

$$\mathbf{M}_\beta = \frac{1}{\delta} \begin{bmatrix} -2\beta_1^3 & 2(\beta_2 + \beta_1^3 + \beta_1^2 + \beta_1) & -2(\beta_2 + \beta_1^2 + \beta_1 + 1) & 2 \\ 6\beta_1^3 & -3(\beta_2 + 2\beta_1^3 + 2\beta_1^2) & 3(\beta_2 + 2\beta_1^2) & 0 \\ -6\beta_1^3 & 6(\beta_1^3 - \beta_1) & 6\beta_1 & 0 \\ 2\beta_1^3 & \beta_2 + 4(\beta_1^2 + \beta_1) & 2 & 0 \end{bmatrix} \quad (10-68)$$

where $\delta = \beta_2 + 2\beta_1^3 + 4\beta_1^2 + 4\beta_1 + 2$.

We obtain the B-spline matrix \mathbf{M}_β when $\beta_1 = 1$ and $\beta_2 = 0$. And we get the B-spline with tension matrix \mathbf{M}_{β_t} when

$$\beta_1 = 1, \quad \beta_2 = \frac{12}{t}(1 - t)$$

10-11

RATIONAL SPLINES

A rational function is simply the ratio of two polynomials. Thus, a **rational spline** is the ratio of two spline functions. For example a rational B-spline curve can be described with the position vector:

$$\mathbf{P}(u) = \frac{\sum_{k=0}^n \omega_k \mathbf{p}_k B_{k,d}(u)}{\sum_{k=0}^n \omega_k B_{k,d}(u)} \quad (10-69)$$

where the \mathbf{p}_k are a set of $n + 1$ control-point positions. Parameters ω_k are weight factors for the control points. The greater the value of a particular ω_k , the closer the curve is pulled toward the control point \mathbf{p}_k weighted by that parameter. When all weight factors are set to the value 1, we have the standard B-spline curve since the denominator in Eq. 10-69 is 1 (the sum of the blending functions).

Rational splines have two important advantages compared to nonrational splines. First, they provide an exact representation for quadric curves (conics), such as circles and ellipses. Nonrational splines, which are polynomials, can only approximate conics. This allows graphics packages to model all curve shapes with one representation—rational splines—without needing a library of curve functions to handle different design shapes. Another advantage of rational splines is that they are invariant with respect to a perspective viewing transformation (Section 12-3). This means that we can apply a perspective viewing transformation to the control points of the rational curve, and we will obtain the correct view of the curve. Nonrational splines, on the other hand, are not invariant with respect to a perspective viewing transformation. Typically, graphics design packages use nonuniform knot-vector representations for constructing rational B-splines. These splines are referred to as NURBs (*nonuniform rational B-splines*).

Homogeneous coordinate representations are used for rational splines, since the denominator can be treated as the homogeneous factor in a four-dimensional representation of the control points. Thus, a rational spline can be thought of as the projection of a four-dimensional nonrational spline into three-dimensional space.

Constructing a rational B-spline representation is carried out with the same procedures for constructing a nonrational representation. Given the set of control points, the degree of the polynomial, the weighting factors, and the knot vector, we apply the recurrence relations to obtain the blending functions.

To plot conic sections with NURBs, we use a quadratic spline function ($d = 3$) and three control points. We can do this with a B-spline function defined with the open knot vector:

$$\{0, 0, 0, 1, 1, 1\}$$

which is the same as a quadratic Bézier spline. We then set the weighting functions to the following values:

$$\begin{aligned}\omega_0 &= \omega_2 = 1 \\ \omega_1 &= \frac{r}{1-r}, \quad 0 \leq r < 1\end{aligned}\quad (10-70)$$

and the rational B-spline representation is

$$\mathbf{P}(u) = \frac{\mathbf{p}_0 B_{0,3}(u) + [r/(1-r)]\mathbf{p}_1 B_{1,3}(u) + \mathbf{p}_2 B_{2,3}(u)}{B_{0,3}(u) + [r/(1-r)]B_{1,3}(u) + B_{2,3}(u)} \quad (10-71)$$

We then obtain the various conics (Fig. 10-50) with the following values for parameter r :

- $r > 1/2, \quad \omega_1 > 1$ (hyperbola section)
- $r = 1/2, \quad \omega_1 = 1$ (parabola section)
- $r < 1/2, \quad \omega_1 < 1$ (ellipse section)
- $r = 0, \quad \omega_1 = 0$ (straight-line segment)

We can generate a one-quarter arc of a unit circle in the first quadrant of the xy plane (Fig. 10-51) by setting $\omega_1 = \cos\phi$ and by choosing the control points as

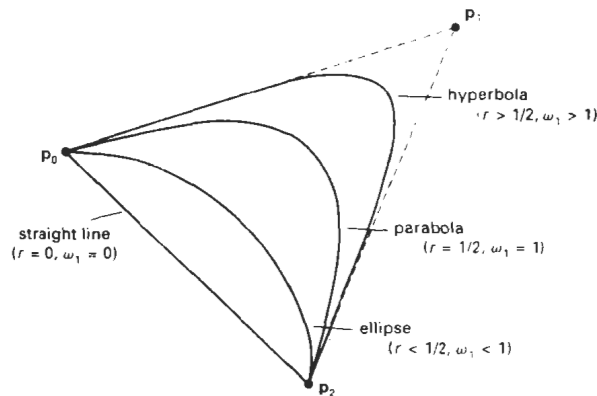


Figure 10-50
Conic sections generated with various values of the rational-spline weighting factor ω_1 .

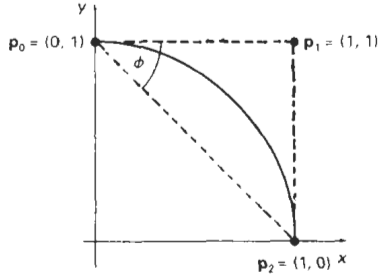


Figure 10-51
A circular arc in the first quadrant of the xy plane.

$$p_0 = (0, 1), \quad p_1 = (1, 1), \quad p_2 = (1, 0)$$

Other sections of a unit circle can be obtained with different control-point positions. A complete circle can be generated using geometric transformation in the xy plane. For example, we can reflect the one-quarter circular arc about the x and y axes to produce the circular arcs in the other three quadrants.

In some CAD systems, we construct a conic section by specifying three points on an arc. A rational homogeneous-coordinate spline representation is then determined by computing control-point positions that would generate the selected conic type. As an example, a homogeneous representation for a unit circular arc in the first quadrant of the xy plane is

$$\begin{bmatrix} x_h(u) \\ y_h(u) \\ z_h(u) \\ h \end{bmatrix} = \begin{bmatrix} 1 - u^2 \\ 2u \\ 0 \\ 1 + u^2 \end{bmatrix}$$

10-12

CONVERSION BETWEEN SPLINE REPRESENTATIONS

Sometimes it is desirable to be able to switch from one spline representation to another. For instance, a Bézier representation is the most convenient one for subdividing a spline curve, while a B-spline representation offers greater design flexibility. So we might design a curve using B-spline sections, then we can convert to an equivalent Bézier representation to display the object using a recursive subdivision procedure to locate coordinate positions along the curve.

Suppose we have a spline description of an object that can be expressed with the following matrix product:

$$P(u) = U \cdot M_{\text{spline1}} \cdot M_{\text{geom1}} \quad (10-72)$$

where M_{spline1} is the matrix characterizing the spline representation, and M_{geom1} is the column matrix of geometric constraints (for example, control-point coordinates). To transform to a second representation with spline matrix M_{spline2} , we need to determine the geometric constraint matrix M_{geom2} that produces the same vector point function for the object. That is,

or

$$P(u) = U \cdot M_{\text{spline2}} \cdot M_{\text{geom2}} \quad (10-73)$$

$$U \cdot M_{\text{spline2}} \cdot M_{\text{geom2}} = U \cdot M_{\text{spline1}} \cdot M_{\text{geom1}}$$

Solving for M_{geom2} , we have

$$\begin{aligned} M_{\text{geom2}} &= M_{\text{spline2}}^{-1} \cdot M_{\text{spline1}} \cdot M_{\text{geom1}} \\ &= M_{s1,s2} \cdot M_{\text{geom1}} \end{aligned} \quad (10-74)$$

and the required transformation matrix that converts from the first spline representation to the second is then calculated as

$$M_{s1,s2} = M_{\text{spline2}}^{-1} \cdot M_{\text{spline1}} \quad (10-75)$$

A nonuniform B-spline cannot be characterized with a general spline matrix. But we can rearrange the knot sequence to change the nonuniform B-spline to a Bézier representation. Then the Bézier matrix could be converted to any other form.

