-space visible-surface algorithms can be adapted easily to visible-line detection.

Although there are major differences in the basic approach taken by the various visible-surface detection algorithms, most use sorting and coherence methods to improve performance. Sorting is used to facilitate depth comparisons by ordering the individual surfaces in a scene according to their distance from the view plane. Coherence methods are used to take advantage of regularities in a scene. An individual scan line can be expected to contain intervals (runs) of constant pixel intensities, and scan-line patterns often change little from one line to the next. Animation frames contain changes only in the vicinity of moving objects. And constant relationships often can be established between objects and surfaces in a scene.

13-2 BACK-FACE DETECTION

A fast and simple object-space method for identifying the **back faces** of a polyhedron is based on the "inside-outside" tests discussed in Chapter 10. A point (x, y, z) is "inside" a polygon surface with plane parameters A, B, C, and D if

$$Ax + By + Cz + D < 0 \tag{13-1}$$

When an inside point is along the line of sight to the surface, the polygon must be a back face (we are inside that face and cannot see the front of it from our viewing position).

We can simplify this test by considering the normal vector **N** to a polygon surface, which has Cartesian components (A, B, C). In general, if **V** is a vector in the viewing direction from the eye (or "camera") position, as shown in Fig. 13-1, then this polygon is a back face if

$$\mathbf{V} \cdot \mathbf{N} > 0 \tag{13-2}$$

Furthermore, if object descriptions have been converted to projection coordinates and our viewing direction is parallel to the viewing z_v axis, then $\mathbf{V} = (0, 0, V_z)$ and

$$\mathbf{V} \cdot \mathbf{N} = V_z C$$

so that we only need to consider the sign of C, the z component of the normal vector **N**

In a right-handed viewing system with viewing direction along the negative z_v axis (Fig. 13-2), the polygon is a back face if $C < 0$. Also, we cannot see any face whose normal has z component $C = 0$, since our viewing direction is grazing that polygon. Thus, in general, we can label any polygon as a back face if its normal vector has a z-component value:

$$C \le 0 \tag{13-3}$$

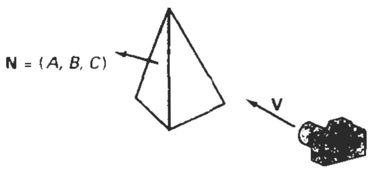


Figure 13-1
Vector **V** in the viewing direction and a back-face normal vector **N** of a polyhedron

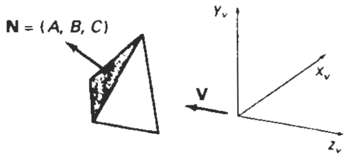


Figure 13-2
A polygon surface with plane parameter $C < 0$ in a right-handed viewing coordinate system is identified as a back face when the viewing direction is along the negative z_v axis.

Similar methods can be used in packages that employ a left-handed viewing system. In these packages, plane parameters A, B, C, and D can be calculated from polygon vertex coordinates specified in a clockwise direction (instead of the counterclockwise direction used in a right-handed system). Inequality 13-1 then remains a valid test for inside points. Also, back faces have normal vectors that point away from the viewing position and are identified by $C \geq 0$ when the viewing direction is along the positive z_v axis.

By examining parameter C for the different planes defining an object, we can immediately identify all the back faces. For a single convex polyhedron, such as the pyramid in Fig. 13-2, this test identifies all the hidden surfaces on the object, since each surface is either completely visible or completely hidden. Also, if a scene contains only nonoverlapping convex polyhedra, then again all hidden surfaces are identified with the back-face method.

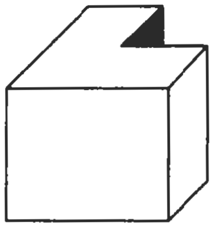


Figure 13-3
View of a concave polyhedron with one face partially hidden by other faces.

For other objects, such as the concave polyhedron in Fig. 13-3, more tests need to be carried out to determine whether there are additional faces that are totally or partly obscured by other faces. And a general scene can be expected to contain overlapping objects along the line of sight. We then need to determine where the obscured objects are partially or completely hidden by other objects. In general, back-face removal can be expected to eliminate about half of the polygon surfaces in a scene from further visibility tests.

13-3 DEPTH-BUFFER METHOD

A commonly used image-space approach to detecting visible surfaces is the **depth-buffer method,** which compares surface depths at each pixel position on the projection plane. This procedure is also referred to as the z-**buffer method,** since object depth is usually measured from the view plane along the z axis of a viewing system. Each surface of a scene is processed separately, one point at a time across the surface. The method is usually applied to scenes containing only polygon surfaces, because depth values can be computed very quickly and the method is easy to implement. But the method can be applied to nonplanar surfaces.

With object descriptions converted to projection coordinates, each (x, y, z) position on a polygon surface corresponds to the orthographic projection point (x, y) on the view plane. Therefore, for each pixel position (x, y) on the view plane, object depths can be compared by comparing z values. Figure 13-4 shows three surfaces at varying distances along the orthographic projection line from position (x, y) in a view plane taken as the $x_v y_v$ plane. Surface S_1 is closest at this position, so its surface intensity value at (x, y) is saved.

We can implement the depth-buffer algorithm in normalized coordinates, so that z values range from 0 at the back clipping plane to z_{max} at the front clip-

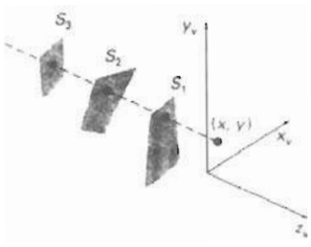


Figure 13-4
At view-plane position (x, y), surface S_1 has the smallest depth from the view plane and so is visible at that position.

ping plane. The value of z_{max} can be set either to 1 (for a unit cube) or to the largest value that can be stored on the system.

As implied by the name of this method, two buffer areas are required. A depth buffer is used to store depth values for each (x, y) position as surfaces are processed, and the refresh buffer stores the intensity values for each position. Initially, all positions in the depth buffer are set to 0 (minimum depth), and the refresh buffer is initialized to the background intensity. Each surface listed in the polygon tables is then processed, one scan line at a time, calculating the depth (z value) at each (x, y) pixel position. The calculated depth is compared to the value previously stored in the depth buffer at that position. If the calculated depth is greater than the value stored in the depth buffer, the new depth value is stored, and the surface intensity at that position is determined and placed in the same xy location in the refresh buffer.

We summarize the steps of a depth-buffer algorithm as follows:

1. Initialize the depth buffer and refresh buffer so that for all buffer positions (x, y),

$$\text{depth}(x, y) = 0, \qquad \text{refresh}(x, y) = I_{\text{backgnd}}$$

2. For each position on each polygon surface, compare depth values to previously stored values in the depth buffer to determine visibility.

    - Calculate the depth z for each (x, y) position on the polygon.
    - If $z > \text{depth}(x, y)$, then set

$$\text{depth}(x, y) = z, \qquad \text{refresh}(x, y) = I_{\text{surf}}(x,y)$$

where I_{backgnd} is the value for the background intensity, and $I_{\text{surf}}(x,y)$ is the projected intensity value for the surface at pixel position (x,y). After all surfaces have been processed, the depth buffer contains depth values for the visible surfaces and the refresh buffer contains the corresponding intensity values for those surfaces.

Depth values for a surface position (x, y) are calculated from the plane equation for each surface:

$$z = \frac{-Ax - By - D}{C} \tag{13-4}$$

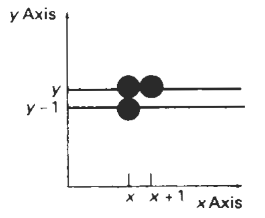


Figure 13-5
From position (x, y) on a scan line, the next position across the line has coordinates $(x + 1, y)$, and the position immediately below on the next line has coordinates $(x, y - 1)$.

For any scan line (Fig. 13-5), adjacent horizontal positions across the line differ by 1, and a vertical y value on an adjacent scan line differs by 1. If the depth of position (x, y) has been determined to be z, then the depth z' of the next position $(x + 1, y)$ along the scan line is obtained from Eq. 13-4 as

$$z' = \frac{-A(x+1) - By - D}{C} \tag{13-5}$$

or

$$z' = z - \frac{A}{C} \tag{13-6}$$

The ratio $-A/C$ is constant for each surface, so succeeding depth values across a scan line are obtained from preceding values with a single addition.

On each scan line, we start by calculating the depth on a left edge of the polygon that intersects that scan line (Fig. 13-6). Depth values at each successive position across the scan line are then calculated by Eq. 13-6.

We first determine the y-coordinate extents of each polygon, and process the surface from the topmost scan line to the bottom scan line, as shown in Fig. 13-6. Starting at a top vertex, we can recursively calculate x positions down a left edge of the polygon as $x' = x - 1/m$, where m is the slope of the edge (Fig. 13-7). Depth values down the edge are then obtained recursively as

$$z' = z + \frac{A/m + B}{C} \tag{13-7}$$

If we are processing down a vertical edge, the slope is infinite and the recursive calculations reduce to

$$z' = z + \frac{B}{C}$$

An alternate approach is to use a midpoint method or Bresenham-type algorithm for determining x values on left edges for each scan line. Also the method can be applied to curved surfaces by determining depth and intensity values at each surface projection point.

For polygon surfaces, the depth-buffer method is very easy to implement, and it requires no sorting of the surfaces in a scene. But it does require the availability of a second buffer in addition to the refresh buffer. A system with a resolu-

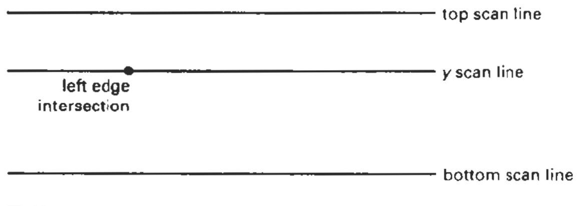


Figure 13-6
Scan lines intersecting a polygon surface.

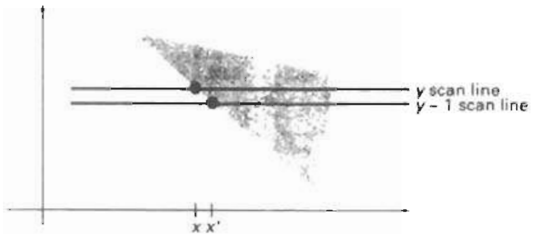


Figure 13-7
Intersection positions on successive scan lines along a left polygon edge.

tion of 1024 by 1024, for example, would require over a million positions in the depth buffer, with each position containing enough bits to represent the number of depth increments needed. One way to reduce storage requirements is to process one section of the scene at a time, using a smaller depth buffer. After each view section is processed, the buffer is reused for the next section.

13-4 A-BUFFER METHOD

An extension of the ideas in the depth-buffer method is the **A-buffer method** (at the other end of the alphabet from "z-buffer", where z represents depth). The A-buffer method represents an *antialiased, area-averaged, accumulation-buffer* method developed by Lucasfilm for implementation in the surface-rendering system called REYES (an acronym for "Renders Everything You Ever Saw").

A drawback of the depth-buffer method is that it can only find one visible surface at each pixel position. In other words, it deals only with opaque surfaces and cannot accumulate intensity values for more than one surface, as is necessary if transparent surfaces are to be displayed (Fig. 13-8). The A-buffer method expands the depth buffer so that each position in the buffer can reference a linked list of surfaces. Thus, more than one surface intensity can be taken into consideration at each pixel position, and object edges can be antialiased.

Each position in the A-buffer has two fields:

- depth field — stores a positive or negative real number
- intensity field — stores surface-intensity information or a pointer value.

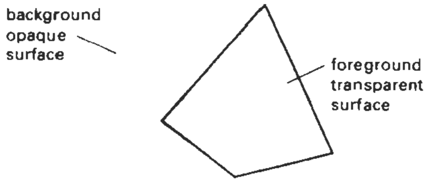


Figure 13-8
Viewing an opaque surface through a transparent surface requires multiple surface-intensity contributions for pixel positions.

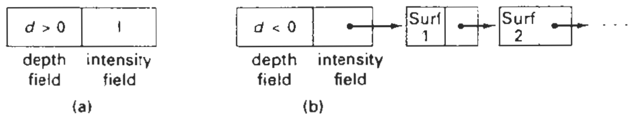


Figure 13-9
Organization of an A-buffer pixel position: (a) single-surface overlap of the corresponding pixel area, and (b) multiple-surface overlap.

If the depth field is positive, the number stored at that position is the depth of a single surface overlapping the corresponding pixel area. The intensity field then stores the RGB components of the surface color at that point and the percent of pixel coverage, as illustrated in Fig. 13-9(a).

If the depth field is negative, this indicates multiple-surface contributions to the pixel intensity. The intensity field then stores a pointer to a linked list of surface data, as in Fig. 13-9(b). Data for each surface in the linked list includes

- RGB intensity components
- opacity parameter (percent of transparency)
- depth
- percent of area coverage
- surface identifier
- other surface-rendering parameters
- pointer to next surface

The A-buffer can be constructed using methods similar to those in the depth-buffer algorithm. Scan lines are processed to determine surface overlaps of pixels across the individual scanlines. Surfaces are subdivided into a polygon mesh and clipped against the pixel boundaries. Using the opacity factors and percent of surface overlaps, we can calculate the intensity of each pixel as an average of the contributions from the overlapping surfaces.

13-5 SCAN-LINE METHOD

This image-space method for removing hidden surfaces is an extension of the scan-line algorithm for filling polygon interiors. Instead of filling just one surface, we now deal with multiple surfaces. As each scan line is processed, all polygon surfaces intersecting that line are examined to determine which are visible. Across each scan line, depth calculations are made for each overlapping surface to determine which is nearest to the view plane. When the visible surface has been determined, the intensity value for that position is entered into the refresh buffer.

We assume that tables are set up for the various surfaces, as discussed in Chapter 10, which include both an edge table and a polygon table. The edge table contains coordinate endpoints for each line in the scene, the inverse slope of each line, and pointers into the polygon table to identify the surfaces bounded by each line. The polygon table contains coefficients of the plane equation for each surface, intensity information for the surfaces, and possibly pointers into the edge table. To facilitate the search for surfaces crossing a given scan line, we can set up an active list of edges from information in the edge table. This active list will contain only edges that cross the current scan line, sorted in order of increasing x. In addition, we define a flag for each surface that is set on or off to indicate whether a position along a scan line is inside or outside of the surface. Scan lines are processed from left to right. At the leftmost boundary of a surface, the surface flag is turned on; and at the rightmost boundary, it is turned off.

Figure 13-10 illustrates the scan-line method for locating visible portions of surfaces for pixel positions along the line. The active list for scan line 1 contains information from the edge table for edges AB, BC, EH, and FG. For positions along this scan line between edges AB and BC, only the flag for surface S_1 is on. Therefore, no depth calculations are necessary, and intensity information for surface S_1 is entered from the polygon table into the refresh buffer. Similarly, between edges EH and FG, only the flag for surface S_2 is on. No other positions along scan line 1 intersect surfaces, so the intensity values in the other areas are set to the background intensity. The background intensity can be loaded throughout the buffer in an initialization routine.

For scan lines 2 and 3 in Fig. 13-10, the active edge list contains edges AD, EH, BC, and FG. Along scan line 2 from edge AD to edge EH, only the flag for surface S_1 is on. But between edges EH and BC, the flags for both surfaces are on. In this interval, depth calculations must be made using the plane coefficients for the two surfaces. For this example, the depth of surface S_1 is assumed to be less than that of S_2, so intensities for surface S_1 are loaded into the refresh buffer until boundary BC is encountered. Then the flag for surface S_1 goes off, and intensities for surface S_2 are stored until edge FG is passed.

We can take advantage of coherence along the scan lines as we pass from one scan line to the next. In Fig. 13-10, scan line 3 has the same active list of edges as scan line 2. Since no changes have occurred in line intersections, it is unnecessary again to make depth calculations between edges EH and BC. The two sur-

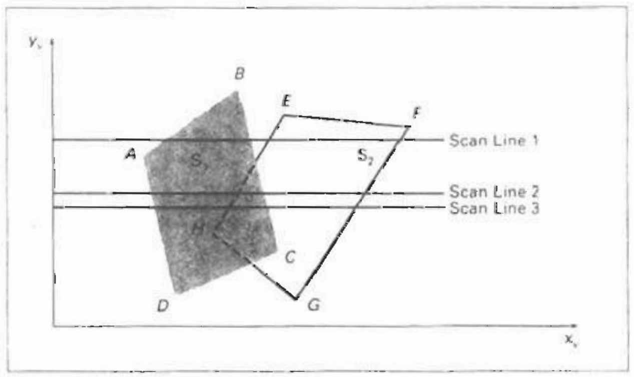


Figure 13-10
Scan lines crossing the projection of two surfaces, S_1 and S_2, in the view plane. Dashed lines indicate the boundaries of hidden surfaces.

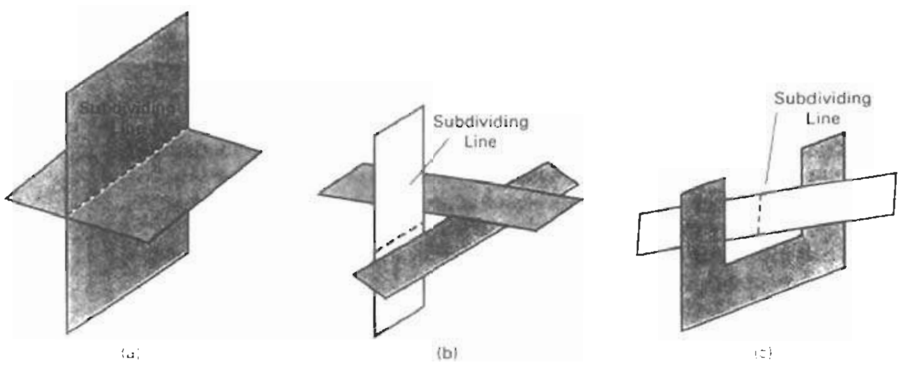


Figure 13-11
Intersecting and cyclically overlapping surfaces that alternately obscure one another.

faces must be in the same orientation as determined on scan line 2, so the intensities for surface S_1 can be entered without further calculations.

Any number of overlapping polygon surfaces can be processed with this scan-line method. Flags for the surfaces are set to indicate whether a position is inside or outside, and depth calculations are performed when surfaces overlap. When these coherence methods are used, we need to be careful to keep track of which surface section is visible on each scan line. This works only if surfaces do not cut through or otherwise cyclically overlap each other (Fig. 13-11). If any kind of cyclic overlap is present in a scene, we can divide the surfaces to eliminate the overlaps. The dashed lines in this figure indicate where planes could be subdivided to form two distinct surfaces, so that the cyclic overlaps are eliminated.

13-6 DEPTH-SORTING METHOD

Using both image-space and object-space operations, the **depth-sorting method** performs the following basic functions:

1. Surfaces are sorted in order of decreasing depth.
2. Surfaces are scan converted in order, starting with the surface of greatest depth.

Sorting operations are carried out in both image and object space, and the scan conversion of the polygon surfaces is performed in image space.

This method for solving the hidden-surface problem is often referred to as the **painter's algorithm**. In creating an oil painting, an artist first paints the background colors. Next, the most distant objects are added, then the nearer objects, and so forth. At the final step, the foreground objects are painted on the canvas over the background and other objects that have been painted on the canvas.

Each layer of paint covers up the previous layer. Using a similar technique, we first sort surfaces according to their distance from the view plane. The intensity values for the farthest surface are then entered into the refresh buffer. Taking each succeeding surface in turn (in decreasing depth order), we "paint" the surface intensities onto the frame buffer over the intensities of the previously processed surfaces.

Painting polygon surfaces onto the frame buffer according to depth is carried out in several steps. Assuming we are viewing along the $-z$ direction, surfaces are ordered on the first pass according to the smallest z value on each surface. Surface S with the greatest depth is then compared to the other surfaces in the list to determine whether there are any overlaps in depth. If no depth overlaps occur, S is scan converted. Figure 13-12 shows two surfaces that overlap in the xy plane but have no depth overlap. This process is then repeated for the next surface in the list. As long as no overlaps occur, each surface is processed in depth order until all have been scan converted. If a depth overlap is detected at any point in the list, we need to make some additional comparisons to determine whether any of the surfaces should be reordered.

We make the following tests for each surface that overlaps with S. If any one of these tests is true, no reordering is necessary for that surface. The tests are listed in order of increasing difficulty.

1. The bounding rectangles in the xy plane for the two surfaces do not overlap.
2. Surface S is completely behind the overlapping surface relative to the viewing position.
3. The overlapping surface is completely in front of S relative to the viewing position.
4. The projections of the two surfaces onto the view plane do not overlap.

We perform these tests in the order listed and proceed to the next overlapping surface as soon as we find one of the tests is true. If all the overlapping surfaces pass at least one of these tests, none of them is behind S. No reordering is then necessary and S is scan converted.

Test 1 is performed in two parts. We first check for overlap in the x direction, then we check for overlap in the y direction. If either of these directions show no overlap, the two planes cannot obscure one other. An example of two

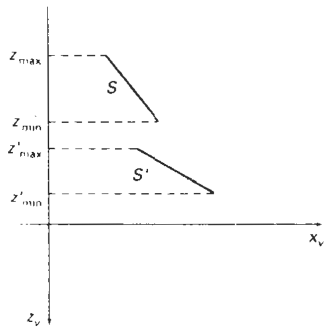


Figure 13-12
Two surfaces with no depth overlap.

surfaces that overlap in the z direction but not in the x direction is shown in Fig. 13-13.

We can perform tests 2 and 3 with an "inside-outside" polygon test. That is, we substitute the coordinates for all vertices of S into the plane equation for the overlapping surface and check the sign of the result. If the plane equations are set up so that the outside of the surface is toward the viewing position, then S is behind S' if all vertices of S are "inside" S' (Fig. 13-14). Similarly, S' is completely in front of S if all vertices of S are "outside" of S'. Figure 13-15 shows an overlapping surface S' that is completely in front of S, but surface S is not completely "inside" S' (test 2 is not true).

If tests 1 through 3 have all failed, we try test 4 by checking for intersections between the bounding edges of the two surfaces using line equations in the xy plane. As demonstrated in Fig. 13-16, two surfaces may or may not intersect even though their coordinate extents overlap in the x, y, and z directions.

Should all four tests fail with a particular overlapping surface S', we interchange surfaces S and S' in the sorted list. An example of two surfaces that

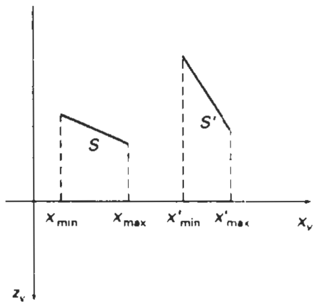


Figure 13-13
Two surfaces with depth overlap but no overlap in the x direction.

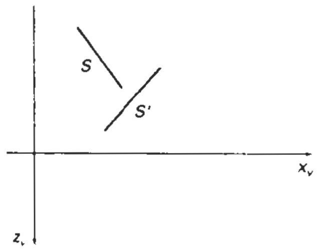


Figure 13-14
Surface S is completely behind ("inside") the overlapping surface S'.

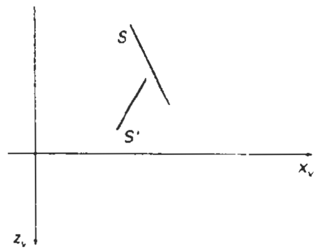


Figure 13-15
Overlapping surface S' is completely in front ("outside") of surface S, but S is not completely behind S'

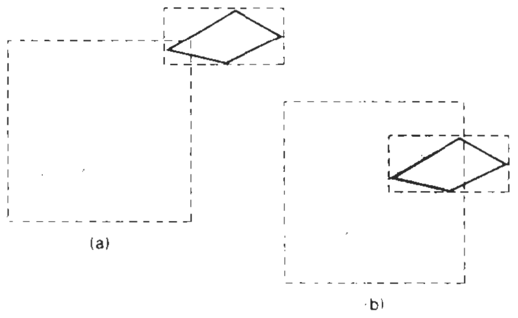


Figure 13-16
Two surfaces with overlapping bounding rectangles in the *xy* plane.

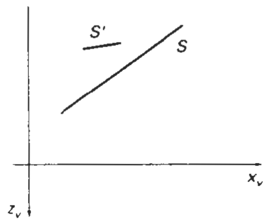


Figure 13-17
Surface *S* has greater depth but obscures surface *S'*.

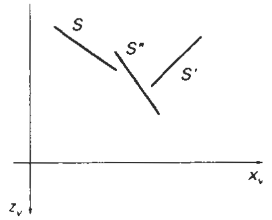


Figure 13-18
Three surfaces entered into the sorted surface list in the order *S*, *S'*, *S''* should be reordered *S'*, *S''*, *S*.

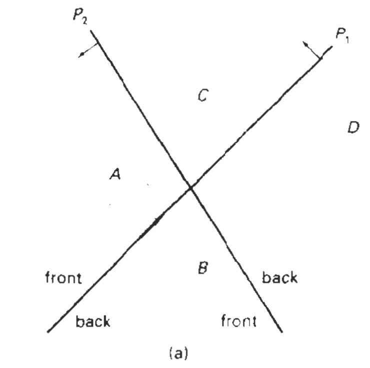
would be reordered with this procedure is given in Fig. 13-17. At this point, we still do not know for certain that we have found the farthest surface from the view plane. Figure 13-18 illustrates a situation in which we would first interchange *S* and *S''*. But since *S''* obscures part of *S'*, we need to interchange *S''* and *S'* to get the three surfaces into the correct depth order. Therefore, we need to repeat the testing process for each surface that is reordered in the list.

It is possible for the algorithm just outlined to get into an infinite loop if two or more surfaces alternately obscure each other, as in Fig. 13-11. In such situations, the algorithm would continually reshuffle the positions of the overlapping surfaces. To avoid such loops, we can flag any surface that has been reordered to a farther depth position so that it cannot be moved again. If an attempt is made to switch the surface a second time, we divide it into two parts to eliminate the cyclic overlap. The original surface is then replaced by the two new surfaces, and we continue processing as before.

13-7 BSP-TREE METHOD

A **binary space-partitioning (BSP)** tree is an efficient method for determining object visibility by painting surfaces onto the screen from back to front, as in the painter's algorithm. The BSP tree is particularly useful when the view reference point changes, but the objects in a scene are at fixed positions.

Applying a BSP tree to visibility testing involves identifying surfaces that are "inside" and "outside" the partitioning plane at each step of the space subdivision, relative to the viewing direction. Figure 13-19 illustrates the basic concept in this algorithm. With plane P_1, we first partition the space into two sets of objects. One set of objects is behind, or in back of, plane P_1 relative to the viewing direction, and the other set is in front of P_1. Since one object is intersected by plane P_1, we divide that object into two separate objects, labeled *A* and *B*. Objects *A* and *C* are in front of P_1, and objects *B* and *D* are behind P_1. We next partition the space again with plane P_2 and construct the binary tree representation shown in Fig. 13-19(b). In this tree, the objects are represented as terminal nodes, with front objects as left branches and back objects as right branches.



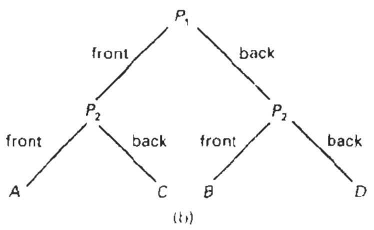


Figure 13-19
A region of space (a) is partitioned with two planes P_1 and P_2 to form the BSP tree representation in (b).

For objects described with polygon facets, we chose the partitioning planes to coincide with the polygon planes. The polygon equations are then used to identify "inside" and "outside" polygons, and the tree is constructed with one partitioning plane for each polygon face. Any polygon intersected by a partitioning plane is split into two parts. When the BSP tree is complete, we process the tree by selecting the surfaces for display in the order back to front, so that foreground objects are painted over the background objects. Fast hardware implementations for constructing and processing BSP trees are used in some systems.

13-8 AREA-SUBDIVISION METHOD

This technique for hidden-surface removal is essentially an image-space method, but object-space operations can be used to accomplish depth ordering of surfaces. The **area-subdivision method** takes advantage of area coherence in a scene by locating those view areas that represent part of a single surface. We apply this method by successively dividing the total viewing area into smaller and smaller rectangles until each small area is the projection of part of a single visible surface or no surface at all.

To implement this method, we need to establish tests tnat can quickly identify the area as part of a single surface or tell us that the area is too complex to analyze easily. Starting with the total view, we apply the tests to determine whether we should subdivide the total area into smaller rectangles. If the tests indicate that the view is sufficiently complex, we subdivide it. Next, we apply the tests to each of the smaller areas, subdividing these if the tests indicate that visibility of a single surface is still uncertain. We continue this process until the subdivisions are easily analyzed as belonging to a single surface or until they are reduced to the size of a single pixel. An easy way to do this is to successively divide the area into four equal parts at each step, as shown in Fig. 13-20. This approach is similar to that used in constructing a quadtree. A viewing area with a resolution of 1024 by 1024 could be subdivided ten times in this way before a subarea is reduced to a point.

Tests to determine the visibility of a single surface within a specified area are made by comparing surfaces to the boundary of the area. There are four possible relationships that a surface can have with a specified area boundary. We can describe these relative surface characteristics in the following way (Fig. 13-21):

Surrounding surface—One that completely encloses the area.
Overlapping surface—One that is partly inside and partly outside the area.
Inside surface—One that is completely inside the area.
Outside surface—One that is completely outside the area.

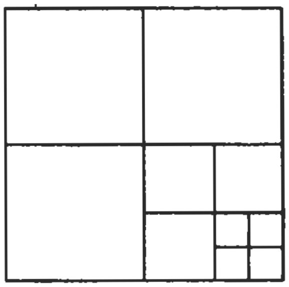


Figure 13-20
Dividing a square area into equal-sized quadrants at each step.

The tests for determining surface visibility within an area can be stated in terms of these four classifications. No further subdivisions of a specified area are needed if one of the following conditions is true:

1. All surfaces are outside surfaces with respect to the area.
2. Only one inside, overlapping, or surrounding surface is in the area.
3. A surrounding surface obscures all other surfaces within the area boundaries.

Test 1 can be carried out by checking the bounding rectangles of all surfaces against the area boundaries. Test 2 can also use the bounding rectangles in the *xy* plane to identify an inside surface. For other types of surfaces, the bounding rectangles can be used as an initial check. If a single bounding rectangle intersects the area in some way, additional checks are used to determine whether the surface is surrounding, overlapping, or outside. Once a single inside, overlapping, or surrounding surface has been identified, its pixel intensities are transferred to the appropriate area within the frame buffer.

One method for implementing test 3 is to order surfaces according to their minimum depth from the view plane. For each surrounding surface, we then compute the maximum depth within the area under consideration. If the maxi-

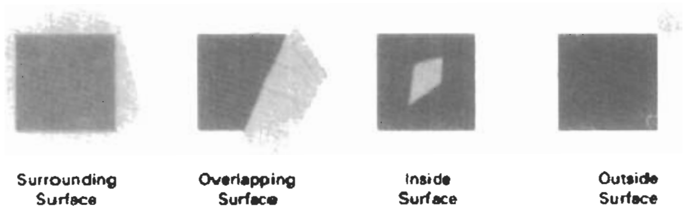


Figure 13-21
Possible relationships between polygon surfaces and a rectangular area.

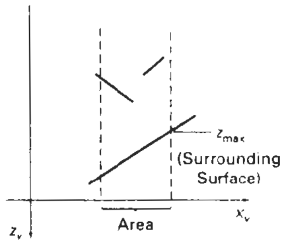


Figure 13-22
Within a specified area, a surrounding surface with a maximum depth of z_{max} obscures all surfaces that have a minimum depth beyond z_{max}.

mum depth of one of these surrounding surfaces is closer to the view plane than the minimum depth of all other surfaces within the area, test 3 is satisfied. Figure 13-22 shows an example of the conditions for this method.

Another method for carrying out test 3 that does not require depth sorting is to use plane equations to calculate depth values at the four vertices of the area for all surrounding, overlapping, and inside surfaces. If the calculated depths for one of the surrounding surfaces is less than the calculated depths for all other surfaces, test 3 is true. Then the area can be filled with the intensity values of the surrounding surface.

For some situations, both methods of implementing test 3 will fail to identify correctly a surrounding surface that obscures all the other surfaces. Further testing could be carried out to identify the single surface that covers the area, but it is faster to subdivide the area than to continue with more complex testing. Once outside and surrounding surfaces have been identified for an area, they will remain outside and surrounding surfaces for all subdivisions of the area. Furthermore, some inside and overlapping surfaces can be expected to be eliminated as the subdivision process continues, so that the areas become easier to analyze. In the limiting case, when a subdivision the size of a pixel is produced, we simply calculate the depth of each relevant surface at that point and transfer the intensity of the nearest surface to the frame buffer.

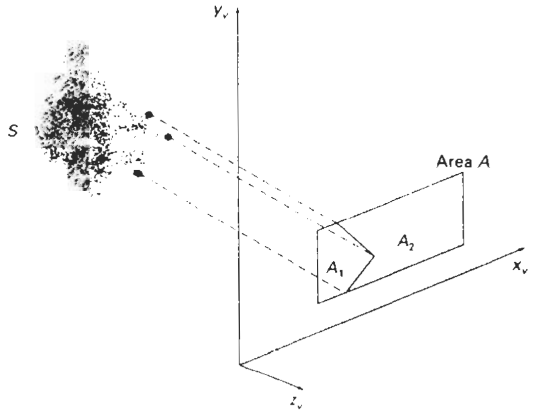


Figure 13-23
Area A is subdivided into A_1 and A_2 using the boundary of surface S on the view plane.

As a variation on the basic subdivision process, we could subdivide areas along surface boundaries instead of dividing them in half. If the surfaces have been sorted according to minimum depth, we can use the surface with the smallest depth value to subdivide a given area. Figure 13-23 illustrates this method for subdividing areas. The projection of the boundary of surface S is used to partition the original area into the subdivisions A_1 and A_2. Surface S is then a surrounding surface for A_1 and visibility tests 2 and 3 can be applied to determine whether further subdividing is necessary. In general, fewer subdivisions are required using this approach, but more processing is needed to subdivide areas and to analyze the relation of surfaces to the subdivision boundaries.

13-9 OCTREE METHODS

When an octree representation is used for the viewing volume, hidden-surface elimination is accomplished by projecting octree nodes onto the viewing surface in a front-to-back order. In Fig. 13-24, the front face of a region of space (the side toward the viewer) is formed with octants 0, 1, 2, and 3. Surfaces in the front of these octants are visible to the viewer. Any surfaces toward the rear of the front octants or in the back octants (4, 5, 6, and 7) may be hidden by the front surfaces.

Back surfaces are eliminated, for the viewing direction given in Fig. 13-24, by processing data elements in the octree nodes in the order 0, 1, 2, 3, 4, 5, 6, 7. This results in a depth-first traversal of the octree, so that nodes representing octants 0, 1, 2, and 3 for the entire region are visited before the nodes representing octants 4, 5, 6, and 7. Similarly, the nodes for the front four suboctants of octant 0 are visited before the nodes for the four back suboctants. The traversal of the octree continues in this order for each octant subdivision.

When a color value is encountered in an octree node, the pixel area in the frame buffer corresponding to this node is assigned that color value only if no values have previously been stored in this area. In this way, only the front colors are loaded into the buffer. Nothing is loaded if an area is void. Any node that is found to be completely obscured is eliminated from further processing, so that its subtrees are not accessed.

Different views of objects represented as octrees can be obtained by applying transformations to the octree representation that reorient the object according

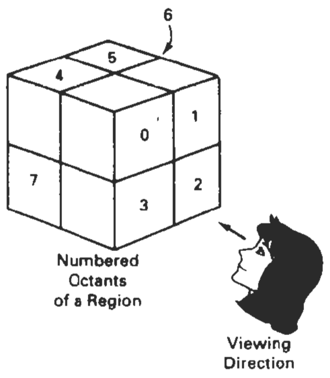
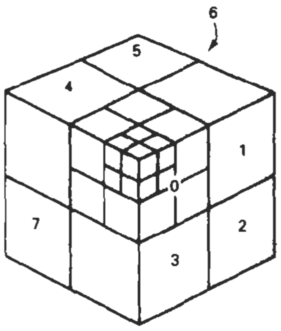


Figure 13-24
Objects in octants 0, 1, 2, and 3 obscure objects in the back octants (4, 5, 6, 7) when the viewing direction is as shown.

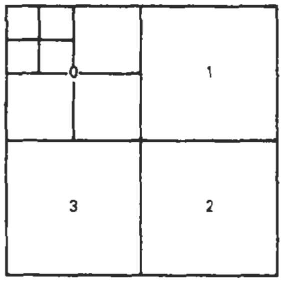
to the view selected. We assume that the octree representation is always set up so that octants 0, 1, 2, and 3 of a region form the front face, as in Fig. 13-24.

A method for displaying an octree is first to map the octree onto a quadtree of visible areas by traversing octree nodes from front to back in a recursive procedure. Then the quadtree representation for the visible surfaces is loaded into the frame buffer. Figure 13-25 depicts the octants in a region of space and the corresponding quadrants on the view plane. Contributions to quadrant 0 come from octants 0 and 4. Color values in quadrant 1 are obtained from surfaces in octants 1 and 5, and values in each of the other two quadrants are generated from the pair of octants aligned with each of these quadrants.

Recursive processing of octree nodes is demonstrated in the following procedure, which accepts an octree description and creates the quadtree representation for visible surfaces in the region. In most cases, both a front and a back octant must be considered in determining the correct color values for a quadrant. But if the front octant is homogeneously filled with some color, we do not process the back octant. For heterogeneous regions, the procedure is recursively called, passing as new arguments the child of the heterogeneous octant and a newly created quadtree node. If the front is empty, the rear octant is processed. Otherwise, two recursive calls are made, one for the rear octant and one for the front octant.



Octants in Space



Quadrants for the View Plane

Figure 13-25
Octant divisions for a region of space and the corresponding quadrant plane.

```
typedef enum { SOLID, MIXED } Status;

#define EMPTY -1

typedef struct tOctree {
  int id;
  Status status;
  union {
    int color;
    struct tOctree * children[8];
  } data;
} Octree;

typedef struct tQuadtree {
  int id;
  Status status;
  union {
    int color;
    struct tQuadtree * children[4];
  } data;
} Quadtree;

int nQuadtree = 0,

void octreeToQuadtree (Octree * oTree, Quadtree * qTree)
{
  Octree * front, * back;
  Quadtree * newQuadtree;
  int i, j;

  if (oTree->status == SOLID) {
    qTree->status = SOLID;
    qTree->data.color = oTree->data.color;
    return;
  }
  qTree->status = MIXED;
  /* Fill in each quad of the quadtree */
  for (i=0; i<4; i++) {
    front = oTree->data.children[i];
```

```
      back = oTree->data.children[i+4];
      newQuadtree = (Quadtree *) malloc (sizeof (Quadtree));
      newQuadtree->id = nQuadtree++;
      newQuadtree->status = SOLID;
      qTree->data.children[i] = newQuadtree;

      if (front->status == SOLID)
        if (front->data.color != EMPTY)
          qTree->data.children[i]->data.color = front->data.color;
        else
          if (back->status == SOLID)
            if (back->data.color != EMPTY)
              qTree->data.children[i]->data.color = back->data.color;
            else
              qTree->data.children[i]->data.color = EMPTY;
          else { /* back node is mixed */
            newQuadtree->status = MIXED;
            octreeToQuadtree (back, newQuadtree);
          }
      else { /* front node is mixed */
        newQuadtree->status = MIXED;
        octreeToQuadtree (back, newQuadtree);
        octreeToQuadtree (front, newQuadtree);
      }
    }
}
```

13-10 RAY-CASTING METHOD

If we consider the line of sight from a pixel position on the view plane through a scene, as in Fig. 13-26, we can determine which objects in the scene (if any) intersect this line. After calculating all ray–surface intersections, we identify the visible surface as the one whose intersection point is closest to the pixel. This visibility-detection scheme uses *ray-casting procedures* that were introduced in Section 10-15. Ray casting, as a visibility-detection tool, is based on geometric optics methods, which trace the paths of light rays. Since there are an infinite number of light rays in a scene and we are interested only in those rays that pass through

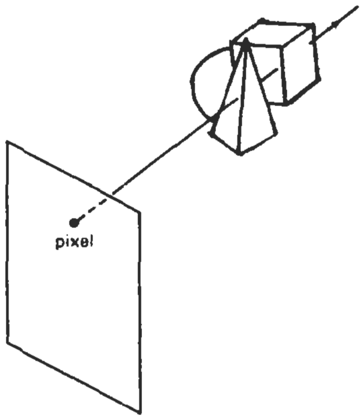


Figure 13-26
A ray along the line of sight from a pixel position through a scene.

pixel positions, we can trace the light-ray paths backward from the pixels through the scene. The ray-casting approach is an effective visibility-detection method for scenes with curved surfaces, particularly spheres.

We can think of ray casting as a variation on the depth-buffer method (Section 13-3). In the depth-buffer algorithm, we process surfaces one at a time and calculate depth values for all projection points over the surface. The calculated surface depths are then compared to previously stored depths to determine visible surfaces at each pixel. In ray-casting, we process pixels one at a time and calculate depths for all surfaces along the projection path to that pixel.

Ray casting is a special case of *ray-tracing algorithms* (Section 14-6) that trace multiple ray paths to pick up global reflection and refraction contributions from multiple objects in a scene. With ray casting, we only follow a ray out from each pixel to the nearest object. Efficient ray–surface intersection calculations have been developed for common objects, particularly spheres, and we discuss these intersection methods in detail in Chapter 14.

13-11 CURVED SURFACES

Effective methods for determining visibility for objects with curved surfaces include ray-casting and octree methods. With ray casting, we calculate ray–surface intersections and locate the smallest intersection distance along the pixel ray. With octrees, once the representation has been established from the input definition of the objects, all visible surfaces are identified with the same processing procedures. No special considerations need be given to different kinds of curved surfaces.

We can also approximate a curved surface as a set of plane, polygon surfaces. In the list of surfaces, we then replace each curved surface with a polygon mesh and use one of the other hidden-surface methods previously discussed. With some objects, such as spheres, it can be more efficient as well as more accurate to use ray casting and the curved-surface equation.

Curved-Surface Representations

We can represent a surface with an implicit equation of the form $f(x, y, z) = 0$ or with a parametric representation (Appendix A). Spline surfaces, for instance, are normally described with parametric equations. In some cases, it is useful to obtain an explicit surface equation, as, for example, a height function over an xy ground plane:

$$z = f(x, y)$$

Many objects of interest, such as spheres, ellipsoids, cylinders, and cones, have quadratic representations. These surfaces are commonly used to model molecular structures, roller bearings, rings, and shafts.

Scan-line and ray-casting algorithms often involve numerical approximation techniques to solve the surface equation at the intersection point with a scan line or with a pixel ray. Various techniques, including parallel calculations and fast hardware implementations, have been developed for solving the curved-surface equations for commonly used objects.

Surface Contour Plots

For many applications in mathematics, physical sciences, engineering and other fields, it is useful to display a surface function with a set of contour lines that show the surface shape. The surface may be described with an equation or with data tables, such as topographic data on elevations or population density. With an explicit functional representation, we can plot the visible-surface contour lines and eliminate those contour sections that are hidden by the visible parts of the surface.

To obtain an xy plot of a functional surface, we write the surface representation in the form

$$y = f(x, z) \tag{13-8}$$

A curve in the xy plane can then be plotted for values of z within some selected range, using a specified interval Δz. Starting with the largest value of z, we plot the curves from "front" to "back" and eliminate hidden sections. We draw the curve sections on the screen by mapping an xy range for the function into an xy pixel screen range. Then, unit steps are taken in x and the corresponding y value for each x value is determined from Eq. 13-8 for a given value of z.

One way to identify the visible curve sections on the surface is to maintain a list of $y_{\min}$ and $y_{\max}$ values previously calculated for the pixel x coordinates on the screen. As we step from one pixel x position to the next, we check the calculated y value against the stored range, $y_{\min}$ and $y_{\max}$, for the next pixel. If $y_{\min} \leq y \leq y_{\max}$, that point on the surface is not visible and we do not plot it. But if the calculated y value is outside the stored y bounds for that pixel, the point is visible. We then plot the point and reset the bounds for that pixel. Similar procedures can be used to project the contour plot onto the xz or the yz plane. Figure 13-27 shows an example of a surface contour plot with color-coded contour lines.

Similar methods can be used with a discrete set of data points by determining isosurface lines. For example, if we have a discrete set of z values for an n_x by n_y grid of xy values, we can determine the path of a line of constant z over the surface using the contour methods discussed in Section 10-21. Each selected contour line can then be projected onto a view plane and displayed with straight-line

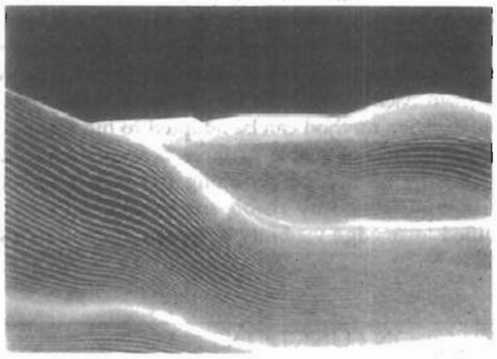


Figure 13-27
A color-coded surface contour plot. (*Courtesy of Los Alamos National Laboratory.*)

segments. Again, lines can be drawn on the display device in a front-to-back depth order, and we eliminate contour sections that pass behind previously drawn (visible) contour lines.

13-12 WIREFRAME METHODS

When only the outline of an object is to be displayed, visibility tests are applied to surface edges. Visible edge sections are displayed, and hidden edge sections can either be eliminated or displayed differently from the visible edges. For example, hidden edges could be drawn as dashed lines, or we could use depth cueing to decrease the intensity of the lines as a linear function of distance from the view plane. Procedures for determining visibility of object edges are referred to as **wireframe-visibility methods.** They are also called **visible-line detection methods** or **hidden-line detection methods.** Special wireframe-visibility procedures have been developed, but some of the visible-surface methods discussed in preceding sections can also be used to test for edge visibility.

A direct approach to identifying the visible lines in a scene is to compare each line to each surface. The process involved here is similar to clipping lines against arbitrary window shapes, except that we now want to determine which sections of the lines are hidden by surfaces. For each line, depth values are compared to the surfaces to determine which line sections are not visible. We can use coherence methods to identify hidden line segments without actually testing each coordinate position. If both line intersections with the projection of a surface boundary have greater depth than the surface at those points, the line segment between the intersections is completely hidden, as in Fig. 13-28(a). This is the usual situation in a scene, but it is also possible to have lines and surfaces intersecting each other. When a line has greater depth at one boundary intersection and less depth than the surface at the other boundary intersection, the line must penetrate the surface interior, as in Fig. 13-28(b). In this case, we calculate the intersection point of the line with the surface using the plane equation and display only the visible sections.

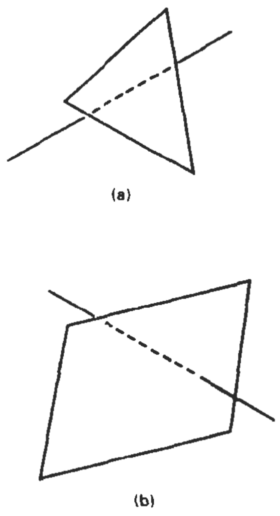


Figure 13-28
Hidden-line sections (dashed) for a line that (a) passes behind a surface and (b) penetrates a surface.