The following example calculates the transformation matrix for conversion from a periodic, cubic B-spline representation to a cubic, Bézier spline representation.

$$\begin{aligned} M_{B, \text{Bez}} &= \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}^{-1} \cdot \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \\ &= \frac{1}{6} \begin{bmatrix} 1 & 4 & 1 & 0 \\ 0 & 4 & 2 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix} \end{aligned} \quad (10-76)$$

And the the transformation matrix for converting from a cubic Bézier representation to a periodic, cubic B-spline representation is

$$\begin{aligned} M_{\text{Bez}, B} &= \begin{bmatrix} -1/6 & 1/2 & -1/2 & 1/6 \\ 1/2 & -1 & 1/2 & 0 \\ -1/2 & 0 & 1/2 & 0 \\ 1/6 & 2/3 & 1/6 & 0 \end{bmatrix}^{-1} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 6 & -7 & 2 & 0 \\ 0 & 2 & -1 & 0 \\ 0 & -1 & 2 & 0 \\ 0 & 2 & -7 & 6 \end{bmatrix} \end{aligned} \quad (10-77)$$

To display a spline curve or surface, we must determine coordinate positions on the curve or surface that project to pixel positions on the display device. This means that we must evaluate the parametric polynomial spline functions in certain increments over the range of the functions. There are several methods we can use to calculate positions over the range of a spline curve or surface.

Horner's Rule

The simplest method for evaluating a polynomial, other than a brute-force calculation of each term in succession, is *Horner's rule*, which performs the calculations by successive factoring. This requires one multiplication and one addition at each step. For a polynomial of degree n , there are n steps.

As an example, suppose we have a cubic spline representation where coordinate positions are expressed as

$$x(u) = a_x u^3 + b_x u^2 + c_x u + d_x \quad (10-78)$$

with similar expressions for the y and z coordinates. For a particular value of parameter u , we evaluate this polynomial in the following factored order:

$$x(u) = [(a_x u + b_x)u + c_x]u + d_x \quad (10-79)$$

The calculation of each x value requires three multiplications and three additions, so that the determination of each coordinate position (x, y, z) along a cubic spline curve requires nine multiplications and nine additions.

Additional factoring tricks can be applied to reduce the number of computations required by Horner's method, especially for higher-order polynomials (degree greater than 3). But repeated determination of coordinate positions over the range of a spline function can be computed much faster using forward-difference calculations or spline-subdivision methods.

Forward-Difference Calculations

A fast method for evaluating polynomial functions is to generate successive values recursively by incrementing previously calculated values as, for example,

$$x_{k+1} = x_k + \Delta x_k \quad (10-80)$$

Thus, once we know the increment and the value of x_k at any step, we get the next value by adding the increment to the value at that step. The increment Δx_k at each step is called the *forward difference*. If we divide the total range of u into subintervals of fixed size δ , then two successive x positions occur at $x_k = x(u_k)$ and $x_{k+1} = x(u_{k+1})$, where

$$u_{k+1} = u_k + \delta, \quad k = 0, 1, 2, \dots \quad (10-81)$$

and $u_0 = 0$.

To illustrate the method, suppose we have the linear spline representation $x(u) = a_x u + b_x$. Two successive x -coordinate positions are represented as

$$\begin{aligned} x_k &= a_x u_k + b_x \\ x_{k+1} &= a_x (u_k + \delta) + b_x \end{aligned} \quad (10-82)$$

Subtracting the two equations, we obtain the forward difference: $\Delta x_k = a_x \delta$. In this case, the forward difference is a constant. With higher-order polynomials, the forward difference is itself a polynomial function of parameter u with degree one less than the original polynomial.

For the cubic spline representation in Eq. 10-78, two successive x -coordinate positions have the polynomial representations

$$\begin{aligned} x_k &= a_x u_k^3 + b_x u_k^2 + c_x u_k + d_x \\ x_{k+1} &= a_x (u_k + \delta)^3 + b_x (u_k + \delta)^2 + c_x (u_k + \delta) + d_x \end{aligned} \quad (10-83)$$

The forward difference now evaluates to

$$\Delta x_k = 3a_x \delta u_k^2 + (3a_x \delta^2 + 2b_x \delta) u_k + (a_x \delta^3 + b_x \delta^2 + c_x \delta) \quad (10-84)$$

which is a quadratic function of parameter u_k . Since Δx_k is a polynomial function of u , we can use the same incremental procedure to obtain successive values of Δx_k . That is,

$$\Delta x_{k+1} = \Delta x_k + \Delta^2 x_k \quad (10-85)$$

where the second forward difference is the linear function

$$\Delta^2 x_k = 6a_x \delta^2 u_k + 6a_x \delta^3 + 2b_x \delta^2 \quad (10-86)$$

Repeating this process once more, we can write

$$\Delta^2 x_{k+1} = \Delta^2 x_k + \Delta^3 x_k \quad (10-87)$$

with the third forward difference as the constant

$$\Delta^3 x_k = 6a_x \delta^3 \quad (10-88)$$

Equations 10-80, 10-85, 10-87, and 10-88 provide an incremental forward-difference calculation of points along the cubic curve. Starting at $u_0 = 0$ with a step size δ , we obtain the initial values for the x coordinate and its first two forward differences as

$$\begin{aligned} x_0 &= d_x \\ \Delta x_0 &= a_x \delta^3 + b_x \delta^2 + c_x \delta \\ \Delta^2 x_0 &= 6a_x \delta^3 + 2b_x \delta^2 \end{aligned} \quad (10-89)$$

Once these initial values have been computed, the calculation for each successive x -coordinate position requires only three additions.

We can apply forward-difference methods to determine positions along spline curves of any degree n . Each successive coordinate position (x, y, z) is evaluated with a series of $3n$ additions. For surfaces, the incremental calculations are applied to both parameter u and parameter v .

Subdivision Methods

Recursive spline-subdivision procedures are used to repeatedly divide a given curve section in half, increasing the number of control points at each step. Subdivision methods are useful for displaying approximation spline curves since we can continue the subdivision process until the control graph approximates the curve path. Control-point coordinates then can be plotted as curve positions. Another application of subdivision is to generate more control points for shaping the curve. Thus, we could design a general curve shape with a few control points, then we could apply a subdivision procedure to obtain additional control points. With the added control points, we can make fine adjustments to small sections of the curve.

Spline subdivision is most easily applied to a Bézier curve section because the curve passes through the first and last control points, the range of parameter u is always between 0 and 1, and it is easy to determine when the control points are "near enough" to the curve path. Bézier subdivision can be applied to other spline representations with the following sequence of operations:

1. Convert the spline representation in use to a Bézier representation.
2. Apply the Bézier subdivision algorithm.
3. Convert the Bézier representation back to the original spline representation.

Figure 10-52 shows the first step in a recursive subdivision of a cubic Bézier curve section. Positions along the Bézier curve are described with the parametric point function $P(u)$ for $0 \leq u \leq 1$. At the first subdivision step, we use the halfway point $P(0.5)$ to divide the original curve into two sections. The first section is then described with the point function $P_1(s)$, and the section is described with $P_2(t)$, where

$$\begin{aligned} s &= 2u, & \text{for } 0 \leq u \leq 0.5 \\ t &= 2u - 1, & \text{for } 0.5 \leq u \leq 1 \end{aligned} \quad (10-90)$$

Each of the two new curve sections has the same number of control points as the original curve section. Also, the boundary conditions (position and parametric

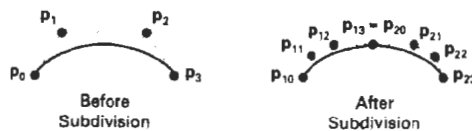


Figure 10-52
Subdividing a cubic Bézier curve section into two sections, each with four control points.

slope) at the two ends of each new curve section must match the position and slope values for the original curve $P(u)$. This gives us four conditions for each curve section that we can use to determine the control-point positions. For the first half of the curve, the four new control points are

$$\begin{aligned} P_{1,0} &= P_0 \\ P_{1,1} &= \frac{1}{2}(P_0 + P_1) \\ P_{1,2} &= \frac{1}{4}(P_0 + 2P_1 + P_2) \\ P_{1,3} &= \frac{1}{8}(P_0 + 3P_1 + 3P_2 + P_3) \end{aligned} \quad (10-91)$$

And for the second half of the curve, we obtain the four control points

$$\begin{aligned} P_{2,0} &= \frac{1}{8}(P_0 + 3P_1 + 3P_2 + P_3) \\ P_{2,1} &= \frac{1}{4}(P_1 + 2P_2 + P_3) \\ P_{2,2} &= \frac{1}{2}(P_2 + P_3) \\ P_{2,3} &= P_3 \end{aligned} \quad (10-92)$$

An efficient order for computing the new control points can be set up with only add and shift (division by 2) operations as

$$\begin{aligned} P_{1,0} &= P_0 \\ P_{1,1} &= \frac{1}{2}(P_0 + P_1) \\ T &= \frac{1}{2}(P_1 + P_2) \\ P_{1,2} &= \frac{1}{2}(P_{1,1} + T) \\ P_{2,3} &= P_3 \\ P_{2,2} &= \frac{1}{2}(P_2 + P_3) \\ P_{2,1} &= \frac{1}{2}(T + P_{2,2}) \\ P_{2,0} &= \frac{1}{2}(P_{1,2} + P_{2,1}) \\ P_{1,3} &= P_{2,0} \end{aligned} \quad (10-93)$$

These steps can be repeated any number of times, depending on whether we are subdividing the curve to gain more control points or whether we are trying to locate approximate curve positions. When we are subdividing to obtain a set of display points, we can terminate the subdivision procedure when the curve sections are small enough. One way to determine this is to check the distances between adjacent pairs of control points for each section. If these distances are "sufficiently" small, we can stop subdividing. Or we could stop subdividing when the set of control points for each section is nearly along a straight-line path.

Subdivision methods can be applied to Bézier curves of any degree. For a Bézier polynomial of degree $n - 1$, the $2n$ control points for each half of the curve at the first subdivision step are

$$\begin{aligned} p_{1,k} &= \frac{1}{2^k} \sum_{i=0}^k C(k, i) p_i, & k = 0, 1, 2, \dots, n \\ p_{2,k} &= \frac{1}{2^{n-k}} \sum_{i=k}^n C(n-k, n-i) p_i \end{aligned} \quad (10-94)$$

where $C(k, i)$ and $C(n-k, n-i)$ are the binomial coefficients.

We can apply subdivision methods directly to nonuniform B-splines by adding values to the knot vector. But, in general, these methods are not as efficient as Bézier subdivision.

10-14

SWEEP REPRESENTATIONS

Solid-modeling packages often provide a number of construction techniques. **Sweep representations** are useful for constructing three-dimensional objects that possess translational, rotational, or other symmetries. We can represent such objects by specifying a two-dimensional shape and a sweep that moves the shape through a region of space. A set of two-dimensional primitives, such as circles and rectangles, can be provided for sweep representations as menu options. Other methods for obtaining two-dimensional figures include closed spline-curve constructions and cross-sectional slices of solid objects.

Figure 10-53 illustrates a translational sweep. The periodic spline curve in Fig. 10-53(a) defines the object cross section. We then perform a translational



Figure 10-53

Constructing a solid with a translational sweep. Translating the control points of the periodic spline curve in (a) generates the solid shown in (b), whose surface can be described with point function $P(u, v)$.

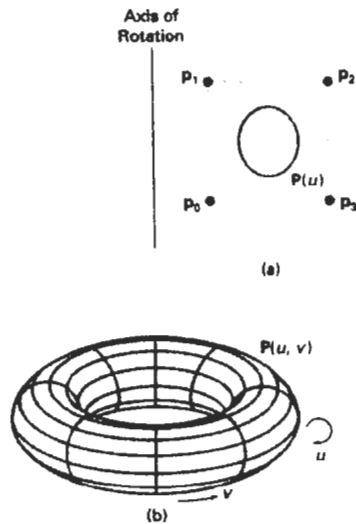


Figure 10-54

Constructing a solid with a rotational sweep. Rotating the control points of the periodic spline curve in (a) about the given rotation axis generates the solid shown in (b), whose surface can be described with point function $P(u, v)$.

sweep by moving the control points p_0 through p_3 a set distance along a straight-line path perpendicular to the plane of the cross section. At intervals along this path, we replicate the cross-sectional shape and draw a set of connecting lines in the direction of the sweep to obtain the wireframe representation shown in Fig. 10-53(b).

An example of object design using a rotational sweep is given in Fig. 10-54. This time, the periodic spline cross section is rotated about an axis of rotation specified in the plane of the cross section to produce the wireframe representation shown in Fig. 10-54(b). Any axis can be chosen for a rotational sweep. If we use a rotation axis perpendicular to the plane of the spline cross section in Fig. 10-54(a), we generate a two-dimensional shape. But if the cross section shown in this figure has depth, then we are using one three-dimensional object to generate another.

In general, we can specify sweep constructions using any path. For rotational sweeps, we can move along a circular path through any angular distance from 0 to 360°. For noncircular paths, we can specify the curve function describing the path and the distance of travel along the path. In addition, we can vary the shape or size of the cross section along the sweep path. Or we could vary the orientation of the cross section relative to the sweep path as we move the shape through a region of space.

10-15

CONSTRUCTIVE SOLID-GEOMETRY METHODS

Another technique for solid modeling is to combine the volumes occupied by overlapping three-dimensional objects using set operations. This modeling method, called **constructive solid geometry (CSG)**, creates a new volume by applying the union, intersection, or difference operation to two specified volumes.

Section 10-15

Constructive Solid-Geometry Methods

Figures 10-55 and 10-56 show examples for forming new shapes using the set operations. In Fig. 10-55(a), a block and pyramid are placed adjacent to each other. Specifying the union operation, we obtain the combined object shown in Fig. 10-55(b). Figure 10-56(a) shows a block and a cylinder with overlapping volumes. Using the intersection operation, we obtain the resulting solid in Fig. 10-56(b). With a difference operation, we can get the solid shown in Fig. 10-56(c).

A CSG application starts with an initial set of three-dimensional objects (primitives), such as blocks, pyramids, cylinders, cones, spheres, and closed spline surfaces. The primitives can be provided by the CSG package as menu selections, or the primitives themselves could be formed using sweep methods, spline constructions, or other modeling procedures. To create a new three-dimensional shape using CSG methods, we first select two primitives and drag them into position in some region of space. Then we select an operation (union, intersection, or difference) for combining the volumes of the two primitives. Now we have a new object, in addition to the primitives, that we can use to form other objects. We continue to construct new shapes, using combinations of primitives and the objects created at each step, until we have the final shape. An object designed with this procedure is represented with a binary tree. An example tree representation for a CSG object is given in Fig. 10-57.

Ray-casting methods are commonly used to implement constructive solid-geometry operations when objects are described with boundary representations. We apply ray casting by constructing composite objects in world coordinates with the xy plane corresponding to the pixel plane of a video monitor. This plane is then referred to as the "firing plane" since we fire a ray from each pixel position through the objects that are to be combined (Fig. 10-58). We then determine surface intersections along each ray path, and sort the intersection points according to the distance from the firing plane. The surface limits for the composite object are then determined by the specified set operation. An example of the ray-casting determination of surface limits for a CSG object is given in Fig. 10-59, which shows yz cross sections of two primitives and the path of a pixel ray perpendicular to the firing plane. For the union operation, the new volume is the combined interior regions occupied by either or both primitives. For the intersection operation, the new volume is the interior region common to both primitives.

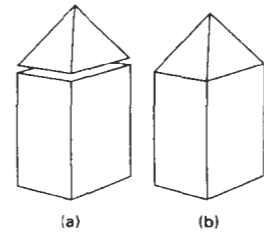


Figure 10-55
Combining two objects
(a) with a union operation
produces a single, composite
solid object (b).