Some visible-surface methods are readily adapted to wireframe visibility testing. Using a back-face method, we could identify all the back surfaces of an object and display only the boundaries for the visible surfaces. With depth sorting, surfaces can be painted into the refresh buffer so that surface interiors are in the background color, while boundaries are in the foreground color. By processing the surfaces from back to front, hidden lines are erased by the nearer surfaces. An area-subdivision method can be adapted to hidden-line removal by displaying only the boundaries of visible surfaces. Scan-line methods can be used to display visible lines by setting points along the scan line that coincide with boundaries of visible surfaces. Any visible-surface method that uses scan conversion can be modified to an edge-visibility detection method in a similar way.

13-13 VISIBILITY-DETECTION FUNCTIONS

Often, three-dimensional graphics packages accommodate several visible-surface detection procedures, particularly the back-face and depth-buffer methods. A particular function can then be invoked with the procedure name, such as `backFace` or `depthBuffer`.

In general programming standards, such as GKS and PHIGS, visibility methods are implementation-dependent. A table of available methods is listed at each installation, and a particular visibility-detection method is selected with the hidden-line-hidden-surface-removal (HLHSR) function:

```
setHLHSRidentifier (visibilityFunctionIndex)
```

Parameter `visibilityFunctionIndex` is assigned an integer code to identify the visibility method that is to be applied to subsequently specified output primitives.

SUMMARY

Here, we give a summary of the visibility-detection methods discussed in this chapter and a comparison of their effectiveness. Back-face detection is fast and effective as an initial screening to eliminate many polygons from further visibility tests. For a single convex polyhedron, back-face detection eliminates all hidden surfaces, but in general, back-face detection cannot completely identify all hidden surfaces. Other, more involved, visibility-detection schemes will correctly produce a list of visible surfaces.

A fast and simple technique for identifying visible surfaces is the depth-buffer (or z-buffer) method. This procedure requires two buffers, one for the pixel intensities and one for the depth of the visible surface for each pixel in the view plane. Fast incremental methods are used to scan each surface in a scene to calculate surface depths. As each surface is processed, the two buffers are updated. An improvement on the depth-buffer approach is the A-buffer, which provides additional information for displaying antialiased and transparent surfaces. Other visible-surface detection schemes include the scan-line method, the depth-sorting method (painter's algorithm), the BSP-tree method, area subdivision, octree methods, and ray casting.

Visibility-detection methods are also used in displaying three-dimensional line drawings. With curved surfaces, we can display contour plots. For wireframe displays of polyhedrons, we search for the various edge sections of the surfaces in a scene that are visible from the view plane.

The effectiveness of a visible-surface detection method depends on the characteristics of a particular application. If the surfaces in a scene are spread out in the z direction so that there is very little depth overlap, a depth-sorting or BSP-tree method is often the best choice. For scenes with surfaces fairly well separated horizontally, a scan-line or area-subdivision method can be used efficiently to locate visible surfaces.

As a general rule, the depth-sorting or BSP-tree method is a highly effective approach for scenes with only a few surfaces. This is because these scenes usually have few surfaces that overlap in depth. The scan-line method also performs well when a scene contains a small number of surfaces. Either the scan-line, depth-sorting, or BSP-tree method can be used effectively for scenes with up to several thousand polygon surfaces. With scenes that contain more than a few thousand surfaces, the depth-buffer method or octree approach performs best. The depth-buffer method has a nearly constant processing time, independent of the number of surfaces in a scene. This is because the size of the surface areas decreases as the number of surfaces in the scene increases. Therefore, the depth-buffer method exhibits relatively low performance with simple scenes and relatively high performance with complex scenes. BSP trees are useful when multiple views are to be generated using different view reference points.

When octree representations are used in a system, the hidden-surface elimination process is fast and simple. Only integer additions and subtractions are used in the process, and there is no need to perform sorting or intersection calculations. Another advantage of octrees is that they store more than surfaces. The entire solid region of an object is available for display, which makes the octree representation useful for obtaining cross-sectional slices of solids.

If a scene contains curved-surface representations, we use octree or ray-casting methods to identify visible parts of the scene. Ray-casting methods are an integral part of ray-tracing algorithms, which allow scenes to be displayed with global-illumination effects.

It is possible to combine and implement the different visible-surface detection methods in various ways. In addition, visibility-detection algorithms are often implemented in hardware, and special systems utilizing parallel processing are employed to increase the efficiency of these methods. Special hardware systems are used when processing speed is an especially important consideration, as in the generation of animated views for flight simulators.

REFERENCES

Additional sources of information on visibility algorithms include Elber and Cohen (1990), Franklin and Kankanhalli (1990), Glassner (1990), Naylor, Amanatides, and Thibault (1990), and Segal (1990).

EXERCISES

13-1. Develop a procedure, based on a back-face detection technique, for identifying all the visible faces of a convex polyhedron that has different-colored surfaces. Assume that the object is defined in a right-handed viewing system with the *xy*-plane as the viewing surface.

13-2. Implement a back-face detection procedure using an orthographic parallel projection to view visible faces of a convex polyhedron. Assume that all parts of the object are in front of the view plane, and provide a mapping onto a screen viewport for display.

13-3. Implement a back-face detection procedure using a perspective projection to view visible faces of a convex polyhedron. Assume that all parts of the object are in front of the view plane, and provide a mapping onto a screen viewport for display.

13-4. Write a program to produce an animation of a convex polyhedron. The object is to be rotated incrementally about an axis that passes through the object and is parallel to the view plane. Assume that the object lies completely in front of the view plane. Use an orthographic parallel projection to map the views successively onto the view plane.

13-5. Implement the depth-buffer method to display the visible surfaces of a given polyhedron. How can the storage requirements for the depth buffer be determined from the definition of the objects to be displayed?

13-6. Implement the depth-buffer method to display the visible surfaces in a scene containing any number of polyhedrons. Set up efficient methods for storing and processing the various objects in the scene.

13-7. Implement the A-buffer algorithm to display a scene containing both opaque and transparent surfaces. As an optional feature, your algorithm may be extended to include antialiasing.

13-8. Develop a program to implement the scan-line algorithm for displaying the visible surfaces of a given polyhedron. Use polygon and edge tables to store the definition of the object, and use coherence techniques to evaluate points along and between scan lines.

13-9. Write a program to implement the scan-line algorithm for a scene containing several polyhedrons. Use polygon and edge tables to store the definition of the object, and use coherence techniques to evaluate points along and between scan lines.

13-10. Set up a program to display the visible surfaces of a convex polyhedron using the painter's algorithm. That is, surfaces are to be sorted on depth and painted on the screen from back to front.

13-11. Write a program that uses the depth-sorting method to display the visible surfaces of any given object with plane faces.

13-12. Develop a depth-sorting program to display the visible surfaces in a scene containing several polyhedrons.

13-13. Write a program to display the visible surfaces of a convex polyhedron using the BSP-tree method.

13-14. Give examples of situations where the two methods discussed for test 3 in the area-subdivision algorithm will fail to identify correctly a surrounding surface that obscures all other surfaces.

13-15. Develop an algorithm that would test a given plane surface against a rectangular area to decide whether it is a surrounding, overlapping, inside, or outside surface.

13-16. Develop an algorithm for generating a quadtree representation for the visible surfaces of an object by applying the area-subdivision tests to determine the values of the quadtree elements.

13-17. Set up an algorithm to load a given quadtree representation of an object into a frame buffer for display.

13-18. Write a program on your system to display an octree representation for an object so that hidden-surfaces are removed.

13-19. Devise an algorithm for viewing a single sphere using the ray-casting method.

13-20. Discuss how antialiasing methods can be incorporated into the various hidden-surface elimination algorithms.

13-21. Write a routine to produce a surface contour plot for a given surface function $f(x, y)$.

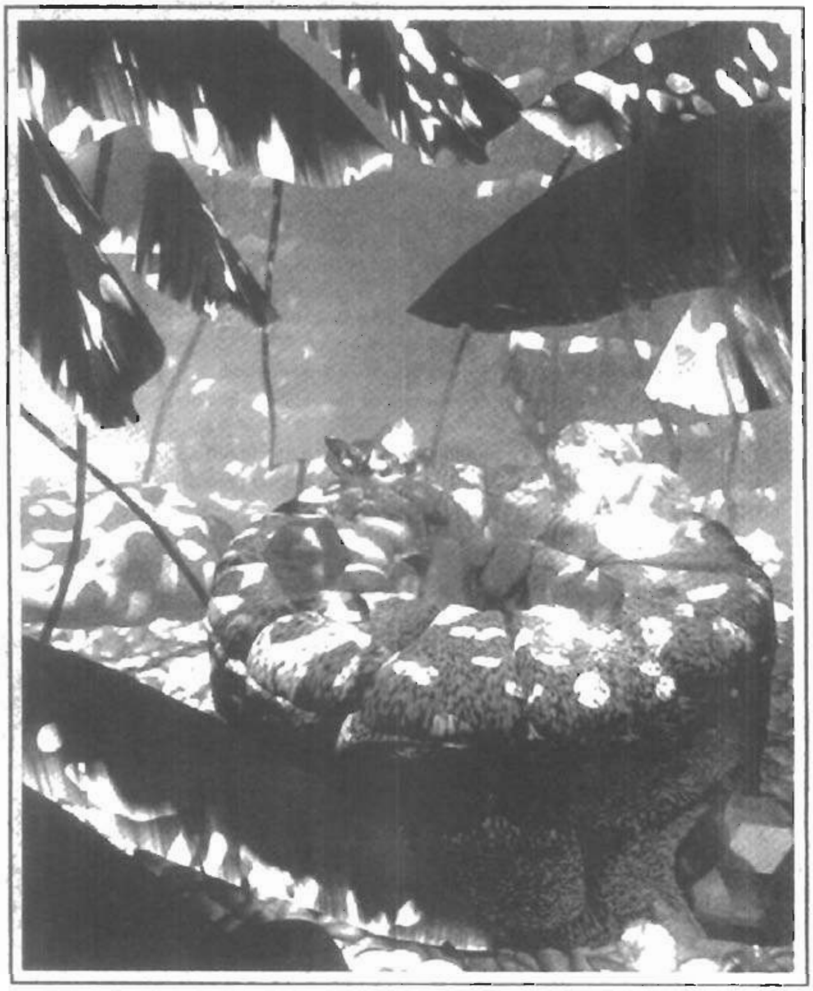
13-22. Develop an algorithm for detecting visible line sections in a scene by comparing each line in the scene to each surface.

13-23. Discuss how wireframe displays might be generated with the various visible-surface detection methods discussed in this chapter.

13-24. Set up a procedure for generating a wireframe display of a polyhedron with the hidden edges of the object drawn with dashed lines.

CHAPTER

14 Illumination Models and Surface-Rendering Methods



Realistic displays of a scene are obtained by generating perspective projections of objects and by applying natural lighting effects to the visible surfaces. An **illumination model,** also called a **lighting model** and sometimes referred to as a **shading model,** is used to calculate the intensity of light that we should see at a given point on the surface of an object. A **surface-rendering algorithm** uses the intensity calculations from an illumination model to determine the light intensity for all projected pixel positions for the various surfaces in a scene. Surface rendering can be performed by applying the illumination model to every visible surface point, or the rendering can be accomplished by interpolating intensities across the surfaces from a small set of illumination-model calculations. Scan-line, image-space algorithms typically use interpolation schemes, while ray-tracing algorithms invoke the illumination model at each pixel position. Sometimes, surface-rendering procedures are termed *surface-shading methods.* To avoid confusion, we will refer to the model for calculating light intensity at a single surface point as an *illumination model* or a *lighting model,* and we will use the term *surface rendering* to mean a procedure for applying a lighting model to obtain pixel intensities for all the projected surface positions in a scene.

Photorealism in computer graphics involves two elements: accurate graphical representations of objects and good physical descriptions of the lighting effects in a scene. Lighting effects include light reflections, transparency, surface texture, and shadows.

Modeling the colors and lighting effects that we see on an object is a complex process, involving principles of both physics and psychology. Fundamentally, lighting effects are described with models that consider the interaction of electromagnetic energy with object surfaces. Once light reaches our eyes, it triggers perception processes that determine what we actually "see" in a scene. Physical illumination models involve a number of factors, such as object type, object position relative to light sources and other objects, and the light-source conditions that we set for a scene. Objects can be constructed of opaque materials, or they can be more or less transparent. In addition, they can have shiny or dull surfaces, and they can have a variety of surface-texture patterns. Light sources, of varying shapes, colors, and positions, can be used to provide the illumination effects for a scene. Given the parameters for the optical properties of surfaces, the relative positions of the surfaces in a scene, the color and positions of the light sources, and the position and orientation of the viewing plane, illumination models calculate the intensity projected from a particular surface point in a specified viewing direction.

Illumination models in computer graphics are often loosely derived from the physical laws that describe surface light intensities. To minimize intensity calculations, most packages use empirical models based on simplified photometric calculations. More accurate models, such as the radiosity algorithm, calculate light intensities by considering the propagation of radiant energy between the surfaces and light sources in a scene. In the following sections, we first take a look at the basic illumination models often used in graphics packages; then we discuss more accurate, but more time-consuming, methods for calculating surface intensities. And we explore the various surface-rendering algorithms for applying the lighting models to obtain the appropriate shading over visible surfaces in a scene.

14-1 LIGHT SOURCES

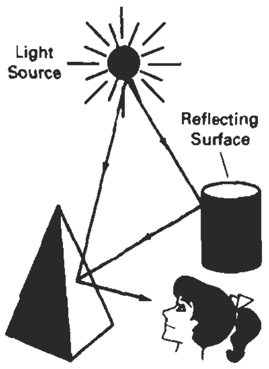


Figure 14-1
Light viewed from an opaque nonluminous surface is in general a combination of reflected light from a light source and reflections of light reflections from other surfaces.

When we view an opaque nonluminous object, we see reflected light from the surfaces of the object. The total reflected light is the sum of the contributions from **light sources** and other reflecting surfaces in the scene (Fig. 14-1). Thus, a surface that is not directly exposed to a light source may still be visible if nearby objects are illuminated. Sometimes, light sources are referred to as *light-emitting sources;* and reflecting surfaces, such as the walls of a room, are termed *light-reflecting sources.* We will use the term *light source* to mean an object that is emitting radiant energy, such as a light bulb or the sun.

A luminous object, in general, can be both a light source and a light reflector. For example, a plastic globe with a light bulb inside both emits and reflects light from the surface of the globe. Emitted light from the globe may then illuminate other objects in the vicinity.

The simplest model for a light emitter is a **point source**. Rays from the source then follow radially diverging paths from the source position, as shown in Fig. 14-2. This light-source model is a reasonable approximation for sources whose dimensions are small compared to the size of objects in the scene. Sources, such as the sun, that are sufficiently far from the scene can be accurately modeled as point sources. A nearby source, such as the long fluorescent light in Fig. 14-3, is more accurately modeled as a **distributed light source**. In this case, the illumination effects cannot be approximated realistically with a point source, because the area of the source is not small compared to the surfaces in the scene. An accurate model for the distributed source is one that considers the accumulated illumination effects of the points over the surface of the source.

When light is incident on an opaque surface, part of it is reflected and part is absorbed. The amount of incident light reflected by a surface depends on the type of material. Shiny materials reflect more of the incident light, and dull surfaces absorb more of the incident light. Similarly, for an illuminated transparent

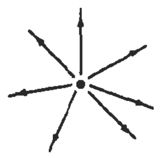


Figure 14-2
Diverging ray paths from a point light source.

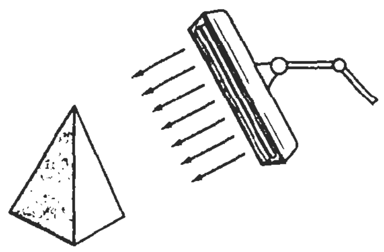


Figure 14-3
An object illuminated with a distributed light source.

surface, some of the incident light will be reflected and some will be transmitted through the material.

Surfaces that are rough, or grainy, tend to scatter the reflected light in all directions. This scattered light is called **diffuse reflection**. A very rough matte surface produces primarily diffuse reflections, so that the surface appears equally bright from all viewing directions. Figure 14-4 illustrates diffuse light scattering from a surface. What we call the color of an object is the color of the diffuse reflection of the incident light. A blue object illuminated by a white light source, for example, reflects the blue component of the white light and totally absorbs all other components. If the blue object is viewed under a red light, it appears black since all of the incident light is absorbed.

In addition to diffuse reflection, light sources create highlights, or bright spots, called **specular reflection**. This highlighting effect is more pronounced on shiny surfaces than on dull surfaces. An illustration of specular reflection is shown in Fig. 14-5.

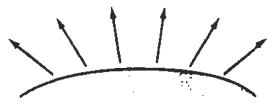


Figure 14-4
Diffuse reflections from a surface.

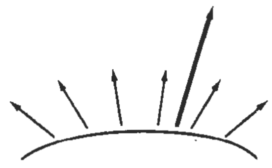


Figure 14-5
Specular reflection superimposed on diffuse reflection vectors.

14-2 BASIC ILLUMINATION MODELS

Here we discuss simplified methods for calculating light intensities. The empirical models described in this section provide simple and fast methods for calculating surface intensity at a given point, and they produce reasonably good results for most scenes. Lighting calculations are based on the optical properties of surfaces, the background lighting conditions, and the light-source specifications. Optical parameters are used to set surface properties, such as glossy, matte, opaque, and transparent. This controls the amount of reflection and absorption of incident light. All light sources are considered to be point sources, specified with a coordinate position and an intensity value (color).

Ambient Light

A surface that is not exposed directly to a light source still will be visible if nearby objects are illuminated. In our basic illumination model, we can set a general level of brightness for a scene. This is a simple way to model the combination of light reflections from various surfaces to produce a uniform illumination called the **ambient light,** or **background light**. Ambient light has no spatial or directional characteristics. The amount of ambient light incident on each object is a constant for all surfaces and over all directions.

We can set the level for the ambient light in a scene with parameter I_a, and each surface is then illuminated with this constant value. The resulting reflected light is a constant for each surface, independent of the viewing direction and the spatial orientation of the surface. But the intensity of the reflected light for each surface depends on the optical properties of the surface; that is, how much of the incident energy is to be reflected and how much absorbed.

Diffuse Reflection

Ambient-light reflection is an approximation of global diffuse lighting effects. Diffuse reflections are constant over each surface in a scene, independent of the viewing direction. The fractional amount of the incident light that is diffusely re-

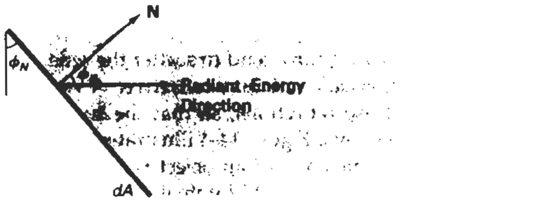


Figure 14-6
Radiant energy from a surface area dA in direction ϕ_N relative to the surface normal direction.

flected can be set for each surface with parameter k_d, the **diffuse-reflection coefficient, or diffuse reflectivity**. Parameter k_d is assigned a constant value in the interval 0 to 1, according to the reflecting properties we want the surface to have. If we want a highly reflective surface, we set the value of k_d near 1. This produces a bright surface with the intensity of the reflected light near that of the incident light. To simulate a surface that absorbs most of the incident light, we set the reflectivity to a value near 0. Actually, parameter k_d is a function of surface color, but for the time being we will assume k_d is a constant.

If a surface is exposed only to ambient light, we can express the intensity of the diffuse reflection at any point on the surface as

$$I_{\text{ambdiff}} = k_d I_a \tag{14-1}$$

Since ambient light produces a flat uninteresting shading for each surface (Fig. 14-19(b)), scenes are rarely rendered with ambient light alone. At least one light source is included in a scene, often as a point source at the viewing position.

We can model the diffuse reflections of illumination from a point source in a similar way. That is, we assume that the diffuse reflections from the surface are scattered with equal intensity in all directions, independent of the viewing direction. Such surfaces are sometimes referred to as *ideal diffuse reflectors*. They are also called *Lambertian reflectors*, since radiated light energy from any point on the surface is governed by *Lambert's cosine law*. This law states that the radiant energy from any small surface area dA in any direction ϕ_N relative to the surface normal is proportional to $\cos\phi_N$ (Fig. 14-6). The light intensity, though, depends on the radiant energy per projected area perpendicular to direction ϕ_N, which is $dA \cos\phi_N$. Thus, for Lambertian reflection, the intensity of light is the same over all viewing directions. We discuss photometry concepts and terms, such as radiant energy, in greater detail in Section 14-7.

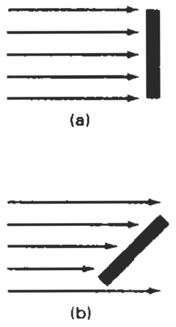


Figure 14-7
A surface perpendicular to the direction of the incident light (a) is more illuminated than an equal-sized surface at an oblique angle (b) to the incoming light direction.

Even though there is equal light scattering in all directions from a perfect diffuse reflector, the brightness of the surface does depend on the orientation of the surface relative to the light source. A surface that is oriented perpendicular to the direction of the incident light appears brighter than if the surface were tilted at an oblique angle to the direction of the incoming light. This is easily seen by holding a white sheet of paper or smooth cardboard parallel to a nearby window and slowly rotating the sheet away from the window direction. As the angle between the surface normal and the incoming light direction increases, less of the incident light falls on the surface, as shown in Fig. 14-7. This figure shows a beam of light rays incident on two equal-area plane surface patches with different spatial orientations relative to the incident light direction from a distant source (par-

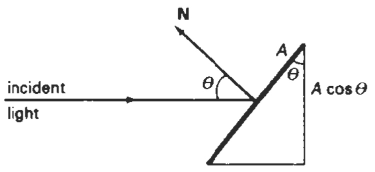


Figure 14-8
An illuminated area projected perpendicular to the path of the incoming light rays.

allel incoming rays). If we denote the **angle of incidence** between the incoming light direction and the surface normal as θ (Fig. 14-8), then the projected area of a surface patch perpendicular to the light direction is proportional to $\cos\theta$. Thus, the amount of illumination (or the "number of incident light rays" cutting across the projected surface patch) depends on $\cos\theta$. If the incoming light from the source is perpendicular to the surface at a particular point, that point is fully illuminated. As the angle of illumination moves away from the surface normal, the brightness of the point drops off. If I_l is the intensity of the point light source, then the diffuse reflection equation for a point on the surface can be written as

$$I_{l,\text{diff}} = k_d I_l \cos\theta \qquad (14\text{-}2)$$

A surface is illuminated by a point source only if the angle of incidence is in the range 0° to 90° ($\cos\theta$ is in the interval from 0 to 1). When $\cos\theta$ is negative, the light source is "behind" the surface.

If **N** is the unit normal vector to a surface and **L** is the unit direction vector to the point light source from a position on the surface (Fig. 14-9), then $\cos\theta = \mathbf{N} \cdot \mathbf{L}$ and the diffuse reflection equation for single point-source illumination is

$$I_{l,\text{diff}} = k_d I_l (\mathbf{N} \cdot \mathbf{L}) \qquad (14\text{-}3)$$

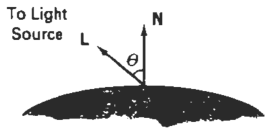


Figure 14-9
Angle of incidence θ between the unit light-source direction vector **L** and the unit surface normal **N**.

Reflections for point-source illumination are calculated in world coordinates or viewing coordinates before shearing and perspective transformations are applied. These transformations may transform the orientation of normal vectors so that they are no longer perpendicular to the surfaces they represent. Transformation procedures for maintaining the proper orientation of surface normals are discussed in Chapter 11.

Figure 14-10 illustrates the application of Eq. 14-3 to positions over the surface of a sphere, using various values of parameter k_d between 0 and 1. Each projected pixel position for the surface was assigned an intensity as calculated by the diffuse reflection equation for a point light source. The renderings in this figure illustrate single point-source lighting with no other lighting effects. This is what we might expect to see if we shined a small light on the object in a completely darkened room. For general scenes, however, we expect some background lighting effects in addition to the illumination effects produced by a direct light source.

We can combine the ambient and point-source intensity calculations to obtain an expression for the total diffuse reflection. In addition, many graphics packages introduce an **ambient-reflection coefficient** k_a to modify the ambient-light intensity I_a for each surface. This simply provides us with an additional parameter to adjust the light conditions in a scene. Using parameter k_a, we can write the total diffuse reflection equation as

$$I_{\text{diff}} = k_a I_a + k_d I_l (\mathbf{N} \cdot \mathbf{L}) \qquad (14\text{-}4)$$

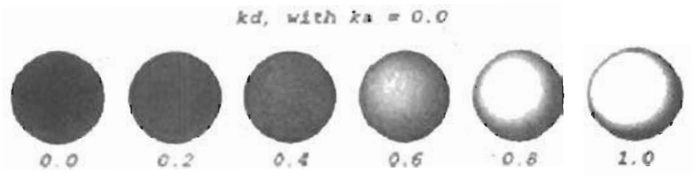


Figure 14-10
Diffuse reflections from a spherical surface illuminated by a point light source for values of the diffuse reflectivity coefficient in the interval $0 \leq k_d \leq 1$.

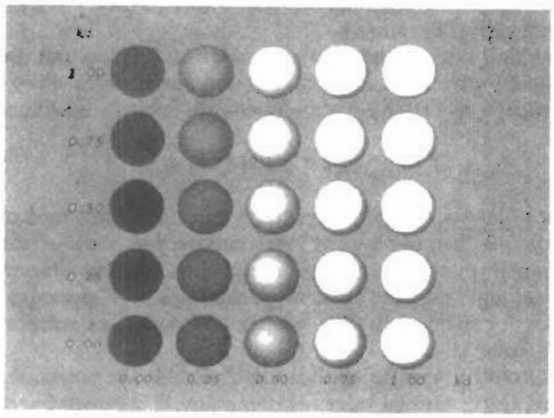


Figure 14-11
Diffuse reflections from a spherical surface illuminated with ambient light and a single point source for values of k_a and k_d in the interval (0, 1).

where both k_a and k_d depend on surface material properties and are assigned values in the range from 0 to 1. Figure 14-11 shows a sphere displayed with surface intensitities calculated from Eq. 14-4 for values of parameters k_a and k_d between 0 and 1.

Specular Reflection and the Phong Model

When we look at an illuminated shiny surface, such as polished metal, an apple, or a person's forehead, we see a highlight, or bright spot, at certain viewing directions. This phenomenon, called *specular reflection*, is the result of total, or near total, reflection of the incident light in a concentrated region around the **specular-reflection angle**. Figure 14-12 shows the specular reflection direction at a point on the illuminated surface. The specular-reflection angle equals the angle of the incident light, with the two angles measured on opposite sides of the unit normal surface vector **N**. In this figure, we use **R** to represent the unit vector in the direction of ideal specular reflection; **L** to represent the unit vector directed toward the point light source; and **V** as the unit vector pointing to the viewer from the surface position. Angle ϕ is the viewing angle relative to the specular-reflection direction **R**. For an ideal reflector (perfect mirror), incident light is reflected only in the specular-reflection direction. In this case, we would only see reflected light when vectors **V** and **R** coincide ($\phi = 0$).

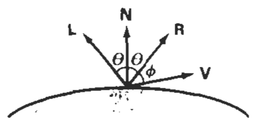


Figure 14-12
Specular-reflection angle equals angle of incidence θ.

Objects other than ideal reflectors exhibit specular reflections over a finite range of viewing positions around vector **R**. Shiny surfaces have a narrow specular-reflection range, and dull surfaces have a wider reflection range. An empirical model for calculating the specular-reflection range, developed by Phong Bui Tuong and called the **Phong specular-reflection model,** or simply the **Phong model,** sets the intensity of specular reflection proportional to $\cos^{n_s}\phi$. Angle ϕ can be assigned values in the range 0° to 90°, so that $\cos\phi$ varies from 0 to 1. The value assigned to *specular-reflection parameter* n_s is determined by the type of surface that we want to display. A very shiny surface is modeled with a large value for n_s (say, 100 or more), and smaller values (down to 1) are used for duller surfaces. For a perfect reflector, n_s is infinite. For a rough surface, such as chalk or cinderblock, n_s would be assigned a value near 1. Figures 14-13 and 14-14 show the effect of n_s on the angular range for which we can expect to see specular reflections.

The intensity of specular reflection depends on the material properties of the surface and the angle of incidence, as well as other factors such as the polarization and color of the incident light. We can approximately model monochromatic specular intensity variations using a **specular-reflection coefficient,** $W(\theta)$, for each surface. Figure 14-15 shows the general variation of $W(\theta)$ over the range $\theta = 0°$ to $\theta = 90°$ for a few materials. In general, $W(\theta)$ tends to increase as the angle of incidence increases. At $\theta = 90°$, $W(\theta) = 1$ and all of the incident light is reflected. The variation of specular intensity with angle of incidence is described by *Fresnel's Laws of Reflection.* Using the spectral-reflection function $W(\theta)$, we can write the Phong specular-reflection model as

$$I_{\text{spec}} = W(\theta) I_l \cos^{n_s} \phi \qquad (14\text{-}5)$$

where I_l is the intensity of the light source, and ϕ is the viewing angle relative to the specular-reflection direction **R**.

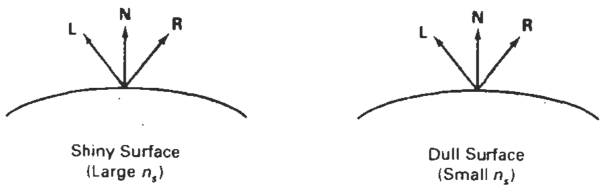


Figure 14-13
Modeling specular reflections (shaded area) with parameter n_s.

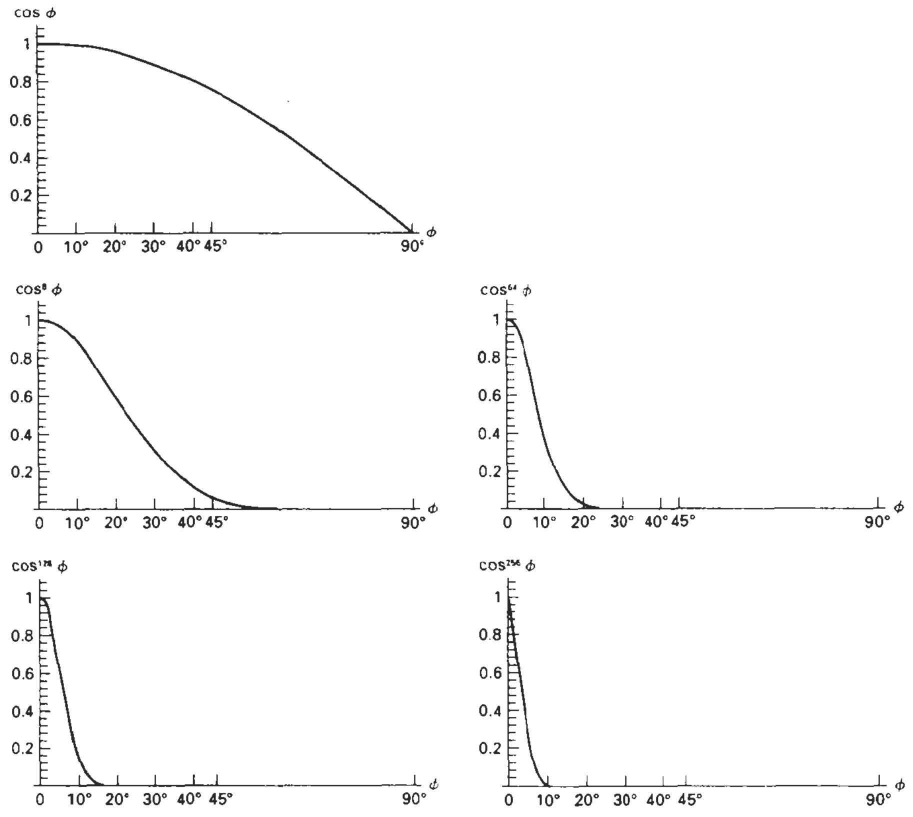


Figure 14-14
Plots of $\cos^{n_s}\phi$ for several values of specular parameter n_s.

As seen in Fig. 14-15, transparent materials, such as glass, only exhibit appreciable specular reflections as θ approaches 90°. At $\theta = 0°$, about 4 percent of the incident light on a glass surface is reflected. And for most of the range of θ, the reflected intensity is less than 10 percent of the incident intensity. But for many opaque materials, specular reflection is nearly constant for all incidence angles. In this case, we can reasonably model the reflected light effects by replacing $W(\theta)$ with a constant specular-reflection coefficient k_s. We then simply set k_s equal to some value in the range 0 to 1 for each surface.

Since **V** and **R** are unit vectors in the viewing and specular-reflection directions, we can calculate the value of $\cos\phi$ with the dot product $\mathbf{V} \cdot \mathbf{R}$. Assuming the specular-reflection coefficient is a constant, we can determine the intensity of the specular reflection at a surface point with the calculation

$$I_{spec} = k_s I_l (\mathbf{V} \cdot \mathbf{R})^{n_s} \qquad (14\text{-}9)$$

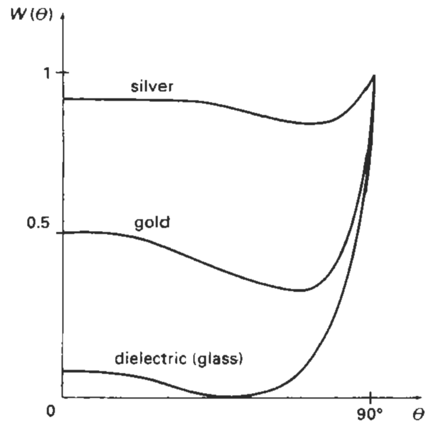


Figure 14-15
Approximate variation of the specular-reflection coefficient as a function of angle of incidence for different materials.

Vector **R** in this expression can be calculated in terms of vectors **L** and **N**. As seen in Fig. 14-16, the projection of **L** onto the direction of the normal vector is obtained with the dot product **N** · **L**. Therefore, from the diagram, we have

$$\mathbf{R} + \mathbf{L} = (2\mathbf{N} \cdot \mathbf{L})\mathbf{N}$$

and the specular-reflection vector is obtained as

$$\mathbf{R} = (2\mathbf{N} \cdot \mathbf{L})\mathbf{N} - \mathbf{L} \tag{14-7}$$

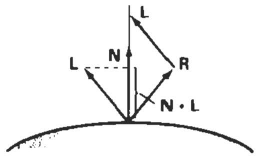


Figure 14-16
Calculation of vector **R** by considering projections onto the direction of the normal vector **N**.

Figure 14-17 illustrates specular reflections for various values of k_s and n_s on a sphere illuminated with a single point light source.

A somewhat simplified Phong model is obtained by using the *halfway vector* **H** between **L** and **V** to calculate the range of specular reflections. If we replace **V** · **R** in the Phong model with the dot product **N** · **H**, this simply replaces the empirical cos ϕ calculation with the empirical cos α calculation (Fig. 14-18). The halfway vector is obtained as

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{|\mathbf{L} + \mathbf{V}|} \tag{14-8}$$

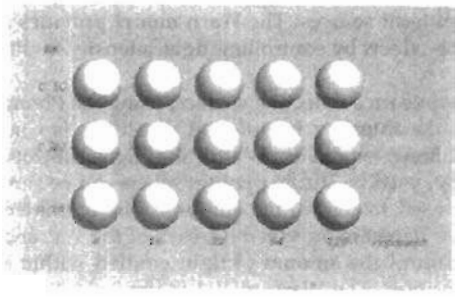


Figure 14-17
Specular reflections from a spherical surface for varying specular parameter values and a single light source.

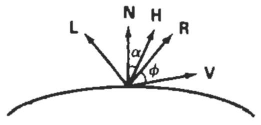


Figure 14-18
Halfway vector **H** along the bisector of the angle between **L** and **V**.

If both the viewer and the light source are sufficiently far from the surface, both **V** and **L** are constant over the surface, and thus **H** is also constant for all surface points. For nonplanar surfaces, **N** · **H** then requires less computation than **V** · **R** since the calculation of **R** at each surface point involves the variable vector **N**.

For given light-source and viewer positions, vector **H** is the orientation direction for the surface that would produce maximum specular reflection in the viewing direction. For this reason, **H** is sometimes referred to as the surface orientation direction for maximum highlights. Also, if vector **V** is coplanar with vectors **L** and **R** (and thus **N**), angle α has the value $\phi/2$. When **V, L,** and **N** are not coplanar, $\alpha > \phi/2$, depending on the spatial relationship of the three vectors.

Combined Diffuse and Specular Reflections with Multiple Light Sources

For a single point light source, we can model the combined diffuse and specular reflections from a point on an illuminated surface as

$$\begin{aligned} I &= I_{\text{diff}} + I_{\text{spec}} \\ &= k_a I_a + k_d I_l (\mathbf{N} \cdot \mathbf{L}) + k_s I_l (\mathbf{N} \cdot \mathbf{H})^{n_s} \end{aligned} \tag{14-9}$$

Figure 14-19 illustrates surface lighting effects produced by the various terms in Eq. 14-9. If we place more than one point source in a scene, we obtain the light reflection at any surface point by summing the contributions from the individual sources:

$$I = k_a I_a + \sum_{i=1}^{n} I_{l_i} [k_d (\mathbf{N} \cdot \mathbf{L}_i) + k_s (\mathbf{N} \cdot \mathbf{H}_i)^{n_s}] \tag{14-10}$$

To ensure that any pixel intensity does not exceed the maximum allowable value, we can apply some type of normalization procedure. A simple approach is to set a maximum magnitude for each term in the intensity equation. If any calculated term exceeds the maximum, we simply set it to the maximum value. Another way to compensate for intensity overflow is to normalize the individual terms by dividing each by the magnitude of the largest term. A more complicated procedure is first to calculate all pixel intensities for the scene, then the calculated intensities are scaled onto the allowable intensity range.

Warn Model

So far we have considered only point light sources. The **Warn model** provides a method for simulating studio lighting effects by controlling light intensity in different directions.

Light sources are modeled as points on a reflecting surface, using the Phong model for the surface points. Then the intensity in different directions is controlled by selecting values for the Phong exponent. In addition, light controls, such as "barn doors" and spotlighting, used by studio photographers can be simulated in the Warn model. *Flaps* are used to control the amount of light emitted by a source in various directions. Two flaps are provided for each of the x, y, and z directions. *Spotlights* are used to control the amount of light emitted within a cone with apex at a point-source position. The Warn model is implemented in

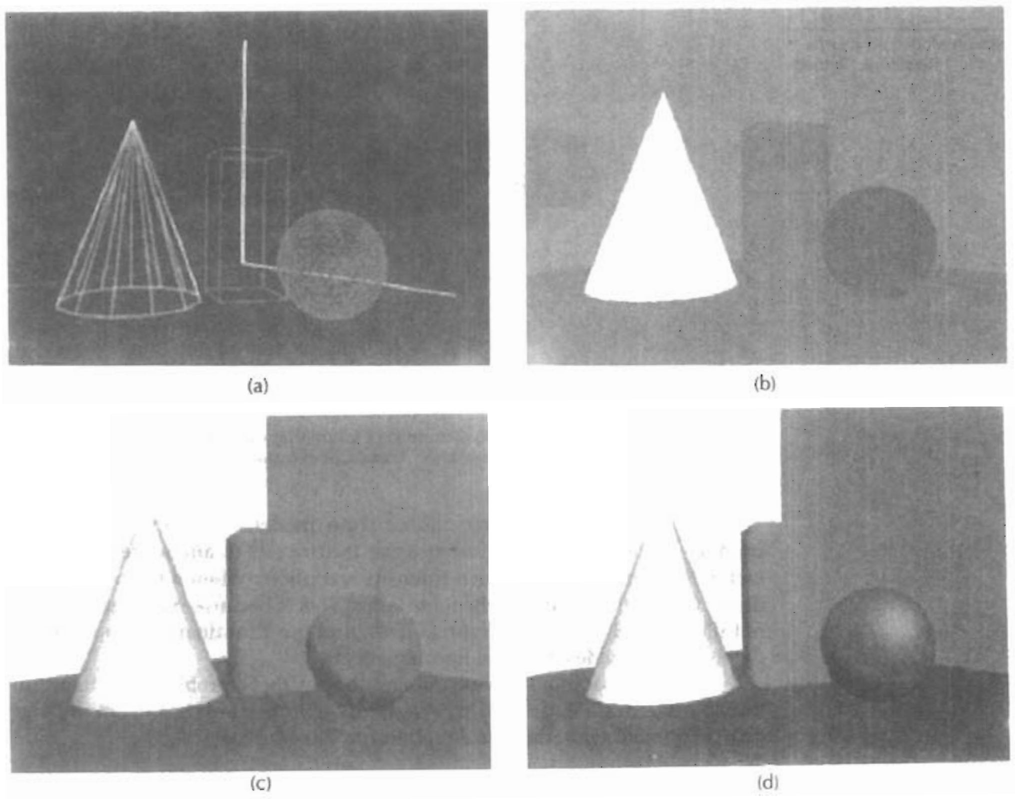


Figure 14-19
A wireframe scene (a) is displayed only with ambient lighting in (b), and the surface of each object is assigned a different color. Using ambient light and diffuse reflections due to a single source with $k_s = 0$ for all surfaces, we obtain the lighting effects shown in (c). Using ambient light and both diffuse and specular reflections due to a single light source, we obtain the lighting effects shown in (d).

PHIGS+, and Fig. 14-20 illustrates lighting effects that can be produced with this model.

Intensity Attenuation

As radiant energy from a point light source travels through space, its amplitude is attenuated by the factor $1/d^2$, where d is the distance that the light has traveled. This means that a surface close to the light source (small d) receives a higher incident intensity from the source than a distant surface (large d). Therefore, to produce realistic lighting effects, our illumination model should take this intensity attenuation into account. Otherwise, we are illuminating all surfaces with the same intensity, no matter how far they might be from the light source. If two parallel surfaces with the same optical parameters overlap, they would be indistinguishable from each other. The two surfaces would be displayed as one surface.

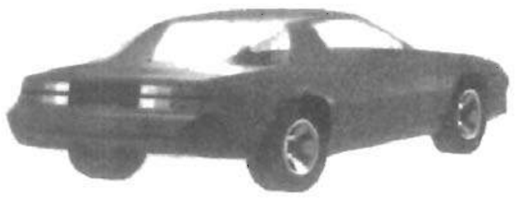


Figure 14-20
Studio lighting effects produced with the Warn model, using five light sources to illuminate a Chevrolet Camaro. *(Courtesy of David R. Warn, General Motors Research Laboratories.)*

Our simple point-source illumination model, however, does not always produce realistic pictures, if we use the factor $1/d^2$ to attenuate intensities. The factor $1/d^2$ produces too much intensity variations when d is small, and it produces very little variation when d is large. This is because real scenes are usually not illuminated with point light sources, and our illumination model is too simple to accurately describe real lighting effects.

Graphics packages have compensated for these problems by using inverse linear or quadratic functions of d to attenuate intensities. For example, a general inverse quadratic **attenuation function** can be set up as

$$f(d) = \frac{1}{a_0 + a_1 d + a_2 d^2} \tag{14-11}$$

A user can then fiddle with the coefficients a_0, a_1, and a_2 to obtain a variety of lighting effects for a scene. The value of the constant term a_0 can be adjusted to prevent $f(d)$ from becoming too large when d is very small. Also, the values for the coefficients in the attenuation function, and the optical surface parameters for a scene, can be adjusted to prevent calculations of reflected intensities from exceeding the maximum allowable value. This is an effective method for limiting intensity values when a single light source is used to illuminate a scene. For multiple light-source illumination, the methods described in the preceding section are more effective for limiting the intensity range.

With a given set of attenuation coefficients, we can limit the magnitude of the attenuation function to 1 with the calculation

$$f(d) = \min\left(1, \frac{1}{a_0 + a_1 d + a_2 d^2}\right) \tag{14-12}$$

Using this function, we can then write our basic illumination model as

$$I = k_a I_a + \sum_{i=1}^{n} f(d_i) I_{li} [k_d(\mathbf{N} \cdot \mathbf{L}_i) + k_s(\mathbf{N} \cdot \mathbf{H}_i)^{n_s}] \tag{14-13}$$

where d_i is the distance light has traveled from light source i.

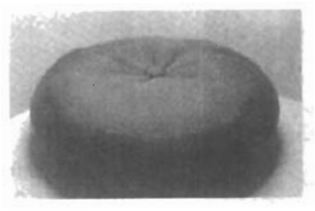


Figure 14-21
Light reflections from the surface of a black nylon cushion, modeled as woven cloth patterns and rendered using Monte Carlo ray-tracing methods. *(Courtesy of Stephen H. Westin, Program of Computer Graphics, Cornell University.)*

Color Considerations

Most graphics displays of realistic scenes are in color. But the illumination model we have described so far considers only monochromatic lighting effects. To incorporate color, we need to write the intensity equation as a function of the color properties of the light sources and object surfaces.

For an RGB description, each color in a scene is expressed in terms of red, green, and blue components. We then specify the RGB components of light-source intensities and surface colors, and the illumination model calculates the RGB components of the reflected light. One way to set surface colors is by specifying the reflectivity coefficients as three-element vectors. The diffuse reflection-coefficient vector, for example, would then have RGB components (k_{dR}, k_{dG}, k_{dB}). If we want an object to have a blue surface, we select a nonzero value in the range from 0 to 1 for the blue reflectivity component, k_{dB}, while the red and green reflectivity components are set to zero ($k_{dR} = k_{dG} = 0$). Any nonzero red or green components in the incident light are absorbed, and only the blue component is reflected. The intensity calculation for this example reduces to the single expression

$$I_B = k_{aB} I_{aB} + \sum_{i=1}^{n} f_i(d) I_{lBi} [k_{dB}(\mathbf{N} \cdot \mathbf{L}_i) + k_{sB}(\mathbf{N} \cdot \mathbf{H}_i)^{n_s}] \tag{14-14}$$

Surfaces typically are illuminated with white light sources, and in general we can set surface color so that the reflected light has nonzero values for all three RGB components. Calculated intensity levels for each color component can be used to adjust the corresponding electron gun in an RGB monitor.

In his original specular-reflection model, Phong set parameter k_s to a constant value independent of the surface color. This produces specular reflections that are the same color as the incident light (usually white), which gives the surface a plastic appearance. For a nonplastic material, the color of the specular reflection is a function of the surface properties and may be different from both the color of the incident light and the color of the diffuse reflections. We can approximate specular effects on such surfaces by making the specular-reflection coefficient color-dependent, as in Eq. 14-14. Figure 14-21 illustrates color reflections from a matte surface, and Figs. 14-22 and 14-23 show color reflections from metal

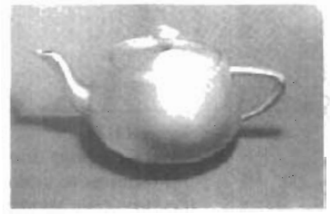


Figure 14-22
Light reflections from a teapot with reflectance parameters set to simulate brushed aluminum surfaces and rendered using Monte Carlo ray-tracing methods. *(Courtesy of Stephen H. Westin, Program of Computer Graphics, Cornell University.)*

Figure 14-23
Light reflections from trombones with reflectance parameters set to simulate shiny brass surfaces. *(Courtesy of SOFTIMAGE, Inc.)*

surfaces. Light reflections from object surfaces due to multiple colored light sources is shown in Fig. 14-24.