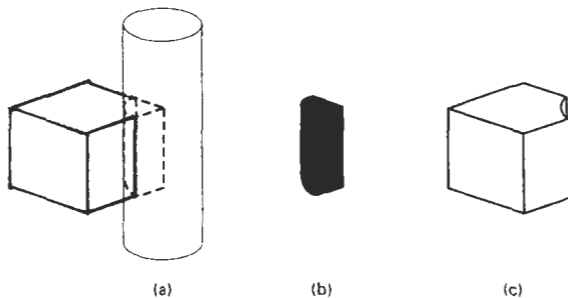


Figure 10-56
(a) Two overlapping objects. (b) A wedge-shaped CSG object
formed with the intersection operation. (c) A CSG object
formed with a difference operation by subtracting the
overlapping volume of the cylinder from the block volume.

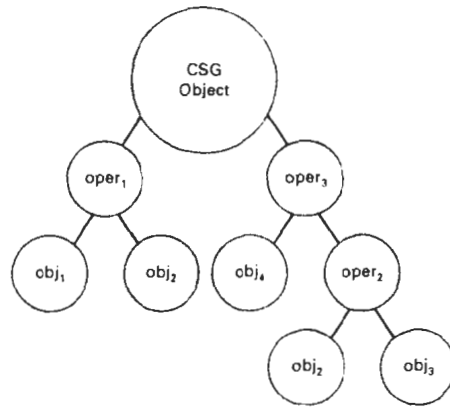


Figure 10-57
A CSG tree representation for an object.

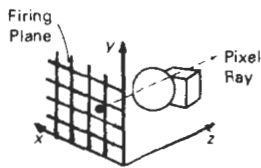
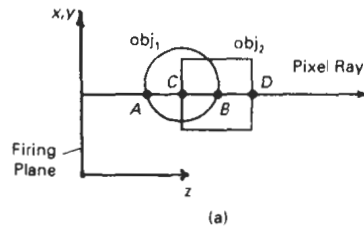


Figure 10-58
Implementing CSG operations using ray casting.



Operation	Surface Limits
Union	A, D
Intersection	C, B
Difference ($obj_2 - obj_1$)	B, D

Figure 10-59
Determining surface limits along a pixel ray.

And a difference operation subtracts the volume of one primitive from the other.

Each primitive can be defined in its own local (modeling) coordinates. Then, a composite shape can be formed by specifying the modeling-transformation matrices that would place two primitives in an overlapping position in world coordinates. The inverse of these modeling matrices can then be used to transform the pixel rays to modeling coordinates, where the surface-intersection calculations are carried out for the individual primitives. Then surface intersections for the two objects are sorted and used to determine the composite object limits according to the specified set operation. This procedure is repeated for each pair of objects that are to be combined in the CSG tree for a particular object.

Once a CSG object has been designed, ray casting is used to determine physical properties, such as volume and mass. To determine the volume of the object, we can divide the firing plane into any number of small squares, as shown in Fig. 10-60. We can then approximate the volume V_{ij} of the object for a cross-sectional slice with area A_{ij} along the path of a ray from the square at position (i, j) as

$$V_{ij} \approx A_{ij} \Delta z_{ij} \quad (10-95)$$

where Δz_{ij} is the depth of the object along the ray from position (i, j) . If the object has internal holes, Δz_{ij} is the sum of the distances between pairs of intersection points along the ray. The total volume of the CSG object is then calculated as

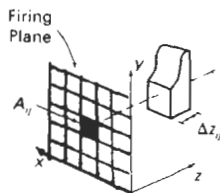


Figure 10-60
Determining object volume along a ray path for a small area A_{ij} on the firing plane.

$$V \approx \sum_{i,j} V_{ij} \quad (10-96)$$

Section 10-16
Octrees

Given the density function, $\rho(x, y, z)$, for the object, we can approximate the mass along the ray from position (i, j) as

$$m_{ij} \approx A_{ij} \int \rho(x_{ij}, y_{ij}, z) dz \quad (10-97)$$

where the one-dimensional integral can often be approximated without actually carrying out the integration, depending on the form of the density function. The total mass of the CSG object is then approximated as

$$m \approx \sum_{m,i,j} M_{ij} \quad (10-98)$$

Other physical properties, such as center of mass and moment of inertia, can be obtained with similar calculations. We can improve the approximate calculations for the values of the physical properties by taking finer subdivisions in the firing plane.

If object shapes are represented with octrees, we can implement the set operations in CSG procedures by scanning the tree structure describing the contents of spatial octants. This procedure, described in the following section, searches the octants and suboctants of a unit cube to locate the regions occupied by the two objects that are to be combined.

10-16

OCTREES

Hierarchical tree structures, called **octrees**, are used to represent solid objects in some graphics systems. Medical imaging and other applications that require displays of object cross sections commonly use octree representations. The tree structure is organized so that each node corresponds to a region of three-dimensional space. This representation for solids takes advantage of spatial coherence to reduce storage requirements for three-dimensional objects. It also provides a convenient representation for storing information about object interiors.

The octree encoding procedure for a three-dimensional space is an extension of an encoding scheme for two-dimensional space, called **quadtree** encoding. Quadtrees are generated by successively dividing a two-dimensional region (usually a square) into quadrants. Each node in the quadtree has four data elements, one for each of the quadrants in the region (Fig. 10-61). If all pixels within a quadrant have the same color (a homogeneous quadrant), the corresponding data element in the node stores that color. In addition, a flag is set in the data element to indicate that the quadrant is homogeneous. Suppose all pixels in quadrant 2 of Fig. 10-61 are found to be red. The color code for red is then placed in data element 2 of the node. Otherwise, the quadrant is said to be heterogeneous, and that quadrant is itself divided into quadrants (Fig. 10-62). The corresponding data element in the node now flags the quadrant as heterogeneous and stores the pointer to the next node in the quadtree.

An algorithm for generating a quadtree tests pixel-intensity values and sets up the quadtree nodes accordingly. If each quadrant in the original space has a

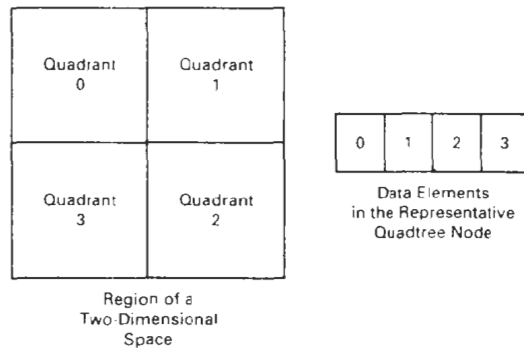


Figure 10-61
Region of a two-dimensional space divided into numbered quadrants and the associated quadtree node with four data elements.

single color specification, the quadtree has only one node. For a heterogeneous region of space, the successive subdivisions into quadrants continues until all quadrants are homogeneous. Figure 10-63 shows a quadtree representation for a region containing one area with a solid color that is different from the uniform color specified for all other areas in the region.

Quadtree encodings provide considerable savings in storage when large color areas exist in a region of space, since each single-color area can be represented with one node. For an area containing 2^n by 2^n pixels, a quadtree representation contains at most n levels. Each node in the quadtree has at most four immediate descendants.

An octree encoding scheme divides regions of three-dimensional space (usually cubes) into octants and stores eight data elements in each node of the tree (Fig. 10-64). Individual elements of a three-dimensional space are called **volume elements**, or **voxels**. When all voxels in an octant are of the same type, this

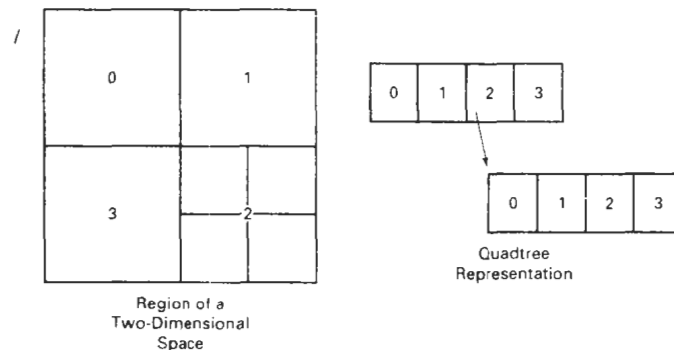


Figure 10-62
Region of a two-dimensional space with two levels of quadrant divisions and the associated quadtree representation

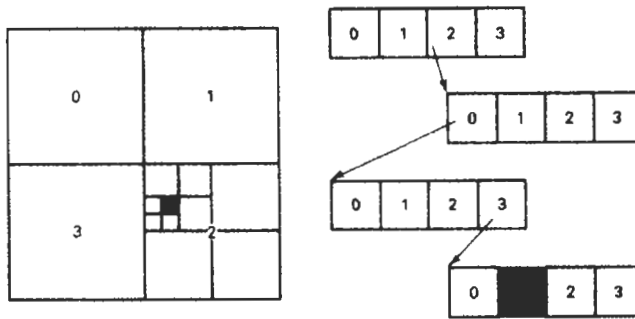


Figure 10-63

Quadtrees representation for a region containing one foreground-color pixel on a solid background.

type value is stored in the corresponding data element of the node. Empty regions of space are represented by voxel type "void." Any heterogeneous octant is subdivided into octants, and the corresponding data element in the node points to the next node in the octree. Procedures for generating octrees are similar to those for quadrees: Voxels in each octant are tested, and octant subdivisions continue until the region of space contains only homogeneous octants. Each node in the octree can now have from zero to eight immediate descendants.

Algorithms for generating octrees can be structured to accept definitions of objects in any form, such as a polygon mesh, curved surface patches, or solid-geometry constructions. Using the minimum and maximum coordinate values of the object, we can define a box (parallelepiped) around the object. This region of three-dimensional space containing the object is then tested, octant by octant, to generate the octree representation.

Once an octree representation has been established for a solid object, various manipulation routines can be applied to the solid. An algorithm for performing set operations can be applied to two octree representations for the same region of space. For a union operation, a new octree is constructed with the combined regions for each of the input objects. Similarly, intersection or differ-

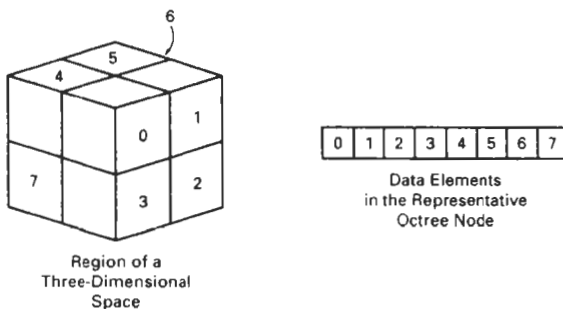


Figure 10-64

Region of a three-dimensional space divided into numbered octants and the associated octree node with eight data elements.

ence operations are performed by looking for regions of overlap in the two octrees. The new octree is then formed by either storing the octants where the two objects overlap or the region occupied by one object but not the other.

Three-dimensional octree rotations are accomplished by applying the transformations to the occupied octants. Visible-surface identification is carried out by searching the octants from front to back. The first object detected is visible, so that information can be transferred to a quadtree representation for display.

10-17

BSP TREES

This representation scheme is similar to octree encoding, except we now divide space into two partitions instead of eight at each step. With a **binary space-partitioning (BSP)** tree, we subdivide a scene into two sections at each step with a plane that can be at any position and orientation. In an octree encoding, the scene is subdivided at each step with three mutually perpendicular planes aligned with the Cartesian coordinate planes.

For adaptive subdivision of space, BSP trees can provide a more efficient partitioning since we can position and orient the cutting planes to suit the spatial distribution of the objects. This can reduce the depth of the tree representation for a scene, compared to an octree, and thus reduce the time to search the tree. In addition, BSP trees are useful for identifying visible surfaces and for space partitioning in ray-tracing algorithms.

10-18

FRACTAL-GEOMETRY METHODS

All the object representations we have considered in the previous sections used Euclidean-geometry methods; that is, object shapes were described with equations. These methods are adequate for describing manufactured objects: those that have smooth surfaces and regular shapes. But natural objects, such as mountains and clouds, have irregular or fragmented features, and Euclidean methods do not realistically model these objects. Natural objects can be realistically described with **fractal-geometry methods**, where procedures rather than equations are used to model objects. As we might expect, procedurally defined objects have characteristics quite different from objects described with equations. Fractal-geometry representations for objects are commonly applied in many fields to describe and explain the features of natural phenomena. In computer graphics, we use fractal methods to generate displays of natural objects and visualizations of various mathematical and physical systems.

A fractal object has two basic characteristics: infinite detail at every point and a certain *self-similarity* between the object parts and the overall features of the object. The self-similarity properties of an object can take different forms, depending on the choice of fractal representation. We describe a fractal object with a procedure that specifies a repeated operation for producing the detail in the object subparts. Natural objects are represented with procedures that theoretically repeat an infinite number of times. Graphics displays of natural objects are, of course, generated with a finite number of steps.

If we zoom in on a continuous Euclidean shape, no matter how complicated, we can eventually get the zoomed-in view to smooth out. But if we zoom

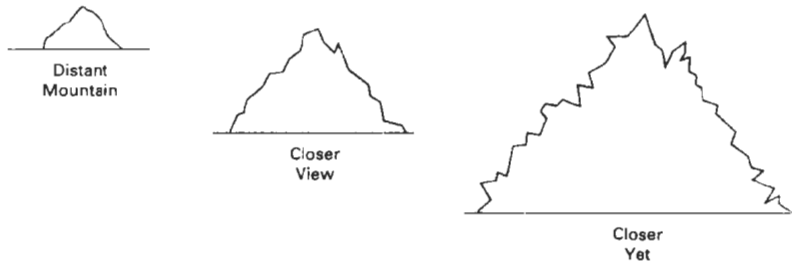


Figure 10-65
The ragged appearance of a mountain outline at different levels of magnification.

in on a fractal object, we continue to see as much detail in the magnification as we did in the original view. A mountain outlined against the sky continues to have the same jagged shape as we view it from a closer and closer position (Fig. 10-65). As we near the mountain, the smaller detail in the individual ledges and boulders becomes apparent. Moving even closer, we see the outlines of rocks, then stones, and then grains of sand. At each step, the outline reveals more twists and turns. If we took the grains of sand and put them under a microscope, we would again see the same detail repeated down through the molecular level. Similar shapes describe coastlines and the edges of plants and clouds.

Zooming in on a graphics display of a fractal object is obtained by selecting a smaller window and repeating the fractal procedures to generate the detail in the new window. A consequence of the infinite detail of a fractal object is that it has no definite size. As we consider more and more detail, the size of an object tends to infinity, but the coordinate extents of the object remain bound within a finite region of space.

We can describe the amount of variation in the object detail with a number called the *fractal dimension*. Unlike the Euclidean dimension, this number is not necessarily an integer. The fractal dimension of an object is sometimes referred to as the *fractional dimension*, which is the basis for the name “fractal”.

Fractal methods have proven useful for modeling a very wide variety of natural phenomena. In graphics applications, fractal representations are used to model terrain, clouds, water, trees and other plants, feathers, fur, and various surface textures, and just to make pretty patterns. In other disciplines, fractal patterns have been found in the distribution of stars, river islands, and moon craters; in rain fields; in stock market variations; in music; in traffic flow; in urban property utilization; and in the boundaries of convergence regions for numerical-analysis techniques.

Fractal-Generation Procedures

A fractal object is generated by repeatedly applying a specified transformation function to points within a region of space. If $P_0 = (x_0, y_0, z_0)$ is a selected initial point, each iteration of a transformation function F generates successive levels of detail with the calculations

$$P_1 = F(P_0), \quad P_2 = F(P_1), \quad P_3 = F(P_2), \quad \dots \quad (10-99)$$

In general, the transformation function can be applied to a specified point set, or we could apply the transformation function to an initial set of primitives, such as straight lines, curves, color areas, surfaces, and solid objects. Also, we can use either deterministic or random generation procedures at each iteration. The transformation function may be defined in terms of geometric transformations (scaling, translation, rotation), or it can be set up with nonlinear coordinate transformations and decision parameters.