Another method for setting surface color is to specify the components of diffuse and specular color vectors for each surface, while retaining the reflectivity coefficients as single-valued constants. For an RGB color representation, for instance, the components of these two surface-color vectors can be denoted as (S_{dR}, S_{dG}, S_{dB}) and (S_{sR}, S_{sG}, S_{sB}). The blue component of the reflected light is then calculated as

$$I_B = k_a S_{dB} I_{aB} + \sum_{i=1}^{n} f_i(d)\, I_{lBi}[k_d S_{dB}(\mathbf{N} \cdot \mathbf{L}_i) + k_s S_{sB}(\mathbf{N} \cdot \mathbf{H}_i)^{n_s}] \tag{14-15}$$

This approach provides somewhat greater flexibility, since surface-color parameters can be set independently from the reflectivity values.

Other color representations besides RGB can be used to describe colors in a scene. And sometimes it is convenient to use a color model with more than three components for a color specification. We discuss color models in detail in the next chapter. For now, we can simply represent any component of a color specification with its spectral wavelength λ. Intensity calculations can then be expressed as

$$I_\lambda = k_a S_{d\lambda} I_{a\lambda} + \sum_{i=1}^{n} f_i(d) I_{l\lambda i}[k_d S_{d\lambda}(\mathbf{N} \cdot \mathbf{L}_i) + k_s S_{s\lambda}(\mathbf{N} \cdot \mathbf{H}_i)^{n_s}] \tag{14-16}$$

Transparency

A transparent surface, in general, produces both reflected and transmitted light. The relative contribution of the transmitted light depends on the degree of trans-

Figure 14-24
Light reflections due to multiple light sources of various colors. *(Courtesy of Sun Microsystems.)*

parency of the surface and whether any light sources or illuminated surfaces are behind the transparent surface. Figure 14-25 illustrates the intensity contributions to the surface lighting for a transparent object.

When a transparent surface is to be modeled, the intensity equations must be modified to include contributions from light passing through the surface. In most cases, the transmitted light is generated from reflecting objects in back of the surface, as in Fig. 14-26. Reflected light from these objects passes through the transparent surface and contributes to the total surface intensity.

Both diffuse and specular transmission can take place at the surfaces of a transparent object. Diffuse effects are important when a partially transparent surface, such as frosted glass, is to be modeled. Light passing through such materials is scattered so that a blurred image of background objects is obtained. Diffuse refractions can be generated by decreasing the intensity of the refracted light and spreading intensity contributions at each point on the refracting surface onto a finite area. These manipulations are time-comsuming, and most lighting models employ only specular effects.

Realistic transparency effects are modeled by considering light refraction. When light is incident upon a transparent surface, part of it is reflected and part is **refracted** (Fig. 14-27). Because the speed of light is different in different materials, the path of the refracted light is different from that of the incident light. The direction of the refracted light, specified by the **angle of refraction**, is a function of the **index of refraction** of each material and the direction of the incident light. Index of refraction for a material is defined as the ratio of the speed of light in a vacuum to the speed of light in the material. Angle of refraction θ_r is calculated from the angle of incidence θ_i, the index of refraction η_i of the "incident" material (usually air), and the index of refraction η_r of the refracting material according to *Snell's law:*

$$\sin\theta_r = \frac{\eta_i}{\eta_r}\sin\theta_i \tag{14-17}$$

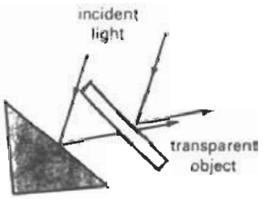


Figure 14-25
Light emission from a transparent surface is in general a combination of reflected and transmitted light.

Figure 14-26
A ray-traced view of a transparent glass surface, showing both light transmission from objects behind the glass and light reflection from the glass surface. (*Courtesy of Eric Haines, 3D/EYE Inc.*)

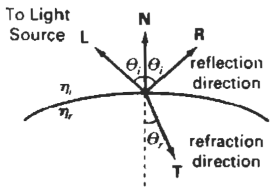


Figure 14-27
Reflection direction **R** and refraction direction **T** for a ray of light incident upon a surface with index of refraction η_r.

Actually, the index of refraction of a material is a function of the wavelength of the incident light, so that the different color components of a light ray will be refracted at different angles. For most applications, we can use an average index of refraction for the different materials that are modeled in a scene. The index of refraction of air is approximately 1, and that of crown glass is about 1.5. Using these values in Eq. 14-17 with an angle of incidence of 30° yields an angle of refraction of about 19°. Figure 14-28 illustrates the changes in the path direction for a light ray refracted through a glass object. The overall effect of the refraction is to shift the incident light to a parallel path. Since the calculations of the trigonometric functions in Eq. 14-17 are time-consuming, refraction effects could be modeled by simply shifting the path of the incident light a small amount.

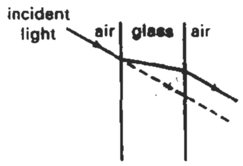


Figure 14-28
Refraction of light through a glass object. The emerging refracted ray travels along a path that is parallel to the incident light path (dashed line).

From Snell's law and the diagram in Fig. 14-27, we can obtain the unit transmission vector **T** in the refraction direction θ_r as

$$\mathbf{T} = \left(\frac{\eta_i}{\eta_r}\cos\theta_i - \cos\theta_r\right)\mathbf{N} - \frac{\eta_i}{\eta_r}\mathbf{L} \tag{14-18}$$

where **N** is the unit surface normal, and **L** is the unit vector in the direction of the light source. Transmission vector **T** can be used to locate intersections of the refraction path with objects behind the transparent surface. Including refraction effects in a scene can produce highly realistic displays, but the determination of refraction paths and object intersections requires considerable computation. Most scan-line image-space methods model light transmission with approximations that reduce processing time. We return to the topic of refraction in our discussion of ray-tracing algorithms (Section 14-6).

A simpler procedure for modeling transparent objects is to ignore the path shifts altogether. In effect, this approach assumes there is no change in the index of refraction from one material to another, so that the angle of refraction is always the same as the angle of incidence. This method speeds up the calculation of intensities and can produce reasonable transparency effects for thin polygon surfaces.

We can combine the transmitted intensity I_{trans} through a surface from a background object with the reflected intensity I_{refl} from the transparent surface (Fig. 14-29) using a **transparency coefficient** k_t. We assign parameter k_t a value between 0 and 1 to specify how much of the background light is to be transmitted. Total surface intensity is then calculated as

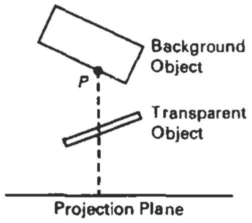


Figure 14-29
The intensity of a background object at point P can be combined with the reflected intensity off the surface of a transparent object along a perpendicular projection line (dashed).

$$I = (1 - k_t)I_{\text{refl}} + k_t I_{\text{trans}} \tag{14-19}$$

The term $(1 - k_t)$ is the **opacity factor**.

For highly transparent objects, we assign k_t a value near 1. Nearly opaque objects transmit very little light from background objects, and we can set k_t to a value near 0 for these materials (opacity near 1). It is also possible to allow k_t to be a function of position over the surface, so that different parts of an object can transmit more or less background intensity according to the values assigned to k_t.

Transparency effects are often implemented with modified depth-buffer (z-buffer) algorithms. A simple way to do this is to process opaque objects first to determine depths for the visible opaque surfaces. Then, the depth positions of the transparent objects are compared to the values previously strored in the depth buffer. If any transparent surface is visible, its reflected intensity is calculated and combined with the opaque-surface intensity previously stored in the frame buffer. This method can be modified to produce more accurate displays by using additional storage for the depth and other parameters of the transparent

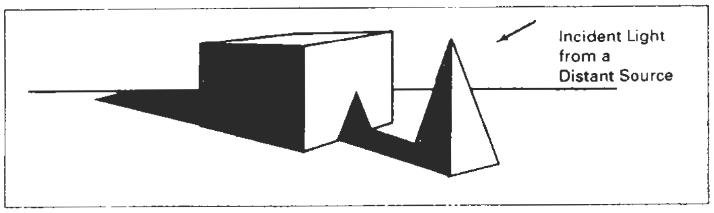


Figure 14-30
Objects modeled with shadow regions.

surfaces. This allows depth values for the transparent surfaces to be compared to each other, as well as to the depth values of the opaque surfaces. Visible transparent surfaces are then rendered by combining their surface intensities with those of the visible and opaque surfaces behind them.

Accurate displays of transparency and antialiasing can be obtained with the *A*-buffer algorithm. For each pixel position, surface patches for all overlapping surfaces are saved and sorted in depth order. Then, intensities for the transparent and opaque surface patches that overlap in depth are combined in the proper visibility order to produce the final averaged intensity for the pixel, as discussed in Chapter 13.

A depth-sorting visibility algorithm can be modified to handle transparency by first sorting surfaces in depth order, then determining whether any visible surface is transparent. If we find a visible transparent surface, its reflected surface intensity is combined with the surface intensity of objects behind it to obtain the pixel intensity at each projected surface point.

Shadows

Hidden-surface methods can be used to locate areas where light sources produce shadows. By applying a hidden-surface method with a light source at the view position, we can determine which surface sections cannot be "seen" from the light source. These are the shadow areas. Once we have determined the shadow areas for all light sources, the shadows could be treated as surface patterns and stored in pattern arrays. Figure 14-30 illustrates the generation of shading patterns for two objects on a table and a distant light source. All shadow areas in this figure are surfaces that are not visible from the position of the light source. The scene in Fig. 14-26 shows shadow effects produced by multiple light sources.

Shadow patterns generated by a hidden-surface method are valid for any selected viewing position, as long as the light-source positions are not changed. Surfaces that are visible from the view position are shaded according to the lighting model, which can be combined with texture patterns. We can display shadow areas with ambient-light intensity only, or we can combine the ambient light with specified surface textures.

14-3
DISPLAYING LIGHT INTENSITIES

Values of intensity calculated by an illumination model must be converted to one of the allowable intensity levels for the particular graphics system in use. Some systems are capable of displaying several intensity levels, while others are capable of only two levels for each pixel (on or off). In the first case, we convert intensities from the lighting model into one of the available levels for storage in the frame buffer. For bilevel systems, we can convert intensities into halftone patterns, as discussed in the next section.

Assigning Intensity Levels

We first consider how grayscale values on a video monitor can be distributed over the range between 0 and 1 so that the distribution corresponds to our perception of equal intensity intervals. We perceive relative light intensities the same way that we perceive relative sound intensities: on a logarithmic scale. This means that if the ratio of two intensities is the same as the ratio of two other intensities, we perceive the difference between each pair of intensities to be the same. As an example, we perceive the difference between intensities 0.20 and 0.22 to be the same as the difference between 0.80 and 0.88. Therefore, to display $n + 1$ successive intensity levels with equal perceived brightness, the intensity levels on the monitor should be spaced so that the ratio of successive intensities is constant:

$$\frac{I_1}{I_0} = \frac{I_2}{I_1} = \cdots = \frac{I_n}{I_{n-1}} = r \qquad (14\text{-}20)$$

Here, we denote the lowest level that can be displayed on the monitor as I_0 and the highest as I_n. Any intermediate intensity can then be expressed in terms of I_0 as

$$I_k = r^k I_0 \qquad (14\text{-}21)$$

We can calculate the value of r, given the values of I_0 and n for a particular system, by substituting $k = n$ in the preceding expression. Since $I_n = 1$, we have

$$r = \left(\frac{1}{I_0}\right)^{1/n} \qquad (14\text{-}22)$$

Thus, the calculation for I_k in Eq. 14-21 can be rewritten as

$$I_k = I_0^{(n-k)/n} \qquad (14\text{-}23)$$

As an example, if $I_0 = 1/8$ for a system with $n = 3$, we have $r = 2$, and the four intensity values are 1/8, 1/4, 1/2, and 1.

The lowest intensity value I_0 depends on the characteristics of the monitor and is typically in the range from 0.005 to around 0.025. As we saw in Chapter 2, a "black" region displayed on a monitor will always have some intensity value above 0 due to reflected light from the screen phosphors. For a black-and-white monitor with 8 bits per pixel ($n = 255$) and $I_0 = 0.01$, the ratio of successive intensities is approximately $r = 1.0182$. The approximate values for the 256 intensities on this system are 0.0100, 0.0102, 0.0104, 0.0106, 0.0107, 0.0109, . . . , 0.9821, and 1.0000.

With a color system, we set up intensity levels for each component of the color model. Using the RGB model, for example, we can relate the blue component of intensity at level k to the lowest attainable blue value as in Eq. 14-21:

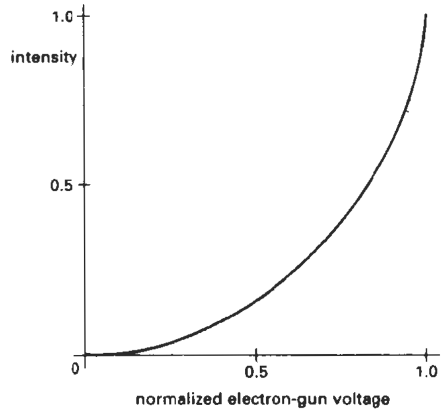


Figure 14-31
A typical monitor response curve, showing the displayed screen intensity as a function of normalized electron-gun voltage.

$$I_{Bk} = r_B^k I_{B0} \tag{14-24}$$

where

$$r_B = \left(\frac{1}{I_{B0}}\right)^{1/n} \tag{14-25}$$

and n is the number of intensity levels. Similar expressions hold for the other color components.

Gamma Correction and Video Lookup Tables

Another problem associated with the display of calculated intensities is the nonlinearity of display devices. Illumination models produce a linear range of intensities. The RGB color (0.25, 0.25, 0.25) obtained from a lighting model represents one-half the intensity of the color (0.5, 0.5, 0.5). Usually, these calculated intensities are then stored in an image file as integer values, with one byte for each of the three RGB components. This intensity file is also linear, so that a pixel with the value (64, 64, 64) has one-half the intensity of a pixel with the value (128, 128, 128). A video monitor, however, is a nonlinear device. If we set the voltages for the electron gun proportional to the linear pixel values, the displayed intensities will be shifted according to the **monitor response curve** shown in Fig. 14-31.

To correct for monitor nonlinearities, graphics systems use a **video lookup table** that adjusts the linear pixel values. The monitor response curve is described by the exponential function

$$I = aV^{\gamma} \tag{14-26}$$

Parameter I is the displayed intensity, and parameter V is the input voltage. Values for parameters a and γ depend on the characteristics of the monitor used in the graphics system. Thus, if we want to display a particular intensity value I, the correct voltage value to produce this intensity is

$$V = \left(\frac{I}{a}\right)^{1/\gamma} \tag{14-27}$$

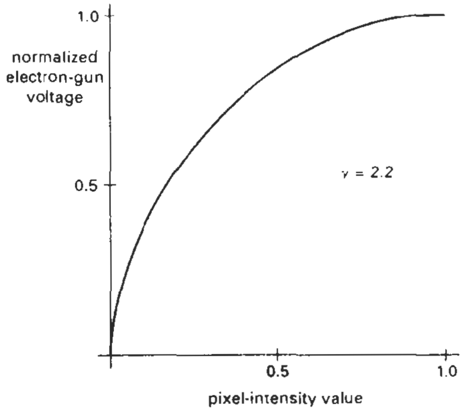


Figure 14-32
A video lookup correction curve for mapping pixel intensities to electron-gun voltages using gamma correction with $\gamma = 2.2$. Values for both pixel intensity and monitor voltages are normalized on the interval 0 to 1.

This calculation is referred to as **gamma correction** of intensity. Monitor gamma values are typically between 2.0 and 3.0. The National Television System Committee (NTSC) signal standard is $\gamma = 2.2$. Figure 14-32 shows a gamma-correction curve using the NTSC gamma value. Equation 14-27 is used to set up the video lookup table that converts integer pixel values in the image file to values that control the electron-gun voltages.

We can combine gamma correction with logarithmic intensity mapping to produce a lookup table that contains both conversions. If I is an input intensity value from an illumination model, we first locate the nearest intensity I_k from a table of values created with Eq. 14-20 or Eq. 14-23. Alternatively, we could determine the level number for this intensity value with the calculation

$$k = \text{round}\left(\log_r \frac{I}{I_0}\right) \tag{14-28}$$

then we compute the intensity value at this level using Eq. 14-23. Once we have the intensity value I_k, we can calculate the electron-gun voltage:

$$V_k = \left(\frac{I_k}{a}\right)^{1/\gamma} \tag{14-29}$$

Values V_k can then be placed in the lookup tables, and values for k would be stored in the frame-buffer pixel positions. If a particular system has no lookup table, computed values for V_k can be stored directly in the frame buffer. The combined conversion to a logarithmic intensity scale followed by calculation of the V_k using Eq.14-29 is also sometimes referred to as gamma correction.

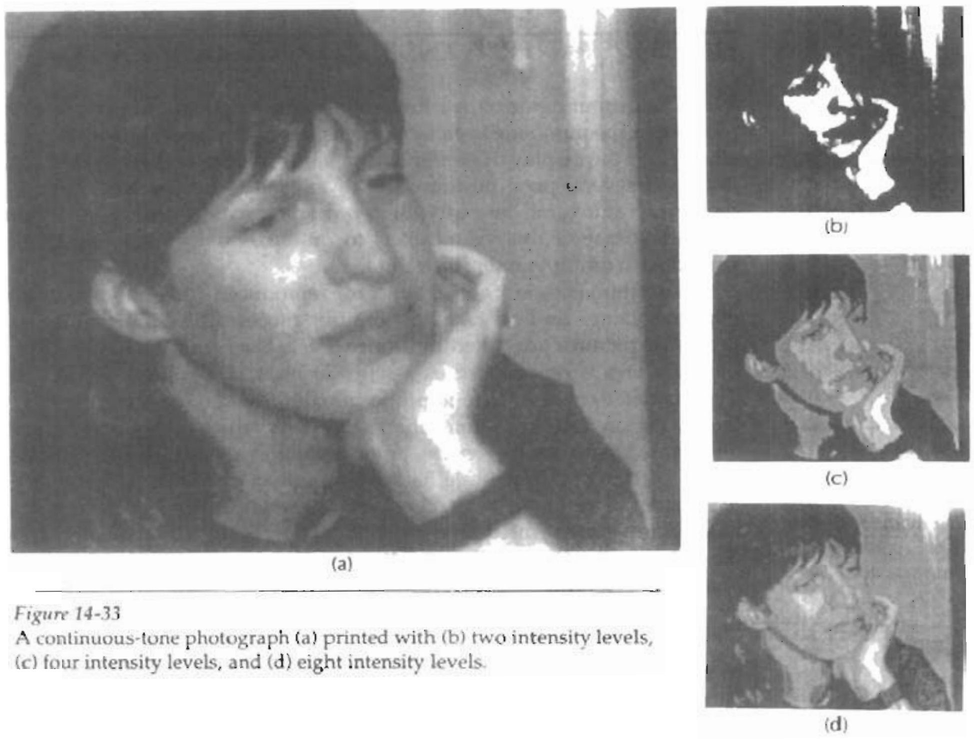


Figure 14-33
A continuous-tone photograph (a) printed with (b) two intensity levels, (c) four intensity levels, and (d) eight intensity levels.

If the video amplifiers of a monitor are designed to convert the linear input pixel values to electron-gun voltages, we cannot combine the two intensity-conversion processes. In this case, gamma correction is built into the hardware, and the logarithmic values I_k must be precomputed and stored in the frame buffer (or the color table).

Displaying Continuous-Tone Images

High-quality computer graphics systems generally provide 256 intensity levels for each color component, but acceptable displays can be obtained for many applications with fewer levels. A four-level system provides minimum shading capability for continuous-tone images, while photorealistic images can be generated on systems that are capable of from 32 to 256 intensity levels per pixel.

Figure 14-33 shows a continuous-tone photograph displayed with various intensity levels. When a small number of intensity levels are used to reproduce a continuous-tone image, the borders between the different intensity regions (called *contours*) are clearly visible. In the two-level reproduction, the features of the photograph are just barely identifiable. Using four intensity levels, we begin to identify the original shading patterns, but the contouring effects are glaring. With eight intensity levels, contouring effects are still obvious, but we begin to have a better indication of the original shading. At 16 or more intensity levels, contouring effects diminish and the reproductions are very close to the original. Reproductions of continuous-tone images using more than 32 intensity levels show only very subtle differences from the original.

14-4 HALFTONE PATTERNS AND DITHERING TECHNIQUES

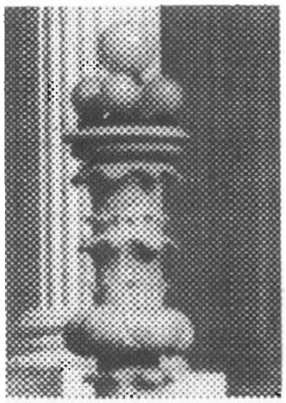


Figure 14-34
An enlarged section of a photograph reproduced with a halftoning method, showing how tones are represented with varying size dots.

When an output device has a limited intensity range, we can create an apparent increase in the number of available intensities by incorporating multiple pixel positions into the display of each intensity value. When we view a small region consisting of several pixel positions, our eyes tend to integrate or average the fine detail into an overall intensity. Bilevel monitors and printers, in particular, can take advantage of this visual effect to produce pictures that appear to be displayed with multiple intensity values.

Continuous-tone photographs are reproduced for publication in newspapers, magazines, and books with a printing process called **halftoning**, and the reproduced pictures are called **halftones**. For a black-and-white photograph, each intensity area is reproduced as a series of black circles on a white background. The diameter of each circle is proportional to the darkness required for that intensity region. Darker regions are printed with larger circles, and lighter regions are printed with smaller circles (more white area). Figure 14-34 shows an enlarged section of a gray-scale halftone reproduction. Color halftones are printed using dots of various sizes and colors, as shown in Fig. 14-35. Book and magazine halftones are printed on high-quality paper using approximately 60 to 80 circles of varying diameter per centimeter. Newspapers use lower-quality paper and lower resolution (about 25 to 30 dots per centimeter).

Halftone Approximations

In computer graphics, halftone reproductions are approximated using rectangular pixel regions, called *halftone patterns* or *pixel patterns*. The number of intensity

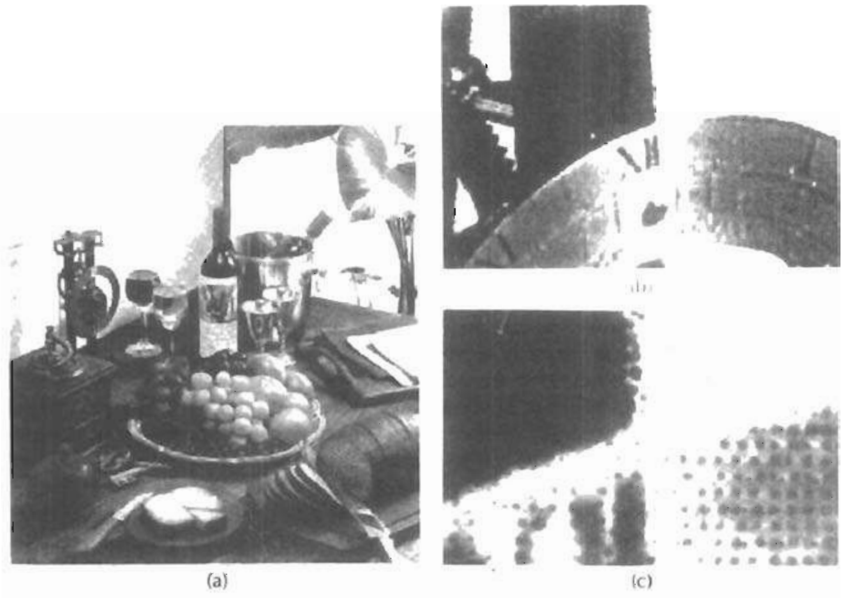


Figure 14-35
Color halftone dot patterns. The top half of the clock in the color halftone (a) is enlarged by a factor of 10 in (b) and by a factor of 50 in (c).

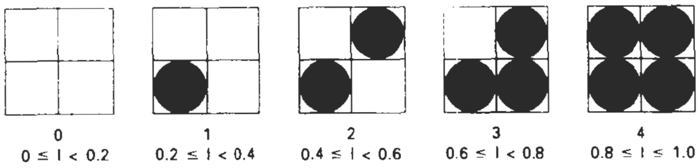


Figure 14-36
A 2 by 2 pixel grid used to display five intensity levels on a bilevel system. The intensity values that would be mapped to each grid are listed below each pixel pattern.

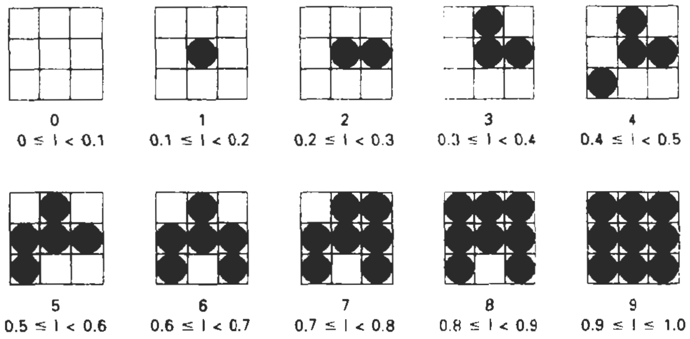


Figure 14-37
A 3 by 3 pixel grid can be used to display 10 intensities on a bilevel system. The intensity values that would be mapped to each grid are listed below each pixel pattern.

levels that we can display with this method depends on how many pixels we include in the rectangular grids and how many levels a system can display. With n by n pixels for each grid on a bilevel system, we can represent $n^2 + 1$ intensity levels. Figure 14-36 shows one way to set up pixel patterns to represent five intensity levels that could be used with a bilevel system. In pattern 0, all pixels are turned off; in pattern 1, one pixel is turned on; and in pattern 4, all four pixels are turned on. An intensity value I in a scene is mapped to a particular pattern according to the range listed below each grid shown in the figure. Pattern 0 is used for $0 \le I < 0.2$, pattern 1 for $0.2 \le I < 0.4$, and pattern 4 is used for $0.8 \le I \le 1.0$.

With 3 by 3 pixel grids on a bilevel system, we can display 10 intensity levels. One way to set up the 10 pixel patterns for these levels is shown in Fig. 14-37. Pixel positions are chosen at each level so that the patterns approximate the increasing circle sizes used in halftone reproductions. That is, the "on" pixel positions are near the center of the grid for lower intensity levels and expand outward as the intensity level increases.

For any pixel-grid size, we can represent the pixel patterns for the various possible intensities with a "mask" of pixel position numbers. As an example, the following mask can be used to generate the nine 3 by 3 grid patterns for intensity levels above 0 shown in Fig. 14-37.

$$\begin{bmatrix} 8 & 3 & 7 \\ 5 & 1 & 2 \\ 4 & 9 & 6 \end{bmatrix} \tag{14-30}$$

To display a particular intensity with level number k, we turn on each pixel whose position number is less than or equal to k.

Although the use of n by n pixel patterns increases the number of intensities that can be displayed, they reduce the resolution of the displayed picture by a factor of $1/n$ along each of the x and y axes. A 512 by 512 screen area, for instance, is reduced to an area containing 256 by 256 intensity points with 2 by 2 grid patterns. And with 3 by 3 patterns, we would reduce the 512 by 512 area to 128 intensity positions along each side.

Another problem with pixel grids is that subgrid patterns become apparent as the grid size increases. The grid size that can be used without distorting the intensity variations depends on the size of a displayed pixel. Therefore, for systems with lower resolution (fewer pixels per centimeter), we must be satisfied with fewer intensity levels. On the other hand, high-quality displays require at least 64 intensity levels. This means that we need 8 by 8 pixel grids. And to achieve a resolution equivalent to that of halftones in books and magazines, we must display 60 dots per centimeter. Thus, we need to be able to display $60 \times 8 = 480$ dots per centimeter. Some devices, for example high-quality film recorders, are able to display this resolution.

Pixel-grid patterns for halftone approximations must also be constructed to minimize contouring and other visual effects not present in the original scene. Contouring can be minimized by evolving each successive grid pattern from the previous pattern. That is, we form the pattern at level k by adding an "on" position to the grid pattern at level $k - 1$. Thus, if a pixel position is on for one grid level, it is on for all higher levels (Figs. 14-36 and 14-37). We can minimize the introduction of other visual effects by avoiding symmetrical patterns. With a 3 by 3 pixel grid, for instance, the third intensity level above zero would be better represented by the pattern in Fig. 14-38(a) than by any of the symmetrical arrangements in Fig. 14-38(b). The symmetrical patterns in this figure would produce vertical, horizontal, or diagonal streaks in any large area shaded with intensity level 3. For hard-copy output on devices such as film recorders and some printers, isolated pixels are not effectly reproduced. Therefore, a grid pattern with a single "on" pixel or one with isolated "on" pixels, as in Fig. 14-39, should be avoided.

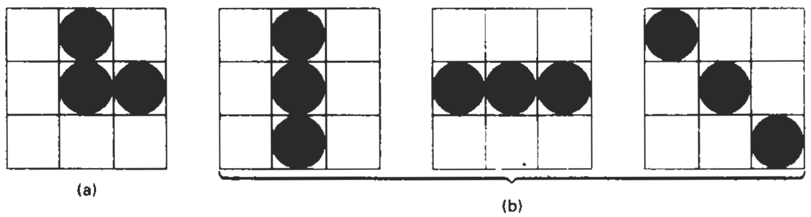


Figure 14-38
For a 3 by 3 pixel grid, pattern (a) is to be preferred to the patterns in (b) for representing the third intensity level above 0.

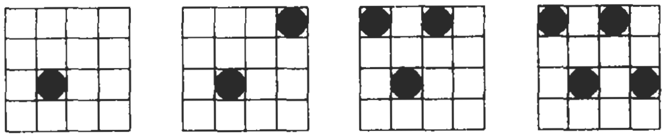


Figure 14-39
Halftone grid patterns with isolated pixels that cannot be effectively reproduced on some hard-copy devices.

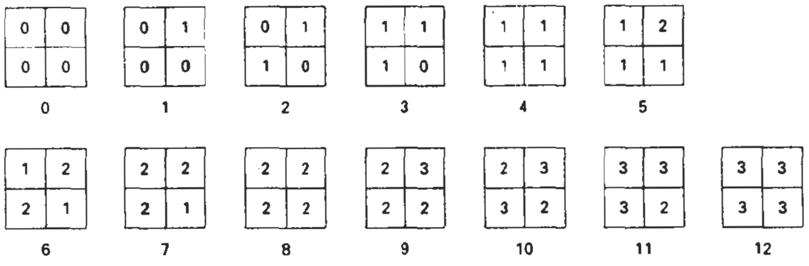


Figure 14-40
Intensity levels 0 through 12 obtained with halftone approximations using 2 by 2 pixel grids on a four-level system.

Halftone approximations also can be used to increase the number of intensity options on systems that are capable of displaying more than two intensities per pixel. For example, on a system that can display four intensity levels per pixel, we can use 2 by 2 pixel grids to extend the available intensity levels from 4 to 13. In Fig. 14-36, the four grid patterns above zero now represent several levels each, since each pixel position can display three intensity values above zero. Figure 14-40 shows one way to assign the pixel intensities to obtain the 13 distinct levels. Intensity levels for individual pixels are labeled 0 through 3, and the overall levels for the system are labeled 0 through 12.

Similarly, we can use pixel-grid patterns to increase the number of intensities that can be displayed on a color system. As an example, suppose we have a three-bit per pixel RGB system. This gives one bit per color gun in the monitor, providing eight colors (including black and white). Using 2 by 2 pixel-grid patterns, we now have 12 phosphor dots that can be used to represent a particular color value, as shown in Fig. 14-41. Each of the three RGB colors has four phosphor dots in the pattern, which allows five possible settings per color. This gives us a total of 125 different color combinations.

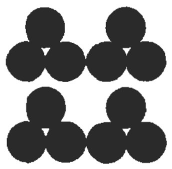


Figure 14-41
An RGB 2 by 2 pixel-grid pattern.

Dithering Techniques

The term **dithering** is used in various contexts. Primarily, it refers to techniques for approximating halftones without reducing resolution, as pixel-grid patterns do. But the term is also applied to halftone-approximation methods using pixel grids, and sometimes it is used to refer to color halftone approximations only.

Random values added to pixel intensities to break up contours are often referred to as *dither noise*. Various algorithms have been used to generate the random distributions. The effect is to add noise over an entire picture, which tends to soften intensity boundaries.

Ordered-dither methods generate intensity variations with a one-to-one mapping of points in a scene to the display pixels. To obtain n^2 intensity levels, we set up an n by n dither matrix D_n, whose elements are distinct positive integers in the range 0 to $n^2 - 1$. For example, we can generate four intensity levels with

$$D_2 = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix} \tag{14-31}$$

and we can generate nine intensity levels with

$$D_3 = \begin{bmatrix} 7 & 2 & 6 \\ 4 & 0 & 1 \\ 3 & 8 & 5 \end{bmatrix} \tag{14-32}$$

The matrix elements for D_2 and D_3 are in the same order as the pixel mask for setting up 2 by 2 and 3 by 3 pixel grids, respectively. For a bilevel system, we then determine display intensity values by comparing input intensities to the matrix elements. Each input intensity is first scaled to the range $0 \leq I \leq n^2$. If the intensity I is to be applied to screen position (x, y), we calculate row and column numbers for the dither matrix as

$$i = (x \bmod n) + 1, \qquad j = (y \bmod n) + 1 \tag{14-33}$$

If $I > D_n(i,j)$, we turn on the pixel at position (x, y). Otherwise, the pixel is not turned on.

Elements of the dither matrix are assigned in accordance with the guidelines discussed for pixel grids. That is, we want to minimize added visual effect in a displayed scene. Order dither produces constant-intensity areas identical to those generated with pixel-grid patterns when the values of the matrix elements correspond to the grid mask. Variations from the pixel-grid displays occur at boundaries of the intensity levels.

Typically, the number of intensity levels is taken to be a multiple of 2. Higher-order dither matrices are then obtained from lower-order matrices with the recurrence relation:

$$D_n = \begin{bmatrix} 4D_{n/2} + D_2(1,1)U_{n/2} & 4D_{n/2} + D_2(1,2)U_{n/2} \\ 4D_{n/2} + D_2(2,1)U_{n/2} & 4D_{n/2} + D_2(2,2)U_{n/2} \end{bmatrix} \tag{14-34}$$

assuming $n \geq 4$. Parameter $U_{n/2}$ is the "unity" matrix (all elements are 1). As an example, if D_2 is specified as in Eq. 14-31, then recurrence relation 14-34 yields

$$D_4 = \begin{bmatrix} 15 & 7 & 13 & 5 \\ 3 & 11 & 1 & 9 \\ 12 & 4 & 14 & 6 \\ 0 & 8 & 2 & 10 \end{bmatrix} \tag{14-35}$$

Another method for mapping a picture with m by n points to a display area with m by n pixels is *error diffusion*. Here, the error between an input intensity value and the displayed pixel intensity level at a given position is dispersed, or diffused, to pixel positions to the right and below the current pixel position. Starting with a matrix **M** of intensity values obtained by scanning a photograph, we want to construct an array I of pixel intensity values for an area of the screen. We do this by first scanning across the rows of **M**, from left to right, top to bottom, and determining the nearest available pixel-intensity level for each element of **M**. Then the error between the value stored in matrix **M** and the displayed intensity level at each pixel position is distributed to neighboring elements in **M**, using the following simplified algorithm:

```
for (i=0; i<m; i++)
        for (j=0; j<n; j++) {
                /* Determine the available intensity level $I_k$ */
                /* that is closest to the value $M_{i,j}$. */
                $I_{i,j} := I_k$;
                err := $M_{i,j} - I_{i,j}$;
                $M_{i,j+1} := M_{i,j+1} + \alpha \cdot$ err;
                $M_{i+1,j-1} := M_{i+1,j-1} + \beta \cdot$ err;
                $M_{i+1,j} := M_{i+1,j} + \gamma \cdot$ err;
                $M_{i+1,j+1} := M_{i+1,j+1} + \delta \cdot$ err;
        }
```

Once the elements of matrix **I** have been assigned intensity-level values, we then map the matrix to some area of a display device, such as a printer or video monitor. Of course, we cannot disperse the error past the last matrix column ($j = n$) or below the last matrix row ($i = m$). For a bilevel system, the available intensity levels are 0 and 1. Parameters for distributing the error can be chosen to satisfy the following relationship

$$\alpha + \beta + \gamma + \delta \leq 1 \tag{14-36}$$

One choice for the error-diffusion parameters that produces fairly good results is $(\alpha, \beta, \gamma, \delta) = (7/16, 3/16, 5/16, 1/16)$. Figure 14-42 illustrates the error distribution using these parameter values. Error diffusion sometimes produces "ghosts" in a picture by repeating, or echoing, certain parts of the picture, particularly with facial features such as hairlines and nose outlines. Ghosting can be re-

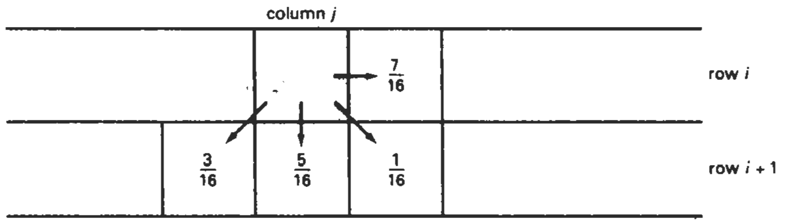


Figure 14-42
Fraction of intensity error that can be distributed to neighboring pixel positions using an error-diffusion scheme.

34	48	40	32	29	15	23	31
42	58	56	53	21	5	7	10
50	62	61	45	13	1	2	18
38	46	54	37	25	17	9	26
28	14	22	30	35	49	41	33
20	4	6	11	43	59	57	52
12	0	3	19	51	63	60	44
24	16	8	27	39	47	55	36

Figure 14-43
One possible distribution scheme for dividing the intensity array into 64 dot-diffusion classes, numbered from 0 through 63.

duced by choosing values for the error-diffusion parameters that sum to a value less than 1 and by rescaling the matrix values after the dispersion of errors. One way to rescale is to multiply all elements of **M** by 0.8 and then add 0.1. Another method for improving picture quality is to alternate the scanning of matrix rows from right to left and left to right.

A variation on the error-diffusion method is *dot diffusion*. In this method, the m by n array of intensity values is divided into 64 classes numbered from 0 to 63, as shown in Fig. 14-43. The error between a matrix value and the displayed intensity is then distributed only to those neighboring matrix elements that have a larger class number. Distribution of the 64 class numbers is based on minimizing the number of elements that are completely surrounded by elements with a lower class number, since this would tend to direct all errors of the surrounding elements to that one position.

14-5 POLYGON-RENDERING METHODS

In this section, we consider the application of an illumination model to the rendering of standard graphics objects: those formed with polygon surfaces. The objects are usually polygon-mesh approximations of curved-surface objects, but they may also be polyhedra that are not curved-surface approximations. Scan-line algorithms typically apply a lighting model to obtain polygon surface rendering in one of two ways. Each polygon can be rendered with a single intensity, or the intensity can be obtained at each point of the surface using an interpolation scheme.

Constant-Intensity Shading

A fast and simple method for rendering an object with polygon surfaces is **constant-intensity shading**, also called **flat shading**. In this method, a single intensity is calculated for each polygon. All points over the surface of the polygon are then displayed with the same intensity value. Constant shading can be useful for quickly displaying the general appearance of a curved surface, as in Fig. 14-47.

In general, flat shading of polygon facets provides an accurate rendering for an object if all of the following assumptions are valid:

- The object is a polyhedron and is not an approximation of an object with a curved surface.

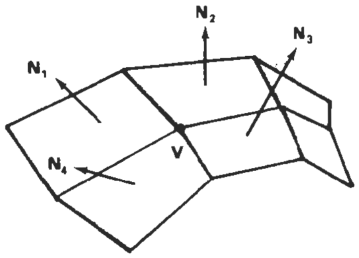


Figure 14-44
The normal vector at vertex **V** is calculated as the average of the surface normals for each polygon sharing that vertex.

- All light sources illuminating the object are sufficiently far from the surface so that **N** · **L** and the attenuation function are constant over the surface.
- The viewing position is sufficiently far from the surface so that **V** · **R** is constant over the surface.

Even if all of these conditions are not true, we can still reasonably approximate surface-lighting effects using small polygon facets with flat shading and calculate the intensity for each facet, say, at the center of the polygon.

Gouraud Shading

This **intensity-interpolation** scheme, developed by Gouraud and generally referred to as **Gouraud shading**, renders a polygon surface by linearly interpolating intensity values across the surface. Intensity values for each polygon are matched with the values of adjacent polygons along the common edges, thus eliminating the intensity discontinuities that can occur in flat shading.

Each polygon surface is rendered with Gouraud shading by performing the following calculations:

- Determine the average unit normal vector at each polygon vertex.
- Apply an illumination model to each vertex to calculate the vertex intensity.
- Linearly interpolate the vertex intensities over the surface of the polygon.

At each polygon vertex, we obtain a normal vector by averaging the surface normals of all polygons sharing that vertex, as illustrated in Fig. 14-44. Thus, for any vertex position V, we obtain the unit vertex normal with the calculation

$$\mathbf{N}_V = \frac{\sum_{k=1}^{n} \mathbf{N}_k}{\left| \sum_{k=1}^{n} \mathbf{N}_k \right|} \qquad (14\text{-}37)$$

Once we have the vertex normals, we can determine the intensity at the vertices from a lighting model.

Figure 14-45 demonstrates the next step: interpolating intensities along the polygon edges. For each scan line, the intensity at the intersection of the scan line with a polygon edge is linearly interpolated from the intensities at the edge endpoints. For the example in Fig. 14-45, the polygon edge with endpoint vertices at positions 1 and 2 is intersected by the scan line at point 4. A fast method for obtaining the intensity at point 4 is to interpolate between intensities I_1 and I_2 using only the vertical displacement of the scan line:

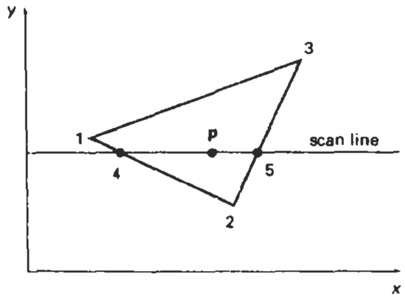


Figure 14-45
For Gouraud shading, the intensity at point 4 is linearly interpolated from the intensities at vertices 1 and 2. The intensity at point 5 is linearly interpolated from intensities at vertices 2 and 3. An interior point **p** is then assigned an intensity value that is linearly interpolated from intensities at positions 4 and 5.

$$I_4 = \frac{y_4 - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y_4}{y_1 - y_2} I_2 \tag{14-38}$$

Similarly, intensity at the right intersection of this scan line (point 5) is interpolated from intensity values at vertices 2 and 3. Once these bounding intensities are established for a scan line, an interior point (such as point **p** in Fig. 14-45) is interpolated from the bounding intensities at points 4 and 5 as

$$I_p = \frac{x_5 - x_p}{x_5 - x_4} I_4 + \frac{x_p - x_4}{x_5 - x_4} I_5 \tag{14-39}$$

Incremental calculations are used to obtain successive edge intensity values between scan lines and to obtain successive intensities along a scan line. As shown in Fig. 14-46, if the intensity at edge position (x, y) is interpolated as

$$I = \frac{y - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y}{y_1 - y_2} I_2 \tag{14-40}$$

then we can obtain the intensity along this edge for the next scan line, $y - 1$, as

$$I' = I + \frac{I_2 - I_1}{y_1 - y_2} \tag{14-41}$$

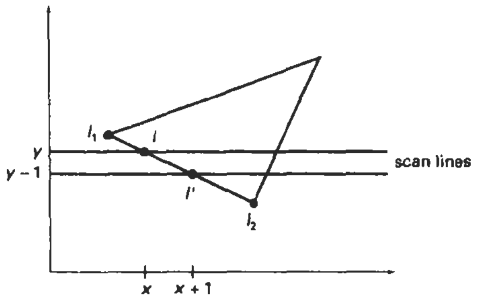


Figure 14-46
Incremental interpolation of intensity values along a polygon edge for successive scan lines.

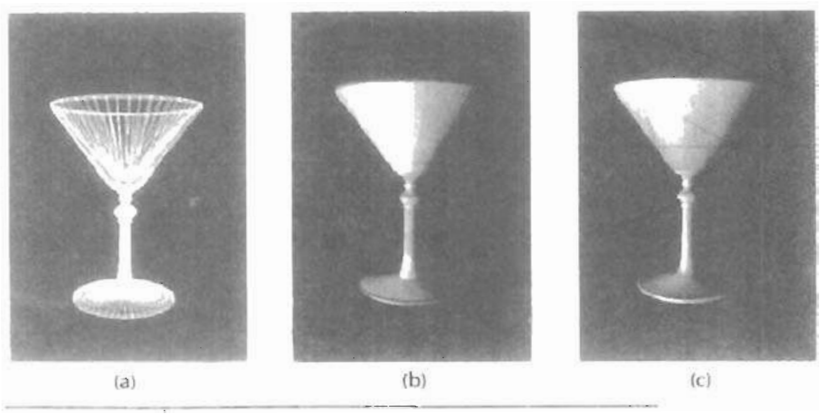


Figure 14-47
A polygon mesh approximation of an object (a) is rendered with flat shading (b) and with Gouraud shading (c).

Similar calculations are used to obtain intensities at successive horizontal pixel positions along each scan line.

When surfaces are to be rendered in color, the intensity of each color component is calculated at the vertices. Gouraud shading can be combined with a hidden-surface algorithm to fill in the visible polygons along each scan line. An example of an object shaded with the Gouraud method appears in Fig. 14-47.

Gouraud shading removes the intensity discontinuities associated with the constant-shading model, but it has some other deficiencies. Highlights on the surface are sometimes displayed with anomalous shapes, and the linear intensity interpolation can cause bright or dark intensity streaks, called Mach bands, to appear on the surface. These effects can be reduced by dividing the surface into a greater number of polygon faces or by using other methods, such as Phong shading, that require more calculations.

Phong Shading

A more accurate method for rendering a polygon surface is to interpolate normal vectors, and then apply the illumination model to each surface point. This method, developed by Phong Bui Tuong, is called **Phong shading,** or **normal-vector interpolation shading.** It displays more realistic highlights on a surface and greatly reduces the Mach-band effect.

A polygon surface is rendered using Phong shading by carrying out the following steps:

- Determine the average unit normal vector at each polygon vertex.
- Linearly interpolate the vertex normals over the surface of the polygon.
- Apply an illumination model along each scan line to calculate projected pixel intensities for the surface points.

Interpolation of surface normals along a polygon edge between two vertices is illustrated in Fig. 14-48. The normal vector **N** for the scan-line intersection point along the edge between vertices 1 and 2 can be obtained by vertically interpolating between edge endpoint normals:

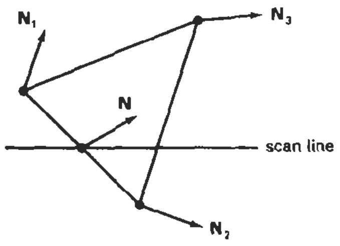


Figure 14-48
Interpolation of surface normals along a polygon edge

$$\mathbf{N} = \frac{y - y_2}{y_1 - y_2}\mathbf{N}_1 + \frac{y_1 - y}{y_1 - y_2}\mathbf{N}_2 \tag{14-42}$$

Incremental methods are used to evaluate normals between scan lines and along each individual scan line. At each pixel position along a scan line, the illumination model is applied to determine the surface intensity at that point.

Intensity calculations using an approximated normal vector at each point along the scan line produce more accurate results than the direct interpolation of intensities, as in Gouraud shading. The trade-off, however, is that Phong shading requires considerably more calculations.

Fast Phong Shading

Surface rendering with Phong shading can be speeded up by using approximations in the illumination-model calculations of normal vectors. **Fast Phong shading** approximates the intensity calculations using a Taylor-series expansion and triangular surface patches.

Since Phong shading interpolates normal vectors from vertex normals, we can express the surface normal **N** at any point (x, y) over a triangle as

$$\mathbf{N} = \mathbf{A}x + \mathbf{B}y + \mathbf{C} \tag{14-43}$$

where vectors **A, B,** and **C** are determined from the three vertex equations:

$$\mathbf{N}_k = \mathbf{A}x_k + \mathbf{B}y_k + \mathbf{C}, \qquad k = 1, 2, 3 \tag{14-44}$$

with (x_k, y_k) denoting a vertex position.

Omitting the reflectivity and attenuation parameters, we can write the calculation for light-source diffuse reflection from a surface point (x, y) as

$$\begin{aligned} I_{\text{diff}}(x, y) &= \frac{\mathbf{L} \cdot \mathbf{N}}{|\mathbf{L}|\,|\mathbf{N}|} \\ &= \frac{\mathbf{L} \cdot (\mathbf{A}x + \mathbf{B}y + \mathbf{C})}{|\mathbf{L}|\,|\mathbf{A}x + \mathbf{B}y + \mathbf{C}|} \\ &= \frac{(\mathbf{L} \cdot \mathbf{A})x + (\mathbf{L} \cdot \mathbf{B})y + \mathbf{L} \cdot \mathbf{C}}{|\mathbf{L}|\,|\mathbf{A}x + \mathbf{B}y + \mathbf{C}|} \end{aligned} \tag{14-45}$$

We can rewrite this expression in the form

$$I_{\text{diff}}(x, y) = \frac{ax + by + c}{(dx^2 + exy + fy^2 + gx + hy + i)^{1/2}} \tag{14-46}$$

where parameters such as a, b, c, and d are used to represent the various dot products. For example,

$$a = \frac{\mathbf{L} \cdot \mathbf{A}}{|\mathbf{L}|} \tag{14-47}$$

Finally, we can express the denominator in Eq. 14-46 as a Taylor-series expansion and retain terms up to second degree in x and y. This yields

$$I_{\text{diff}}(x, y) = T_5x^2 + T_4xy + T_3y^2 + T_2x + T_1y + T_0 \tag{14-48}$$

where each T_k is a function of parameters a, b, c, and so forth.

Using forward differences, we can evaluate Eq. 14-48 with only two additions for each pixel position (x, y) once the initial forward-difference parameters have been evaluated. Although fast Phong shading reduces the Phong-shading calculations, it still takes approximately twice as long to render a surface with fast Phong shading as it does with Gouraud shading. Normal Phong shading using forward differences takes about six to seven times longer than Gouraud shading.

Fast Phong shading for diffuse reflection can be extended to include specular reflections. Calculations similar to those for diffuse reflections are used to evaluate specular terms such as $(\mathbf{N} \cdot \mathbf{H})^{n_s}$ in the basic illumination model. In addition, we can generalize the algorithm to include polygons other than triangles and finite viewing positions.

14-6 RAY-TRACING METHODS

In Section 10-15, we introduced the notion of *ray casting*, where a ray is sent out from each pixel position to locate surface intersections for object modeling using constructive solid geometry methods. We also discussed the use of ray casting as a method for determining visible surfaces in a scene (Section 13-10). **Ray tracing** is an extension of this basic idea. Instead of merely looking for the visible surface for each pixel, we continue to bounce the ray around the scene, as illustrated in Fig. 14-49, collecting intensity contributions. This provides a simple and powerful rendering technique for obtaining global reflection and transmission effects. The basic ray-tracing algorithm also provides for visible-surface detection, shadow effects, transparency, and multiple light-source illumination. Many extensions to the basic algorithm have been developed to produce photorealistic displays. Ray-traced displays can be highly realistic, particularly for shiny objects, but they require considerable computation time to generate. An example of the global reflection and transmission effects possible with ray tracing is shown in Fig. 14-50.

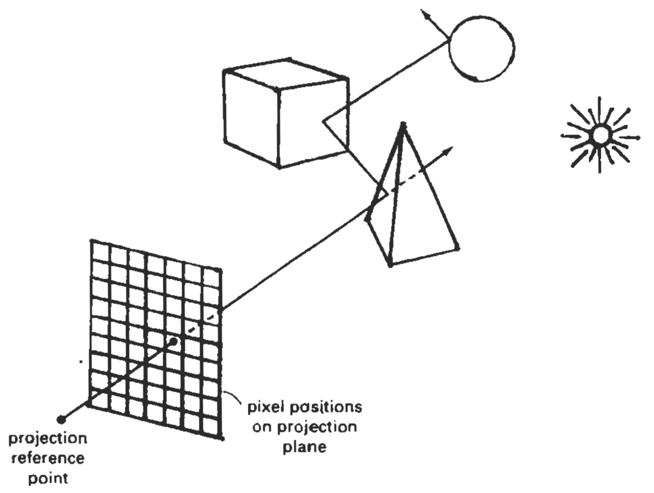

pixel positions on projection plane
projection reference point

Figure 14-49
Tracing a ray from the projection reference point through a pixel position with multiple reflections and transmissions.

Basic Ray-Tracing Algorithm

We first set up a coordinate system with the pixel positions designated in the xy plane. The scene description is given in this reference frame (Fig. 14-51). From the center of projection, we then determine a ray path that passes through the center of each screen-pixel position. Illumination effects accumulated along this ray path are then assigned to the pixel. This rendering approach is based on the principles of geometric optics. Light rays from the surfaces in a scene emanate in all directions, and some will pass through the pixel positions in the projection plane. Since there are an infinite number of ray paths, we determine the contributions to a particular pixel by tracing a light path backward from the pixel to the scene. We first consider the basic ray-tracing algorithm with one ray per pixel, which is equivalent to viewing the scene through a pinhole camera.

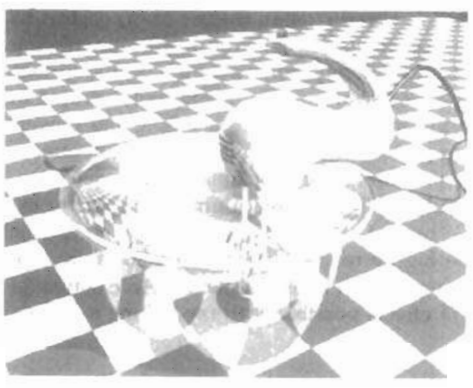


Figure 14-50
A ray-traced scene, showing global reflection and transmission illumination effects from object surfaces. *(Courtesy of Evans & Sutherland.)*

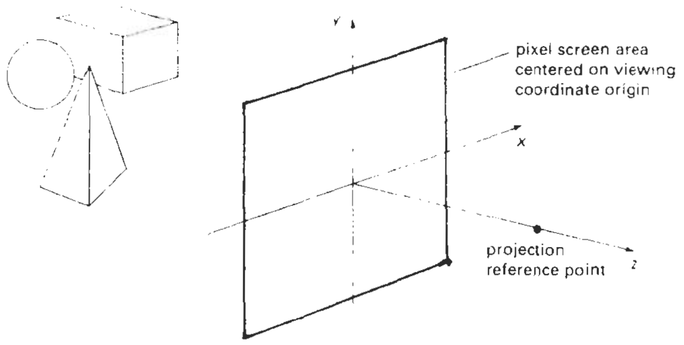

y
pixel screen area centered on viewing coordinate origin
x
projection reference point
z

Figure 14-51
Ray-tracing coordinate-reference frame.

For each pixel ray, we test each surface in the scene to determine if it is intersected by the ray. If a surface is intersected, we calculate the distance from the pixel to the surface-intersection point. The smallest calculated intersection distance identifies the visible surface for that pixel. We then reflect the ray off the visible surface along a specular path (angle of reflection equals angle of incidence). If the surface is transparent, we also send a ray through the surface in the refraction direction. Reflection and refraction rays are referred to as *secondary rays*.

This procedure is repeated for each secondary ray: Objects are tested for intersection, and the nearest surface along a secondary ray path is used to recursively produce the next generation of reflection and refraction paths. As the rays from a pixel ricochet through the scene, each successively intersected surface is added to a binary *ray-tracing tree*, as shown in Fig. 14-52. We use left branches in the tree to represent reflection paths, and right branches represent transmission paths. Maximum depth of the ray-tracing trees can be set as a user option, or it can be determined by the amount of storage available. A path in the tree is then terminated if it reaches the preset maximum or if the ray strikes a light source.

The intensity assigned to a pixel is then determined by accumulating the intensity contributions, starting at the bottom (terminal nodes) of its ray-tracing tree. Surface intensity from each node in the tree is attenuated by the distance from the "parent" surface (next node up the tree) and added to the intensity of the parent surface. Pixel intensity is then the sum of the attenuated intensities at the root node of the ray tree. If no surfaces are intersected by a pixel ray, the ray-tracing tree is empty and the pixel is assigned the intensity value of the background. If a pixel ray intersects a nonreflecting light source, the pixel can be assigned the intensity of the source, although light sources are usually placed beyond the path of the initial rays.

Figure 14-53 shows a surface intersected by a ray and the unit vectors needed for the reflected light-intensity calculations. Unit vector **u** is in the direction of the ray path. **N** is the unit surface normal, **R** is the unit reflection vector, **L** is the unit vector pointing to the light source, and **H** is the unit vector halfway between **V** (opposite to **u**) and **L**. The path along **L** is referred to as the **shadow ray**. If any object intersects the shadow ray between the surface and the point light

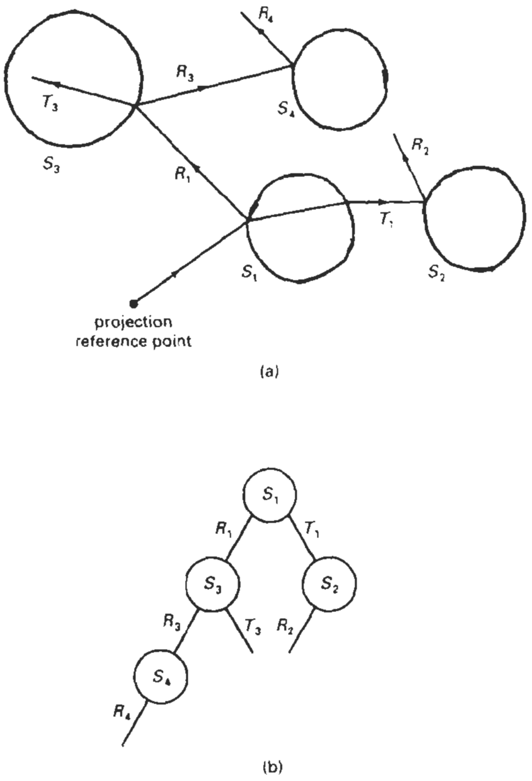


Figure 14-52
(a) Reflection and refraction ray paths through a scene for a screen pixel. (b) Binary ray-tracing tree for the paths shown in (a).

source, the surface is in shadow with respect to that source. Ambient light at the surface is calculated as $k_a I_a$; diffuse reflection due to the source is proportional to $k_d(\mathbf{N} \cdot \mathbf{L})$; and the specular-reflection component is proportional to $k_s(\mathbf{H} \cdot \mathbf{N})^{n_s}$. As discussed in Section 14-2, the specular-reflection direction for the secondary ray path **R** depends on the surface normal and the incoming ray direction:

$$\mathbf{R} = \mathbf{u} - (2\mathbf{u} \cdot \mathbf{N})\mathbf{N} \tag{14-49}$$

For a transparent surface, we also need to obtain intensity contributions from light transmitted through the material. We can locate the source of this contribution by tracing a secondary ray along the transmission direction **T**, as shown in Fig. 14-54. The unit transmission vector can be obtained from vectors **u** and **N** as

$$\mathbf{T} = \frac{\eta_i}{\eta_r}\mathbf{u} - \left(\cos\theta_r - \frac{\eta_i}{\eta_r}\cos\theta_i\right)\mathbf{N} \tag{14-50}$$

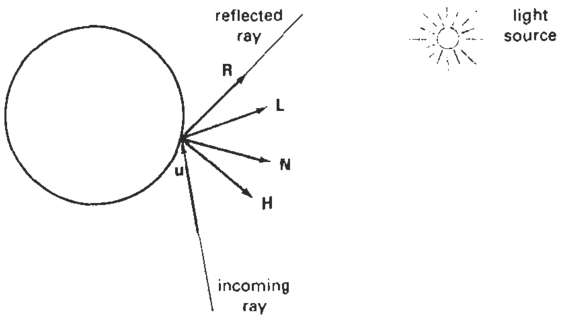


Figure 14-53
Unit vectors at the surface of an object intersected by an incoming ray along direction **u**.

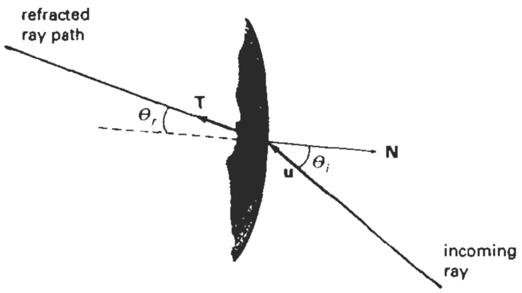


Figure 14-54
Refracted ray path **T** through a transparent material.

Parameters η_i and η_r are the indices of refraction in the incident material and the refracting material, respectively. Angle of refraction θ_r can be calculated from Snell's law:

$$\cos\theta_r = \sqrt{1 - \left(\frac{\eta_i}{\eta_r}\right)^2 (1 - \cos^2\theta_i)} \qquad (14\text{-}51)$$

Ray-Surface Intersection Calculations

A ray can be described with an initial position $\mathbf{P}_0$ and unit direction vector **u**, as illustrated in Fig. 14-55. The coordinates of any point **P** along the ray at a distance s from $\mathbf{P}_0$ is computed from the **ray equation**:

$$\mathbf{P} = \mathbf{P}_0 + s\mathbf{u} \qquad (14\text{-}52)$$

Initailly, $\mathbf{P}_0$ can be set to the position of the pixel on the projection plane, or it could be chosen to be the projection reference point. Unit vector **u** is initially ob-

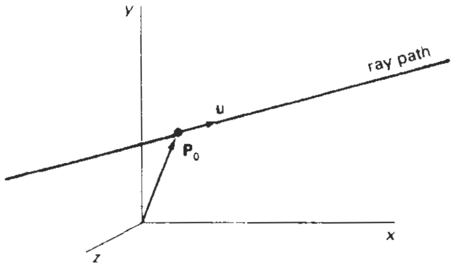


Figure 14-55
Describing a ray with an initial-position vector P_0 and unit direction vector **u**.

tained from the position of the pixel through which the ray passes and the projection reference point:

$$\mathbf{u} = \frac{\mathbf{P}_{\text{pix}} - \mathbf{P}_{\text{prp}}}{|\mathbf{P}_{\text{pix}} - \mathbf{P}_{\text{prp}}|} \tag{14-53}$$

At each intersected surface, vectors $\mathbf{P}_0$ and **u** are updated for the secondary rays at the ray-surface intersection point. For the secondary rays, reflection direction for **u** is **R** and the transmission direction is **T**. To locate surface intersections, we simultaneously solve the ray equation and the surface equation for the individual objects in the scene.

The simplest objects to ray trace are spheres. If we have a sphere of radius r and center position $\mathbf{P}_c$ (Fig. 14-56), then any point **P** on the surface must satisfy the sphere equation:

$$|\mathbf{P} - \mathbf{P}_c|^2 - r^2 = 0 \tag{14-54}$$

Substituting the ray equation 14-52, we have

$$|\mathbf{P}_0 - s\mathbf{u} - \mathbf{P}_c|^2 - r^2 = 0 \tag{14-55}$$

If we let $\Delta\mathbf{P} = \mathbf{P}_c - \mathbf{P}_0$ and expand the dot product, we obtain the quadratic equation

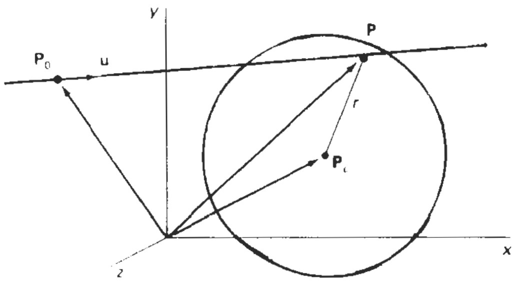


Figure 14-56
A ray intersecting a sphere with radius r centered on position $\mathbf{P}_c$.

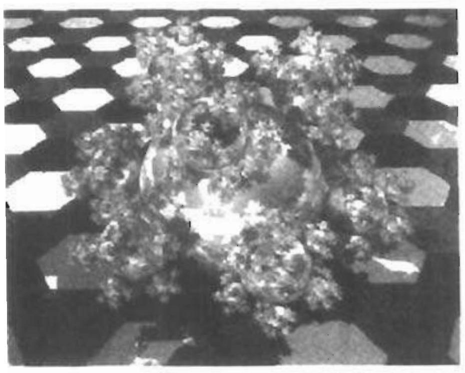


Figure 14-57
A "sphereflake" rendered with ray tracing using 7381 spheres and 3 light sources. (*Courtesy of Eric Haines, 3D/EYE Inc.*)

$$s^2 - 2(\mathbf{u} \cdot \Delta\mathbf{P})s + (|\Delta\mathbf{P}|^2 - r^2) = 0 \tag{14-56}$$

whose solution is

$$s = \mathbf{u} \cdot \Delta\mathbf{P} \pm \sqrt{(\mathbf{u} \cdot \Delta\mathbf{P})^2 - |\Delta\mathbf{P}|^2 + r^2} \tag{14-57}$$

If the discriminant is negative, the ray does not intersect the sphere. Otherwise, the surface-intersection coordinates are obtained from the ray equation 14-52 using the smaller of the two values from Eq. 14-57.

For small spheres that are far from the initial ray position, Eq. 14-57 is susceptible to roundoff errors. That is, if

$$r^2 << |\Delta\mathbf{P}|^2$$

we could lose the r^2 term in the precision error of $|\Delta\mathbf{P}|^2$. We can avoid this for most cases by rearranging the calculation for distance s as

$$s = \mathbf{u} \cdot \Delta\mathbf{P} \pm \sqrt{r^2 - |\Delta\mathbf{P} - (\mathbf{u} \cdot \Delta\mathbf{P})\mathbf{u}|^2} \tag{14-58}$$

Figure 14-57 shows a snowflake pattern of shiny spheres rendered with ray tracing to display global surface reflections.

Polyhedra require more processing than spheres to locate surface intersections. For that reason, it is often better to do an initial intersection test on a bounding volume. For example, Fig. 14-58 shows a polyhedron bounded by a sphere. If a ray does not intersect the sphere, we do not need to do any further testing on the polyhedron. But if the ray does intersect the sphere, we first locate "front" faces with the test

$$\mathbf{u} \cdot \mathbf{N} < 0 \tag{14-59}$$

where $\mathbf{N}$ is a surface normal. For each face of the polyhedron that satisifies inequality 14-59, we solve the plane equation

$$\mathbf{N} \cdot \mathbf{P} = -D \tag{14-60}$$

for surface position $\mathbf{P}$ that also satisfies the ray equation 14-52. Here, $\mathbf{N} = (A, B, C)$

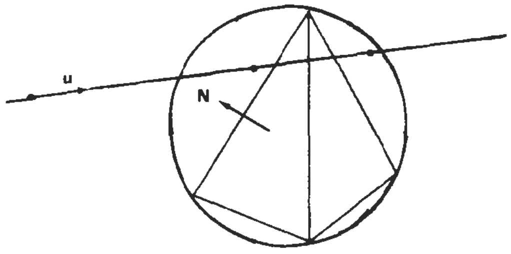


Figure 14-58
Polyhedron enclosed by a bounding sphere.

and D is the fourth plane parameter. Position $\mathbf{P}$ is both on the plane and on the ray path if

$$\mathbf{N} \cdot (\mathbf{P}_0 + s\mathbf{u}) = -D \tag{14-61}$$

And the distance from the initial ray position to the plane is

$$s = -\frac{D + \mathbf{N} \cdot \mathbf{P}_0}{\mathbf{N} \cdot \mathbf{u}} \tag{14-62}$$

This gives us a position on the infinite plane that contains the polygon face, but this position may not be inside the polygon boundaries (Fig. 14-59). So we need to perform an "inside-outside" test (Chapter 3) to determine whether the ray intersected this face of the polyhedron. We perform this test for each face satisfying inequality 14-59. The smallest distance s to an inside point identifies the intersected face of the polyhedron. If no intersection positions from Eq. 14-62 are inside points, the ray does not intersect the object.

Similar procedures are used to calculate ray-surface intersection positions for other objects, such as quadric or spline surfaces. We combine the ray equation with the surface definition and solve for parameter s. In many cases, numerical root-finding methods and incremental calculations are used to locate intersection

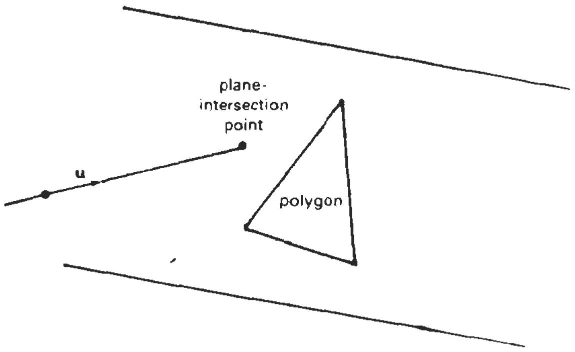


Figure 14-59
Ray intersection with the plane of a polygon.

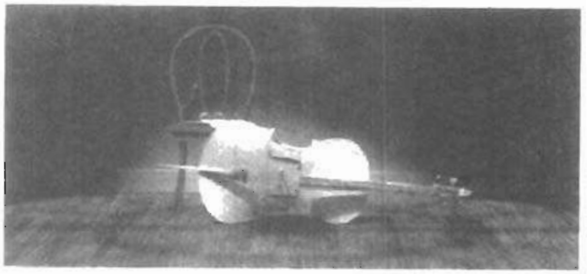


Figure 14-60
A ray-traced scene showing global reflection of surface-texture patterns. *(Courtesy of Sun Microsystems.)*

points over a surface. Figure 14-60 shows a ray-traced scene containing multiple objects and texture patterns.

Reducing Object-Intersection Calculations

Ray-surface intersection calculations can account for as much as 95 percent of the processing time in a ray tracer. For a scene with many objects, most of the processing time for each ray is spent checking objects that are not visible along the ray path. Therefore, several methods have been developed for reducing the processing time spent on these intersection calculations.

One method for reducing the intersection calculations is to enclose groups of adjacent objects within a bounding volume, such as a sphere or a box (Fig. 14-61). We can then test for ray intersections with the bounding volume. If the ray does not intersect the bounding object, we can eliminate the intersection tests with the enclosed surfaces. This approach can be extended to include a hierarchy of bounding volumes. That is, we enclose several bounding volumes within a larger volume and carry out the intersection tests hierarchically. First, we test the outer bounding volume; then, if necessary, we test the smaller inner bounding volumes; and so on.

Space-Subdivision Methods

Another way to reduce intersection calculations, is to use *space-subdivision methods.* We can enclose a scene within a cube, then we successively subdivide the cube until each subregion (cell) contains no more than a preset maximum number of surfaces. For example, we could require that each cell contain no more than one surface. If parallel- and vector-processing capabilities are available, the maximum number of surfaces per cell can be determined by the size of the vector

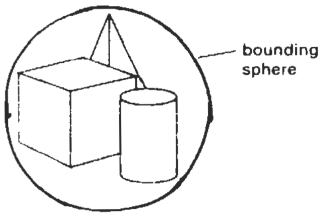


Figure 14-61
A group of objects enclosed within a bounding sphere.

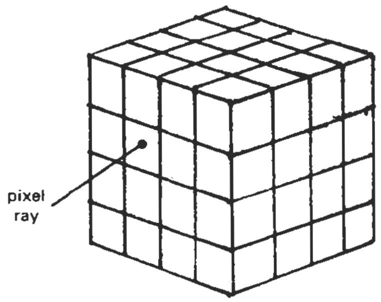


Figure 14-62
Ray intersection with a cube enclosing all objects in a scene.

registers and the number of processors. Space subdivision of the cube can be stored in an octree or in a binary-partition tree. In addition, we can perform a *uniform subdivision* by dividing the cube into eight equal-size octants at each step, or we can perform an *adaptive subdivision* and subdivide only those regions of the cube containing objects.

We then trace rays through the individual cells of the cube, performing intersection tests only within those cells containing surfaces. The first object surface intersected by a ray is the visible surface for that ray. There is a trade-off between the cell size and the number of surfaces per cell. If we set the maximum number of surfaces per cell too low, cell size can become so small that much of the savings in reduced intersection tests goes into cell-traversal processing.

Figure 14-62 illustrates the intersection of a pixel ray with the front face of the cube enclosing a scene. Once we calculate the intersection point on the front face of the cube, we determine the initial cell intersection by checking the intersection coordinates against the cell boundary positions. We then need to process the ray through the cells by determining the entry and exit points (Fig. 14-63) for each cell traversed by the ray until we intersect an object surface or exit the cube enclosing the scene.

Given a ray direction $\mathbf{u}$ and a ray entry position $\mathbf{P}_{\text{in}}$ for a cell, the potential exit faces are those for which

$$\mathbf{u} \cdot \mathbf{N}_k > 0 \tag{14-63}$$

If the normal vectors for the cell faces in Fig. 14-63 are aligned with the coordinates axes, then

$$N_k = \begin{cases} (\pm 1, 0, 0) \\ (0, \pm 1, 0) \\ (0, 0, \pm 1) \end{cases}$$

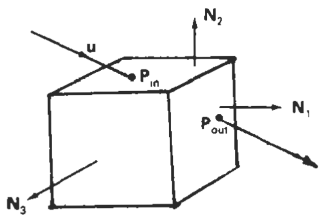


Figure 14-63
Ray traversal through a subregion (cell) of a cube enclosing a scene.

and we only need to check the sign of each component of **u** to determine the three candidate exit planes. The exit position on each candidate plane is obtained from the ray equation:

$$\mathbf{P}_{\text{out},k} = \mathbf{P}_{\text{in}} + s_k\mathbf{u} \qquad (14\text{-}64)$$

where s_k is the distance along the ray from $\mathbf{P}_{\text{in}}$ to $\mathbf{P}_{\text{out},k}$. Substituting the ray equation into the plane equation for each cell face:

$$\mathbf{N}_k \cdot \mathbf{P}_{\text{out},k} = -D \qquad (14\text{-}65)$$

we can solve for the ray distance to each candidate exit face as

$$s_k = \frac{-D - \mathbf{N}_k \cdot \mathbf{P}_{\text{in}}}{\mathbf{N}_k \cdot \mathbf{u}} \qquad (14\text{-}66)$$

and then select smallest s_k. This calculation can be simplified if the normal vectors $\mathbf{N}_k$ are aligned with the coordinate axes. For example, if a candidate normal vector is (1, 0, 0), then for that plane we have

$$s_k = \frac{x_k - x_0}{u_x} \qquad (14\text{-}67)$$

where $\mathbf{u} = (u_x, u_y, u_z)$, and x_k is the value of the right boundary face for the cell.

Various modifications can be made to the cell-traversal procedures to speed up the processing. One possibility is to take a trial exit plane k as the one perpendicular to the direction of the largest component of **u**. The sector on the trial plane (Fig. 14-64) containing $\mathbf{P}_{\text{out},k}$ determines the true exit plane. If the intersection point $\mathbf{P}_{\text{out},k}$ is in sector 0, the trial plane is the true exit plane and we are done. If the intersection point is sector 1, the true exit plane is the top plane and we simply need to calculate the exit point on the top boundary of the cell. Similarly, sector 3 identifies the bottom plane as the true exit plane; and sectors 4 and 2 identify the true exit plane as the left and right cell planes, respectively. When the trial exit point falls in sector 5, 6, 7, or 8, we need to carry out two additional intersection calculations to identify the true exit plane. Implementation of these methods on parallel vector machines provides further improvements in performance.

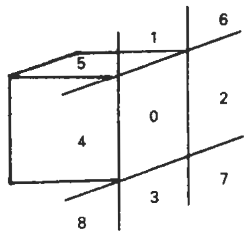


Figure 14-64
Sectors of the trial exit plane.

The scene in Fig. 14-65 was ray traced using space-subdivision methods. Without space subdivision, the ray-tracing calculations took 10 times longer. Eliminating the polygons also speeded up the processing. For a scene containing 2048 spheres and no polygons, the same algorithm executed 46 times faster than the basic ray tracer.

Figure 14-66 illustrates another ray-traced scene using spatial subdivision and parallel-processing methods. This image of Rodin's Thinker was ray traced with over 1.5 million rays in 24 seconds.

The scene shown in Fig. 14-67 was rendered with a *light-buffer technique*, a form of spatial partitioning. Here, a cube is centered on each point light source, and each side of the cube is partitioned with a grid of squares. A sorted list of objects that are visible to the light through each square is then maintained by the ray tracer to speed up processing of shadow rays. To determine surface-illumination effects, the square for each shadow ray is computed and the shadow ray is then processed against the list of objects for that square.

Intersection tests in ray-tracing programs can also be reduced with directional subdivision procedures, by considering sectors that contain a bundle of rays. Within each sector, we can sort surfaces in depth order, as in Fig. 14-68. Each ray then only needs to test objects within the sector that contains that ray.

Antialiased Ray Tracing

Two basic techniques for antialiasing in ray-tracing algorithms are *supersampling* and *adaptive sampling*. Sampling in ray tracing is an extension of the sampling methods we discussed in Chapter 4. In supersampling and adaptive sampling,

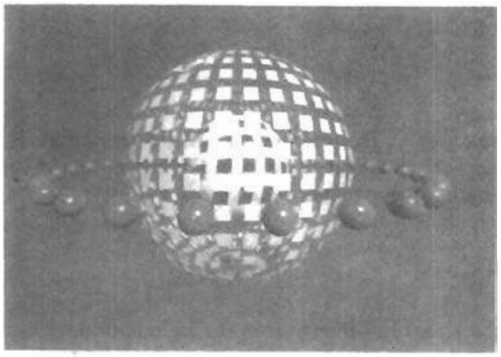


Figure 14-65
A parallel ray-traced scene containing 37 spheres and 720 polygon surfaces. The ray-tracing algorithm used 9 rays per pixel and a tree depth of 5. Spatial subdivision methods processed the scene 10 times faster than the basic ray-tracing algorithm on an Alliant FX/8. *(Courtesy of Lee-Hian Quek, Information Technology Institute, Republic of Singapore.)*

Figure 14-66
This ray-traced scene took 24 seconds to render on a Kendall Square Research KSR1 parallel computer with 32 processors. Rodin's Thinker was modeled with 3036 primitives. Two light sources and one primary ray per pixel were used to obtain the global illumination effects from the 1,675,776 rays processed. *(Courtesy of M. J. Keates and R. J. Hubbold, Department of Computer Science, University of Manchester.)*

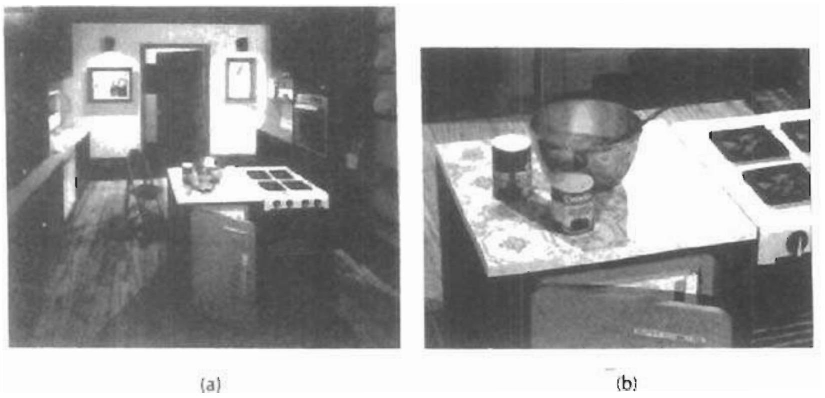


Figure 14-67
A room scene illuminated with 5 light sources (a) was rendered using the ray-tracing light-buffer technique to process shadow rays. A closeup (b) of part of the room shown in (a) illustrates the global illumination effects. The room is modeled with 1298 polygons, 4 spheres, 76 cylinders, and 35 quadrics. Rendering time was 246 minutes on a VAX 11/780, compared to 602 minutes without using light buffers. *(Courtesy of Eric Haines and Donald P. Greenberg, Program of Computer Graphics, Cornell University.)*

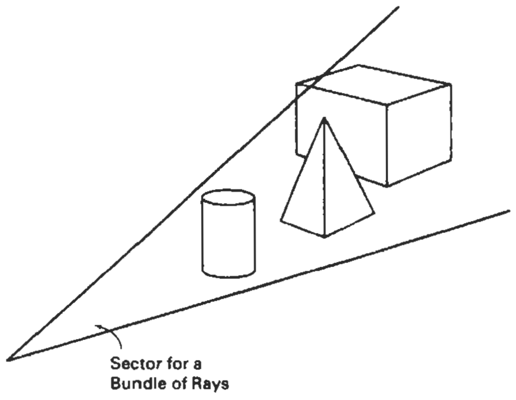


Figure 14-68
Directional subdivision of space. All rays in this sector only need to test the surfaces within the sector in depth order.

the pixel is treated as a finite square area instead of a single point. Supersampling uses multiple, evenly spaced rays (samples) over each pixel area. Adaptive sampling uses unevenly spaced rays in some regions of the pixel area. For example, more rays can be used near object edges to obtain a better estimate of the pixel intensities. Another method for sampling is to randomly distribute the rays over the pixel area. We discuss this approach in the next section. When multiple rays

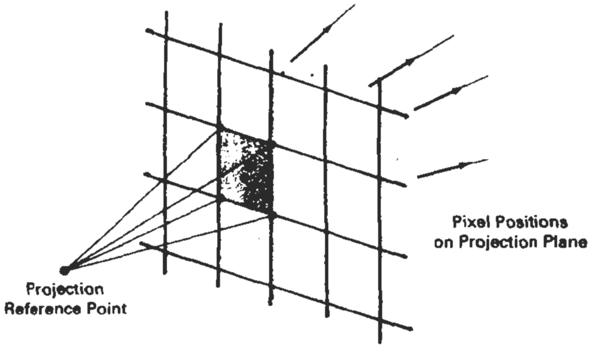

Pixel Positions on Projection Plane
Projection Reference Point

Figure 14-69
Supersampling with four rays per pixel, one at each pixel corner.

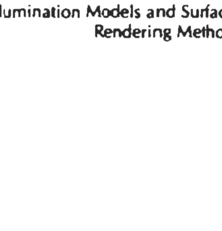


Figure 14-70
Subdividing a pixel into nine subpixels with one ray at each subpixel corner.

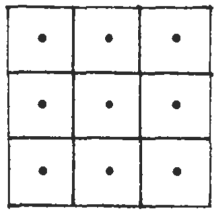


Figure 14-71
Ray positions centered on subpixel areas.

per pixel are used, the intensities of the pixel rays are averaged to produce the overall pixel intensity.

Figure 14-69 illustrates a simple supersampling procedure. Here, one ray is generated through each corner of the pixel. If the intensities for the four rays are not approximately equal, or if some small object lies between the four rays, we divide the pixel area into subpixels and repeat the process. As an example, the pixel in Fig. 14-70 is divided into nine subpixels using 16 rays, one at each subpixel corner. Adaptive sampling is then used to further subdivide those subpixels that do not have nearly equal-intensity rays or that subtend some small object. This subdivision process can be continued until each subpixel has approximately equal-intensity rays or an upper bound, say, 256, has been reached for the number of rays per pixel.

The cover picture for this book was rendered with adaptive-subdivision ray tracing, using Rayshade version 3 on a Macintosh II. An extended light source was used to provide realistic soft shadows. Nearly 26 million primary rays were generated, with 33.5 million shadow rays and 67.3 million reflection rays. Wood grain and marble surface patterns were generated using solid texturing methods with a noise function. Total rendering time with the extended light source was 213 hours. Each image of the stereo pair shown in Fig. 2-20 was generated in 45 hours using a point light source.

Instead of passing rays through pixel corners, we can generate rays through subpixel centers, as in Fig. 14-71. With this approach, we can weight the rays according to one of the sampling schemes discussed in Chapter 4.

Another method for antialiasing displayed scenes is to treat a pixel ray as a cone, as shown in Fig. 14-72. Only one ray is generated per pixel, but the ray now has a finite cross section. To determine the percent of pixel-area coverage with objects, we calculate the intersection of the pixel cone with the object surface. For a sphere, this requires finding the intersection of two circles. For a polyhedron, we must find the intersection of a circle with a polygon.

Distributed Ray Tracing

This is a stochastic sampling method that randomly distributes rays according to the various parameters in an illumination model. Illumination parameters in-

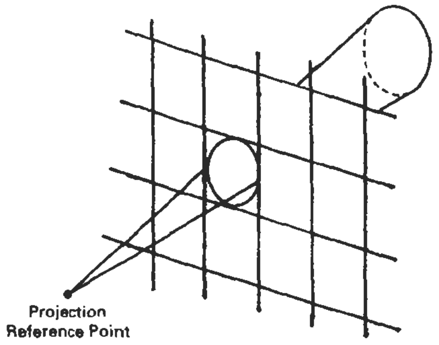


Figure 14-72
A pixel ray cone.

clude pixel area, reflection and refraction directions, camera lens area, and time. Aliasing effects are thus replaced with low-level "noise", which improves picture quality and allows more accurate modeling of surface gloss and translucency, finite camera apertures, finite light sources, and motion-blur displays of moving objects. **Distributed ray tracing** (also referred to as *distribution ray tracing*) essentially provides a Monte Carlo evaluation of the multiple integrals that occur in an accurate description of surface lighting.

Pixel sampling is accomplished by randomly distributing a number of rays over the pixel surface. Choosing ray positions completely at random, however, can result in the rays clustering together in a small region of the pixel area, and leaving other parts of the pixel unsampled. A better approximation of the light distribution over a pixel area is obtained by using a technique called *jittering* on a regular subpixel grid. This is usually done by initially dividing the pixel area (a unit square) into the 16 subareas shown in Fig. 14-73 and generating a random *jitter position* in each subarea. The random ray positions are obtained by jittering the center coordinates of each subarea by small amounts, δ_x and δ_y, where both δ_x and δ_y are assigned values in the interval $(-0.5, 0.5)$. We then choose the ray position in a cell with center coordinates (x, y) as the jitter position $(x + \delta_x, y + \delta_y)$.

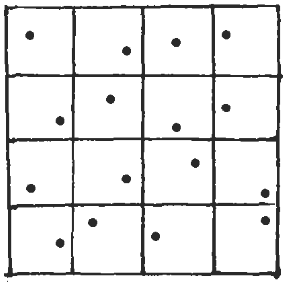


Figure 14-73
Pixel sampling using 16 subpixel areas and a jittered ray position from the center coordinates for each subarea.

Integer codes 1 through 16 are randomly assigned to each of the 16 rays, and a table lookup is used to obtain values for the other parameters (reflection angle, time, etc.), as explained in the following discussion. Each subpixel ray is then processed through the scene to determine the intensity contribution for that ray. The 16 ray intensities are then averaged to produce the overall pixel intensity. If the subpixel intensities vary too much, the pixel is further subdivided.

To model camera-lens effects, we set a lens of assigned focal length f in front of the projection plane and distribute the subpixel rays over the lens area. Assuming we have 16 rays per pixel, we can subdivide the lens area into 16 zones. Each ray is then sent to the zone corresponding to its assigned code. The ray position within the zone is set to a jittered position from the zone center. Then the ray is projected into the scene from the jittered zone position through the focal point of the lens. We locate the focal point for a ray at a distance f from the lens along the line from the center of the subpixel through the lens center, as shown in Fig. 14-74. Objects near the focal plane are projected as sharp images. Objects in front or in back of the focal plane are blurred. To obtain better displays of out-of-focus objects, we increase the number of subpixel rays.

Ray reflections at surface-intersection points are distributed about the specular reflection direction **R** according to the assigned ray codes (Fig. 14-75). The

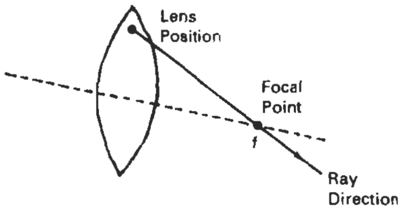


Figure 14-74
Distributing subpixel rays over a camera lens of focal length *f*.

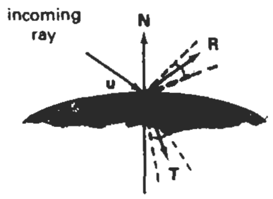


Figure 14-75
Distributing subpixel rays about the reflection direction **R** and the transmission direction **T**.

maximum spread about **R** is divided into 16 angular zones, and each ray is reflected in a jittered position from the zone center corresponding to its integer code. We can use the Phong model, $\cos^{n_s}\phi$, to determine the maximum reflection spread. If the material is transparent, refracted rays are distributed about the transmission direction T in a similar manner.

Extended light sources are handled by distributing a number of shadow rays over the area of the light source, as demonstrated in Fig. 14-76. The light source is divided into zones, and shadow rays are assigned jitter directions to the various zones. Additionally, zones can be weighted according to the intensity of the light source within that zone and the size of the projected zone area onto the object surface. More shadow rays are then sent to zones with higher weights. If some shadow rays are blocked by opaque objects between the surface and the light source, a penumbra is generated at that surface point. Figure 14-77 illustrates the regions for the umbra and penumbra on a surface partially shielded from a light source.

We create motion blur by distributing rays over time. A total frame time and the frame-time subdivisions are determined according to the motion dynamics required for the scene. Time intervals are labeled with integer codes, and each ray is assigned to a jittered time within the interval corresponding to the ray code. Objects are then moved to their positions at that time, and the ray is traced

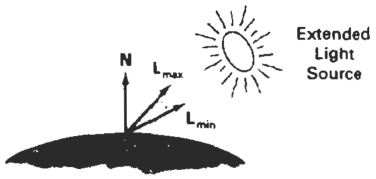


Figure 14-76
Distributing shadow rays over a finite-sized light source.

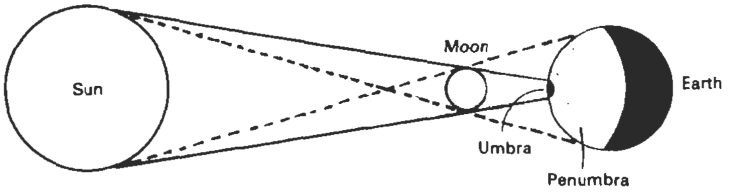


Figure 14-77
Umbra and penumbra regions created by a solar eclipse on the surface of the earth.

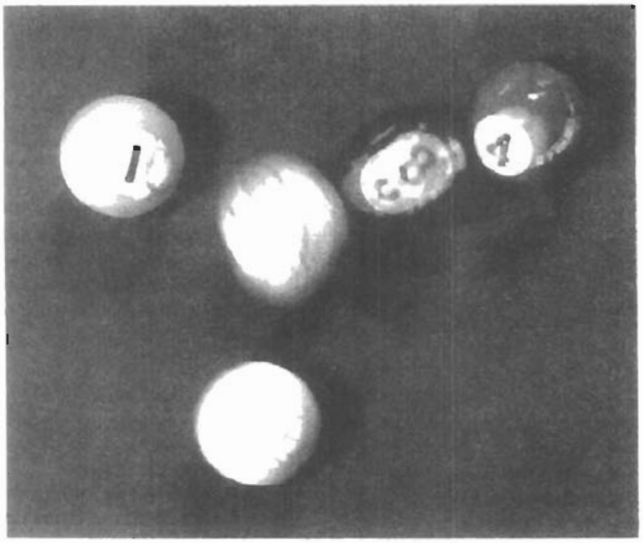


Figure 14-78
A scene, entitled *1984*, rendered with distributed ray tracing, illustrating motion-blur and penumbra effects. (*Courtesy of Pixar. © 1984 Pixar. All rights reserved.*)

through the scene. Additional rays are used for highly blurred objects. To reduce calculations, we can use bounding boxes or spheres for initial ray-intersection tests. That is, we move the bounding object according to the motion requirements and test for intersection. If the ray does not intersect the bounding object, we do not need to process the individual surfaces within the bourding volume. Figure 14-78 shows a scene displayed with motion blur. This image was rendered using distributed ray tracing with 4096 by 3550 pixels and 16 rays per pixel. In addition to the motion-blurred reflections, the shadows are displayed with penumbra areas resulting from the extended light sources around the room that are illuminating the pool table.

Additional examples of objects rendered with distributed ray-tracing methods are given in Figs. 14-79 and 14-80. Figure 14-81 illustrates focusing, refraction, and antialiasing effects with distributed ray tracing.

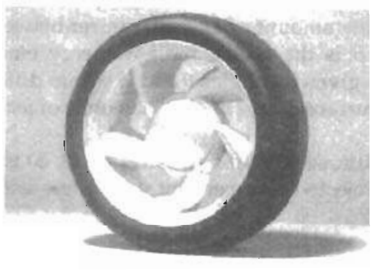


Figure 14-79
A brushed aluminum wheel showing reflectance and shadow effects generated with distributed ray-tracing techniques. (*Courtesy of Stephen H. Westin, Program of Computer Graphics, Cornell University.*)

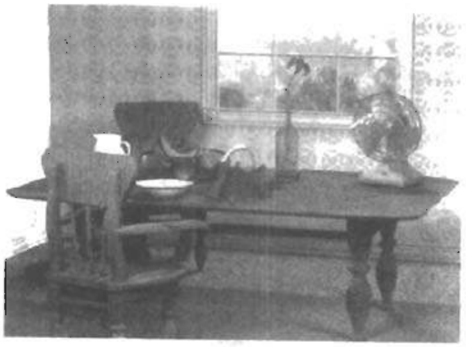


Figure 14-80
A room scene rendered with distributed ray-tracing methods. *(Courtesy of John Snyder, Jed Lengyel, Devendra Kalra, and Al Barr, Computer Graphics Lab, California Institute of Technology. Copyright © 1988 Caltech.)*

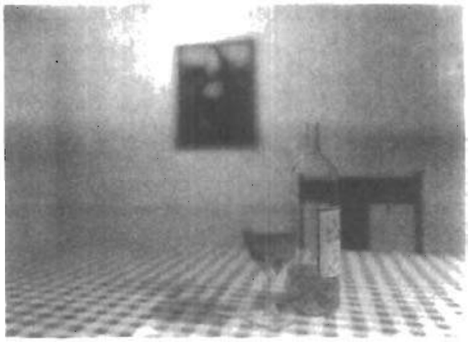


Figure 14-81
A scene showing the focusing, antialiasing, and illumination effects possible with a combination of ray-tracing and radiosity methods. Realistic physical models of light illumination were used to generate the refraction effects, including the caustic in the shadow of the glass. *(Courtesy of Peter Shirley, Department of Computer Science, Indiana University.)*

14-7 RADIOSITY LIGHTING MODEL

We can accurately model diffuse reflections from a surface by considering the radiant energy transfers between surfaces, subject to conservation of energy laws. This method for describing diffuse reflections is generally referred to as the **radiosity model**.