Although fractal objects, by definition, contain infinite detail, we apply the transformation function a finite number of times. Therefore, the objects we display actually have finite dimensions. A procedural representation approaches a “true” fractal as the number of transformations is increased to produce more and more detail. The amount of detail included in the final graphical display of an object depends on the number of iterations performed and the resolution of the display system. We cannot display detail variations that are smaller than the size of a pixel. To see more of the object detail, we zoom in on selected sections and repeat the transformation function iterations.

Classification of Fractals

Self-similar fractals have parts that are scaled-down versions of the entire object. Starting with an initial shape, we construct the object subparts by apply a scaling parameter s to the overall shape. We can use the same scaling factor s for all subparts, or we can use different scaling factors for different scaled-down parts of the object. If we also apply random variations to the scaled-down subparts, the fractal is said to be *statistically self-similar*. The parts then have the same statistical properties. Statistically self-similar fractals are commonly used to model trees, shrubs, and other plants.

Self-affine fractals have parts that are formed with different scaling parameters, s_x, s_y, s_z , in different coordinate directions. And we can also include random variations to obtain *statistically self-affine* fractals. Terrain, water, and clouds are typically modeled with statistically self-affine fractal construction methods.

Invariant fractal sets are formed with nonlinear transformations. This class of fractals includes *self-squaring* fractals, such as the Mandelbrot set, which are formed with squaring functions in complex space; and *self-inverse* fractals, formed with inversion procedures.

Fractal Dimension

The detail variation in a fractal object can be described with a number D , called the **fractal dimension**, which is a measure of the roughness, or fragmentation, of the object. More jagged-looking objects have larger fractal dimensions. We can set up some iterative procedures to generate fractal objects using a given value for the fractal dimension D . With other procedures, we may be able to determine the fractal dimension from the properties of the constructed object, although, in general, the fractal dimension is difficult to calculate.

An expression for the fractal dimension of a self-similar fractal, constructed with a single scalar factor s , is obtained by analogy with the subdivision of a Euclidean object. Figure 10-66 shows the relationships between the scaling factor s and the number of subparts n for subdivision of a unit straight-line segment, a square, and a cube. With $s = 1/2$, the unit line segment (Fig. 10-66(a)) is divided into two equal-length subparts. Similarly, the square in Fig. 10-66(b) is divided into four equal-area subparts, and the cube (Fig. 10-66(c)) is divided into eight equal-volume subparts. For each of these objects, the relationship between the

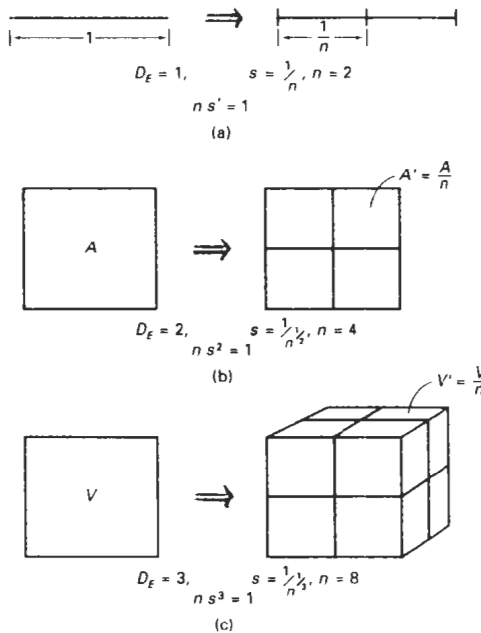


Figure 10-66
Subdividing objects with Euclidean dimensions
(a) $D_E = 1$, (b) $D_E = 2$, and (c) $D_E = 3$ using scaling
factor $s = 1/2$.

number of subparts and the scaling factor is $n \cdot s^{D_F} = 1$. In analogy with Euclidean objects, the fractal dimension D for self-similar objects can be obtained from

$$n s^D = 1 \quad (10-100)$$

Solving this expression for D , the **fractal similarity dimension**, we have

$$D = \frac{\ln n}{\ln (1/s)} \quad (10-101)$$

For a self-similar fractal constructed with different scaling factors for the different parts, the fractal similarity dimension is obtained from the implicit relationship

$$\sum_{k=1}^n s_k^D = 1 \quad (10-102)$$

where s_k is the scaling factor for subpart number k .

In Fig. 10-66, we considered subdivision of simple shapes (straight line, rectangle, box). If we have more complicated shapes, including curved lines and objects with nonplanar surfaces, determining the structure and properties of the subparts is more difficult. For general object shapes, we can use *topological cover-*

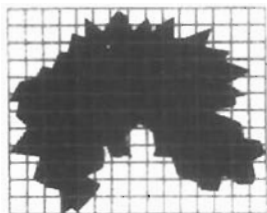


Figure 10-67
Box covering of an irregularly
shaped object.

ing methods that approximate object subparts with simple shapes. A subdivided curve, for example, can be approximated with straight-line sections, and a subdivided polygon could be approximated with small squares or rectangles. Other covering shapes, such as circles, spheres, and cylinders, can also be used to approximate the features of an object divided into a number of smaller parts. Covering methods are commonly used in mathematics to determine geometric properties, such as length, area, or volume, of an object by summing the properties of a set of smaller covering objects. We can also use covering methods to determine the fractal dimension D of some objects.

Topological covering concepts were originally used to extend the meaning of geometric properties to nonstandard shapes. An extension of covering methods using circles or spheres led to the notion of a *Hausdorff-Besicovitch dimension*, or *fractional dimension*. The Hausdorff-Besicovitch dimension can be used as the fractal dimension of some objects, but, in general, it is difficult to evaluate. More commonly, the fractal dimension of an object is estimated with *box-covering methods* using rectangles or parallelepipeds. Figure 10-67 illustrates the notion of a box covering. Here, the area inside the large irregular boundary can be approximated by the sum of the areas of the small covering rectangles.

We apply box-covering methods by first determining the coordinate extents of an object, then we subdivide the object into a number of small boxes using the given scaling factors. The number of boxes n that it takes to cover an object is called the *box dimension*, and n is related to the fractal dimension D of the object. For statistically self-similar objects with a single scaling factor s , we can cover the object with squares or cubes. We then count the number n of covering boxes and use Eq. 10-101 to estimate the fractal dimension. For self-affine objects, we cover the object with rectangular boxes, since different directions are scaled differently. In this case, the number of boxes n is used with the *affine-transformation* parameters to estimate the fractal dimension.

The fractal dimension of an object is always greater than the corresponding Euclidean dimension (or topological dimension), which is simply the least number of parameters needed to specify the object. A Euclidean curve is one-dimensional, a Euclidean surface is two-dimensional, and a Euclidean solid is three-dimensional.

For a fractal curve that lies completely within a two-dimensional plane, the fractal dimension D is greater than 1 (the Euclidean dimension of a curve). The closer D is to 1, the smoother the fractal curve. If $D = 2$, we have a *Peano curve*; that is, the "curve" completely fills a finite region of two-dimensional space. For $2 < D < 3$, the curve self-intersects and the area could be covered an infinite number of times. Fractal curves can be used to model natural-object boundaries, such as shorelines.

Spatial fractal curves (those that do not lie completely within a single plane) also have fractal dimension D greater than 1, but D can be greater than 2 without self-intersecting. A curve that fills a volume of space has dimension $D = 3$, and a self-intersecting space curve has fractal dimension $D > 3$.

Fractal surfaces typically have a dimension within the range $2 < D \leq 3$. If $D = 3$, the "surface" fills a volume of space. And if $D > 3$, there is an overlapping coverage of the volume. Terrain, clouds, and water are typically modeled with fractal surfaces.

The dimension of a fractal solid is usually in the range $3 < D \leq 4$. Again, if $D > 4$, we have a self-overlapping object. Fractal solids can be used, for example, to model cloud properties such as water-vapor density or temperature within a region of space.

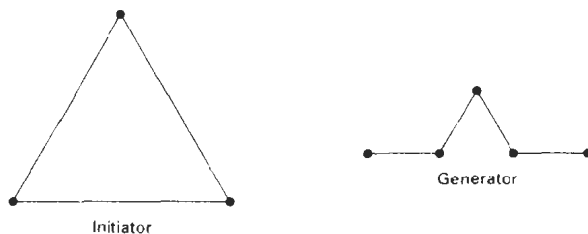


Figure 10-68
Initiator and generator for the Koch curve.