Basic Radiosity Model

In this method, we need to consider the radiant-energy interactions between all surfaces in a scene. We do this by determining the differential amount of radiant energy dB leaving each surface point in the scene and summing the energy contributions over all surfaces to obtain the amount of energy transfer between surfaces. With reference to Fig. 14-82, dB is the visible radiant energy emanating from the surface point in the direction given by angles θ and ϕ within differential solid angle $d\omega$ per unit time per unit surface area. Thus, dB has units of *joules/(second · meter²)*, or *watts/meter²*.

Intensity I, or *luminance*, of the diffuse radiation in direction (θ, ϕ) is the radiant energy per unit time per unit projected area per unit solid angle with units *watts/(meter² · steradians)*:

$$I = \frac{dB}{d\omega \cos \phi} \tag{14-68}$$

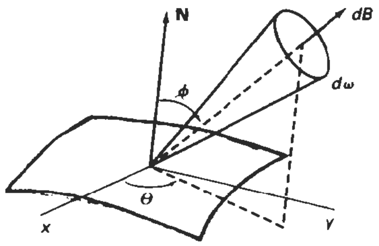


Figure 14-82
Visible radiant energy emitted from a surface point in direction (θ, ϕ) within solid angle $d\omega$.

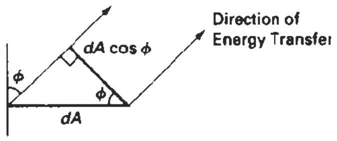


Figure 14-83
For a unit surface element, the projected area perpendicular to the direction of energy transfer is equal to $\cos\phi$.

Assuming the surface is an ideal diffuse reflector, we can set intensity I to a constant for all viewing directions. Thus, $dB/d\omega$ is proportional to the projected surface area (Fig. 14-83). To obtain the total rate of energy radiation from the surface point, we need to sum the radiation for all directions. That is, we want the total energy emanating from a hemisphere centered on the surface point, as in Fig. 14-84:

$$B = \int_{\text{hemi}} dB \tag{14-69}$$

For a perfect diffuse reflector, I is a constant, so we can express radiant energy B as

$$B = I \int_{\text{hemi}} \cos\phi \, d\omega \tag{14-70}$$

Also, the differential element of solid angle $d\omega$ can be expressed as (Appendix A)

$$d\omega = \frac{dS}{r^2} = \sin\phi \, d\phi \, d\theta$$

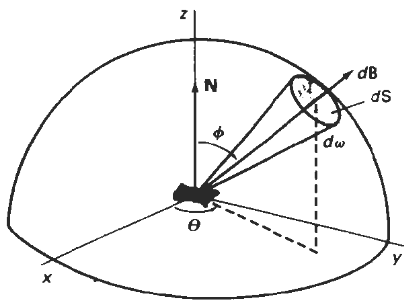


Figure 14-84
Total radiant energy from a surface point is the sum of the contributions in all directions over a hemisphere centered on the surface point.

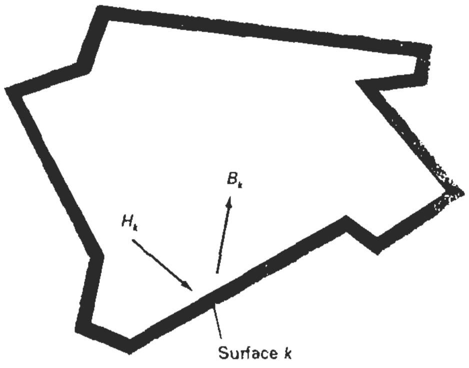


Figure 14-85
An enclosure of surfaces for the radiosity model.

so that

$$\begin{aligned} B &= I\int_0^{2\pi}\int_0^{\pi/2} \cos\phi \sin\phi \, d\phi \, d\theta \\ &= I\pi \end{aligned} \tag{14-71}$$

A model for the light reflections from the various surfaces is formed by setting up an "enclosure" of surfaces (Fig. 14-85). Each surface in the enclosure is either a reflector, an emitter (light source), or a combination reflector-emitter. We designate radiosity parameter B_k as the total rate of energy leaving surface k per unit area. Incident-energy parameter H_k is the sum of the energy contributions from all surfaces in the enclosure arriving at surface k per unit time per unit area. That is,

$$H_k = \sum_j B_j F_{jk} \tag{14-72}$$

where parameter F_{jk} is the *form factor* for surfaces j and k. Form factor F_{jk} is the fractional amount of radiant energy from surface j that reaches surface k.

For a scene with n surfaces in the enclosure, the radiant energy from surface k is described with the **radiosity equation**:

$$\begin{aligned} B_k &= E_k + \rho_k H_k \\ &= E_k + \rho_k \sum_{j=1}^{n} B_j F_{jk} \end{aligned} \tag{14-73}$$

If surface k is not a light source, $E_k = 0$. Otherwise, E_k is the rate of energy emitted from surface k per unit area (*watts/meter*^{2}). Parameter ρ_k is the reflectivity factor for surface k (percent of incident light that is reflected in all directions). This reflectivity factor is related to the diffuse reflection coefficient used in empirical illumination models. Plane and convex surfaces cannot "see" themselves, so that no self-incidence takes place and the form factor F_{kk} for these surfaces is 0.

To obtain the illumination effects over the various surfaces in the enclosure, we need to solve the simultaneous radiosity equations for the n surfaces given the array values for E_k, ρ_k, and F_{jk}. That is, we must solve

$$(1 - \rho_k F_{kk})B_k - \rho_k \sum_{j \neq k} B_j F_{jk} = E_k, \qquad k = 1, 2, 3, \ldots, n \tag{14-74}$$

or

$$\begin{bmatrix} 1 - \rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1 - \rho_2 F_{22} & \cdots & -\rho_2 F_{2n} \\ \vdots & \vdots & & \vdots \\ -\rho_n F_{n1} & -\rho_2 F_{n2} & \cdots & 1 - \rho_n F_{nn} \end{bmatrix} \cdot \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix} \tag{14-75}$$

We then convert to intensity values I_k by dividing the radiosity values B_k by π. For color scenes, we can calculate the individual RGB components of the radiosity (B_{kR}, B_{kG}, B_{kB}) from the color components of ρ_k and E_k.

Before we can solve Eq. 14-74, we need to determine the values for form factors F_{jk}. We do this by considering the energy transfer from surface j to surface k (Fig. 14-86). The rate of radiant energy falling on a small surface element dA_k from area element dA_j is

$$dB_j\, dA_j = (I_j \cos\phi_j\, d\omega) dA_j \tag{14-76}$$

But solid angle $d\omega$ can be written in terms of the projection of area element dA_k perpendicular to the direction dB_j:

$$d\omega = \frac{dA}{r^2} = \frac{\cos\phi_k dA_k}{r^2} \tag{14-77}$$

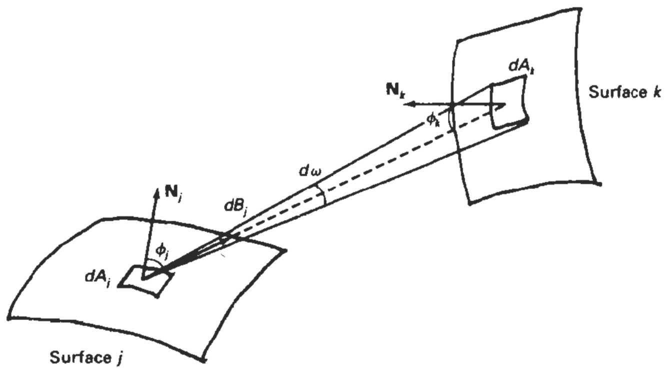


Figure 14-86
Rate of energy transfer dB_j from a surface element with area dA_j to surface element dA_k.

so we can express Eq. 14-76 as

$$dB_j\,dA_j = \frac{I_j \cos\phi_j \cos\phi_k\, dA_j\, dA_k}{r^2} \tag{14-78}$$

The form factor between the two surfaces is the percent of energy emanating from area dA_j that is incident on dA_k:

$$\begin{aligned} F_{dA_j,dA_k} &= \frac{\text{energy incident on } dA_k}{\text{total energy leaving } dA_j} \\ &= \frac{I_j \cos\phi_j \cos\phi_k\, dA_j\, dA_k}{r^2} \cdot \frac{1}{B_j\, dA_j} \end{aligned} \tag{14-79}$$

Also $B_j = \pi I_j$, so that

$$F_{dA_j,dA_k} = \frac{\cos\phi_j \cos\phi_k\, dA_k}{\pi r^2} \tag{14-80}$$

The fraction of emitted energy from area dA_j incident on the entire surface k is then

$$F_{dA_j,A_k} = \int_{\text{surf}_j} \frac{\cos\phi_j \cos\phi_k}{\pi r^2} dA_k \tag{14-81}$$

where A_k is the area of surface k. We now can define the form factor between the two surfaces as the area average of the previous expression:

$$F_{jk} = \frac{1}{A_j} \int_{\text{surf}_j} \int_{\text{surf}_k} \frac{\cos\phi_j \cos\phi_k}{\pi r^2} dA_k\, dA_j \tag{14-82}$$

Integrals 14-82 are evaluated using numerical integration techniques and stipulating the following conditions:

- $\sum_{k=1}^{n} F_{jk} = 1$, for all k (conservation of energy)
- $A_j F_{jk} = A_k F_{kj}$ (uniform light reflection)
- $F_{jj} = 0$, for all j (assuming only plane or convex surface patches)

Each surface in the scene can be subdivided into many small polygons, and the smaller the polygon areas, the more realistic the display appears. We can speed up the calculation of the form factors by using a hemicube to approximate the hemisphere. This replaces the spherical surface with a set of linear (plane) surfaces. Once the form factors are evaluated, we can solve the simultaneous linear equations 14-74 using, say, Gaussian elimination or LU decomposition methods (Appendix A). Alternatively, we can start with approximate values for the B_j and solve the set of linear equations iteratively using the Gauss–Seidel method. At each iteration, we calculate an estimate of the radiosity for surface patch k using the previously obtained radiosity values in the radiosity equation:

$$B_k = E_k + \rho_k \sum_{j=1}^{n} B_j F_{jk}$$

We can then display the scene at each step, and an improved surface rendering is viewed at each iteration until there is little change in the calculated radiosity values.

Progressive Refinement Radiosity Method

Although the radiosity method produces highly realistic surface rendings, there are tremendous storage requirements, and considerable processing time is needed to calculate the form factors. Using *progressive refinement*, we can restructure the iterative radiosity algorithm to speed up the calculations and reduce storage requirements at each iteration.

From the radiosity equation, the radiosity contribution between two surface patches is calculated as

$$B_k \text{ due to } B_j = \rho_k B_j F_{jk} \tag{14-83}$$

Reciprocally,

$$B_j \text{ due to } B_k = \rho_j B_k F_{kj}, \qquad \text{for all } j \tag{14-84}$$

which we can rewrite as

$$B_j \text{ due to } B_k = \rho_j B_k F_{jk} \frac{A_j}{A_k}, \qquad \text{for all } j \tag{14-85}$$

This relationship is the basis for the progressive refinement approach to the radiosity calculations. Using a single surface patch k, we can calculate all form factors F_{jk} and "shoot" light from that patch to all other surfaces in the environment. Thus, we need only to compute and store one hemicube and the associated form factors at a time. We then discard these values and choose another patch for the next iteration. At each step, we display the approximation to the rendering of the scene.

Initially, we set $B_k = E_k$ for all surface patches. We then select the patch with the highest radiosity value, which will be the brightest light emitter, and calculate the next approximation to the radiosity for all other patches. This process is repeated at each step, so that light sources are chosen first in order of highest radiant energy, and then other patches are selected based on the amount of light received from the light sources. The steps in a simple progressive refinement approach are given in the following algorithm.

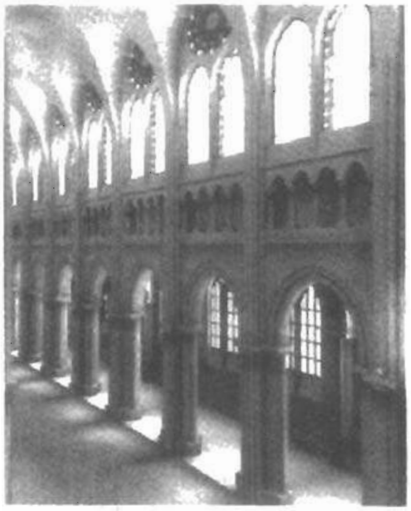


Figure 14-87
Nave of Chartres Cathedral rendered with a progressive-refinement radiosity model by John Wallace and John Lin, using the Hewlett-Packard Starbase Radiosity and Ray Tracing software. Radiosity form factors were computed with ray-tracing methods. (*Courtesy of Eric Haines, 3D/EYE Inc. © 1989, Hewlett-Packard Co.*)

```
for each patch $k$
/* set up hemicube, calculate form factors $F_{jk}$ */

for each patch $j$ {
        $\Delta$rad := $\rho_j B_k F_{jk} A_j / A_k$;
        $\Delta B_j$ := $\Delta B_j$ + $\Delta$rad;
        $B_j$ := $B_j$ + $\Delta$rad;
}

$\Delta B_k = 0$;
```

At each step, the surface patch with the highest value for $\Delta B_k A_k$ is selected as the shooting patch, since radiosity is a measure of radiant energy per unit area. And we choose the initial values as $\Delta B_k = B_k = E_k$ for all surface patches. This progressive refinement algorithm approximates the actual propagation of light through a scene.

Displaying the rendered surfaces at each step produces a sequence of views that proceeds from a dark scene to a fully illuminated one. After the first step, the only surfaces illuminated are the light sources and those nonemitting patches that are visible to the chosen emitter. To produce more useful initial views of the scene, we can set an ambient light level so that all patches have some illumination. At each stage of the iteration, we then reduce the ambient light according to the amount of radiant energy shot into the scene.

Figure 14-87 shows a scene rendered with the progressive-refinement radiosity model. Radiosity renderings of scenes with various lighting conditions are illustrated in Figs. 14-88 to 14-90. Ray-tracing methods are often combined with the radiosity model to produce highly realistic diffuse and specular surface shadings, as in Fig. 14-81.

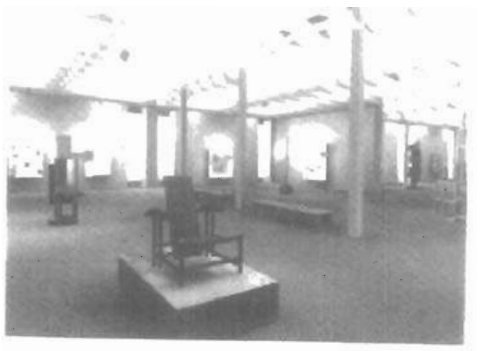


Figure 14-88
Image of a constructivist museum rendered with a progressive-refinement radiosity method. *(Courtesy of Shenchang Eric Chen, Stuart I. Feldman, and Julie Dorsey, Program of Computer Graphics, Cornell University. © 1988, Cornell University, Program of Computer Graphics.)*

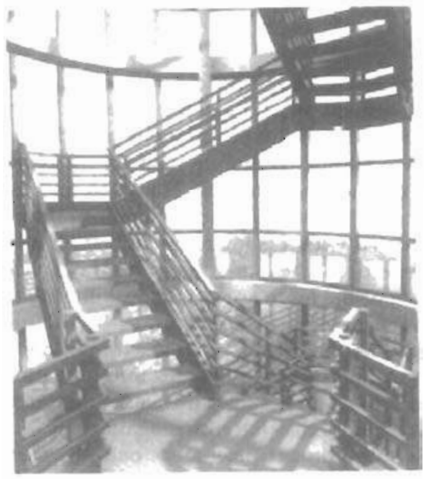


Figure 14-89
Simulation of the stair tower of the Engineering Theory Center Building at Cornell University rendered with a progressive-refinement radiosity method. *(Courtesy of Keith Howie and Ben Trumbore, Program of Computer Graphics, Cornell University. © 1990, Cornell University, Program of Computer Graphics.)*

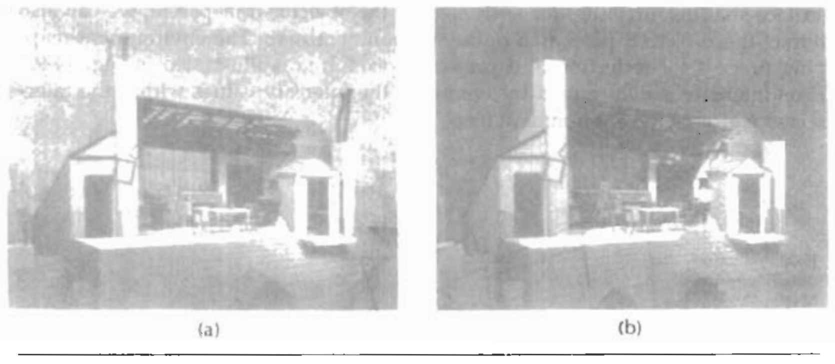


Figure 14-90
Simulation of two lighting schemes for the Parisian garret from the Metropolitan Opera's production of *La Boheme*: (a) day view and (b) night view. *(Courtesy of Julie Dorsey and Mark Shepard, Program of Computer Graphics, Cornell University. © 1991, Cornell University, Program of Computer Graphics.)*

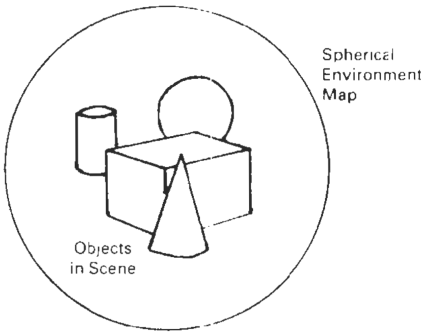


Figure 14-91
A spherical enclosing universe containing the environment map

14-8 ENVIRONMENT MAPPING

An alternate procedure for modeling global reflections is to define an array of intensity values that describes the environment around a single object or a set of objects. Instead of interobject ray tracing or radiosity calculations to pick up the global specular and diffuse illumination effects, we simply map the *environment array* onto an object in relationship to the viewing direction. This procedure is referred to as **environment mapping,** also called **reflection mapping** although transparency effects could also be modeled with the environment map. Environment mapping is sometimes referred to as the "poor person's ray-tracing" method, since it is a fast approximation of the more accurate global-illumination rendering techniques we discussed in the previous two sections.

The environment map is defined over the surface of an enclosing universe. Information in the environment map includes intensity values for light sources, the sky, and other background objects. Figure 14-91 shows the enclosing universe as a sphere, but a cube or a cylinder is often used as the enclosing universe.

To render the surface of an object, we project pixel areas onto the surface and then reflect the projected pixel area onto the environment map to pick up the surface-shading attributes for each pixel. If the object is transparent, we can also refract the projected pixel area to the environment map. The environment-mapping process for reflection of a projected pixel area is illustrated in Fig. 14-92. Pixel intensity is determined by averaging the intensity values within the intersected region of the environment map.

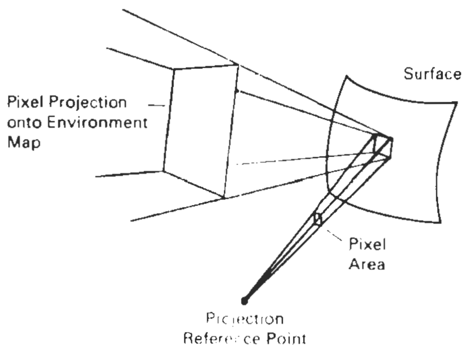

Surface
Pixel Projection onto Environment Map
Pixel Area
Projection Reference Point

Figure 14-92
Projecting a pixel area to a surface, then reflecting the area to the environment map.

14-9 ADDING SURFACE DETAIL

So far we have discussed rendering techniques for displaying smooth surfaces, typically polygons or splines. However, most objects do not have smooth, even surfaces. We need surface texture to model accurately such objects as brick walls, gravel roads, and shag carpets. In addition, some surfaces contain patterns that must be taken into account in the rendering procedures. The surface of a vase could contain a painted design; a water glass might have the family crest engraved into the surface; a tennis court contains markings for the alleys, service areas, and base line; and a four-lane highway has dividing lines and other markings, such as oil spills and tire skids. Figure 14-93 illustrates objects displayed with various surface detail.

Modeling Surface Detail with Polygons

A simple method for adding surface detail is to model structure and patterns with polygon facets. For large-scale detail, polygon modeling can give good results. Some examples of such large-scale detail are squares on a checkerboard, dividing lines on a highway, tile patterns on a linoleum floor, floral designs in a smooth low-pile rug, panels in a door, and lettering on the side of a panel truck. Also, we could model an irregular surface with small, randomly oriented polygon facets, provided the facets were not too small.

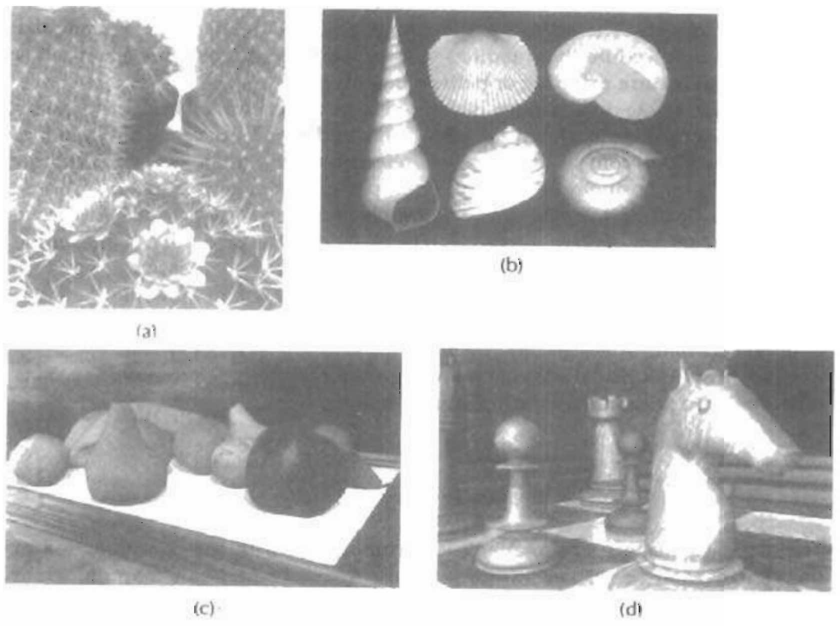


Figure 14-93
Scenes illustrating computer graphics generation of surface detail.
((a) © 1992 Deborah R. Fowler, Przemyslaw Prusinkiewicz, and Johannes Battjes; (b) © 1992 Deborah R. Fowler, Hans Meinhardt, and Przemyslaw Prusinkiewicz, University of Calgary; (c) and (d) Courtesy of SOFTIMAGE, Inc.)

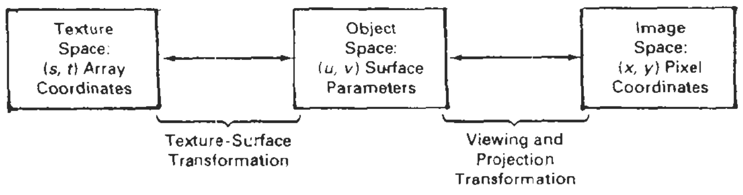

Texture Space: (s, t) Array Coordinates
Object Space: (u, v) Surface Parameters
Image Space: (x, y) Pixel Coordinates
Texture-Surface Transformation
Viewing and Projection Transformation

Figure 14-94
Coordinate reference systems for texture space, object space, and image space.

Surface-pattern polygons are generally overlaid on a larger surface polygon and are processed with the parent surface. Only the parent polygon is processed by the visible-surface algorithms, but the illumination parameters for the surface-detail polygons take precedence over the parent polygon. When intricate or fine surface detail is to be modeled, polygon methods are not practical. For example, it would be difficult to accurately model the surface structure of a raisin with polygon facets.

Texture Mapping

A common method for adding surface detail is to map texture patterns onto the surfaces of objects. The texture pattern may either be defined in a rectangular array or as a procedure that modifies surface intensity values. This approach is referred to as **texture mapping** or **pattern mapping**.

Usually, the texture pattern is defined with a rectangular grid of intensity values in a *texture space* referenced with (s, t) coordinate values, as shown in Fig. 14-94. Surface positions in the scene are referenced with uv object-space coordinates, and pixel positions on the projection plane are referenced in xy Cartesian coordinates. Texture mapping can be accomplished in one of two ways. Either we can map the texture pattern to object surfaces, then to the projection plane; or we can map pixel areas onto object surfaces, then to texture space. Mapping a texture pattern to pixel coordinates is sometimes called *texture scanning*, while the mapping from pixel coordinates to texture space is referred to as *pixel-order scanning* or *inverse scanning* or *image-order scanning*.

To simplify calculations, the mapping from texture space to object space is often specified with parametric linear functions

$$\begin{aligned} u &= f_u(s,t) = a_u s + b_u t + c_u \\ v &= f_v(s,t) = a_v s + b_v t + c_v \end{aligned} \qquad (14\text{-}86)$$

The object-to-image space mapping is accomplished with the concatenation of the viewing and projection transformations. A disadvantage of mapping from texture space to pixel space is that a selected texture patch usually does not match up with the pixel boundaries, thus requiring calculation of the fractional area of pixel coverage. Therefore, mapping from pixel space to texture space (Fig. 14-95) is the most commonly used texture-mapping method. This avoids pixel-subdivision calculations, and allows antialiasing (filtering) procedures to be eas-

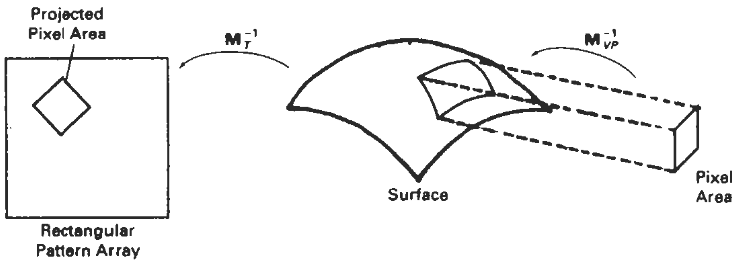


Figure 14-95
Texture mapping by projecting pixel areas to texture space.

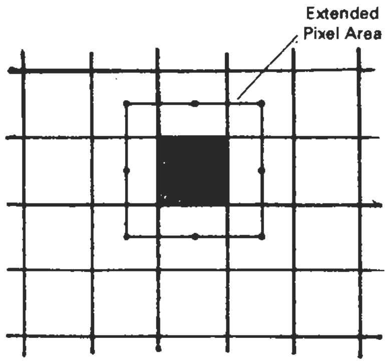


Figure 14-96
Extended area for a pixel that includes centers of adjacent pixels.

ily applied. An effective antialiasing procedure is to project a slightly larger pixel area that includes the centers of neighboring pixels, as shown in Fig. 14-96, and applying a pyramid function to weight the intensity values in the texture pattern. But the mapping from image space to texture space does require calculation of the inverse viewing-projection transformation $\mathbf{M}_{VP}^{-1}$ and the inverse texture-map transformation $\mathbf{M}_T^{-1}$. In the following example, we illustrate this approach by mapping a defined pattern onto a cylindrical surface.

Example 14-1 Texture Mapping

To illustrate the steps in texture mapping, we consider the transfer of the pattern shown in Fig. 14-97 to a cylindrical surface. The surface parameters are

$$u = \theta, \qquad v = z$$

with

$$0 \le \theta \le \pi/2, \qquad 0 \le z \le 1$$

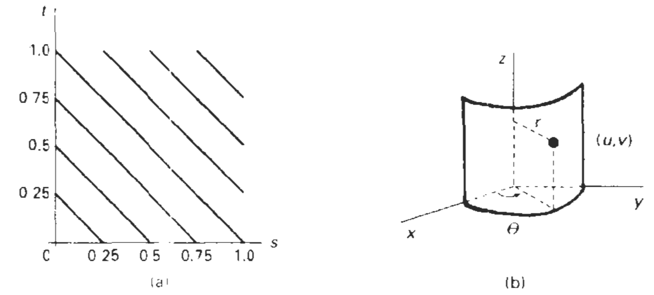


Figure 14-97
Mapping a texture pattern defined on a unit square (a) to a cylindrical surface (b).

And the parametric representation for the surface in the Cartesian reference frame is

$$x = r\cos u, \qquad y = r\sin u, \qquad z = v$$

We can map the array pattern to the surface with the following linear transformation, which maps the pattern origin to the lower left corner of the surface.

$$u = s\pi/2, \qquad v = t$$

Next, we select a viewing position and perform the inverse viewing transformation from pixel coordinates to the Cartesian reference for the cylindrical surface. Cartesian coordinates are then mapped to the surface parameters with the transformation

$$u = \tan^{-1}(y/x), \qquad v = z$$

and projected pixel positions are mapped to texture space with the inverse transformation

$$s = 2u/\pi, \qquad t = v$$

Intensity values in the pattern array covered by each projected pixel area are then averaged to obtain the pixel intensity.

Procedural Texturing Methods

Another method for adding surface texture is to use procedural definitions of the color variations that are to be applied to the objects in a scene. This approach avoids the transformation calculations involved in transferring two-dimensional texture patterns to object surfaces.

When values are assigned throughout a region of three-dimensional space, the object color variations are referred to as **solid textures**. Values from *texture*

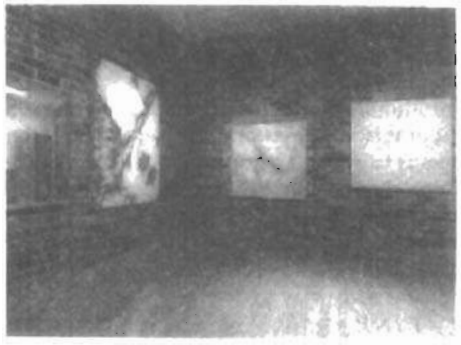


Figure 14-98
A scene with surface characteristics generated using solid-texture methods. *(Courtesy of Peter Shirley, Computer Science Department, Indiana University.)*

space are transferred to object surfaces using procedural methods, since it is usually impossible to store texture values for all points throughout a region of space. Other procedural methods can be used to set up texture values over two-dimensional surfaces. Solid texturing allows cross-sectional views of three-dimensional objects, such as bricks, to be rendered with the same texturing as the outside surfaces.

As examples of procedural texturing, wood grains or marble patterns can be created using harmonic functions (sine curves) defined in three-dimensional space. Random variations in the wood or marble texturing can be attained by superimposing a noise function on the harmonic variations. Figure 14-98 shows a scene displayed using solid textures to obtain wood-grain and other surface patterns. The scene in Fig. 14-99 was rendered using procedural descriptions of materials such as stone masonry, polished gold, and banana leaves.

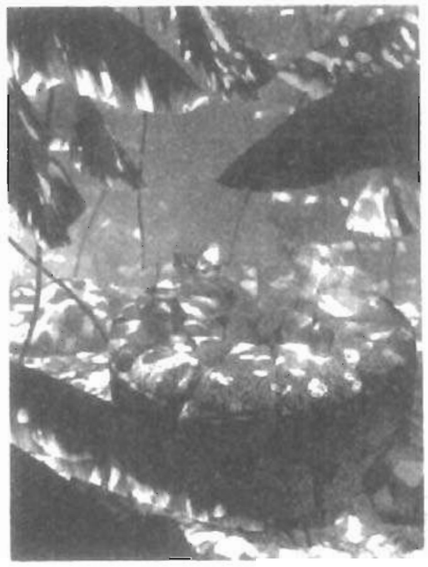


Figure 14-99
A scene rendered with VG Shaders and modeled with RenderMan using polygonal facets for the gem faces, quadric surfaces, and bicubic patches. In addition to surface texturing, procedural methods were used to create the steamy jungle atmosphere and the forest canopy dappled lighting effect. *(Courtesy of the VALIS Group. Reprinted from Graphics Gems III, edited by David Kirk. Copyright © 1992, Academic Press, Inc.)*

Bump Mapping

Although texture mapping can be used to add fine surface detail, it is not a good method for modeling the surface roughness that appears on objects such as oranges, strawberries, and raisins. The illumination detail in the texture pattern usually does not correspond to the illumination direction in the scene. A better method for creating surface bumpiness is to apply a perturbation function to the surface normal and then use the perturbed normal in the illumination-model calculations. This techniques is called **bump mapping**.

If $\mathbf{P}(u, v)$ represents a position on a parametric surface, we can obtain the surface normal at that point with the calculation

$$\mathbf{N} = \mathbf{P}_u \times \mathbf{P}_v \tag{14-87}$$

where $\mathbf{P}_u$ and $\mathbf{P}_v$ are the partial derivatives of $\mathbf{P}$ with respect to parameters u and v. To obtain a perturbed normal, we modify the surface-position vector by adding a small perturbation function, called a *bump function*:

$$\mathbf{P}'(u, v) = \mathbf{P}(u, v) + b(u, v)\mathbf{n} \tag{14-88}$$

This adds bumps to the surface in the direction of the unit surface normal $\mathbf{n} = \mathbf{N}/|\mathbf{N}|$. The perturbed surface normal is then obtained as

$$\mathbf{N}' = \mathbf{P}'_u \times \mathbf{P}'_v \tag{14-89}$$

We calculate the partial derivative with respect to u of the perturbed position vector as

$$\begin{aligned} \mathbf{P}'_u &= \frac{\partial}{\partial u}(\mathbf{P} + b\mathbf{n}) \\ &= \mathbf{P}_u + b_u\mathbf{n} + b\mathbf{n}_u \end{aligned} \tag{14-90}$$

Assuming the bump function b is small, we can neglect the last term and write:

$$\mathbf{P}'_u \approx \mathbf{P}_u - b_u\mathbf{n} \tag{14-91}$$

Similarly,

$$\mathbf{P}'_v \approx \mathbf{P}_v + b_v\mathbf{n} \tag{14-92}$$

And the perturbed surface normal is

$$\mathbf{N}' = \mathbf{P}_u \times \mathbf{P}_v + b_v(\mathbf{P}_u \times \mathbf{n}) + b_u(\mathbf{n} \times \mathbf{P}_v) + b_ub_v(\mathbf{n} \times \mathbf{n})$$

But $\mathbf{n} \times \mathbf{n} = 0$, so that

$$\mathbf{N}' = \mathbf{N} + b_v(\mathbf{P}_u \times \mathbf{n}) + b_u(\mathbf{n} \times \mathbf{P}_v) \tag{14-93}$$

The final step is to normalize $\mathbf{N}'$ for use in the illumination-model calculations.

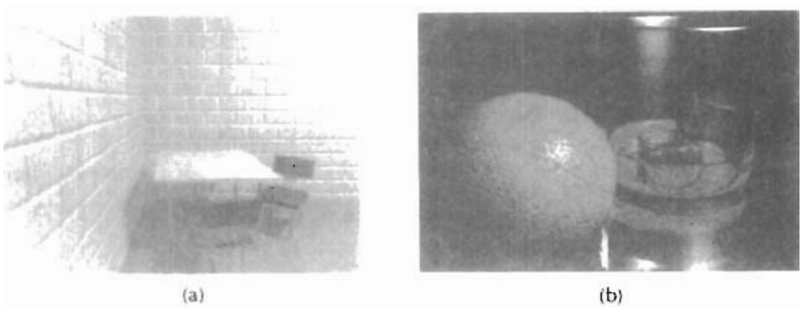


Figure 14-100
Surface roughness characteristics rendered with bump mapping. *(Courtesy of (a) Peter Shirley, Computer Science Department, Indiana University and (b) SOFTIMAGE, Inc.)*

Figure 14-101
The stained-glass knight from the motion picture *Young Sherlock Holmes*. A combination of bump mapping, environment mapping, and texture mapping was used to render the armor surface. *(Courtesy of Industrial Light & Magic. Copyright © 1985 Paramount Pictures/Amblin.)*

There are several ways in which we can specify the bump function $b(u, v)$. We can actually define an analytic expression, but bump values are usually obtained with table lookups. With a bump table, values for b can be obtained quickly with linear interpolation and incremental calculations. Partial derivatives b_u and b_v are approximated with finite differences. The bump table can be set up with random patterns, regular grid patterns, or character shapes. Random patterns are useful for modeling irregular surfaces, such as a raisin, while a repeating pattern could be used to model the surface of an orange, for example. To antialiase, we subdivide pixel areas and average the computed subpixel intensities.

Figure 14-100 shows examples of surfaces rendered with bump mapping. An example of combined surface-rendering methods is given in Fig. 14-101. The armor for the stained-glass knight in the film *Young Sherlock Holmes* was rendered with a combination of bump mapping, environment mapping, and texture mapping. An environment map of the surroundings was combined with a bump map to produce background illumination reflections and surface roughness. Then additional color and surface illumination, bumps, spots of dirt, and stains for the seams and rivets were added to produce the overall effect shown in Fig. 14-101.

Frame Mapping

This technique is an extension of bump mapping. In **frame mapping**, we perturb both the surface normal **N** and a local coordinate system (Fig. 14-102) attached to

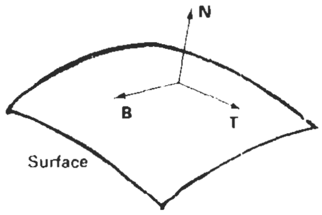


Figure 14-102
A local coordinate system at a surface point.

N. The local coordinates are defined with a surface-tangent vector **T** and a binormal vector **B** = **T** × **N**.

Frame mapping is used to model anisotropic surfaces. We orient **T** along the "grain" of the surface and apply directional perturbations, in addition to bump perturbations in the direction of **N**. In this way, we can model wood-grain patterns, cross-thread patterns in cloth, and streaks in marble or similar materials. Both bump and directional perturbations can be obtained with table lookups.

SUMMARY

In general, an object is illuminated with radiant energy from light-emitting sources and from the reflective surfaces of other objects in the scene. Light sources can be modeled as point sources or as distributed (extended) sources. Objects can be either opaque or transparent. And lighting effects can be described in terms of diffuse and specular components for both reflections and refractions.

An empirical, point light-source, illumination model can be used to describe diffuse reflections with Lambert's cosine law and to describe specular reflections with the Phong model. General background (ambient) lighting can be modeled with a fixed intensity level and a coefficient of reflection for each surface. In this basic model, we can approximate transparency effects by combining surface intensities using a transparency coefficient. Accurate geometric modeling of light paths through transparent materials is obtained by calculating refraction angles using Snell's law. Color is incorporated into the model by assigning a triple of RGB values to intensities and surface reflection coefficients. We can also extend the basic model to incorporate distributed light sources, studio lighting effects, and intensity attenuation.

Intensity values calculated with an illumination model must be mapped to the intensity levels available on the display system in use. A logarithmic intensity scale is used to provide a set of intensity levels with equal perceived brightness. In addition, gamma correction is applied to intensity values to correct for the nonlinearity of diaplay devices. With bilevel monitors, we can use halftone patterns and dithering techniques to simulate a range of intensity values. Halftone approximations can also be used to increase the number of intensity options on systems that are capable of displaying more than two intensities per pixel. Ordered-dither, error-diffusion, and dot-diffusion methods are used to simulate a range of intensities when the number of points to be plotted in a scene is equal to the number of pixels on the display device.

Surface rendering can be accomplished by applying a basic illumination model to the objects in a scene. We apply an illumination model using either constant-intensity shading, Gouraud shading, or Phong shading. Constant shading is accurate for polyhedrons or for curved-surface polygon meshes when the viewing and light-source positions are far from the objects in a scene. Gouraud shading approximates light reflections from curved surfaces by calculating intensity values at polygon vertices and interpolating these intensity values across the polygon facets. A more accurate, but slower, surface-rendering procedure is Phong shading, which interpolates the average normal vectors for polygon vertices over the polygon facets. Then, surface intensities are calculated using the interpolated normal vectors. Fast Phong shading can be used to speed up the calculations using Taylor series approximations.

Ray tracing provides an accurate method for obtaining global, specular reflection and transmission effects. Pixel rays are traced through a scene, bouncing from object to object while accumulating intensity contributions. A ray-tracing tree is constructed for each pixel, and intensity values are combined from the terminal nodes of the tree back up to the root. Object-intersection calculations in ray tracing can be reduced with space-subdivision methods that test for ray-object intersections only within subregions of the total space. Distributed (or distribution) ray tracing traces multiple rays per pixel and distributes the rays randomly over the various ray parameters, such as direction and time. This provides an accurate method for modeling surface gloss and translucency, finite camera apertures, distributed light sources, shadow effects, and motion blur.

Radiosity methods provide accurate modeling of diffuse-reflection effects by calculating radiant energy transfer between the various surface patches in a scene. Progressive refinement is used to speed up the radiosity calculations by considering energy transfer from one surface patch at a time. Highly photorealistic scenes are generated using a combination of ray tracing and radiosity.

A fast method for approximating global illumination effects is environment mapping. An environment array is used to store background intensity information for a scene. This array is then mapped to the objects in a scene based on the specified viewing direction.

Surface detail can be added to objects using polygon facets, texture mapping, bump mapping, or frame mapping. Small polygon facets can be overlaid on larger surfaces to provide various kinds of designs. Alternatively, texture patterns can be defined in a two-dimensional array and mapped to object surfaces. Bump mapping is a means for modeling surface irregularities by applying a bump function to perturb surface normals. Frame mapping is an extension of bump mapping that allows for horizontal surface variations, as well as vertical variations.

REFERENCES

A general discussion of energy propagation, transfer equations, rendering processes, and our perception of light and color is given in Glassner (1994). Algorithms for various surface-rendering techniques are presented in Glassner (1990), Arvo (1991), and Kirk (1992). For further discussion of ordered dither, error diffusion, and dot diffusion see Knuth (1987). Additional information on ray-tracing methods can be found in Quek and Hearn (1988), Glassner (1989), Shirley (1990), and Koh and Hearn (1992). Radiosity methods are discussed in Goral et al. (1984), Cohen and Greenberg (1985), Cohen et al. (1988), Wallace, Elmquist, and Haines (1989), Chen et al. (1991), Dorsey, Sillion, and Greenberg (1991), He et al. (1992), Sillion et al. (1991), Schoeneman et al (1993), and Lischinski, Tampieri, and Greenberg (1993).

EXERCISES

14.1 Write a routine to implement Eq. 14-4 of the basic illumination model using a single point light source and constant surface shading for the faces of a specified polyhedron. The object description is to be given as a set of polygon tables, including surface normals for each of the polygon faces. Additional input parameters include the ambient intensity, light-source intensity, and the surface reflection coefficients. All coordinate information can be specified directly in the viewing reference frame.

14-2. Modify the routine in Exercise 14-1 to render a polygon surface mesh using Gouraud shading

14-3. Modify the routine in Exercise 14-1 to render a polygon surface mesh using Phong shading

14-4. Write a routine to implement Eq. 14-9 of the basic illumination model using a single point light source and Gouraud surface shading for the faces of a specified polygon mesh. The object description is to be given as a set of polygon tables, including surface normals for each of the polygon faces. Additional input includes values for the ambient intensity, light-source intensity, surface reflection coefficients, and the specular-reflection parameter. All coordinate information can be specified directly in the viewing reference frame.

14-5. Modify the routine in Exercise 14-4 to render the polygon surfaces using Phong shading.

14-6. Modify the routine in Exercise 14-4 to include a linear intensity attenuation function.

14-7. Modify the routine in Exercise 14-4 to render the polygon surfaces using Phong shading and a linear intensity attenuation function.

14-8. Modify the routine in Exercise 14-4 to implement Eq. 14-13 with any specified number of polyhedrons and light sources in the scene.

14-9. Modify the routine in Exercise 14-4 to implement Eq. 14-14 with any specified number of polyhedrons and light sources in the scene.

14-10. Modify the routine in Exercise 14-4 to implement Eq. 14-15 with any specified number of polyhedrons and light sources in the scene.

14-11. Modify the routine in Exercise 14-4 to implement Eqs. 14-15 and 14-19 with any specified number of light sources and polyhedrons (either opaque or transparent) in the scene.

14-12. Discuss the differences you might expect to see in the appearance of specular reflections modeled with $(\mathbf{N} \cdot \mathbf{H})^{n_s}$ compared to specular reflections modeled with $(\mathbf{V} \cdot \mathbf{R})^{n_s}$

14-13. Verify that $2\alpha = \phi$ in Fig. 14-18 when all vectors are coplanar, but that in general, $2\alpha \neq \phi$.

14-14. Discuss how the different visible-surface detection methods can be combined with an intensity model for displaying a set of polyhedrons with opaque surfaces

14-15 Discuss how the various visible-surface detection methods can be modified to process transparent objects. Are there any visible-surface detection methods that cannot handle transparent surfaces?

14-16 Set up an algorithm, based on one of the visible-surface detection methods, that will identify shadow areas in a scene illuminated by a distant point source

14-17 How many intensity levels can be displayed with halftone approximations using n by n pixel grids where each pixel can be displayed with m different intensities?

14-18 How many different color combinations can be generated using halftone approximations on a two-level RGB system with a 3 by 3 pixel grid?

14-19. Write a routine to display a given set of surface-intensity variations using halftone approximations with 3 by 3 pixel grids and two intensity levels (0 and 1) per pixel

14-20 Write a routine to generate ordered-dither matrices using the recurrence relation in Eq. 14-34

14-21. Write a procedure to display a given array of intensity values using the ordered-dither method.

14-22. Write a procedure to implement the error-diffusion algorithm for a given m by n array of intensity values.

14-23. Write a program to implement the basic ray-tracing algorithm for a scene containing a single sphere hovering over a checkerboard ground square. The scene is to be illuminated with a single point light source at the viewing position.

14-24. Write a program to implement the basic ray-tracing algorithm for a scene containing any specified arrangement of spheres and polygon surfaces illuminated by a given set of point light sources.

14-25. Write a program to implement the basic ray-tracing algorithm using space-subdivision methods for any specified arrangement of spheres and polygon surfaces illuminated by a given set of point light sources.

14-26. Write a program to implement the following features of distributed ray tracing: pixel sampling with 16 jittered rays per pixel, distributed reflection directions, distributed refraction directions, and extended light sources.

14-27. Set up an algorithm for modeling the motion blur of a moving object using distributed ray tracing.

14-28. Implement the basic radiosity algorithm for rendering the inside surfaces of a cube when one inside face of the cube is a light source.

14-29. Devise an algorithm for implementing the progressive refinement radiosity method.

14-30. Write a routine to transform an environment map to the surface of a sphere.