Geometric Construction of Deterministic Self-Similar Fractals

To geometrically construct a deterministic (nonrandom) self-similar fractal, we start with a given geometric shape, called the *initiator*. Subparts of the initiator are then replaced with a pattern, called the *generator*.

As an example, if we use the initiator and generator shown in Fig. 10-68, we can construct the snowflake pattern, or Koch curve, shown in Fig. 10-69. Each straight-line segment in the initiator is replaced with four equal-length line segments at each step. The scaling factor is $1/3$, so the fractal dimension is $D = \ln 4 / \ln 3 \approx 1.2619$. Also, the length of each line segment in the initiator increases by

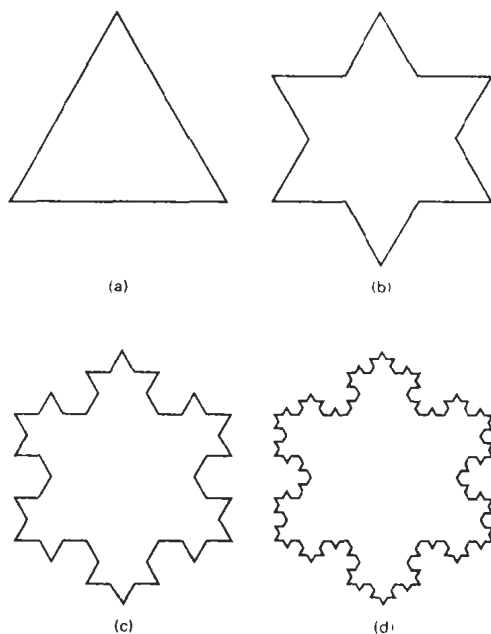


Figure 10-69
First three iterations in the generation of the Koch curve.

Segment Length = 1

Segment Length = $\frac{1}{3}$

Segment Length = $\frac{1}{9}$

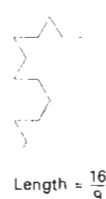


Figure 10-70

Length of each side of the Koch curve increases by a factor of $4/3$ at each step, while the line-segment lengths are reduced by a factor of $1/3$.



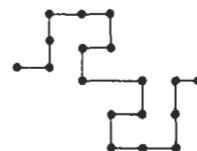
Segment Length = $1/7$

$D \approx 1.129$
(a)



Segment Length = $1/4$

$D \approx 1.500$
(b)

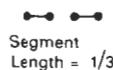


Segment Length = $1/6$

$D \approx 1.613$
(c)

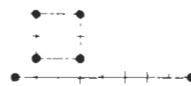
Figure 10-71

Self-similar curve constructions and associated fractal dimensions.



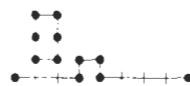
Segment Length = $1/3$

$D \approx 0.631$



Segment Length = $1/8$

$D \approx 1.333$



Segment Length = $1/8$

$D \approx 1.333$

Figure 10-72

Generators with multiple, disjoint parts.



Figure 10-73

A snowflake-filling Peano curve.

a factor of $4/3$ at each step, so that the length of the fractal curve tends to infinity as more detail is added to the curve (Fig. 10-70). Examples of other self-similar, fractal-curve constructions are shown in Fig. 10-71. These examples illustrate the more jagged appearance of objects with higher fractal dimensions.

We can also use generators with multiple disjoint components. Some examples of compound generators are shown in Fig. 10-72. Using random variations with compound generators, we can model various natural objects that have compound parts, such as island distributions along coastlines.

Figure 10-73 shows an example of a self-similar construction using multiple scaling factors. The fractal dimension of this object is determined from Eq. 10-102.

As an example of self-similar fractal construction for a surface, we scale the regular tetrahedron shown in Fig. 10-74 by a factor of $1/2$, then place the scaled

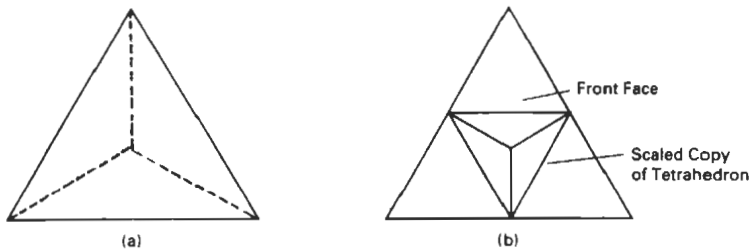


Figure 10-74

Scaling the tetrahedron in (a) by a factor of $1/2$ and positioning the scaled version on one face of the original tetrahedron produces the fractal surface (b).

object on each of the original four surfaces of the tetrahedron. Each face of the original tetrahedron is converted to 6 smaller faces and the original face area is increased by a factor of $3/2$. The fractal dimension of this surface is

$$D = \frac{\ln 6}{\ln 2} \approx 2.58496$$

which indicates a fairly fragmented surface.

Another way to create self-similar fractal objects is to punch holes in a given initiator, instead of adding more surface area. Fig. 10-75 shows some examples of fractal objects created in this way.

Geometric Construction of Statistically Self-Similar Fractals

One way we can introduce some randomness into the geometric construction of a self-similar fractal is to choose a generator randomly at each step from a set of predefined shapes. Another way to generate random self-similar objects is to compute coordinate displacements randomly. For example, in Fig. 10-76 we create a random snowflake pattern by selecting a random, midpoint-displacement distance at each step.

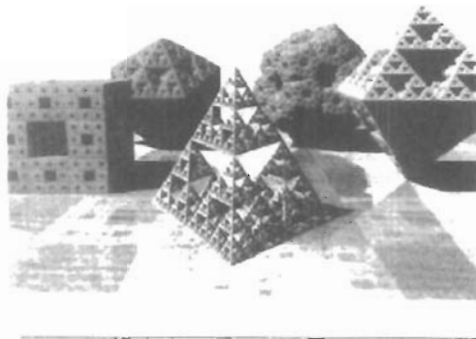


Figure 10-75

Self-similar, three-dimensional fractals formed with generators that subtract subparts from an initiator.
(Courtesy of John C. Hart, Washington State University)

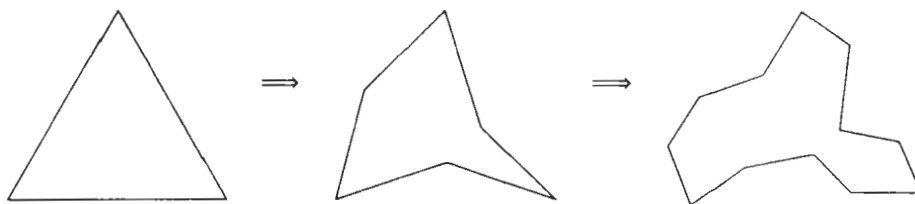


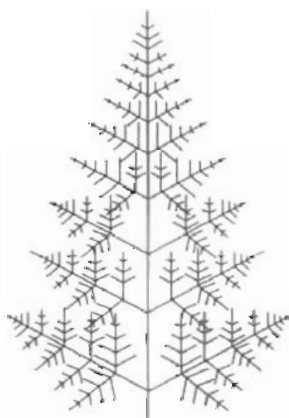
Figure 10-76

A modified "snowflake" pattern using random midpoint displacement.

Displays of trees and other plants can be constructed with similar geometric methods. Figure 10-77 shows a self-similar construction for a fern. In (a) of this figure, each branch is a scaled version of the total object, and (b) shows a fully rendered fern with a twist applied to each branch. Another example of this method is shown in Fig. 10-78. Here, random scaling parameters and branching directions are used to model the vein patterns in a leaf.

Once a set of fractal objects has been created, we can model a scene by placing several transformed instances of the fractal objects together. Figure 10-79 illustrates instancing with a simple fractal tree. In Fig. 10-80, a fractal forest is displayed.

To model the gnarled and contorted shapes of some trees, we can apply twisting functions as well as scaling to create the random, self-similar branches.



(b)

Figure 10-77

Self-similar constructions for a fern. (Courtesy of Peter Oppenheimer, Computer Graphics Lab, New York Institute of Technology.)