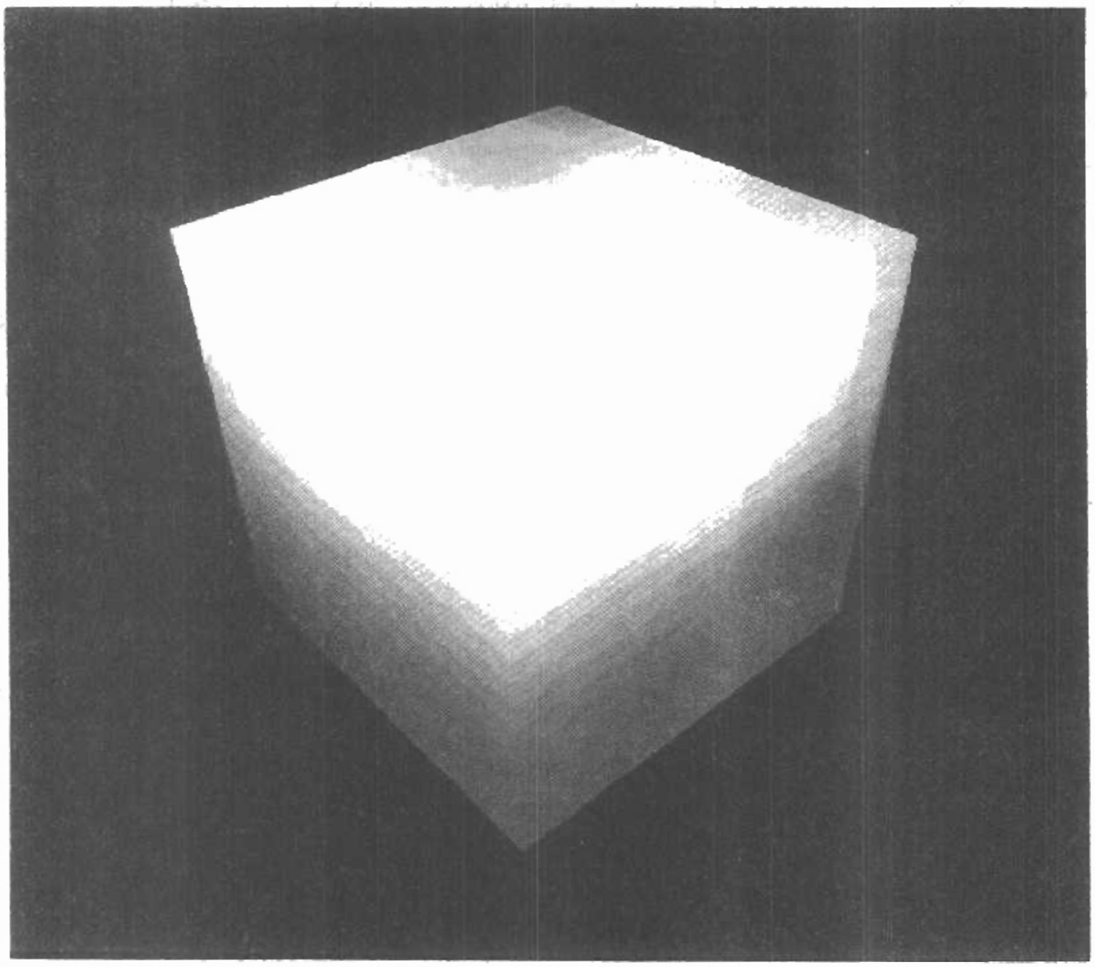
14-31. Write a program to implement texture mapping for (a) spherical surfaces and (b) polyhedrons.

14-32. Given a spherical surface, write a bump-mapping procedure to generate the bumpy surface of an orange.

14-33. Write a bump-mapping routine to produce surface-normal variations for any specified bump function.

CHAPTER

15 Color Models and Color Applications



Our discussions of color up to this point have concentrated on the mechanisms for generating color displays with combinations of red, green, and blue light. This model is helpful in understanding how color is represented on a video monitor, but several other color models are useful as well in graphics applications. Some models are used to describe color output on printers and plotters, and other models provide a more intuitive color-parameter interface for the user.

A **color model** is a method for explaining the properties or behavior of color within some particular context. No single color model can explain all aspects of color, so we make use of different models to help describe the different perceived characteristics of color.

15-1 PROPERTIES OF LIGHT

What we perceive as "light", or different colors, is a narrow frequency band within the electromagnetic spectrum. A few of the other frequency bands within this spectrum are called radio waves, microwaves, infrared waves, and X-rays. Figure 15-1 shows the approximate frequency ranges for some of the electromagnetic bands.

Each frequency value within the visible band corresponds to a distinct color. At the low-frequency end is a red color (4.3×10^{14} hertz), and the highest frequency we can see is a violet color (7.5×10^{14} hertz). Spectral colors range from the reds through orange and yellow at the low-frequency end to greens, blues, and violet at the high end.

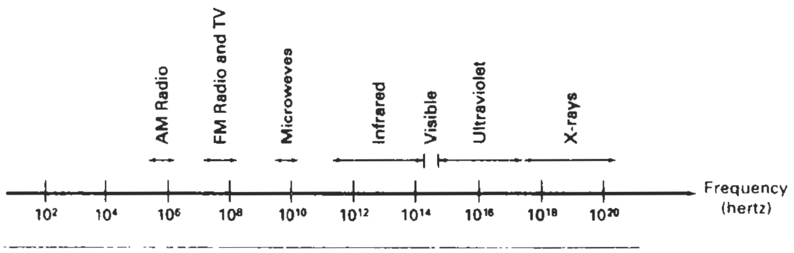


Figure 15-1
Electromagnetic spectrum.

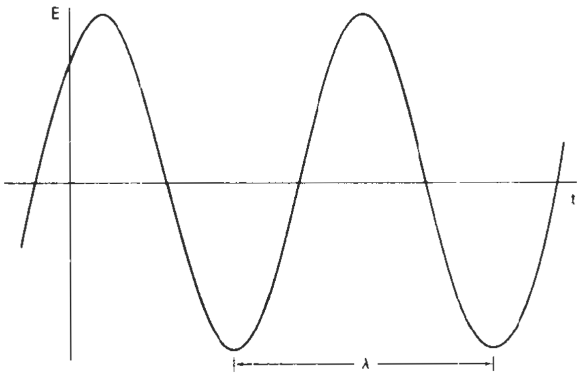


Figure 15-2
Time variations for one electric frequency component of a plane-polarized electromagnetic wave.

Since light is an electromagnetic wave, we can describe the various colors in terms of either the frequency f or the wavelength λ of the wave. In Fig. 15-2, we illustrate the oscillations present in a monochromatic electromagnetic wave, polarized so that the electric oscillations are in one plane. The wavelength and frequency of the monochromatic wave are inversely proportional to each other, with the proportionality constant as the speed of light c:

$$c = \lambda f \tag{15-1}$$

Frequency is constant for all materials, but the speed of light and the wavelength are material-dependent. In a vacuum, $c = 3 \times 10^{10}$ cm/sec. Light wavelengths are very small, so length units for designating spectral colors are usually either angstroms ($1\text{Å} = 10^{-8}$ cm) or nanometers (1 nm $= 10^{-7}$ cm). An equivalent term for nanometer is millimicron. Light at the red end of the spectrum has a wavelength of approximately 700 nanometers (nm), and the wavelength of the violet light at the other end of the spectrum is about 400 nm. Since wavelength units are somewhat more convenient to deal with than frequency units, spectral colors are typically specified in terms of wavelength.

A light source such as the sun or a light bulb emits all frequencies within the visible range to produce white light. When white light is incident upon an object, some frequencies are reflected and some are absorbed by the object. The combination of frequencies present in the reflected light determines what we perceive as the color of the object. If low frequencies are predominant in the reflected light, the object is described as red. In this case, we say the perceived light has a **dominant frequency** (or **dominant wavelength**) at the red end of the spectrum. The dominant frequency is also called the **hue**, or simply the **color**, of the light.

Other properties besides frequency are needed to describe the various characteristics of light. When we view a source of light, our eyes respond to the color (or dominant frequency) and two other basic sensations. One of these we call the **brightness**, which is the perceived intensity of the light. Intensity is the radiant energy emitted per unit time, per unit solid angle, and per unit projected area of the source. Radiant energy is related to the **luminance** of the source. The second

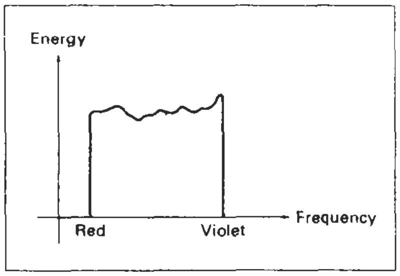


Figure 15-3
Energy distribution of a white-light source.

perceived characteristic is the **purity**, or **saturation**, of the light. Purity describes how washed out or how "pure" the color of the light appears. Pastels and pale colors are described as less pure. These three characteristics, dominant frequency, brightness, and purity, are commonly used to describe the different properties we perceive in a source of light. The term **chromaticity** is used to refer collectively to the two properties describing color characteristics: purity and dominant frequency.

Energy emitted by a white-light source has a distribution over the visible frequencies as shown in Fig. 15-3. Each frequency component within the range from red to violet contributes more or less equally to the total energy, and the color of the source is described as white. When a dominant frequency is present, the energy distribution for the source takes a form such as that in Fig. 15-4. We would now describe the light as having the color corresponding to the dominant frequency. The energy density of the dominant light component is labeled as E_D in this figure, and the contributions from the other frequencies produce white light of energy density E_W. We can calculate the brightness of the source as the area under the curve, which gives the total energy density emitted. Purity depends on the difference between E_D and E_W. The larger the energy E_D of the dominant frequency compared to the white-light component E_W, the more pure the light. We have a purity of 100 percent when $E_W = 0$ and a purity of 0 percent when $E_W = E_D$.

When we view light that has been formed by a combination of two or more sources, we see a resultant light with characteristics determined by the original sources. Two different-color light sources with suitably chosen intensities can be used to produce a range of other colors. If the two color sources combine to pro-

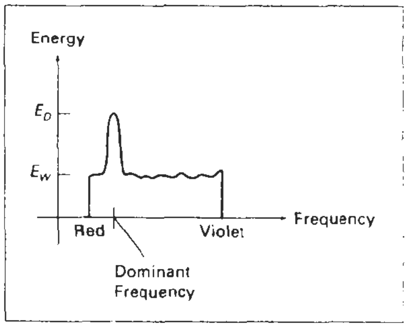


Figure 15-4
Energy distribution of a light source with a dominant frequency near the red end of the frequency range.

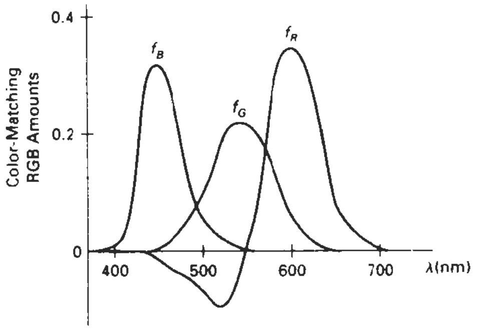


Figure 15-5
Amounts of RGB primaries needed to display spectral colors.

duce white light, they are referred to as **complementary colors**. Examples of complementary color pairs are red and cyan, green and magenta, and blue and yellow. With a judicious choice of two or more starting colors, we can form a wide range of other colors. Typically, color models that are used to describe combinations of light in terms of dominant frequency (hue) use three colors to obtain a reasonably wide range of colors, called the **color gamut** for that model. The two or three colors used to produce other colors in such a color model are referred to as **primary colors**.

No finite set of real primary colors can be combined to produce all possible visible colors. Nevertheless, three primaries are sufficient for most purposes, and colors not in the color gamut for a specified set of primaries can still be described by extended methods. If a certain color cannot be produced by combining the three primaries, we can mix one or two of the primaries with that color to obtain a match with the combination of remaining primaries. In this extended sense, a set of primary colors can be considered to describe all colors. Figure 15-5 shows the amounts of red, green, and blue needed to produce any spectral color. The curves plotted in Fig. 15-5, called *color-matching functions*, were obtained by averaging the judgments of a large number of observers. Colors in the vicinity of 500 nm can only be matched by "subtracting" an amount of red light from a combination of blue and green lights. This means that a color around 500 nm is described only by combining that color with an amount of red light to produce the blue–green combination specified in the diagram. Thus, an RGB color monitor cannot display colors in the neighborhood of 500 nm.

15-2 STANDARD PRIMARIES AND THE CHROMATICITY DIAGRAM

Since no finite set of color light sources can be combined to display all possible colors, three standard primaries were defined in 1931 by the International Commission on Illumination, referred to as the CIE (Commission Internationale de l'Éclairage). The three standard primaries are imaginary colors. They are defined mathematically with positive color-matching functions (Fig. 15-6) that specify the

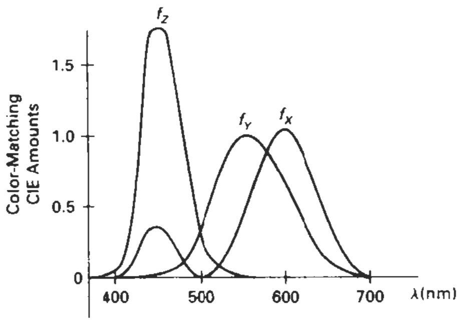


Figure 15-6
Amounts of CIE primaries needed to display spectral colors.

amount of each primary needed to describe any spectral color. This provides an international standard definition for all colors, and the CIE primaries eliminate negative-value color matching and other problems associated with selecting a set of real primaries.

XYZ Color Model

The set of CIE primaries is generally referred to as the XYZ, or (**X**, **Y**, **Z**), color model, where **X**, **Y**, and **Z** represent vectors in a three-dimensional, additive color space. Any color C_λ is then expressed as

$$C_\lambda = X\mathbf{X} + Y\mathbf{Y} + Z\mathbf{Z} \tag{15-2}$$

where X, Y, and Z designate the amounts of the standard primaries needed to match C_λ.

In discussing color properties, it is convenient to normalize the amounts in Eq. 15-2 against luminance ($X + Y + Z$). Normalized amounts are thus calculated as

$$x = \frac{X}{X+Y+Z}, \qquad y = \frac{Y}{X+Y+Z}, \qquad z = \frac{Z}{X+Y+Z} \tag{15-3}$$

with $x + y + z = 1$. Thus, any color can be represented with just the x and y amounts. Since we have normalized against luminance, parameters x and y are called the ***chromaticity values*** because they depend only on hue and purity. Also, if we specify colors only with x and y values, we cannot obtain the amounts X, Y, and Z. Therefore, a complete description of a color is typically given with the three values x, y, and Y. The remaining CIE amounts are then calculated as

$$X = \frac{x}{y}Y, \qquad Z = \frac{z}{y}Y \tag{15-4}$$

where $z = 1 - x - y$. Using chromaticity coordinates (x, y), we can represent all colors on a two-dimensional diagram.

CIE Chromaticity Diagram

When we plot the normalized amounts x and y for colors in the visible spectrum, we obtain the tongue-shaped curve shown in Fig. 15-7. This curve is called the **CIE chromaticity diagram**. Points along the curve are the "pure" colors in the

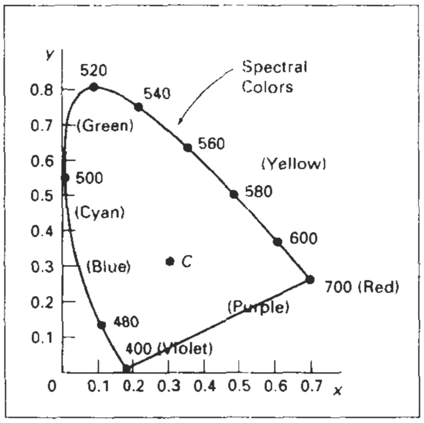


Figure 15-7
CIE chromaticity diagram. Spectral color positions along the curve are labeled in wavelength units (nm).

electromagnetic spectrum, labeled according to wavelength in nanometers from the red end to the violet end of the spectrum. The line joining the red and violet spectral points, called the ***purple line***, is not part of the spectrum. Interior points represent all possible visible color combinations. Point C in the diagram corresponds to the white-light position. Actually, this point is plotted for a white-light source known as **illuminant** ***C***, which is used as a standard approximation for "average" daylight.

Luminance values are not available in the chromaticity diagram because of normalization. Colors with different luminance but the same chromaticity map to the same point. The chromaticity diagram is useful for the following:

- Comparing color gamuts for different sets of primaries.
- Identifying complementary colors.
- Determining dominant wavelength and purity of a given color.

Color gamuts are represented on the chromaticity diagram as straight line segments or as polygons. All colors along the line joining points C_1 and C_2 in Fig. 15-8 can be obtained by mixing appropriate amounts of the colors C_1 and C_2. If a greater proportion of C_1 is used, the resultant color is closer to C_1 than to C_2. The color gamut for three points, such as C_3, C_4, and C_5 in Fig. 15-8, is a triangle with vertices at the three color positions. Three primaries can only generate colors inside or on the bounding edges of the triangle. Thus, the chromaticity diagram helps us understand why no set of three primaries can be additively combined to generate all colors, since no triangle within the diagram can encompass all colors. Color gamuts for video monitors and hard-copy devices are conveniently compared on the chromaticity diagram.

Since the color gamut for two points is a straight line, complementary colors must be represented on the chromaticity diagram as two points situated on opposite sides of C and connected with a straight line. When we mix proper amounts of the two colors C_1 and C_2 in Fig. 15-9, we can obtain white light.

We can also use the interpretation of color gamut for two primaries to determine the dominant wavelength of a color. For color point C_1 in Fig. 15-10, we can draw a straight line from C through C_1 to intersect the spectral curve at point

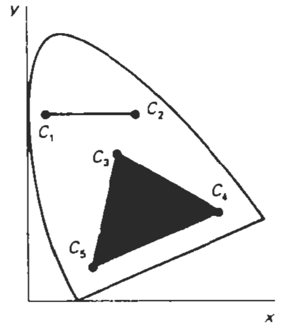


Figure 15-8
Color gamuts defined on the chromaticity diagram for a two-color and a three-color system of primaries.

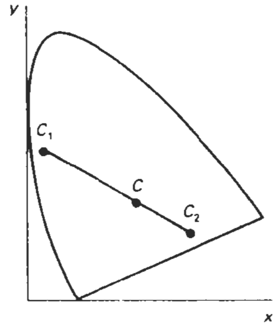


Figure 15-9
Representing complementary colors on the chromaticity diagram.

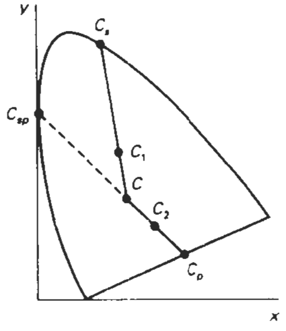


Figure 15-10
Determining dominant wavelength and purity with the chromaticity diagram.

C_s. Color C_1 can then be represented as a combination of white light C and the spectral color C_s. Thus, the dominant wavelength of C_1 is C_s. This method for determining dominant wavelength will not work for color points that are between C and the purple line. Drawing a line from C through point C_2 in Fig. 15-10 takes us to point C_p on the purple line, which is not in the visible spectrum. Point C_2 is referred to as a *nonspectral* color, and its dominant wavelength is taken as the compliment of C_p that lies on the spectral curve (point C_{sp}). Nonspectral colors are in the purple–magenta range and have spectral distributions with subtractive dominant wavelengths. They are generated by subtracting the spectral dominant wavelength (such as C_{sp}) from white light.

For any color point, such as C_1 in Fig. 15-10, we determine the purity as the relative distance of C_1 from C along the straight line joining C to C_s. If d_{c1} denotes the distance from C to C_1 and d_{cs} is the distance from C to C_s, we can calculate purity as the ratio d_{c1}/d_{cs}. Color C_1 in this figure is about 25 percent pure, since it is situated at about one-fourth the total distance from C to C_s. At position C_s, the color point would be 100 percent pure.

15-3 INTUITIVE COLOR CONCEPTS

An artist creates a color painting by mixing color pigments with white and black pigments to form the various shades, tints, and tones in the scene. Starting with the pigment for a "pure color" (or "pure hue"), the artist adds a black pigment to produce different **shades** of that color. The more black pigment, the darker the shade. Similarly, different **tints** of the color are obtained by adding a white pigment to the original color, making it lighter as more white is added. **Tones** of the color are produced by adding both black and white pigments.

To many, these color concepts are more intuitive than describing a color as a set of three numbers that give the relative proportions of the primary colors. It is generally much easier to think of making a color lighter by adding white and making a color darker by adding black. Therefore, graphics packages providing color palettes to a user often employ two or more color models. One model provides an intuitive color interface for the user, and others describe the color components for the output devices.

15-4 RGB COLOR MODEL

Based on the *tristimulus theory* of vision, our eyes perceive color through the stimulation of three visual pigments in the cones of the retina. These visual pigments have a peak sensitivity at wavelengths of about 630 nm (red), 530 nm (green), and 450 nm (blue). By comparing intensities in a light source, we perceive the color of the light. This theory of vision is the basis for displaying color output on a video monitor using the three color primaries, red, green, and blue, referred to as the RGB color model.

We can represent this model with the unit cube defined on R, G, and B axes, as shown in Fig. 15-11. The origin represents black, and the vertex with coordinates (1, 1, 1) is white. Vertices of the cube on the axes represent the primary colors, and the remaining vertices represent the complementary color for each of the primary colors.

As with the XYZ color system, the RGB color scheme is an additive model. Intensities of the primary colors are added to produce other colors. Each color point within the bounds of the cube can be represented as the triple (R, G, B), where values for R, G, and B are assigned in the range from 0 to 1. Thus, a color C_λ is expressed in RGB components as

$$C_\lambda = R\mathbf{R} + G\mathbf{G} + B\mathbf{B} \tag{15-5}$$

The magenta vertex is obtained by adding red and blue to produce the triple (1, 0, 1), and white at (1, 1, 1) is the sum of the red, green, and blue vertices. Shades of gray are represented along the main diagonal of the cube from the origin (black) to the white vertex. Each point along this diagonal has an equal contribution from each primary color, so that a gray shade halfway between black and

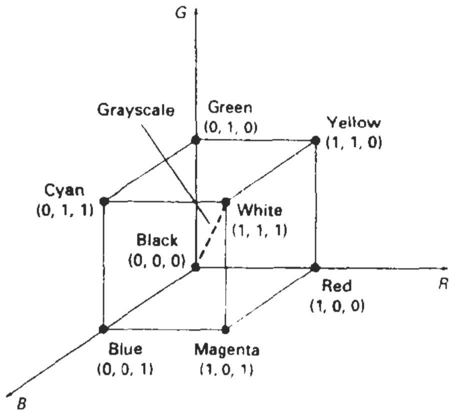


Figure 15-11
The RGB color model, defining colors with an additive process within the unit cube.

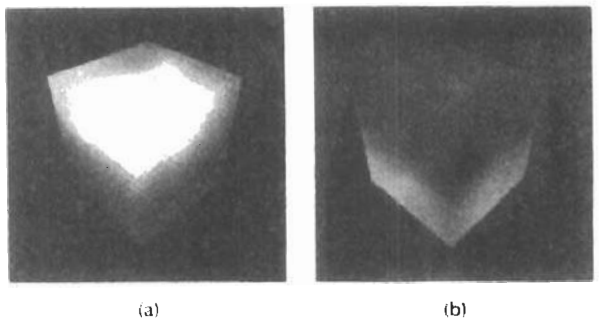


Figure 15-12
Two views of the RGB color cube: (a) along the grayscale diagonal from white to black and (b) along the grayscale diagonal from black to white.

TABLE 15-1
RGB (*X*, *Y*) CHROMACITY COORDINATES

	NTSC Standard	*CIE Model*	*Approx. Color Monitor Values*
R	(0.670,0.330)	(0.735, 0.265)	(0.628, 0.346)
G	(0.210, 0.710)	(0.274, 0.717)	(0.268, 0.588)
B	(0.140, 0.080)	(0.167, 0.009)	(0.150, 0.070)

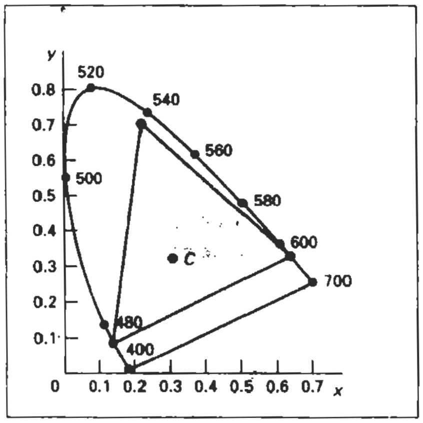


Figure 15-13
RGB color gamut.

white is represented as (0.5, 0.5, 0.5). The color graduations along the front and top planes of the RGB cube are illustrated in Fig. 15-12.

Chromaticity coordinates for an NTSC standard RGB phosphor are listed in Table 15-1. Also listed are the RGB chromaticity coordinates for the CIE RGB color model and the approximate values used for phosphors in color monitors. Figure 15-13 shows the color gamut for the NTSC standard RGB primaries.

15-5 YIQ COLOR MODEL

Whereas an RGB monitor requires separate signals for the red, green, and blue components of an image, a television monitor uses a single composite signal. The National Television System Committee (NTSC) color model for forming the composite video signal is the YIQ model, which is based on concepts in the CIE XYZ model.

In the YIQ color model, parameter Y is the same as in the XYZ model. Luminance (brightness) information is contained in the Y parameter, while chromaticity information (hue and purity) is incorporated into the I and Q parameters. A combination of red, green, and blue intensities are chosen for the Y parameter to yield the standard luminosity curve. Since Y contains the luminance information, black-and-white television monitors use only the Y signal. The largest bandwidth in the NTSC video signal (about 4 MHz) is assigned to the Y information. Parameter I contains orange–cyan hue information that provides the flesh-tone shading, and occupies a bandwidth of approximately 1.5 MHz. Parameter Q carries green–magenta hue information in a bandwidth of about 0.6 MHz.

An RGB signal can be converted to a television signal using an NTSC encoder, which converts RGB values to YIQ values, then modulates and superimposes the I and Q information on the Y signal. The conversion from RGB values to YIQ values is accomplished with the transformation

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.144 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.528 & 0.311 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{15-6}$$

This transformation is based on the NTSC standard RGB phosphor, whose chromaticity coordinates were given in the preceding section. The larger proportions of red and green assigned to parameter Y indicate the relative importance of these hues in determining brightness, compared to blue.

An NTSC video signal can be converted to an RGB signal using an NTSC decoder, which separates the video signal into the YIQ components, then converts to RGB values. We convert from YIQ space to RGB space with the inverse matrix transformation from Eq. 15-6:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.956 & 0.620 \\ 1.000 & -0.272 & -0.647 \\ 1.000 & -1.108 & 1.705 \end{bmatrix} \cdot \begin{bmatrix} Y \\ I \\ Q \end{bmatrix} \tag{15-7}$$

15-6 CMY COLOR MODEL

A color model defined with the primary colors cyan, magenta, and yellow (CMY) is useful for describing color output to hard-copy devices. Unlike video monitors, which produce a color pattern by combining light from the screen phosphors, hard-copy devices such as plotters produce a color picture by coating a paper with color pigments. We see the colors by reflected light, a subtractive process.

As we have noted, cyan can be formed by adding green and blue light. Therefore, when white light is reflected from cyan-colored ink, the reflected light must have no red component. That is, red light is absorbed, or subtracted, by the ink. Similarly, magenta ink subtracts the green component from incident light, and yellow subtracts the blue component. A unit cube representation for the CMY model is illustrated in Fig. 15-14.

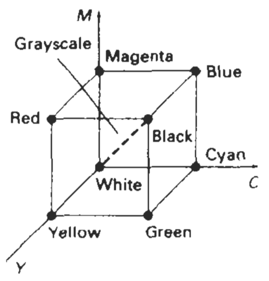


Figure 15-14
The CMY color model, defining colors with a subtractive process inside a unit cube.

In the CMY model, point (1, 1, 1) represents black, because all components of the incident light are subtracted. The origin represents white light. Equal amounts of each of the primary colors produce grays, along the main diagonal of the cube. A combination of cyan and magenta ink produces blue light, because the red and green components of the incident light are absorbed. Other color combinations are obtained by a similar subtractive process.

The printing process often used with the CMY model generates a color point with a collection of four ink dots, somewhat as an RGB monitor uses a collection of three phosphor dots. One dot is used for each of the primary colors (cyan, magenta, and yellow), and one dot is black. A black dot is included because the combination of cyan, magenta, and yellow inks typically produce dark gray instead of black. Some plotters produce different color combinations by spraying the ink for the three primary colors over each other and allowing them to mix before they dry.

We can express the conversion from an RGB representation to a CMY representation with the matrix transformation

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{15-8}$$

where the white is represented in the RGB system as the unit column vector. Similarly, we convert from a CMY color representation to an RGB representation with the matrix transformation

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix} \tag{15-9}$$

where black is represented in the CMY system as the unit column vector.

15-7 HSV COLOR MODEL

Instead of a set of color primaries, the HSV model uses color descriptions that have a more intuitive appeal to a user. To give a color specification, a user selects a spectral color and the amounts of white and black that are to be added to obtain different shades, tints, and tones. Color parameters in this model are *hue* (H), *saturation* (S), and *value* (V).

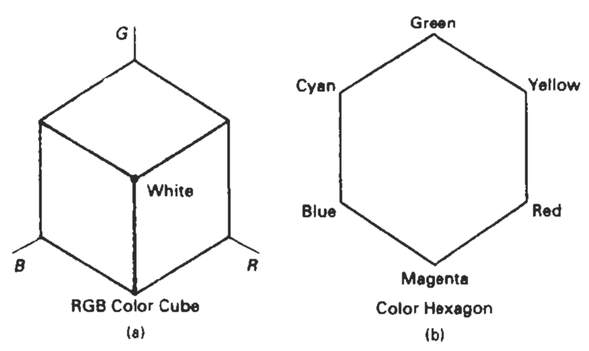


Figure 15-15
When the RGB color cube (a) is viewed along the diagonal from white to black, the color-cube outline is a hexagon (b).

The three-dimensional representation of the HSV model is derived from the RGB cube. If we imagine viewing the cube along the diagonal from the white vertex to the origin (black), we see an outline of the cube that has the hexagon shape shown in Fig. 15-15. The boundary of the hexagon represents the various hues, and it is used as the top of the HSV hexcone (Fig. 15-16). In the hexcone, saturation is measured along a horizontal axis, and value is along a vertical axis through the center of the hexcone.

Hue is represented as an angle about the vertical axis, ranging from 0° at red through 360°. Vertices of the hexagon are separated by 60° intervals. Yellow is at 60°, green at 120°, and cyan opposite red at $H = 180°$. Complementary colors are 180° apart.

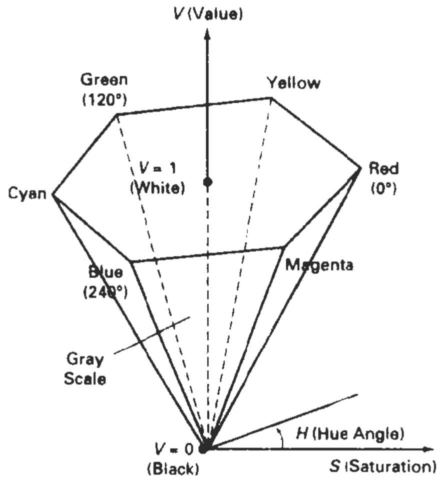


Figure 15-16
The HSV hexcone.

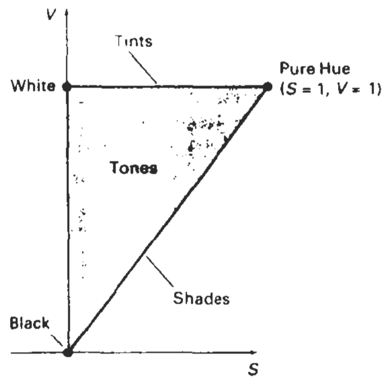


Figure 15-17
Cross section of the HSV hexcone, showing regions for shades, tints, and tones.

Saturation S varies from 0 to 1. It is represented in this model as the ratio of the purity of a selected hue to its maximum purity at $S = 1$. A selected hue is said to be one-quarter pure at the value $S = 0.25$. At $S = 0$, we have the gray scale.

Value V varies from 0 at the apex of the hexcone to 1 at the top. The apex represents black. At the top of the hexcone, colors have their maximum intensity. When $V = 1$ and $S = 1$, we have the "pure" hues. White is the point at $V = 1$ and $S = 0$.

This is a more intuitive model for most users. Starting with a selection for a pure hue, which specifies the hue angle H and sets $V = S = 1$, we describe the color we want in terms of adding either white or black to the pure hue. Adding black decreases the setting for V while S is held constant. To get a dark blue, V could be set to 0.4 with $S = 1$ and $H = 240°$. Similarly, when white is to be added to the hue selected, parameter S is decreased while keeping V constant. A light blue could be designated with $S = 0.3$ while $V = 1$ and $H = 240°$. By adding some black and some white, we decrease both V and S. An interface for this model typically presents the HSV parameter choices in a color palette.

Color concepts associated with the terms shades, tints, and tones are represented in a cross-sectional plane of the HSV hexcone (Fig. 15-17). Adding black to a pure hue decreases V down the side of the hexcone. Thus, various shades are represented with values $S = 1$ and $0 \leq V \leq 1$. Adding white to a pure tone produces different tints across the top plane of the hexcone, where parameter values are $V = 1$ and $0 \leq S \leq 1$. Various tones are specified by adding both black and white, producing color points within the triangular cross-sectional area of the hexcone.

The human eye can distinguish about 128 different hues and about 130 different tints (saturation levels). For each of these, a number of shades (value settings) can be detected, depending on the hue selected. About 23 shades are discernible with yellow colors, and about 16 different shades can be seen at the blue end of the spectrum. This means that we can distinguish about $128 \times 130 \times 23 = 82{,}720$ different colors. For most graphics applications, 128 hues, 8 saturation levels, and 15 value settings are sufficient. With this range of parameters in the HSV color model, 16,384 colors would be available to a user, and the system would need 14 bits of color storage per pixel. Color lookup tables could be used to reduce the storage requirements per pixel and to increase the number of available colors.

15-8 CONVERSION BETWEEN HSV AND RGB MODELS

If HSV color parameters are made available to a user of a graphics package, these parameters are transformed to the RGB settings needed for the color monitor. To determine the operations needed in this transformation, we first consider how the HSV hexcone can be derived from the RGB cube. The diagonal of this cube from black (the origin) to white corresponds to the V axis of the hexcone. Also, each subcube of the RGB cube corresponds to a hexagonal cross-sectional area of the hexcone. At any cross section, all sides of the hexagon and all radial lines from the V axis to any vertex have the value V. For any set of RGB values, V is equal to the maximum value in this set. The HSV point corresponding to the set of RGB values lies on the hexagonal cross section at value V. Parameter S is then determined as the relative distance of this point from the V axis. Parameter H is determined by calculating the relative position of the point within each sextant of the hexagon. An algorithm for mapping any set of RGB values into the corresponding HSV values is given in the following procedure:

```
#include <math.h>

/* Input:   h, s, v in range [0..1]
   Outputs: r, g, b in range [0..1] */
void hsvToRgb(float h, float s, float v, float * r, float * g, float * b)
{
  int i;
  float aa, bb, cc, f;

  if (s == 0) /* Grayscale */
    *r = *g = *b = v;
  else {
    if (h == 1.0) h = 0;
    h *= 6.0;
    i = floor (h);
    f = h - i;
    aa = v * (1 - s);
    bb = v * (1 - (s * f));
    cc = v * (1 - (s * (1 - f)));
    switch (i) {
    case 0: *r = v;  *g = cc; *b = aa; break;
    case 1: *r = bb; *g = v;  *b = aa; break;
    case 2: *r = aa; *g = v;  *b = cc; break;
    case 3: *r = aa; *g = bb; *b = v;  break;
    case 4: *r = cc; *g = aa; *b = v;  break;
    case 5: *r = v;  *g = aa; *b = bb; break;
    }
  }
}
```

We obtain the transformation from HSV parameters to RGB parameters by determining the inverse of the equations in `rgbToHsv procedure`. These inverse operations are carried out for each sextant of the hexcone. The resulting transformation equations are summarized in the following algorithm:

```
#include <math.h>

#define MIN(a,b) (a<b?a:b)
#define MAX(a,b) (a>b?a:b)
```

```
#define NO_HUE   -1

/* Input:   r, g, b in range [0..1]
   Outputs: h, s, v in range [0..1]
*/
void rgbToHsv (float r, float g, float b, float * h, float * s, float * v)
{
  float max = MAX (r, MAX (g, b)), min = MIN (r, MIN (g, b));
  float delta = max - min;

  *v = max;
  if (max != 0.0)
    *s = delta / max;
  else
    *s = 0.0;
  if (*s == 0.0) *h = NO_HUE;
  else {
    if (r == max)
      *h = (g - b) / delta;
    else if (g == max)
      *h = 2 + (b - r) / delta;
    else if (b == max)
      *h = 4 + (r - g) / delta;
    *h *= 60.0;
    if (*h < 0) *h += 360.0;
    *h /= 360.0;
  }
}
```

15-9 HLS COLOR MODEL

Another model based on intuitive color parameters is the HLS system used by Tektronix. This model has the double-cone representation shown in Fig. 15-18. The three color parameters in this model are called *hue* (H), *lightness* (L), and *saturation* (S).

Hue has the same meaning as in the HSV model. It specifies an angle about the vertical axis that locates a chosen hue. In this model, $H = 0°$ corresponds to blue. The remaining colors are specified around the perimeter of the cone in the same order as in the HSV model. Magenta is at 60°, red is at 120°, and cyan is located at $H = 180°$. Again, complementary colors are 180° apart on the double cone.

The vertical axis in this model is called lightness, L. At $L = 0$, we have black, and white is at $L = 1$. Gray scale is along the L axis, and the "pure hues" lie on the $L = 0.5$ plane.

Saturation parameter S again specifies relative purity of a color. This parameter varies from 0 to 1, and pure hues are those for which $S = 1$ and $L = 0.5$. As S decreases, the hues are said to be less pure. At $S = 0$, we have the gray scale.

As in the HSV model, the HLS system allows a user to think in terms of making a selected hue darker or lighter. A hue is selected with hue angle H, and the desired shade, tint, or tone is obtained by adjusting L and S. Colors are made lighter by increasing L and made darker by decreasing L. When S is decreased, the colors move toward gray.

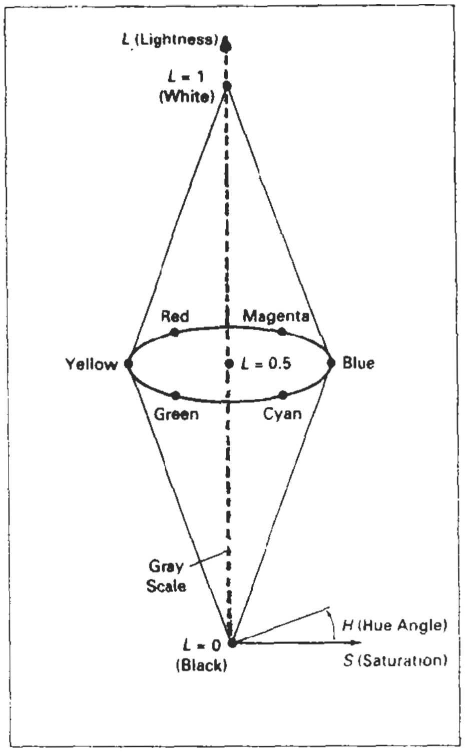


Figure 15-18
The HLS double cone.

15-10 COLOR SELECTION AND APPLICATIONS

A graphics package can provide color capabilities in a way that aids us in making color selections. Various combinations of colors can be selected using sliders and color wheels, and the system can also be designed to aid in the selection of harmonizing colors. In addition, the designer of a package can follow some basic color rules when designing the color displays that are to be presented to a user.

One method for obtaining a set of coordinating colors is to generate the set from some subspace of a color model. If colors are selected at regular intervals along any straight line within the RGB or CMY cube, for example, we can expect to obtain a set of well-matched colors. Randomly selected hues can be expected to produce harsh and clashing color combinations. Another consideration in the selection of color combinations is that different colors are perceived at different depths. This occurs because our eyes focus on colors according to their frequency. Blues, in particular, tend to recede. Displaying a blue pattern next to a red pattern can cause eye fatigue, because we continually need to refocus when our attention is switched from one area to the other. This problem can be reduced by separating these colors or by using colors from one-half or less of the color hexagon in the HSV model. With this technique, a display contains either blues and greens or reds and yellows.

As a general rule, the use of a smaller number of colors produces a more pleasing display than a large number of colors, and tints and shades blend better than pure hues. For a background, gray or the complement of one of the foreground colors is usually best.

SUMMARY

In this chapter, we have discussed the basic properties of light and the concept of a color model. Visible light can be characterized as a narrow frequency distribution within the electromagnetic spectrum. Light sources are described in terms of their dominant frequency (or hue), luminance (or brightness), and purity (or saturation). Complementary color sources are those that combine to produce white light.

One method for defining a color model is to specify a set of two or more primary colors that are combined to produce various other colors. Common color models defined with three primary colors are the RGB and CMY models. Video monitor displays use the RGB model, while hardcopy devices produce color output using the CMY model. Other color models, based on specification of luminance and purity values, include the YIQ, HSV, and HLS color models. Intuitive color models, such as the HSV and HLS models, allow colors to be specified by selecting a value for hue and the amounts of white and black to be added to the selected hue.

Since no model specified with a finite set of color parameters is capable of describing all possible colors, a set of three hypothetical colors, called the CIE primaries, has been adopted as the standard for defining all color combinations. The set of CIE primaries is commonly referred to as the XYZ color model. Plotting normalized values for the X and Y standards produces the CIE chromaticity diagram, which gives a representation for any color in terms of hue and purity. We can use this diagram to compare color gamuts for different color models, to identify complementary colors, and to determine dominant frequency and purity for a given color.

An important consideration in the generation of a color display is the selection of harmonious color combinations. We can do this by following a few simple rules. Coordinating colors usually can be selected from within a small subspace of a color model. Also, we should avoid displaying adjacent colors that differ widely in dominant frequency. And we should limit displays to a small number of color combinations formed with tints and shades, rather than with pure hues.

REFERENCES

A comprehensive discussion of the science of color is given in Wyszecki and Stiles (1982). Color models and color display techniques are discussed in Durrett (1987), Hall (1989), and Travis (1991). Algorithms for various color applications are presented in Glassner (1990), Arvo (1991), and Kirk (1992). For additional information on the human visual system and our perception of light and color, see Glassner (1994).

EXERCISES

15.1. Derive expressions for converting RGB color parameters to HSV values.

15.2. Derive expressions for converting HSV color values to RGB values.

15-3. Write an interactive procedure that allows selection of HSV color parameters from a displayed menu, then the HSV values are to be converted to RGB values for storage in a frame buffer.

15-4. Derive expressions for converting RGB color values to HLS color parameters.

15-5. Derive expressions for converting HLS color values to RGB values.

15-6. Write a program that allows interactive selection of HLS values from a color menu then converts these values to corresponding RGB values.

15-7. Write a program that will produce a set of colors that are linearly interpolated between any two specified positions in RGB space.

15-8. Write an interactive routine for selecting color values from within a specified subspace of RGB space.

15-9. Write a program that will produce a set of colors that are linearly interpolated between any two specified positions in HSV space.

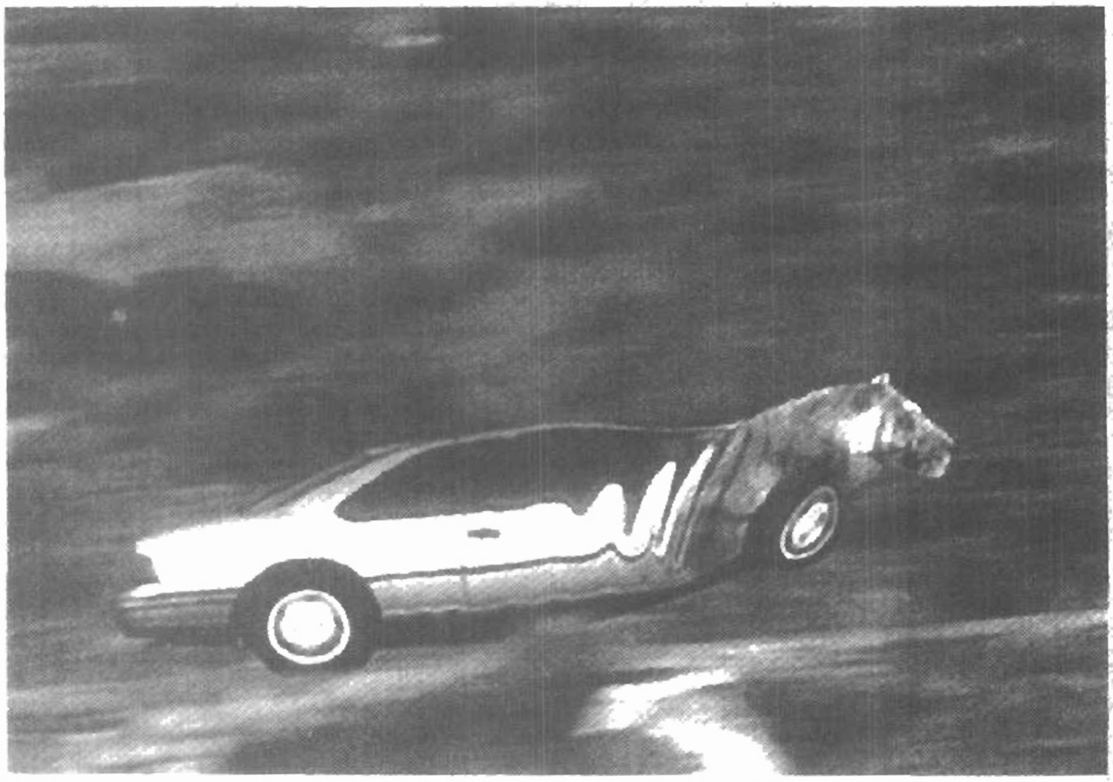
15-10. Write a program that will produce a set of colors that are linearly interpolated between any two specified positions in HLS space.

15-11. Display two RGB color grids, side by side on a video monitor. Fill one grid with a set of randomly selected RGB colors, and fill the other grid with a set of colors that are selected from a small RGB subspace. Experiment with different random selections and different RGB subspaces and compare the two color grids.

15-12. Display the two color grids in Exercise 15-11 using color selections from either the HSV or the HLS color space.

CHAPTER

16 Computer Animation



Some typical applications of computer-generated animation are entertainment (motion pictures and cartoons), advertising, scientific and engineering studies, and training and education. Although we tend to think of animation as implying object motions, the term **computer animation** generally refers to any time sequence of visual changes in a scene. In addition to changing object position with translations or rotations, a computer-generated animation could display time variations in object size, color, transparency, or surface texture. Advertising animations often transition one object shape into another: for example, transforming a can of motor oil into an automobile engine. Computer animations can also be generated by changing camera parameters, such as position, orientation, and focal length. And we can produce computer animations by changing lighting effects or other parameters and procedures associated with illumination and rendering.

Many applications of computer animation require realistic displays. An accurate representation of the shape of a thunderstorm or other natural phenomena described with a numerical model is important for evaluating the reliability of the model Also, simulators for training aircraft pilots and heavy-equipment operators must produce reasonably accurate representations of the environment. Entertainment and advertising applications, on the other hand, are sometimes more interested in visual effects. Thus, scenes may be displayed with exaggerated shapes and unrealistic motions and transformations. There are many entertainment and advertising applications that do require accurate representations for computer-generated scenes. And in some scientific and engineering studies, realism is not a goal. For example, physical quantities are often displayed with pseudo-colors or abstract shapes that change over time to help the researcher understand the nature of the physical process.

16-1 DESIGN OF ANIMATION SEQUENCES

In general, an animation sequence is designed with the following steps:

- Storyboard layout
- Object definitions
- Key-frame specifications
- Generation of in-between frames

This standard approach for animated cartoons is applied to other animation applications as well, although there are many special applications that do not follow this sequence. Real-time computer animations produced by flight simulators, for instance, display motion sequences in response to settings on the aircraft controls. And visualization applications are generated by the solutions of the numerical models. For *frame-by-frame animation*, each frame of the scene is separately generated and stored. Later, the frames can be recorded on film or they can be consecutively displayed in "real-time playback" mode.

The *storyboard* is an outline of the action. It defines the motion sequence as a set of basic events that are to take place. Depending on the type of animation to be produced, the storyboard could consist of a set of rough sketches or it could be a list of the basic ideas for the motion.

An *object definition* is given for each participant in the action. Objects can be defined in terms of basic shapes, such as polygons or splines. In addition, the associated movements for each object are specified along with the shape.

A *key frame* is a detailed drawing of the scene at a certain time in the animation sequence. Within each key frame, each object is positioned according to the time for that frame. Some key frames are chosen at extreme positions in the action; others are spaced so that the time interval between key frames is not too great. More key frames are specified for intricate motions than for simple, slowly varing motions.

In-betweens are the intermediate frames between the key frames. The number of in-betweens needed is determined by the media to be used to display the animation. Film requires 24 frames per second, and graphics terminals are refreshed at the rate of 30 to 60 frames per second. Typically, time intervals for the motion are set up so that there are from three to five in-betweens for each pair of key frames. Depending on the speed specified for the motion, some key frames can be duplicated. For a 1-minute film sequence with no duplication, we would need 1440 frames. With five in-betweens for each pair of key frames, we would need 288 key frames. If the motion is not too complicated, we could space the key frames a little farther apart.

There are several other tasks that may be required, depending on the application. They include motion verification, editing, and production and synchronization of a soundtrack. Many of the functions needed to produce general animations are now computer-generated. Figures 16-1 and 16-2 show examples of computer-generated frames for animation sequences.

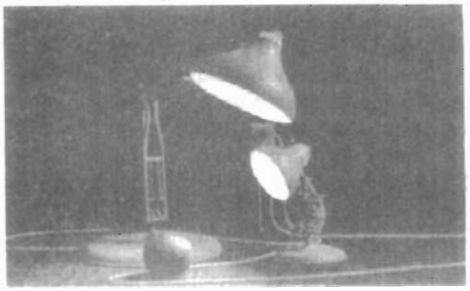


Figure 16-1
One frame from the award-winning computer-animated short film *Luxo Jr.* The film was designed using a key-frame animation system and cartoon animation techniques to provide lifelike actions of the lamps. Final images were rendered with multiple light sources and procedural texturing techniques. *(Courtesy of Pixar. © 1986 Pixar.)*

Figure 16-2
One frame from the short film *Tin Toy*, the first computer-animated film to win an Oscar. Designed using a key-frame animation system, the film also required extensive facial expression modeling. Final images were rendered using procedural shading, self-shadowing techniques, motion blur, and texture mapping. *(Courtesy of Pixar. © 1988 Pixar.)*

16-2 GENERAL COMPUTER-ANIMATION FUNCTIONS

Some steps in the development of an animation sequence are well-suited to computer solution. These include object manipulations and rendering, camera motions, and the generation of in-betweens. Animation packages, such as Wavefront, for example, provide special functions for designing the animation and processing individual objects.

One function available in animation packages is provided to store and manage the object database. Object shapes and associated parameters are stored and updated in the database. Other object functions include those for motion generation and those for object rendering. Motions can be generated according to specified constraints using two-dimensional or three-dimensional transformations. Standard functions can then be applied to identify visible surfaces and apply the rendering algorithms.

Another typical function simulates camera movements. Standard motions are zooming, panning, and tilting. Finally, given the specification for the key frames, the in-betweens can be automatically generated.

16-3 RASTER ANIMATIONS

On raster systems, we can generate real-time animation in limited applications using *raster operations*. As we have seen in Section 5-8, a simple method for translation in the *xy* plane is to transfer a rectangluar block of pixel values from one location to another. Two-dimensional rotations in multiples of 90° are also simple to perform, although we can rotate rectangular blocks of pixels through arbitrary angles using antialiasing procedures. To rotate a block of pixels, we need to determine the percent of area coverage for those pixels that overlap the rotated block. Sequences of raster operations can be executed to produce real-time animation of either two-dimensional or three-dimensional objects, as long as we restrict the animation to motions in the projection plane. Then no viewing or visible-surface algorithms need be invoked.

We can also animate objects along two-dimensional motion paths using the *color-table transformations*. Here we predefine the object at successive positions along the motion path, and set the successive blocks of pixel values to color-table

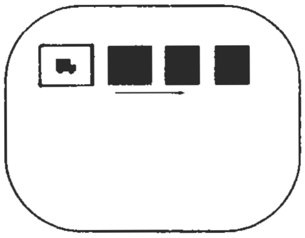


Figure 16-3
Real-time raster color-table animation.

entries. We set the pixels at the first position of the object to "on" values, and we set the pixels at the other object positions to the background color. The animation is then accomplished by changing the color-table values so that the object is "on" at successively positions along the animation path as the preceding position is set to the background intensity (Fig. 16-3).

16-4 COMPUTER-ANIMATION LANGUAGES

Design and control of animation sequences are handled with a set of animation routines. A general-purpose language, such as C, Lisp, Pascal, or FORTRAN, is often used to program the animation functions, but several specialized animation languages have been developed. Animation functions include a graphics editor, a key-frame generator, an in-between generator, and standard graphics routines. The graphics editor allows us to design and modify object shapes, using spline surfaces, constructive solid-geometry methods, or other representation schemes.

A typical task in an animation specification is *scene description*. This includes the positioning of objects and light sources, defining the photometric parameters (light-source intensities and surface-illumination properties), and setting the camera parameters (position, orientation, and lens characteristics). Another standard function is *action specification*. This involves the layout of motion paths for the objects and camera. And we need the usual graphics routines: viewing and perspective transformations, geometric transformations to generate object movements as a function of accelerations or kinematic path specifications, visible-surface identification, and the surface-rendering operations.

Key-frame systems are specialized animation languages designed simply to generate the in-betweens from the user-specified key frames. Usually, each object in the scene is defined as a set of rigid bodies connected at the joints and with a limited number of degrees of freedom. As an example, the single-arm robot in Fig. 16-4 has six degrees of freedom, which are called arm sweep, shoulder swivel, elbow extension, pitch, yaw, and roll. We can extend the number of degrees of freedom for this robot arm to nine by allowing three-dimensional translations for the base (Fig. 16-5). If we also allow base rotations, the robot arm can have a total of 12 degrees of freedom. The human body, in comparison, has over 200 degrees of freedom.

Parameterized systems allow object-motion characteristics to be specified as part of the object definitions. The adjustable parameters control such object characteristics as degrees of freedom, motion limitations, and allowable shape changes.

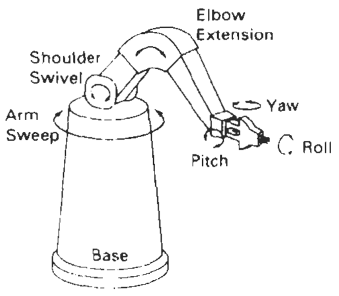


Figure 16-4
Degrees of freedom for a stationary, single-arm robot

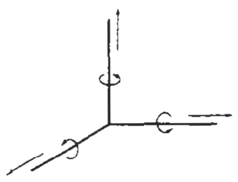


Figure 16-5
Translational and rotational degrees of freedom for the base of the robot arm.

Scripting systems allow object specifications and animation sequences to be defined with a user-input *script*. From the script, a library of various objects and motions can be constructed.

16-5 KEY-FRAME SYSTEMS

We generate each set of in-betweens from the specification of two (or more) key frames. Motion paths can be given with a *kinematic description* as a set of spline curves, or the motions can be *physically based* by specifying the forces acting on the objects to be animated.

For complex scenes, we can separate the frames into individual components or objects called *cels* (celluloid transparencies), an acronym from cartoon animation. Given the animation paths, we can interpolate the positions of individual objects between any two times.

With complex object transformations, the shapes of objects may change over time. Examples are clothes, facial features, magnified detail, evolving shapes, exploding or disintegrating objects, and transforming one object into another object. If all surfaces are described with polygon meshes, then the number of edges per polygon can change from one frame to the next. Thus, the total number of line segments can be different in different frames.

Morphing

Transformation of object shapes from one form to another is called **morphing,** which is a shortened form of metamorphosis. Morphing methods can be applied to any motion or transition involving a change in shape.

Given two key frames for an object transformation, we first adjust the object specification in one of the frames so that the number of polygon edges (or the number of vertices) is the same for the two frames. This preprocessing step is illustrated in Fig. 16-6. A straight-line segment in key frame k is transformed into two line segments in key frame $k + 1$. Since key frame $k + 1$ has an extra vertex, we add a vertex between vertices 1 and 2 in key frame k to balance the number of vertices (and edges) in the two key frames. Using linear interpolation to generate the in-betweens, we transition the added vertex in key frame k into vertex 3' along the straight-line path shown in Fig. 16-7. An example of a triangle linearly expanding into a quadrilateral is given in Fig. 16-8. Figures 16-9 and 16-10 show examples of morphing in television advertising.

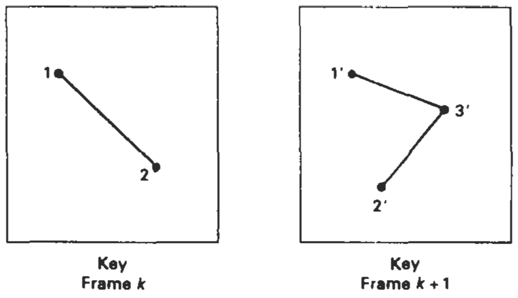


Figure 16-6
An edge with vertex positions 1 and 2 in key frame k evolves into two connected edges in key frame $k + 1$.

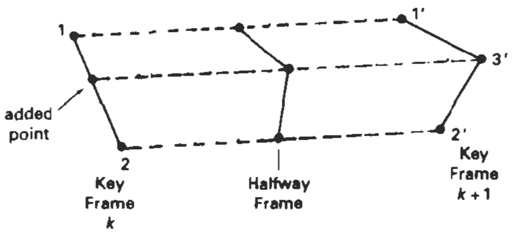


Figure 16-7
Linear interpolation for transforming a line segment in key frame k into two connected line segments in key frame $k + 1$.

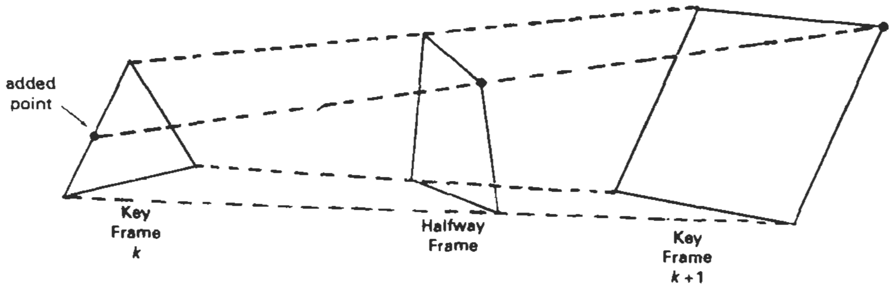


Figure 16-8
Linear interpolation for transforming a triangle into a quadrilateral.

We can state general preprocessing rules for equalizing key frames in terms of either the number of edges or the number of vertices to be added to a key frame. Suppose we equalize the edge count, and parameters L_k and L_{k+1} denote the number of line segments in two consecutive frames. We then define

$$L_{\max} = \max(L_k, L_{k+1}), \qquad L_{\min} = \min(L_k, L_{k+1}) \tag{16-1}$$

and

$$N_e = L_{\max} \bmod L_{\min}$$

$$N_s = \operatorname{int}\left(\frac{L_{\max}}{L_{\min}}\right) \tag{16-2}$$

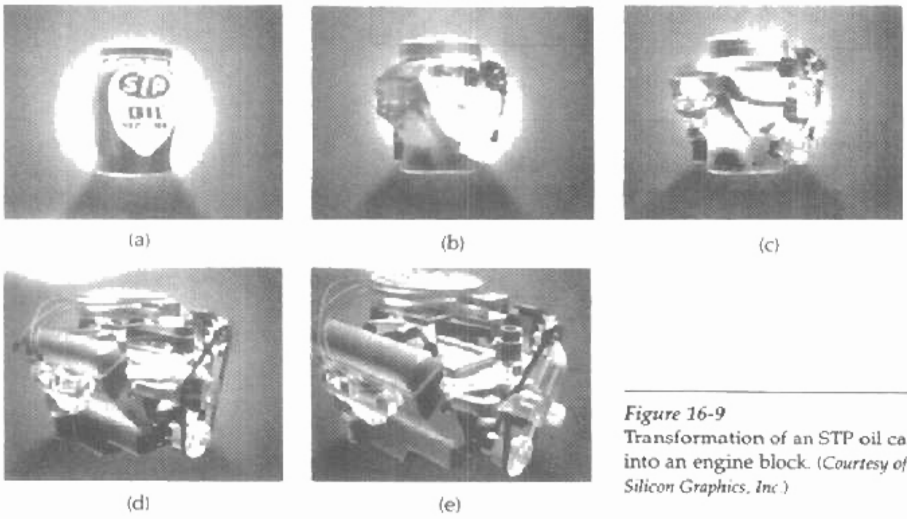


Figure 16-9
Transformation of an STP oil can into an engine block. (*Courtesy of Silicon Graphics, Inc.*)

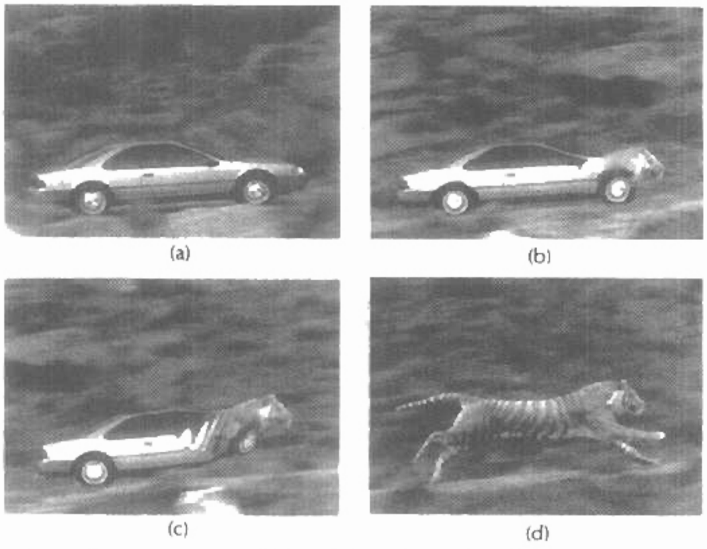


Figure 16-10
Transformation of a moving automobile into a running tiger. (*Courtesy of Exxon Company USA and Pacific Data Images.*)

Then the preprocessing is accomplished by

1. dividing N_e edges of $keyframe_{\min}$ into $N_s + 1$ sections
2. dividing the remaining lines of $keyframe_{\min}$ into N_s sections

As an example, if $L_k = 15$ and $L_{k+1} = 11$, we would divide 4 lines of $keyframe_{k+1}$ into 2 sections each. The remaining lines of $keyframe_{k+1}$ are left intact.

If we equalize the vertex count, we can use parameters V_k and V_{k+1} to denote the number of vertices in the two consecutive frames. In this case, we define

$$V_{\max} = \max(V_k, V_{k+1}), \qquad V_{\min} = \min(V_k, V_{k+1}) \tag{16-3}$$

and

$$N_{ls} = (V_{\max} - 1) \bmod (V_{\min} - 1)$$
$$N_p = \text{int}\left(\frac{V_{\max} - 1}{V_{\min} - 1}\right) \tag{16-4}$$

Preprocessing using vertex count is performed by

1. adding N_p points to N_{ls} line sections of $keyframe_{\min}$
2. adding $N_p - 1$ points to the remaining edges of $keyframe_{\min}$

For the triangle-to-quadrilateral example, $V_k = 3$ and $V_{k+1} = 4$. Both N_{ls} and N_p are 1, so we would add one point to one edge of $keyframe_k$. No points would be added to the remaining lines of $keyframe_{k+1}$.

Simulating Accelerations

Curve-fitting techniques are often used to specify the animation paths between key frames. Given the vertex positions at the key frames, we can fit the positions with linear or nonlinear paths. Figure 16-11 illustrates a nonlinear fit of key-frame positions. This determines the trajectories for the in-betweens. To simulate accelerations, we can adjust the time spacing for the in-betweens.

For constant speed (zero acceleration), we use equal-interval time spacing for the in-betweens. Suppose we want n in-betweens for key frames at times t_1 and t_2 (Fig. 16-12). The time interval between key frames is then divided into $n + 1$ subintervals, yielding an in-between spacing of

$$\Delta t = \frac{t_2 - t_1}{n + 1} \tag{16-5}$$

We can calculate the time for any in-between as

$$tB_j = t_1 + j\,\Delta t, \qquad j = 1, 2, \ldots, n \tag{16-6}$$

and determine the values for coordinate positions, color, and other physical parameters.

Nonzero accelerations are used to produce realistic displays of speed changes, particularly at the beginning and end of a motion sequence. We can model the start-up and slow-down portions of an animation path with spline or

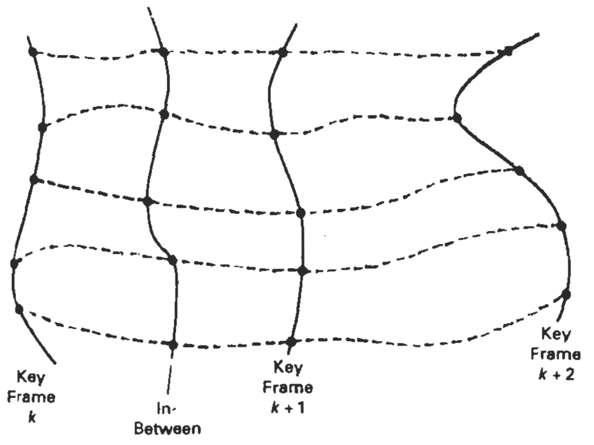


Figure 16-11
Fitting key-frame vertex positions with nonlinear splines.

trignometric functions. Parabolic and cubic time functions have been applied to acceleration modeling, but trignometric functions are more commonly used in animation packages.

To model increasing speed (positive acceleration), we want the time spacing between frames to increase so that greater changes in position occur as the object moves faster. We can obtain an increasing interval size with the function

$$1 - \cos\theta, \qquad 0 < \theta < \pi/2$$

For n in-betweens, the time for the jth in-between would then be calculated as

$$tB_j = t_1 + \Delta t\left[1 - \cos\frac{j\pi}{2(n+1)}\right], \qquad j = 1, 2, \ldots, n \tag{16-7}$$

where Δt is the time difference between the two key frames. Figure 16-13 gives a plot of the trigonometric acceleration function and the in-between spacing for $n = 5$.

We can model decreasing speed (deceleration) with $\sin\theta$ in the range $0 < \theta < \pi/2$. The time position of an in-between is now defined as

$$tB_j = t_1 + \Delta t \sin\frac{j\pi}{2(n+1)}, \qquad j = 1, 2, \ldots, n \tag{16-8}$$

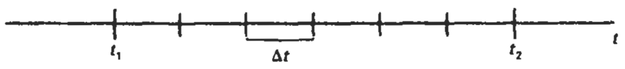


Figure 16-12
In-between positions for motion at constant speed.

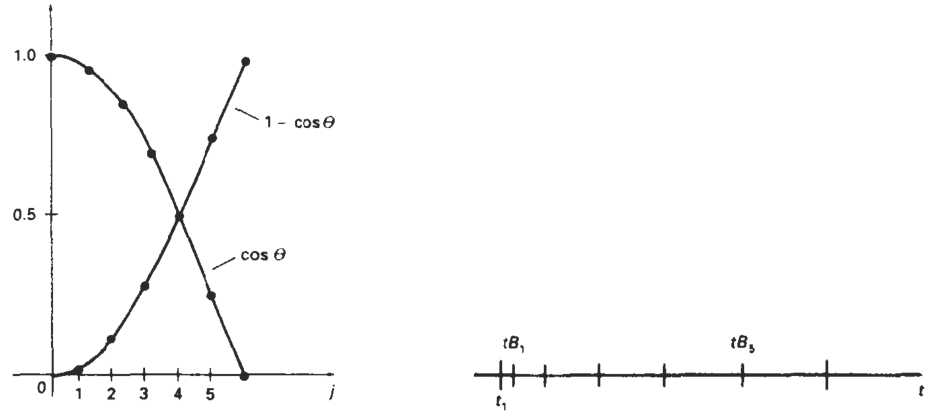


Figure 16-13
A trigonometric acceleration function and the corresponding in-between spacing for $n = 5$ and $\theta = j\pi/12$ in Eq. 16-7, producing increased coordinate changes as the object moves through each time interval.

A plot of this function and the decreasing size of the time intervals is shown in Fig. 16-14 for five in-betweens.

Often, motions contain both speed-ups and slow-downs. We can model a combination of increasing–decreasing speed by first increasing the in-between time spacing, then we decrease this spacing. A function to accomplish these time changes is

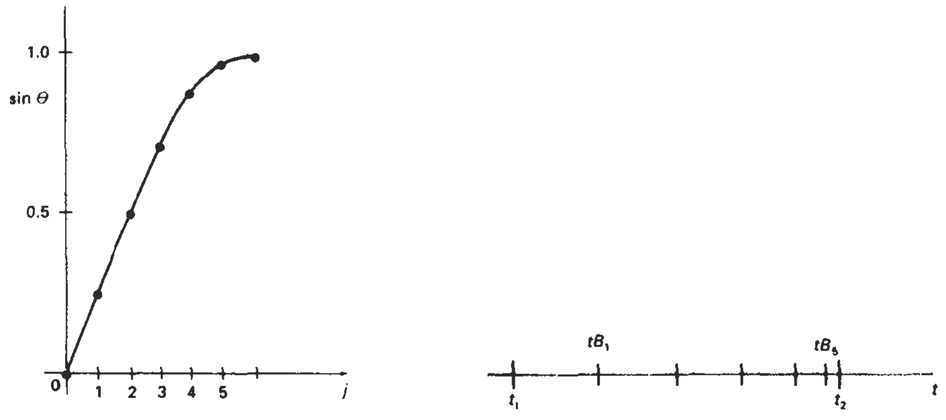


Figure 16-14
A trigonometric deceleration function and the corresponding in-between spacing for $n = 5$ and $\theta = j\pi/12$ in Eq. 16-8, producing decreased coordinate changes as the object moves through each time interval.

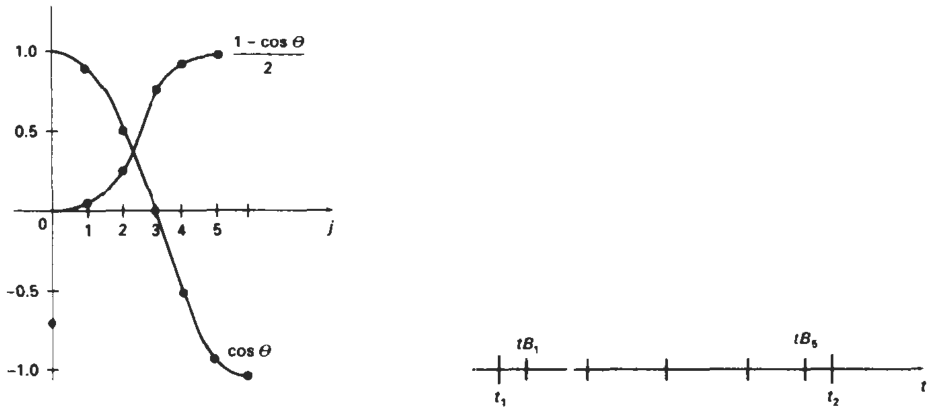


Figure 16-15
A trigonometric accelerate–decelerate function and the corresponding in-between spacing for $n = 5$ in Eq. 16-9.

$$\frac{1}{2}(1 - \cos\theta), \qquad 0 < \theta < \pi/2$$

The time for the jth in-between is now calculated as

$$tB_j = t_1 + \Delta t\left\{\frac{1 - \cos[j\pi/(n + 1)]}{2}\right\}, \qquad j = 1, 2, \ldots, n \qquad (16\text{-}9)$$

with Δt denoting the time difference for the two key frames. Time intervals for the moving object first increase, then the time intervals decrease, as shown in Fig. 16-15.

Processing the in-betweens is simplified by initially modeling "skeleton" (wireframe) objects. This allows interactive adjustment of motion sequences. After the animation sequence is completely defined, objects can be fully rendered.

16-6 MOTION SPECIFICATIONS

There are several ways in which the motions of objects can be specified in an animation system. We can define motions in very explicit terms, or we can use more abstract or more general approaches.

Direct Motion Specification

The most straightforward method for defining a motion sequence is *direct specification* of the motion parameters. Here, we explicitly give the rotation angles and translation vectors. Then the geometric transformation matrices are applied to transform coordinate positions. Alternatively, we could use an approximating

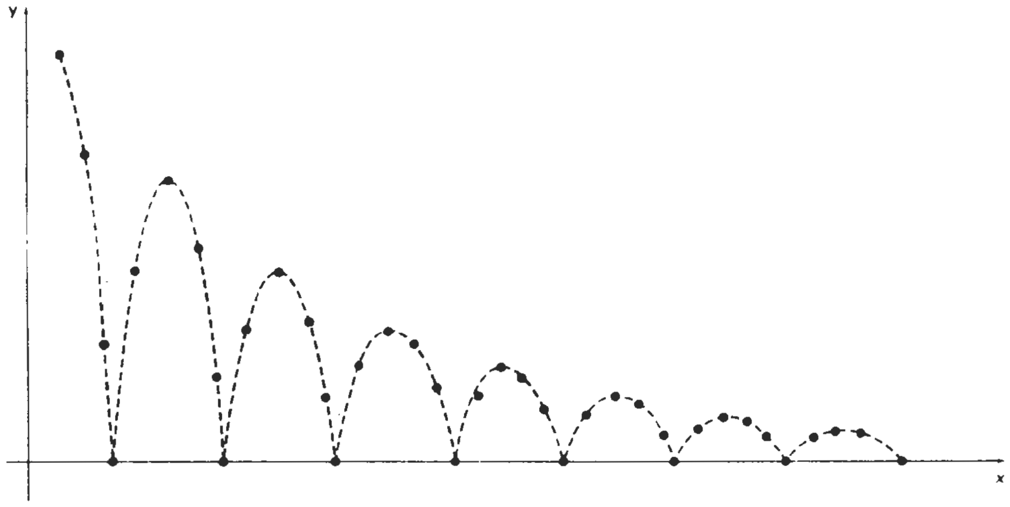


Figure 16-16
Approximating the motion of a bouncing ball with a damped *sine* function (Eq. 16-10).

equation to specify certain kinds of motions. We can approximate the path of a bouncing ball, for instance, with a damped, rectified, *sine* curve (Fig. 16-16):

$$y(x) = A\,|\sin(\omega x + \theta_0)|\,e^{-kx} \qquad (16\text{-}10)$$

where A is the initial amplitude, ω is the angular frequence, θ_0 is the phase angle, and k is the damping constant. These methods can be used for simple user-programmed animation sequences.

Goal-Directed Systems

At the opposite extreme, we can specify the motions that are to take place in general terms that abstractly describe the actions. These systems are referred to as *goal directed* because they determine specific motion parameters given the goals of the animation. For example, we could specify that we want an object to "walk" or to "run" to a particular destination. Or we could state that we want an object to "pick up" some other specified object. The input directives are then interpreted in terms of component motions that will accomplish the selected task. Human motions, for instance, can be defined as a hierarchical structure of submotions for the torso, limbs, and so forth.

Kinematics and Dynamics

We can also construct animation sequences using *kinematic* or *dynamic* descriptions. With a kinematic description, we specify the animation by giving motion parameters (position, velocity, and acceleration) without reference to the forces that cause the motion. For constant velocity (zero acceleration), we designate the motions of rigid bodies in a scene by giving an initial position and velocity vector for each object. As an example, if a velocity is specified as (3, 0, −4) km/sec, then this vector gives the direction for the straight-line motion path and the speed (magnitude of velocity) is 5 km/sec. If we also specify accelerations (rate of change of velocity), we can generate speed-ups, slow-downs, and curved motion paths. Kinematic specification of a motion can also be given by simply describing the motion path. This is often done using spline curves.

An alternate approach is to use *inverse kinematics*. Here, we specify the initial and final positions of objects at specified times and the motion parameters are computed by the system. For example, assuming zero accelerations, we can determine the constant velocity that will accomplish the movement of an object from the initial position to the final position. This method is often used with complex objects by giving the positions and orientations of an end node of an object, such as a hand or a foot. The system then determines the motion parameters of other nodes to accomplish the desired motion.

Dynamic descriptions on the other hand, require the specification of the forces that produce the velocities and accelerations. Descriptions of object behavior under the influence of forces are generally referred to as a *physically based modeling* (Chapter 10). Examples of forces affecting object motion include electromagnetic, gravitational, friction, and other mechanical forces.

Object motions are obtained from the force equations describing physical laws, such as Newton's laws of motion for gravitational and friction processes, Euler or Navier–Stokes equations describing fluid flow, and Maxwell's equations for electromagnetic forces. For example, the general form of Newton's second law for a particle of mass m is

$$\mathbf{F} = \frac{d}{dt}(m\mathbf{v}) \tag{16-11}$$

with **F** as the force vector, and **v** as the velocity vector. If mass is constant, we solve the equation $\mathbf{F} = m\mathbf{a}$, where **a** is the acceleration vector. Otherwise, mass is a function of time, as in relativistic motions or the motions of space vehicles that consume measurable amounts of fuel per unit time. We can also use *inverse dynamics* to obtain the forces, given the initial and final positions of objects and the type of motion.

Applications of physically based modeling include complex rigid-body systems and such nonrigid systems as cloth and plastic materials. Typically, numerical methods are used to obtain the motion parameters incrementally from the dynamical equations using initial conditions or boundary values.

SUMMARY

A computer-animation sequence can be set up by specifying the storyboard, the object definitions, and the key frames. The storyboard is an outline of the action, and the key frames define the details of the object motions for selected positions in the animation. Once the key frames have been established, a sequence of in-betweens can be generated to construct a smooth motion from one key frame to the next. A computer animation can involve motion specifications for the objects in a scene as well as motion paths for a camera that moves through the scene. Computer-animation systems include key-frame systems, parameterized systems, and scripting systems. For motion in two-dimensions, we can use the raster-animation techniques discussed in Chapter 5.

For some applications, key frames are used to define the steps in a morphing sequence that changes one object shape into another. Other in-between methods include generation of variable time intervals to simulate accelerations and decelerations in the motion.

Motion specifications can be given in terms of translation and rotation parameters, or motions can be described with equations or with kinematic or dynamic parameters. Kinematic motion descriptions specify positions, velocities, and accelerations. Dynamic motion descriptions are given in terms of the forces acting on the objects in a scene.

REFERENCES

For additional information on computer animation systems and techniques, see Magnenat-Thalmann and Thalmann (1985), Barzel (1992), and Watt and Watt (1992). Algorithms for animation applications are presented in Glassner (1990), Arvo (1991), Kirk (1992), Gascuel (1993), Ngo and Marks (1993), van de Panne and Fiume (1993), and in Snyder et al. (1993). Morphing techniques are discussed in Beier and Neely (1992), Hughes (1992), Kent, Carlson, and Parent (1992), and in Sederberg and Greenwood (1992). A discussion of animation techniques in PHIGS is given in Gaskins (1992).

EXERCISES

16-1. Design a storyboard layout and accompanying key frames for an animation of a single polyhedron.

16-2. Write a program to generate the in-betweens for the key frames specified in Exercise 16-1 using linear interpolation.

16-3. Expand the animation sequence in Exercise 16-1 to include two or more moving objects.

16-4. Write a program to generate the in-betweens for the key frames in Exercise 16-3 using linear interpolation.

16-5. Write a morphing program to transform a sphere into a specified polyhedron.

16-6. Set up an animation specification involving accelerations and implement Eq. 16-7.

16-7. Set up an animation specification involving both accelerations and decelerations and implement the in-between spacing calculations given in Eqs. 16-7 and 16-8.

16-8. Set up an animation specification implementing the acceleration-deceleration calculations of Eq. 16-9.

16-9. Write a program to simulate the linear, two-dimensional motion of a filled circle inside a given rectangular area. The circle is to be given an initial velocity, and the circle is to rebound from the walls with the angle of reflection equal to the angle of incidence.

16-10. Convert the program of Exercise 16-9 into a ball and paddle game by replacing one side of the rectangle with a short line segment that can be moved back and forth to intercept the circle path. The game is over when the circle escapes from the interior of the rectangle. Initial input parameters include circle position, direction, and speed. The game score can include the number of times the circle is intercepted by the paddle.

16-11. Expand the program of Exercise 16-9 to simulate the three-dimensional motion of a sphere moving inside a parallelepiped. Interactive viewing parameters can be set to view the motion from different directions.

16-12. Write a program to implement the simulation of a bouncing ball using Eq. 16-10.

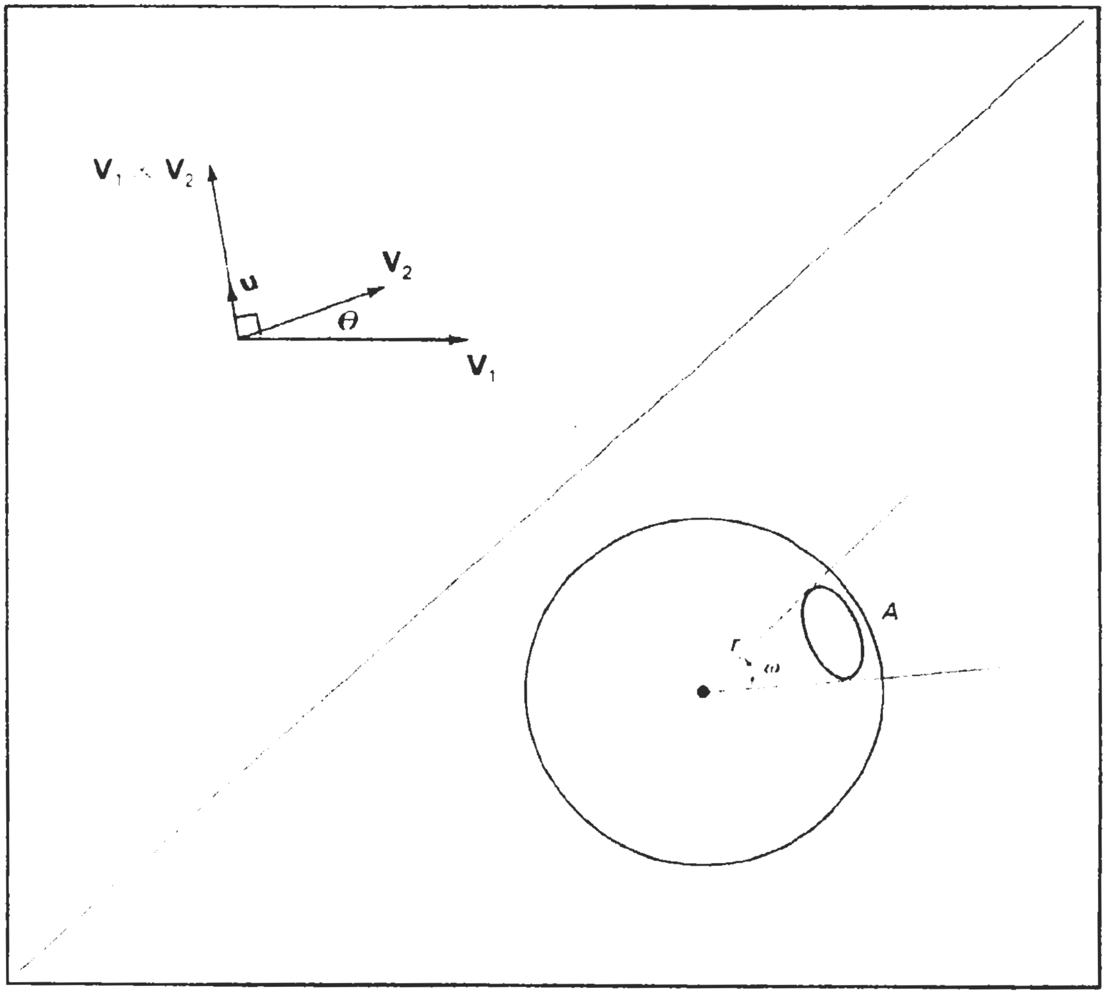
16-13. Write a program to implement the motion of a bouncing ball using a downward gravitational force and a ground-plane friction force. Initially, the ball is to be projected into space with a given velocity vector.

16-14. Write a program to implement the two-player pillbox game. The game can be implemented on a flat plane with fixed pillbox positions, or random terrain features and pillbox placements can be generated at the start of the game.

16-15. Write a program to implement dynamic motion specifications. Specify a scene with two or more objects, initial motion parameters, and specified forces. Then generate the animation from the solution of the force equations. (For example, the objects could be the earth, moon, and sun with attractive gravitational forces that are proportional to mass and inversely proportional to distance squared.)

APPENDIX

A Mathematics for Computer Graphics



Computer graphics algorithms make use of many mathematical concepts and techniques. Here, we provide a brief reference for the topics from analytic geometry, linear algebra, vector analysis, tensor analysis, complex numbers, numerical analysis, and other areas that are referred to in the graphics algorithms discussed throughout this book.

A-1 COORDINATE REFERENCE FRAMES

Graphics packages typically require that coordinate parameters be specified with respect to Cartesian reference frames. But in many applications, non-Cartesian coordinate systems are useful. Spherical, cylindrical, or other symmetries often can be exploited to simplify expressions involving object descriptions or manipulations. Unless a specialized graphics system is available, however, we must first convert any non-Cartesian descriptions to Cartesian coordinates. In this section, we first review standard Cartesian coordinate systems, then we consider a few common non-Cartesian systems.

Two-Dimensional Cartesian Reference Frames

Figure A-1 shows two possible orientations for a Cartesian screen reference system. The standard coordinate orientation shown in Fig. A-1(a), with the coordinate origin in the lower-left corner of the screen, is a commonly used reference

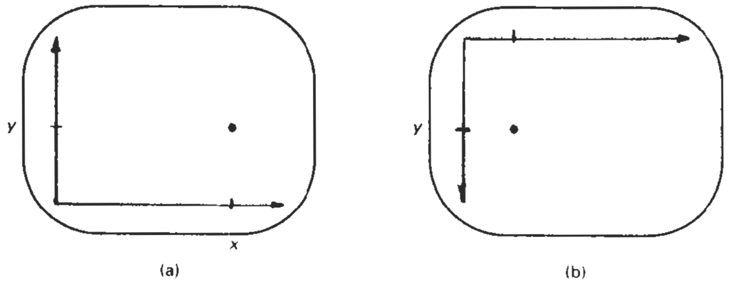


Figure A-1
Screen Cartesian reference systems: (a) coordinate origin at the lower-left screen corner and (b) coordinate origin in the upper-left corner.

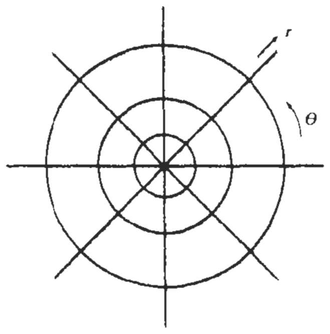


Figure A-2
A polar coordinate reference frame, formed with concentric circles and radial lines.

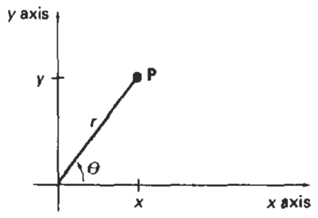


Figure A-3
Relationship between polar and Cartesian coordinates.

frame. Some systems, particularly personal computers, orient the Cartesian reference frame as in Fig. A-1(b), with the origin at the upper left corner. In addition, it is possible in some graphics packages to select a position, such as the center of the screen, for the coordinate origin.

Polar Coordinates in the *xy* Plane

A frequently used non-Cartesian system is a polar-coordinate reference frame (Fig. A-2), where a coordinate position is specified with a radial distance r from the coordinate origin, and an angular displacement θ from the horizontal. Positive angular displacements are counterclockwise, and negative angular displacements are clockwise. Angle θ can be measured in degrees, with one complete counterclockwise revolution about the origin as 360°. The relation between Cartesian and polar coordinates is shown in Fig. A-3. Considering the right triangle in Fig. A-4, and using the definition of the trigonometric functions, we transform from polar coordinates to Cartesian coordinates with the expressions

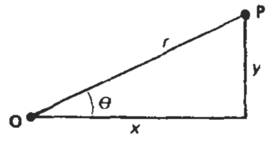


Figure A-4
Right triangle with hypotenuse r and sides x and y.

$$x = r\cos\theta, \qquad y = r\sin\theta \tag{A-1}$$

The inverse transformation from Cartesian to polar coordinates is

$$r = \sqrt{x^2 + y^2}, \qquad \theta = \tan^{-1}\left(\frac{y}{x}\right) \tag{A-2}$$

Other conics, besides circles, can be used to specify coordinate positions. For example, using concentric ellipses instead of circles, we can give coordinate positions in elliptical coordinates. Similarly, other types of symmetries can be exploited with hyperbolic or parabolic plane coordinates.

Angular values can be specified in degrees or they can be given in dimensionless units (radians). Figure A-5 shows two intersecting lines in a plane and a circle centered on the intersection point **P**. The value of angle θ in radians is then given by

$$\theta = \frac{s}{r} \tag{A-3}$$

where s is the length of the circular arc subtending θ, and r is the radius of the circle. Total angular distance around point **P** is the length of the circle perimeter ($2\pi r$) divided by r, or 2π radians.

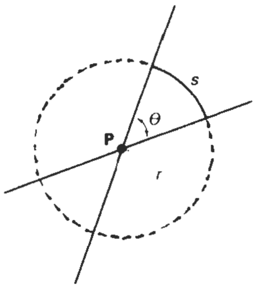


Figure A-5
An angle θ subtended by a circular arc of length s and radius r.

Three-Dimensional Cartesian Reference Frames

Figure A-6(a) shows the conventional orientation for the coordinate axes in a three-dimensional Cartesian reference system. This is called a right-handed system because the right-hand thumb points in the positive z direction when we imagine grasping the z axis with the fingers curling from the positive x axis to the positive y axis (through 90°), as illustrated in Fig. A-6(b). Most computer graphics packages require object descriptions and manipulations to be specified in right-handed Cartesian coordinates. For discussions throughout this book (including the appendix), we assume that all Cartesian reference frames are right-handed.

Another possible arrangement of Cartesian axes is the left-handed system shown in Fig. A-7. For this system, the left-hand thumb points in the positive z direction when we imagine grasping the z axis so that the fingers of the left hand curl from the positive x axis to the positive y axis through 90°. This orientation of axes is sometimes convenient for describing depth of objects relative to a display screen. If screen locations are described in the xy plane of a left-handed system with the coordinate origin in the lower-left screen corner, positive z values indicate positions behind the screen, as in Fig. A-7(a). Larger values along the positive z axis are then interpreted as being farther from the viewer.

Three-Dimensional Curvilinear Coordinate Systems

Any non-Cartesian reference frame is referred to as a **curvilinear coordinate system**. The choice of coordinate system for a particular graphics application depends on a number of factors, such as symmetry, ease of computation, and visu-

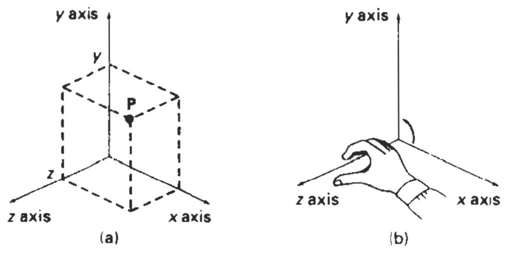


Figure A-6
Coordinate representation of a point **P** at position (x, y, z) in a right-handed Cartesian reference system.

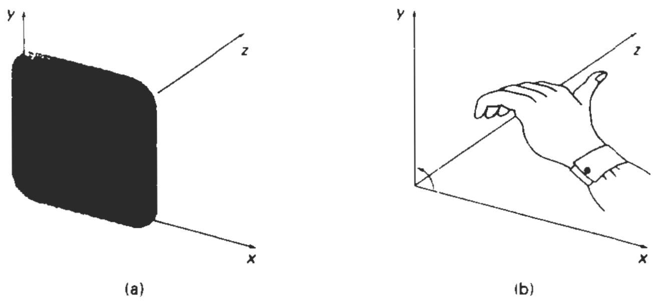


Figure A-7
Left-handed Cartesian coordinate system superimposed on the surface of a video monitor.

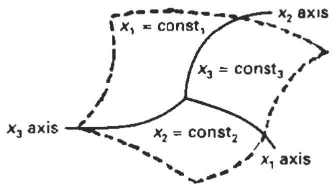


Figure A-8
A general curvilinear coordinate reference frame.

alization advantages. Figure A-8 shows a general curvilinear coordinate reference frame formed with three *coordinate surfaces,* where each surface has one coordinate held constant. For instance, the x_1x_2 surface is defined with x_3 held constant. *Coordinate axes* in any reference frame are the intersection curves of the coordinate surfaces. If the coordinate surfaces intersect at right angles, we have an **orthogonal curvilinear coordinate system**. Nonorthogonal reference frames are useful for specialized spaces, such as visualizations of motions governed by the laws of general relativity, but in general, they are used less frequently in graphics applications than orthogonal systems.

A *cylindrical-coordinate* specification of a spatial position is shown in Fig. A-9 in relation to a Cartesian reference frame. The surface of constant ρ is a vertical

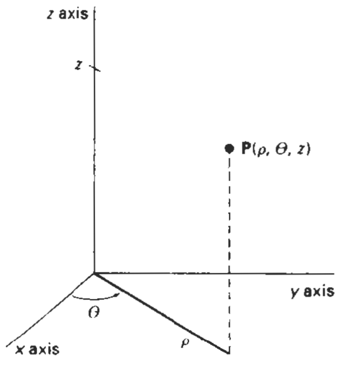


Figure A-9
Cylindrical coordinates: ρ, θ, z.

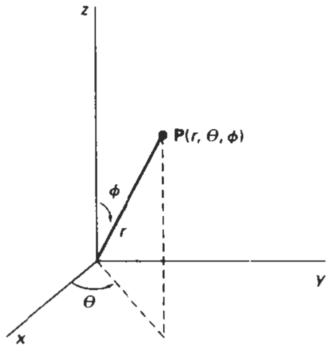


Figure A-10
Spherical coordinates: r, θ, ϕ.

cylinder; the surface of constant θ is a vertical plane containing the z axis; and the surface of constant z is a horizontal plane parallel to the Cartesian xy plane. We transform from a cylindrical coordinate specification to a Cartesian reference frame with the calculations

$$x = \rho\cos\theta, \qquad y = \rho\sin\theta, \qquad z = z \tag{A-4}$$

Figure A-10 shows a *spherical-coordinate* specification of a spatial position in reference to a Cartesian reference frame. Spherical coordinates are sometimes referred to as *polar coordinates in space*. The surface of constant r is a sphere; the surface of constant θ is a vertical plane containing the z axis (same θ surface as in cylindrical coordinates); and the surface of constant ϕ is a cone with apex at the coordinate origin. If $\phi < 90°$, the cone is above the xy plane. If $\phi > 90°$, the cone is below the xy plane. We transfrom from a spherical-coordinate specification to a Cartesian reference frame with the calculations

$$x = r\cos\theta\sin\phi, \qquad y = r\sin\theta\sin\phi, \qquad z = r\cos\phi \tag{A-5}$$

Solid Angle

We define a solid angle in analogy with that for a two-dimensional angle θ between two intersecting lines (Eq. A-3). Instead of a circle, we consider any sphere with center position **P**. The solid angle ω within a cone-shaped region with apex at **P** is defined as

$$\omega = \frac{A}{r^2} \tag{A-6}$$

where A is the area of the spherical surface intersected by the cone (Fig. A-11), and r is the radius of the sphere.

Also, in analogy with two-dimensional polar coordinates, the dimensionless unit for solid angles is called the **steradian**. The total solid angle about a point is the total area of the spherical surface ($4\pi r^2$) divided by r^2, or 4π steradians.

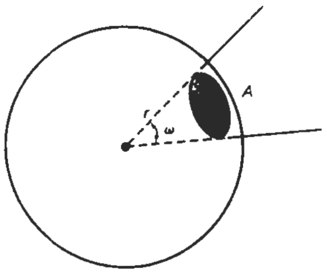


Figure A-11
A solid angle ω subtended by a spherical surface patch of area A with radius r.

A-2 POINTS AND VECTORS

There is a fundamental difference between the concept of a point and that of a vector. A point is a position specified with coordinate values in some reference frame, so that the distance from the origin depends on the choice of reference frame. Figure A-12 illustrates coordinate specification in two reference frames. In frame A, point coordinates are given by the values of the ordered pair (x, y). In frame B, the same point has coordinates (0, 0) and the distance to the origin of frame B is 0.

A vector, on the other hand, is defined as the difference between two point positions. Thus, for a two-dimensional vector (Fig. A-13), we have

$$\begin{aligned} \mathbf{V} &= \mathbf{P}_2 - \mathbf{P}_1 \\ &= (x_2 - x_1, y_2 - y_1) \\ &= (V_x, V_y) \end{aligned} \tag{A-7}$$

where the Cartesian *components* (or Cartesian *elements*) V_x and V_y are the projections of $\mathbf{V}$ onto the x and y axes. Given two point positions, we can obtain vector components in the same way for any coordinate reference frame.

We can describe a vector as a *directed line segment* that has two fundamental properties: magnitude and direction. For the two-dimensional vector in Fig. A-13, we calculate vector magnitude using the Pythagorean theorem:

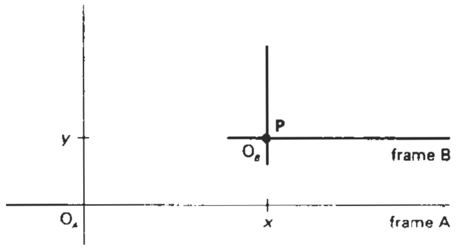


Figure A-12
Position of point **P** with respect to two different Cartesian reference frames.

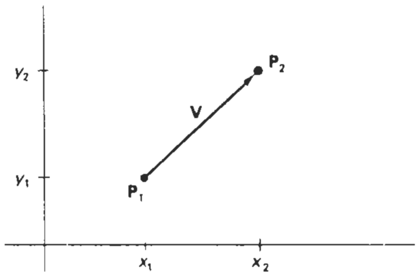


Figure A-13
Vector V in the *xy* plane of a Cartesian reference frame.

$$|\mathbf{V}| = \sqrt{V_x^2 + V_y^2} \tag{A-8}$$

The direction for this two-dimensional vector can be given in terms of the angular displacement from the *x* axis as

$$\alpha = \tan^{-1}\left(\frac{V_y}{V_x}\right) \tag{A-9}$$

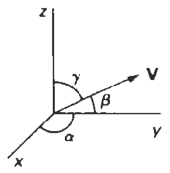


Figure A-14
Direction angles α, β, and γ.

A vector has the same properties (magnitude and direction) no matter where we position the vector within a single coordinate system. And the vector magnitude is independent of the coordinate representation. Of course, if we change the coordinate representation, the values for the vector components change.

For a three-dimensional Cartesian space, we calculate the vector magnitude as

$$|\mathbf{V}| = \sqrt{V_x^2 + V_y^2 + V_z^2} \tag{A-10}$$

Vector direction is given with the *direction angles*, α, β, and γ, that the vector makes with each of the coordinate axes (Fig. A-14). Direction angles are the positive angles that the vector makes with each of the positive coordinate axes. We calculate these angles as

$$\cos\alpha = \frac{V_x}{|\mathbf{V}|}, \qquad \cos\beta = \frac{V_y}{|\mathbf{V}|}, \qquad \cos\gamma = \frac{V_z}{|\mathbf{V}|} \tag{A-11}$$

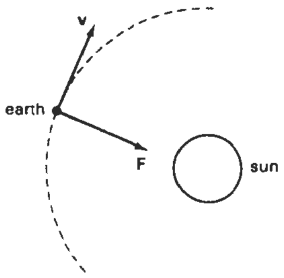


Figure A-15
A gravitational force vector F and a velocity vector v.

The values $\cos\alpha$, $\cos\beta$, and $\cos\gamma$ are called the *direction cosines* of the vector. Actually, we only need to specify two of the direction cosines to give the direction of **V**, since

$$\cos^2\alpha + \cos^2\beta + \cos^2\gamma = 1 \tag{A-12}$$

Vectors are used to represent any quantities that have the properties of magnitude and direction. Two common examples are force and velocity (Fig. A-15). A force can be thought of as a push or a pull of a certain amount in a par-

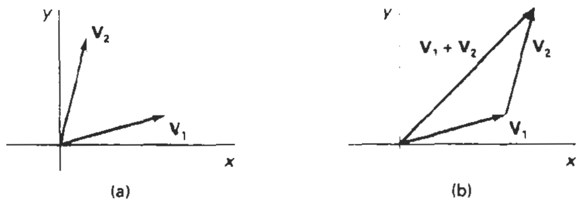


Figure A-16
Two vectors (a) can be added geometrically by positioning the two vectors end to end (b) and drawing the resultant vector from the start of the first vector to the tip of the second vector.

ticular direction. A velocity vector specifies how fast (*speed*) an object is moving in a certain direction.

Vector Addition and Scalar Multiplication

By definition, the sum of two vectors is obtained by adding corresponding components:

$$\mathbf{V}_1 + \mathbf{V}_2 = (V_{1x} + V_{2x}, V_{1y} + V_{2y}, V_{1z} + V_{2z}) \tag{A-13}$$

Vector addition is illustrated geometrically in Fig. A-16. We obtain the vector sum by placing the start position of one vector at the tip of the other vector and drawing the summation vector as in Fig. A-16.

Addition of vectors and scalars is undefined, since a scalar always has only one numerical value while a vector has n numerical components in an n-dimensional space. Scalar multiplication of a three-dimensional vector is defined as

$$a\mathbf{V} = (aV_x, aV_y, aV_z) \tag{A-14}$$

For example, if the scalar parameter a has the value 2, each component of $\mathbf{V}$ is doubled.

We can also multiply two vectors, but there are two possible ways to do this. The multiplication can be carried out so that either we obtain another vector or we obtain a scalar quantity.

Scalar Product of Two Vectors

Vector multiplication for producing a scalar is defined as

$$\mathbf{V}_1 \cdot \mathbf{V}_2 = |\mathbf{V}_1|\,|\mathbf{V}_2| \cos\theta, \qquad 0 \le \theta \le \pi \tag{A-15}$$

where θ is the angle between the two vectors (Fig. A-17). This product is called the **scalar product** (or **dot product**) of two vectors. It is also referred to as the *inner product*, particularly in discussing scalar products in tensor analysis. Equation A-15 is valid in any coordinate representation and can be interpreted as the product of parallel components of the two vectors.

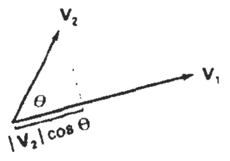


Figure A-17
The dot product of two vectors is obtained by multiplying parallel components.

In addition to the coordinate-independent form of the scalar product, we can express this product in specific coordinate representations. For a Cartesian reference frame, the scalar product is calculated as

$$\mathbf{V}_1 \cdot \mathbf{V}_2 = V_{1x}V_{2x} + V_{1y}V_{2y} + V_{1z}V_{2z} \tag{A-16}$$

The dot product of a vector with itself is simply another statement of the Pythagorean theorem. Also, the scalar product of two vectors is zero if and only if the two vectors are perpendicular (**orthogonal**). Dot products are commutative

$$\mathbf{V}_1 \cdot \mathbf{V}_2 = \mathbf{V}_2 \cdot \mathbf{V}_1 \tag{A-17}$$

because this operation produces a scalar, and dot products are distributive with respect to vector addition

$$\mathbf{V}_1 \cdot (\mathbf{V}_2 + \mathbf{V}_3) = \mathbf{V}_1 \cdot \mathbf{V}_2 + \mathbf{V}_1 \cdot \mathbf{V}_3 \tag{A-18}$$

Vector Product of Two Vectors

Multiplication of two vectors to produce another vector is defined as

$$\mathbf{V}_1 \times \mathbf{V}_2 = \mathbf{u}\,|\mathbf{V}_1|\,|\mathbf{V}_2|\sin\theta, \qquad 0 \le \theta \le \pi \tag{A-19}$$

where **u** is a unit vector (magnitude 1) that is perpendicular to both $\mathbf{V}_1$ and $\mathbf{V}_2$ (Fig. A-18). The direction for u is determined by the *right-hand rule*: We grasp an axis that is perpendicular to the plane of $\mathbf{V}_1$ and $\mathbf{V}_2$ so that the fingers of the right hand curl from $\mathbf{V}_1$ to $\mathbf{V}_2$. Our right thumb then points in the direction of **u**. This product is called the **vector product** (or **cross product**) of two vectors, and Equation A-19 is valid in any coordinate representation. The cross product of two vectors is a vector that is perpendicular to the plane of the two vectors and with magnitude equal to the area of the parallelogram formed by the two vectors.

We can also express the cross product in terms of vector components in a specific reference frame. In a Cartesian coordinate system, we calculate the components of the cross product as

$$\mathbf{V}_1 \times \mathbf{V}_2 = (V_{1y}V_{2z} - V_{1z}V_{2y},\ V_{1z}V_{2x} - V_{1x}V_{2z},\ V_{1x}V_{2y} - V_{1y}V_{2x}) \tag{A-20}$$

If we let $\mathbf{u}_x$, $\mathbf{u}_y$, and $\mathbf{u}_z$ represent unit vectors (magnitude 1) along the x, y, and z axes, we can write the cross product in terms of Cartesian components using determinant notation:

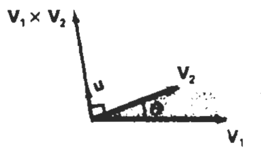


Figure A-18
The cross product of two vectors is a vector in a direction perpendicular to the two original vectors and with a magnitude equal to the area of the shaded parallelogram.

$$\mathbf{V}_1 \times \mathbf{V}_2 = \begin{vmatrix} \mathbf{u}_x & \mathbf{u}_y & \mathbf{u}_z \\ V_{1x} & V_{1y} & V_{1z} \\ V_{2x} & V_{2y} & V_{2z} \end{vmatrix} \qquad (A\text{-}21)$$

The cross product of any two parallel vectors is zero. Therefore, the cross product of a vector with itself is zero. Also, the cross product is not commutative; it is anticommutative:

$$\mathbf{V}_1 \times \mathbf{V}_2 = -(\mathbf{V}_2 \times \mathbf{V}_1) \qquad (A\text{-}22)$$

And the cross product is not associative:

$$\mathbf{V}_1 \times (\mathbf{V}_2 \times \mathbf{V}_3) \neq (\mathbf{V}_1 \times \mathbf{V}_2) \times \mathbf{V}_3 \qquad (A\text{-}23)$$

But the cross product is distributive with respect to vector addition; that is,

$$\mathbf{V}_1 \times (\mathbf{V}_2 + \mathbf{V}_3) = (\mathbf{V}_1 \times \mathbf{V}_2) + (\mathbf{V}_1 \times \mathbf{V}_3) \qquad (A\text{-}24)$$

A-3 BASIS VECTORS AND THE METRIC TENSOR

We can specify the coordinate axes in any reference frame with a set of vectors, one for each axis (Fig. A-19). Each coordinate-axis vector gives the direction of that axis at any point along the axis. These vectors form a linearly independent set of vectors. That is, the axis vectors cannot be written as linear combinations of each other. Also, any other vector in that space can be written as a linear combination of the axis vectors, and the set of axis vectors is called a **basis** (or **a set of base vectors**) for the space. In general, the space is referred to as a *vector space* and the basis contains the minimum number of vectors to represent any other vector in the space as a linear combination of the base vectors.

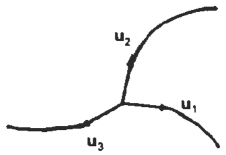


Figure A-19
Curvilinear coordinate-axis vectors.

Orthonormal Basis

Often, vectors in a basis are normalized so that each vector has a magnitude of 1. In this case, the set of unit vectors is called a **normal basis**. Also, for Cartesian reference frames and other commonly used coordinate systems, the coordinate axes are mutually perpendicular, and the set of base vectors is referred to as an **orthogonal basis**. If, in addition, the base vectors are all unit vectors, we have an **orthonormal basis** that satisfies the following conditions:

$$\begin{aligned} \mathbf{u}_k \cdot \mathbf{u}_k &= 1, \qquad \text{for all } k \\ \mathbf{u}_j \cdot \mathbf{u}_k &= 0, \qquad \text{for all } j \neq k \end{aligned} \qquad (A\text{-}25)$$

Most commonly used reference frames are orthogonal, but nonorthogonal coordinate reference frames are useful in some applications including relativity theory and visualization of certain data sets.

For a two-dimensional Cartesian system, the orthonormal basis is

$$\mathbf{u}_x = (1, 0), \qquad \mathbf{u}_y = (0, 1) \tag{A-26}$$

And the orthonormal basis for a three-dimensional Cartesian reference frame is

$$\mathbf{u}_x = (1, 0, 0), \qquad \mathbf{u}_y = (0, 1, 0), \qquad \mathbf{u}_z = (0, 0, 1) \tag{A-27}$$

Metric Tensor

Tensors are generalizations of the notion of a vector. Specifically, a **tensor** is a quantity having a number of components, depending on the tensor rank and the dimension of the space, that satisfy certain transformation properties when converted from one coordinate representation to another. For orthogonal systems, the transformation properties are straightforward. Formally, a vector is a tensor of rank one, and a scalar is a tensor of rank zero. Another way to view this classification is to note that the components of a vector are specified with one subscript, while a scalar always has a single value and, hence, no subscripts. A tensor of rank two thus has two subscripts, and in three-dimensional space, a tensor of rank two has nine components (three values for each subscript).

For any general (curvilinear) coordinate system, the elements (or coefficients) of the **metric tensor** for that space are defined as

$$g_{jk} = \mathbf{u}_j \cdot \mathbf{u}_k \tag{A-28}$$

Thus, the metric tensor is of rank two and it is symmetric: $g_{jk} = g_{kj}$. Metric tensors have several useful properties. The elements of a metric tensor can be used to determine (1) distance between two points in that space, (2) transformation equations for conversion to another space, and (3) components of various differential vector operators (such as gradient, divergence, and curl) within that space.

In an orthogonal space:

$$g_{jk} = 0, \qquad \text{for } j \neq k \tag{A-29}$$

And in a Cartesian coordinate system (assuming unit base vectors):

$$g_{jk} = \begin{cases} 1, & \text{if } j = k \\ 0, & \text{otherwise} \end{cases} \tag{A-30}$$

The unit base vectors in polar coordinates can be expressed in terms of Cartesian base vectors as

$$\mathbf{u}_r = \mathbf{u}_x \cos\theta + \mathbf{u}_y \sin\theta, \qquad \mathbf{u}_\theta = -\mathbf{u}_x r \sin\theta - \mathbf{u}_y r \cos\theta \tag{A-31}$$

Substituting these expressions into Eq. A-28, we obtain the elements of the metric tensor, which can be written in the matrix form:

$$\mathbf{g} = \begin{bmatrix} 1 & 0 \\ 0 & r^2 \end{bmatrix} \tag{A-32}$$

For a cylindrical coordinate reference frame, the base vectors are

$$\mathbf{u}_\rho = \mathbf{u}_x \cos\theta + \mathbf{u}_y \sin\theta, \qquad \mathbf{u}_\theta = -\mathbf{u}_x \rho \sin\theta + \mathbf{u}_y \rho \cos\theta, \qquad \mathbf{u}_z \tag{A-33}$$

And the matrix representation for the metric tensor in cylindrical coordinates is

$$\mathbf{g} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \rho & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{A-34}$$

We can write the base vectors in spherical coordinates as

$$\begin{aligned} \mathbf{u}_r &= \mathbf{u}_x \cos\theta \sin\phi + \mathbf{u}_y \sin\theta \sin\phi + \mathbf{u}_z \cos\phi \\ \mathbf{u}_\theta &= -\mathbf{u}_x r \sin\theta \sin\phi + \mathbf{u}_y r \cos\theta \sin\phi \\ \mathbf{u}_\phi &= \mathbf{u}_x r \cos\theta \cos\phi + \mathbf{u}_y r \sin\theta \cos\phi - \mathbf{u}_z r \sin\phi \end{aligned} \tag{A-35}$$

Then the matrix representation for the metric tensor in spherical coordinates is

$$\mathbf{g} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & r^2 \sin^2\phi & 0 \\ 0 & 0 & r^2 \end{bmatrix} \tag{A-36}$$

A-4 MATRICES

A matrix is a rectangular array of quantities (numbers, functions, or numerical expressions), called the elements of the matrix. Some examples of matrices are

$$\begin{bmatrix} 3.60 & -0.01 & 2.00 \\ -5.46 & 0.00 & 1.63 \end{bmatrix}, \quad \begin{bmatrix} e^x & x \\ e^{2x} & x^2 \end{bmatrix}, \quad [a_1\ a_2\ a_3], \quad \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{A-37}$$

We identify matrices according to the number of rows and number of columns. For these examples, the matrices in left-to-right order are 2 by 3, 2 by 2, 1 by 3, and 3 by 1. When the number of rows is the same as the number of columns, as in the second example, the matrix is called a *square matrix*.

In general, we can write an m by n matrix as

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \tag{A-38}$$

where the a_{jk} represent the elements of matrix $\mathbf{A}$. The first subscript of any element gives the row number, and the second subscript gives the column number.

A matrix with a single row or a single column represents a vector. Thus, the last two matrix examples in A-37 are, respectively, a *row vector* and a *column vector*. In general, a matrix can be viewed as a collection of row vectors or as a collection of column vectors.

When various operations are expressed in matrix form, the standard mathematical convention is to represent a vector with a column matrix. Following this convention, we write the matrix representation for a three-dimensional vector in

Cartesian coordinates as

$$\mathbf{V} = \begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix} \qquad (A\text{-}39)$$

We will use this matrix representation for both points and vectors, but we must keep in mind the distinction between them. It is often convenient to consider a point as a vector with start position at the coordinate origin within a single coordinate reference frame, but points do not have the properties of vectors that remain invariant when switching from one coordinate system to another. Also, in general, we cannot apply vector operations, such as vector addition, dot product, and cross product, to points.

Scalar Multiplication and Matrix Addition

To multiply a matrix **A** by a scalar value s, we multiply each element a_{jk} by the scalar. As an example, if

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

then

$$3\mathbf{A} = \begin{bmatrix} 3 & 6 & 9 \\ 12 & 15 & 18 \end{bmatrix}$$

Matrix addition is defined only for matrices that have the same number of rows m and the same number of columns n. For any two m by n matrices, the sum is obtained by adding corresponding elements. For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 0 & 1.5 & 0.2 \\ -6 & 1.1 & -10 \end{bmatrix} = \begin{bmatrix} 1 & 3.5 & 3.2 \\ -2 & 6.1 & -4 \end{bmatrix}$$

Matrix Multiplication

The product of two matrices is defined as a generalization of the vector dot product. We can multiply an m by n matrix **A** by a p by q matrix **B** to form the matrix product **AB**, providing that the number of columns in A is equal to the number of rows in B (i.e., $n = p$). We then obtain the product matrix by forming sums of the products of the elements in the row vectors of **A** with the corresponding elements in the column vectors of **B**. Thus, for the following product

$$\mathbf{C} = \mathbf{AB} \qquad (A\text{-}40)$$

we obtain an m by q matrix **C** whose elements are calculated as

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \qquad (A\text{-}41)$$

In the following example, a 3 by 2 matrix is postmultiplied by a 2 by 2 matrix to produce a 3 by 2 product matrix:

$$\begin{bmatrix} 0 & -1 \\ 5 & 7 \\ -2 & 8 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 0 \cdot 1 + (-1) \cdot 3 & 0 \cdot 2 + (-1) \cdot 4 \\ 5 \cdot 1 + 7 \cdot 3 & 5 \cdot 2 + 7 \cdot 4 \\ -2 \cdot 1 + 8 \cdot 3 & -2 \cdot 2 + 8 \cdot 4 \end{bmatrix} = \begin{bmatrix} -3 & -4 \\ 26 & 38 \\ 22 & 28 \end{bmatrix}$$

Vector multiplication in matrix notation produces the same result as the dot product, providing the first vector is expressed as a row vector and the second vector is expressed as a column vector:

$$[1\ 2\ 3] \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = [32]$$

This vector product results in a matrix with a single element (a 1-by-1 matrix). If we multiply the vectors in reverse order, we obtain a 3 by 3 matrix:

$$\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} [1\ 2\ 3] = \begin{bmatrix} 4 & 8 & 12 \\ 5 & 10 & 15 \\ 6 & 12 & 18 \end{bmatrix}$$

As the previous two vector products illustrate, matrix multiplication, in general, is not commutative. That is,

$$\mathbf{AB} \neq \mathbf{BA} \tag{A-42}$$

But matrix multiplication is distributive with respect to matrix addition:

$$\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC} \tag{A-43}$$

Matrix Transpose

The **transpose** $\mathbf{A}^T$ of a matrix is obtained by interchanging rows and columns. For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}, \qquad [a\ b\ c]^T = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \tag{A-44}$$

For a matrix product, the transpose is

$$(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T \tag{A-45}$$

Determinant of a Matrix

For a square matrix, we can combine the matrix elements to produce a single number called the **determinant**. Determinants are defined recursively. For a 2 by 2 matrix, the **second-order determinant** is defined to be

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21} \tag{A-46}$$

We then calculate higher-order determinants in terms of lower-order determinants. To calculate the determinants of order 3 or greater, we can select any column k of an n by n matrix and compute the determinant as

$$\det \mathbf{A} = \sum_{j=1}^{n} (-1)^{j+k} a_{jk} \det \mathbf{A}_{jk} \qquad (A\text{-}47)$$

where $\det \mathbf{A}_{jk}$ is the $(n-1)$ by $(n-1)$ determinant of the submatrix obtained from $\mathbf{A}$ by deleting the jth row and the kth column. Alternatively, we can select any row j and calculate the determinant as

$$\det \mathbf{A} = \sum_{k=1}^{n} (-1)^{j+k} a_{jk} \det \mathbf{A}_{jk} \qquad (A\text{-}48)$$

Calculating determinants for large matrices ($n > 4$, say) can be done more efficiently using numerical methods. One way to compute a determinant is to decompose the matrix into two factors: $\mathbf{A} = \mathbf{LU}$, where all elements of matrix $\mathbf{L}$ that are above the diagonal are zero, and all elements of matrix $\mathbf{U}$ that are below the diagonal are zero. We then compute the product of the diagonals for both $\mathbf{L}$ and $\mathbf{U}$, and we obtain $\det \mathbf{A}$ by multiplying these two products together. This method is based on the following property of determinants:

$$\det(\mathbf{AB}) = (\det \mathbf{A})(\det \mathbf{B}) \qquad (A\text{-}49)$$

Another method for calculating determinants is based on Gaussian elimination procedures (Section A-9).

Matrix Inverse

With square matrices, we can obtain an *inverse matrix* if and only if the determinant of the matrix is nonzero. If an inverse exists, the matrix is said to be a **nonsingular matrix**. Otherwise, the matrix is called a **singular matrix**. For most practical applications, where a matrix represents a physical operation, we can expect the inverse to exist.

The inverse of an n by n square matrix $\mathbf{A}$ is denoted as $\mathbf{A}^{-1}$ and

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I} \qquad (A\text{-}50)$$

where $\mathbf{I}$ is the identiy matrix. All diagonal elements of $\mathbf{I}$ have the value 1, and all other (off diagonal) elements are zero.

Elements for the inverse matrix $\mathbf{A}^{-1}$ can be calculated from the elements of $\mathbf{A}$ as

$$a_{jk}^{-1} = \frac{(-1)^{j+k} \det \mathbf{A}_{kj}}{\det \mathbf{A}} \qquad (A\text{-}51)$$

where a_{jk}^{-1} is the element in the jth row and kth column of $\mathbf{A}^{-1}$, and $\mathbf{A}_{kj}$ is the $(n-1)$ by $(n-1)$ submatrix obtained by deleting the kth row and jth column of matrix $\mathbf{A}$. Again, numerical methods can be used to evaluate the determinant and the elements of the inverse matrix for large values of n.

A-5 COMPLEX NUMBERS

By definition, a **complex number** z is an ordered pair of real numbers:

$$z = (x, y) \qquad (A\text{-}52)$$

where x is called the **real part** of z, and y is called the **imaginary part** of z. Real and imaginary parts of a complex number are designated as

$$x = \text{Re}(z), \qquad y = \text{Im}(z) \qquad (A\text{-}53)$$

Geometrically, a complex number is represented in the *complex plane*, as in Fig. A-20.

Complex numbers arise from solutions of equations such as

$$x^2 + 1 = 0, \qquad x^2 - 2x + 5 = 0$$

which have no real-number solutions. Thus, complex numbers and complex arithmetic are set up as extensions of real numbers that provide solutions to such equations.

Addition, subtraction, and scalar multiplication of complex numbers are carried out using the same rules as for two-dimensional vectors. Multiplication of complex numbers is defined as

$$(x_1, y_1)(x_2, y_2) = (x_1x_2 - y_1y_2, x_1y_2 + x_2y_1) \qquad (A\text{-}54)$$

This definition for complex numbers gives the same result as for real-number multiplication when the imaginary parts are zero:

$$(x_1, 0)(x_2, 0) = (x_1x_2, 0)$$

Thus, we can write a real number in complex form as

$$x = (x, 0)$$

Similarly, a *pure imaginary number* has a real part equal to 0: $(0, y)$.

The complex number (0, 1) is called the *imaginary unit*, and it is denoted by

$$i = (0, 1) \qquad (A\text{-}55)$$

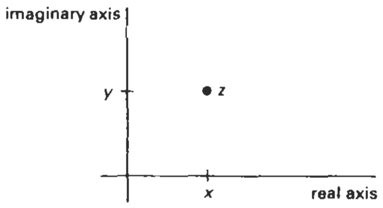


Figure A-20
Position of a point z in the complex plane.

Electrical engineers often use the symbol j for the imaginary unit, because the symbol i is used to represent electrical current. From the rule for complex multiplication, we have

$$i^2 = (0, 1)(0, 1) = (-1, 0)$$

Therefore, i^2 is the real number -1, and

$$i = \sqrt{-1} \tag{A-56}$$

Using the rule for complex multiplication, we can write any pure imaginary number in the form

$$iy = (0, 1)(0, y) = (0, y)$$

Also, by the addition rule, we can write any complex number as the sum

$$z = (x, 0) + (0, y)$$

Therefore, another representation for a complex number is

$$z = x + iy \tag{A-57}$$

which is the usual form used in practical applications.

Another concept associated with a complex number is the *complex conjugate*:

$$\bar{z} = x - iy \tag{A-58}$$

Modulus, or *absolute value*, of a complex number is defined to be

$$|z| = z\bar{z} = \sqrt{x^2 + y^2} \tag{A-59}$$

which gives the length of the "vector" representing the complex number (i.e., the distance from the origin of the complex plane to point z). Real and imaginary parts for the division of two complex numbers is obtained as

$$\begin{aligned} \frac{z_1}{z_2} &= \frac{z_1\overline{z_2}}{z_2\overline{z_2}} \\ &= \frac{(x_1, y_1)(x_2, -y_2)}{x_2^2 + y_2^2}, \\ &= \left(\frac{x_1x_2 + y_1y_2}{x_2^2 + y_2^2}, \frac{x_2y_1 - x_1y_2}{x_2^2 + y_2^2}\right) \end{aligned} \tag{A-60}$$

A particularly useful representation for complex numbers is to express the real and imaginary parts in terms of polar coordinates (Fig. A-21):

$$z = r(\cos\theta + i\sin\theta) \tag{A-61}$$

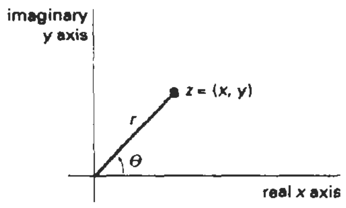


Figure A-21
Polar coordinate position of a complex number z.

We can also write the polar form of z as

$$z = re^{i\theta} \tag{A-62}$$

where e is the base of the natural logarithms ($e = 2.718281828\ldots$), and

$$e^{i\theta} = \cos\theta + i\sin\theta \tag{A-63}$$

which is *Euler's formula*. Complex multiplications and divisions are easily obtained as

$$z_1z_2 = r_1r_2e^{i(\theta_1+\theta_2)}, \qquad \frac{z_1}{z_2} = r_1r_2e^{i(\theta_1 - \theta_2)}$$

And the nth roots of a complex number are calculated as

$$\sqrt[n]{z} = \sqrt[n]{r}\left[\cos\left(\frac{\theta + 2k\pi}{n}\right) + i\sin\left(\frac{\theta + 2k\pi}{n}\right)\right], \qquad k = 0, 1, 2, \ldots, n - 1 \tag{A-64}$$

The n roots lie on a circle of radius $\sqrt[n]{r}$ with center at the origin of the complex plane and form the vertices of a regular polygon with n sides.

A-6 QUATERNIONS

Complex number concepts are extended to higher dimensions with **quaternions**, which are numbers with one real part and three imaginary parts, written as

$$q = s + ia + jb + kc \tag{A-65}$$

where the coefficients a, b, and c in the imaginary terms are real numbers, and parameter s is a real number called the *scalar part*. Parameters i, j, k are defined with the properties

$$i^2 = j^2 = k^2 = -1, \qquad ij = -ji = k \tag{A-66}$$

From these properties, it follows that

$$jk = -kj = i, \qquad ki = -ik = j \tag{A-67}$$

Scalar multiplication is defined in analogy with the corresponding operations for vectors and complex numbers. That is, each of the four components of the quaternion is multiplied by the scalar value. Similarly, quaternion addition is defined as

$$q_1 + q_2 = (s_1 + s_2) + i(a_1 + a_2) + j(b_1 + b_2) + k(c_1 + c_2) \qquad (A\text{-}68)$$

Multiplication of two quaternions is carried out using the operations in Eqs. A-66 and A-67.

An ordered-pair notation for a quaternion is also formed in analogy with complex-number notation:

$$q = (s, \mathbf{v}) \qquad (A\text{-}69)$$

where $\mathbf{v}$ is the vector (a, b, c). In this notation, quaternion addition is expressed as

$$q_1 + q_2 = (s_1 + s_2, \mathbf{v}_1 + \mathbf{v}_2) \qquad (A\text{-}70)$$

Quaternion multiplication can then be expressed in terms of vector dot and cross products as

$$q_1q_2 = (s_1s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, s_1\mathbf{v}_2 + s_2\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2) \qquad (A\text{-}71)$$

As an extension of complex operations, the magnitude squared of a quaternion is defined using the vector dot product as

$$|q|^2 = s^2 + \mathbf{v} \cdot \mathbf{v} \qquad (A\text{-}72)$$

And the inverse of a quaternion is

$$q^{-1} = \frac{1}{|q|^2}(s, \; -\mathbf{v}) \qquad (A\text{-}73)$$

so that

$$qq^{-1} = q^{-1}q = (1, 0)$$

A-7
NONPARAMETRIC REPRESENTATIONS

When we write object descriptions directly in terms of the coordinates of the reference frame in use, the respresentation is called **nonparametric**. For example, we can represent a surface with either of the following Cartesian functions:

$$f(x, y, z) = 0, \qquad \text{or} \qquad z = f(x, y) \qquad (A\text{-}74)$$

The first form in A-74 gives an *implicit* expression for the surface, and the second form gives an *explicit* representation, with x and y as the independent variables, and with z as the dependent variable.

Similarly, we can represent a three-dimensional curved line in nonparametric form as the intersection of two surface functions, or we could represent the curve with the pair of functions

$$y = f(x), \qquad z = g(x) \tag{A-75}$$

where coordinate x is selected as the independent variable. Values for the dependent variables y and z are then determined from Eqs. A-75 as we step through values for x from one line endpoint to the other endpoint.

Nonparametric representations are useful in describing objects within a given reference frame, but they have some disadvantages when used in graphics algorithms. If we want a smooth plot, we must change the independent variable whenever the first derivative (slope) of either $f(x)$ or $g(x)$ becomes greater than 1. This means that we must continually check values of the derivatives, which may become infinite at some points. Also, Eqs. A-75 provide an awkward format for representing multiple-valued functions. For instance, the implicit equation of a circle centered on the origin in the xy plane is

$$x^2 + y^2 = r^2$$

and the explicit expression for y is the multivalued function

$$y = \pm\sqrt{r^2 - x^2}$$

In general, a more convenient representation for object descriptions in graphics algorithms is in terms of parametric equations.

A-8 PARAMETRIC REPRESENTATIONS

Euclidean curves are one-dimensional objects, and positions along the path of a three-dimensional curve can be described with a single parameter u. That is, we can express each of the three Cartesian coordinates in terms of parameter u, and any point on the curve can then be represented with the following vector point function (relative to a particular Cartesian reference frame):

$$\mathbf{P}(u) = (x(u), y(u), z(u)) \tag{A-76}$$

Often, the coordinate equations can be set up so that parameter u is defined over the unit interval from 0 to 1. For example, a circle in the xy plane with center at the coordinate origin could be defined in parametric form as

$$x(u) = r\cos(2\pi u), \qquad y(u) = r\sin(2\pi u), \qquad z(u) = 0, \qquad 0 \le u \le 1 \tag{A-77}$$

Other parametric forms are also possible for describing circles and circular arcs.

Curved (or plane) Euclidean surfaces are two-dimensional objects, and positions on a surface can be described with two parameters u and v. A coordinate position on the surface is then represented with the parametric vector function

$$\mathbf{P}(u, v) = (x(u, v), y(u, v), z(u, v)) \tag{A-78}$$

where the Cartesian coordinate values for x, y, and z are expressed as functions of parameters u and v. As with curves, it is often possible to arrange the parametric descriptions so that parameters u and v are defined over the range from 0 to 1. A spherical surface with center at the coordinate origin, for example, can be described with the equations

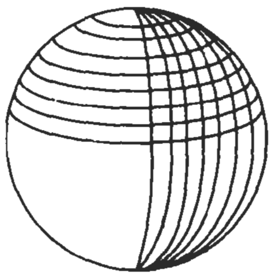


Figure A-22
Section of a spherical surface described by lines of constant u and lines of constant v in Eqs. A-79.

$$\begin{aligned} x(u,v) &= r\sin(\pi u)\cos(2\pi v) \\ y(u,v) &= r\sin(\pi u)\sin(2\pi v) \\ z(u,v) &= r\cos(\pi u) \end{aligned} \tag{A-79}$$

where r is the radius of the sphere. Parameter u describes lines of constant latitude over the surface, and parameter v describes lines of constant longitude. By keeping one of these parameters fixed while varying the other over a subinterval of the range from 0 to 1, we could plot latitude and longitude lines for any spherical section (Fig. A-22).

A-9 NUMERICAL METHODS

In computer graphics algorithms, it is often necessary to solve sets of linear equations, nonlinear equations, integral equations, and other functional forms. Also, to visualize a discrete set of data points, it may be useful to display a continuous curve or surface function that approximates the points of the data set. In this section, we briefly summarize some common algorithms for solving various numerical problems.

Solving Sets of Linear Equations

For variables x_k, $k = 1, 2, \ldots, n$, we can write a system of n linear equations as

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots \qquad\qquad & \quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \tag{A-80}$$

where the values for parameters a_{jk} and b_j are known. This set of equations can be expressed in the matrix form:

$$\mathbf{A}\mathbf{X} = \mathbf{B} \tag{A-81}$$

with $\mathbf{A}$ as an n by n square matrix whose elements are the coefficients a_{jk}, $\mathbf{X}$ as the column matrix of x_j values, and $\mathbf{B}$ as the column matrix of b_j values. The solution for the set of simultaneous linear equation can be expressed in matrix form as

$$\mathbf{X} = \mathbf{A}^{-1}\mathbf{B} \tag{A-82}$$

which depends on the inverse of the coefficient matrix $\mathbf{A}$. Thus the system of equations can be solved if and only if $\mathbf{A}$ is a nonsingular matrix; that is, its determinant is nonzero.

One method for solving the set of equations is *Cramer's Rule*:

$$x_k = \frac{\det \mathbf{A}_k}{\det \mathbf{A}} \tag{A-83}$$

where $\mathbf{A}_k$ is the matrix **A** with the kth column replaced with the elements of **B**. This method is adequate for problems with a few variables. For more than three or four variables, the method is extremely inefficient due to the large number of multiplications needed to evaluate each determinant. Evaluation of a single n by n determinant requires more that $n!$ multiplications.

We can solve the system of equations more efficiently using variations of *Gaussian elimination*. The basic ideas in Gaussian elimination can be illustrated with the following set of two simultaneous equations

$$\begin{aligned} x_1 + 2x_2 &= -4 \\ 3x_1 + 4x_2 &= 1 \end{aligned} \tag{A-84}$$

To solve this set of equations, we can multiply the first equation by -3, then we add the two equations to eliminate the x_1 term, yielding the equation

$$-2x_2 = 13$$

which has the solution $x_2 = -13/2$. This value can then be substituted into either of the original equations to obtain the solution for x_1, which is 9. Efficient algorithms have been devised to carry out the elimination and back-substitution steps.

Gaussian elimination is sometimes susceptable to high roundoff errors, and it may not be possible to obtain an accurate solution. In those cases, we may be able to obtain a solution using the *Gauss–Seidel method*. We start with an initial "guess" for the values of variables x_k, then we repeatedly calculate successive approximations until the difference between successive values is "small." At each iteration, we calculate the approximate values for the variables as

$$\begin{aligned} x_1 &= \frac{b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n}{a_{11}} \\ x_2 &= \frac{b_2 - a_{11}x_1 - a_{12}x_2 - \cdots - a_{1n}x_n}{a_{12}} \\ &\vdots \end{aligned} \tag{A-85}$$

If we can rearrange matrix **A** so that each diagonal element has a magnitude greater than the sum of the magnitudes of the other elements across that row, than the Gauss–Seidel method is guaranteed to converge to a solution.

Finding Roots of Nonlinear Equations

A root of a function $f(x)$ is a value for x that satisfies the equation $f(x) = 0$. One of the most popular methods for finding roots of nonlinear equations is the *Newton–Raphson algorithm*. This algorithm is an iterative procedure that approximates a function $f(x)$ with a straight line at each step of the iteration, as shown in Fig. A-23. We start with an initial "guess" x_0 for the value of the root, then we calcu-

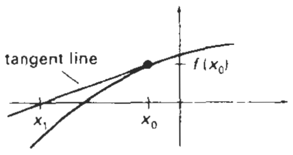


Figure A-23
Approximating a curve at an initial value x_0 with a straight line that is tangent to the curve at that point.

late the next approximation to the root as x_1 by determining where the tangent line from x_0 crosses the x axis. At x_0, the slope (first derivative) of the curve is

$$\frac{df}{dx} = \frac{f(x_0)}{x_0 - x_1} \tag{A-86}$$

Thus, the next approximation to the root is

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{A-87}$$

We repeat this procedure at each calculated approximation until the difference between successive approximations is "small enough".

If the Newton–Raphson algorithm converges to a root, it will converge faster than any other root-finding method. But it may not always converge. For example, the method fails if the derivative $f'(x)$ is 0 at some point in the iteration. Also, depending on the oscillations of the curve, successive approximation may diverge from the position of a root. The Newton–Raphson algorithm can be applied to a function of a complex variable, $f(z)$, and to sets of simultaneous nonlinear functions, real or complex.

Another method, slower but guaranteed to converge, is the *bisection method*. Here we need to first determine an x interval that contains a root, then we apply a binary search procedure to close in on the root. We first look at the midpoint of the interval to determine whether the root is in the lower or upper half of the interval. This procedure is repeated for each successive subinterval until the difference between successive midpoint positions is smaller than some preset value. A speedup can be attained by interpolating successive x positions instead of halving each subinterval (*false-position method*).

Evaluating Integrals

Integration is a summation process. For a function of a single variable x, the integral of $f(x)$ is the area "under" the curve, as illustrated in Fig. A-24.

An integral of $f(x)$ can be numerically approximated with the following summation

$$\int_b^a f(x)\,dx \approx \sum_{k=1}^{n} f_k(x)\,\Delta x_k \tag{A-88}$$

where $f_k(x)$ is an approximation to $f(x)$ over the interval Δx_k. For example, we can approximate the curve with a constant value in each subinterval and add the areas of the resulting rectangles (Fig. A-25). The smaller the subdivisions for the interval from a to b, the better the approximation (up to a point). Actually, if

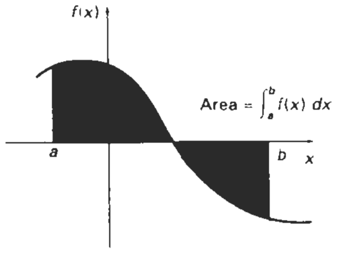


Figure A-24
The integral of $f(x)$ is equal to the amount of area between the function and the x axis over the interval from a to b.

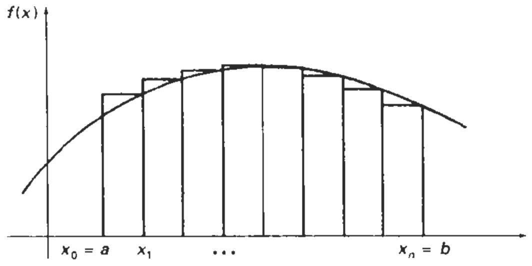


Figure A-25
Approximating an integral as the sum of the areas of small rectangles.

the intervals get too small, the values of successive rectangular areas can get lost in the roundoff error.

Polynomial approximations for the function in each subinterval generally give better results than the rectangle approach. Using a linear approximation, we obtain subareas that are trapezoids, and the approximation method is then referred to as the *trapezoid rule*. If we use a quadratic polynomial (parabola) to approximate the function in each subinterval, the method is called *Simpson's rule* and the integral approximation is

$$\int_a^b f(x)dx \approx \frac{\Delta x}{3}\left[f(a) + f(b) + 4\sum_{\text{odd}\,k=1}^{n-1} f(x_k) + 2\sum_{\text{even}\,k=2}^{n-2} f(x_k)\right] \qquad (A\text{-}89)$$

where the interval from a to b is divided into n equal-width intervals:

$$\Delta x = \frac{b-a}{n} \qquad (A\text{-}90)$$

where n is a multiple of 2, and with

$$x_0 = a, \qquad x_k = x_{k-1} + \Delta x, \qquad k = 1, 2, \ldots, n$$

For functions with high-frequency oscillations (Fig. A-26), the approximation methods previously discussed may not give accurate results. Also, multiple integrals (involving several integration variables) are difficult to solve with Simp-

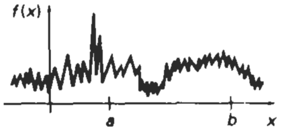


Figure A-26
A function with high-frequency oscillations.

son's rule or the other approximation methods. In these cases, we can apply *Monte Carlo* integration techniques. The term Monte Carlo is applied to any method that uses random numbers to solve deterministic problems.

We apply a Monte Carlo method to evaluate the integral of a function such as the one shown in Fig. A-26 by generating n random positions in a rectangular area that contains $f(x)$ over the interval from a to b (Fig. A-27). An approximation for the integral is then calculated as

$$\int_a^b f(x)dx \approx h(b-a)\frac{n_{\text{count}}}{n} \tag{A-91}$$

where parameter n_{count} is the count of the number of random points that are between $f(x)$ and the x axis. A random position (x, y) in the rectangular region is computed by first generating two random numbers, r_1 and r_2, and then carrying out the calculations

$$h = y_{\max} - y_{\min}, \qquad x = a + r_1(b-a), \qquad y = y_{\min} + r_2 h \tag{A-92}$$

Similar methods can be applied to multiple integrals.

Random numbers r_1 and r_2 are uniformly distributed over the interval (0, 1). We can obtain random numbers from a random-number function in a high-level language, or from a statistical package, or we can use the following algorithm, called the *linear congruential generator*:

$$\begin{aligned} i_k &= a i_{k-1} + c(\operatorname{mod} m), \qquad k = 1, 2, 3, \ldots \\ r_k &= \frac{i_k}{m} \end{aligned} \tag{A-93}$$

where parameters a, c, m, and i_0 are integers, and i_0 is a starting value called the *seed*. Parameter m is chosen to be as large as possible on a particular machine, with values for a and c chosen to make the string of random numbers as long as possible before a value is repeated. For example, on a machine with 32-bit integer representations, we can set $m = 2^{32}$, $a = 1664525$, and $c = 1013904223$.

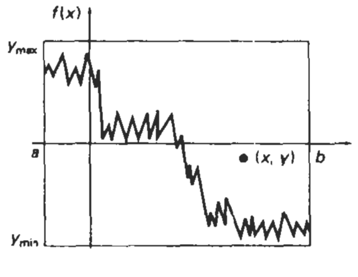


Figure A-27
A rectangular area enclosing a function $f(x)$ over the interval (a, b).

Fitting Curves to Data Sets

A standard method for fitting a function (linear or nonlinear) to a set of data points is the *least-squares algorithm*. For a two-dimensional set of data points (x_k, y_k), $k = 1, 2, \ldots$, we first select a functional form $f(x)$, which could be a straight-line function, a polynomial function, or some other curve shape. We then determine the differences (deviations) between $f(x)$ and the y_k values at each x_k and compute the sum of deviations squared:

$$E = \sum_{k=1}^{n} [y_k - f(x_k)]^2 \qquad (A\text{-}94)$$

Parameters in the function $f(x)$ are determined by minimizing the expression for E. For example, for the linear function

$$f(x) = a_0 + a_1 x$$

parameters a_0 and a_1 are assigned values that minimize E. We determine the values for a_0 and a_1 by solving the two simultaneous linear equations that result from the minimization requirements. That is, E will be minimum if the partial derivative with respect to a_0 is 0 and the partial derivative with respect to a_1 is 0:

$$\frac{\partial E}{\partial a_0} = 0, \qquad \frac{\partial E}{\partial a_1} = 0$$

Similar calculations are carried out for other functions. For the polynomial

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

we need to solve a set of n linear equations to determine values for parameters a_k. And we can also apply least-squares fitting to functions of several variables $f(x_1, x_2, \ldots, x_m)$ that can be linear or nonlinear in each of the variables.