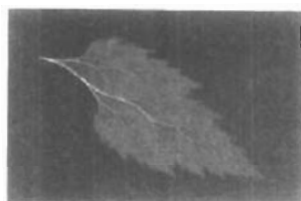


Figure 10-78

Random, self-similar construction of vein branching in a fall leaf. Boundary of the leaf is the limit of the vein growth. (Courtesy of Peter Oppenheimer, Computer Graphics Lab, New York Institute of Technology.)

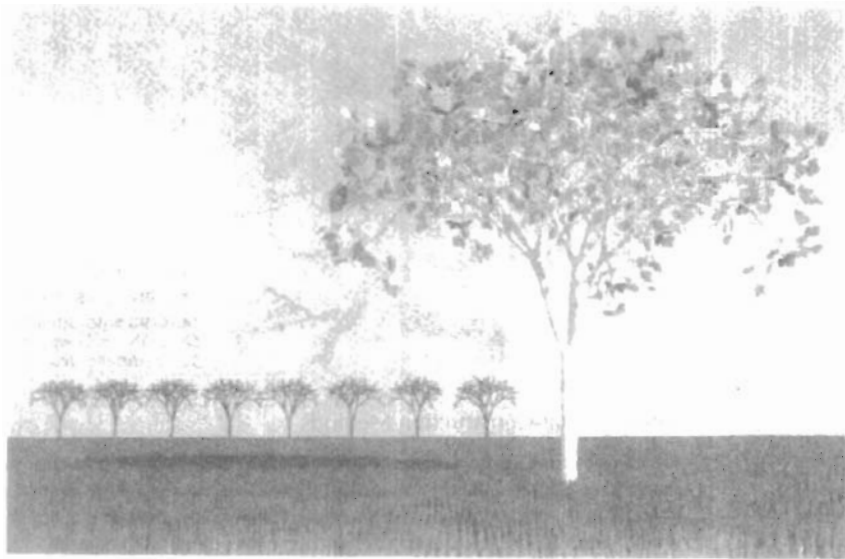


Figure 10-79

Modeling a scene using multiple object instancing. Fractal leaves are attached to a tree, and several instances of the tree are used to form a grove. The grass is modeled with multiple instances of green cones. (Courtesy of John C. Hart, Washington State University.)

This technique is illustrated in Fig. 10-81. Starting with the tapered cylinder on the left of this figure, we can apply transformations to produce (in succession from left to right) a spiral, a helix, and a random twisting pattern. A tree modeled with random twists is shown in Fig. 10-82. The tree bark in this display is modeled using bump mapping and fractal Brownian variations on the bump patterns, as discussed in the following section.



Figure 10-80

A fractal forest created with multiple instances of leaves, pine needles, grass, and tree bark. (Courtesy of John C. Hart, Washington State University.)

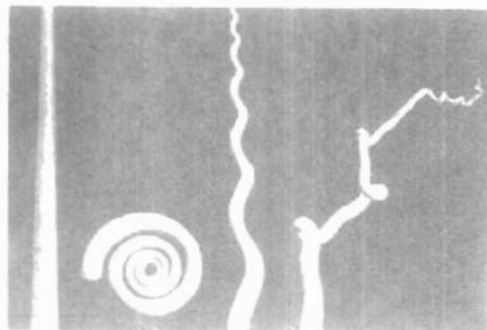


Figure 10-81

Modeling tree branches with spiral, helical, and random twists. (Courtesy of Peter Oppenheimer, Computer Graphics Lab, New York Institute of Technology.)



Figure 10-82
Tree branches modeled with random squiggles. (Courtesy of Peter Oppenheimer, Computer Graphics Lab, New York Institute of Technology.)

Affine Fractal-Construction Methods

We can obtain highly realistic representations for terrain and other natural objects using affine fractal methods that model object features as *fractional Brownian motion*. This is an extension of standard Brownian motion, a form of "random walk", that describes the erratic, zigzag movement of particles in a gas or other fluid. Figure 10-83 illustrates a random-walk path in the xy plane. Starting from a given position, we generate a straight-line segment in a random direction and with a random length. We then move to the endpoint of the first line segment and repeat the process. This procedure is repeated for any number of line segments, and we can calculate the statistical properties of the line path over any time interval t . Fractional Brownian motion is obtained by adding an additional parameter to the statistical distribution describing Brownian motion. This additional parameter sets the fractal dimension for the "motion" path.

A single fractional Brownian path can be used to model a fractal curve. With a two-dimensional array of random fractional Brownian elevations over a

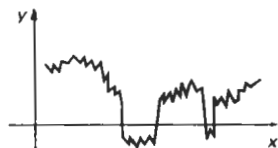


Figure 10-83
An example of Brownian motion (random walk) in the xy plane.

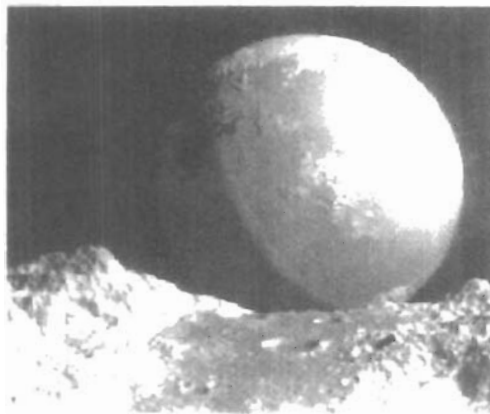


Figure 10-84
A Brownian-motion planet observed from the surface of a fractional Brownian-motion planet, with added craters, in the foreground. (Courtesy of R. V. Voss and B. B. Mandelbrot, adapted from *The Fractal Geometry of Nature* by Benoit B. Mandelbrot (New York: W. H. Freeman and Co., 1983).)

ground plane grid, we can model the surface of a mountain by connecting the elevations to form a set of polygon patches. If random elevations are generated on the surface of a sphere, we can model the mountains, valleys, and oceans of a planet. In Fig. 10-84, Brownian motion was used to create the elevation variations on the planet surface. The elevations were then color coded so that lowest elevations were painted blue (the oceans) and the highest elevations white (snow on the mountains). Fractional Brownian motion was used to create the terrain features in the foreground. Craters were created with random diameters and random positions, using affine fractal procedures that closely describe the distribution of observed craters, river islands, rain patterns, and other similar systems of objects.

By adjusting the fractal dimension in the fractional Brownian-motion calculations, we can vary the ruggedness of terrain features. Values for the fractal dimension in the neighborhood of $D \approx 2.15$ produce realistic mountain features, while higher values close to 3.0 can be used to create unusual-looking extraterrestrial landscapes. We can also scale the calculated elevations to deepen the valleys and to increase the height of mountain peaks. Some examples of terrain features that can be modeled with fractal procedures are given in Fig. 10-85. A scene modeled with fractal clouds over a fractal mountain is shown in Fig. 10-86.

Random Midpoint-Displacement Methods

Fractional Brownian-motion calculations are time-consuming, because the elevation coordinates of the terrain above a ground plane are calculated with Fourier series, which are sums of cosine and sine terms. Fast Fourier transform (FFT) methods are typically used, but it is still a slow process to generate fractal-mountain scenes. Therefore, faster **random midpoint-displacement methods**, similar to the random displacement methods used in geometric constructions, have been developed to approximate fractional Brownian-motion representations for terrain and other natural phenomena. These methods were originally used to generate animation frames for science-fiction films involving unusual terrain and planet features. Midpoint-displacement methods are now commonly used in many applications, including television advertising animations.

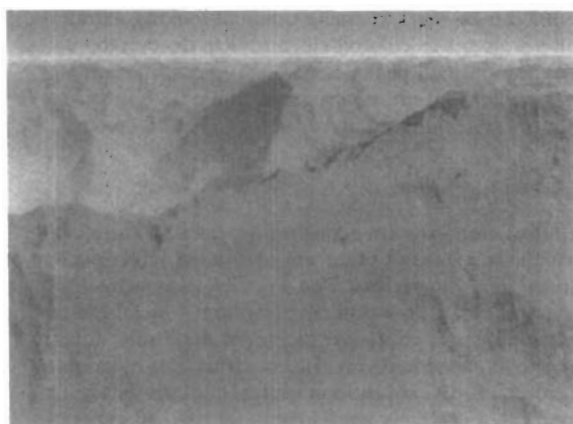
Although random midpoint-displacement methods are faster than fractional Brownian-motion calculations, they produce less realistic-looking terrain features. Figure 10-87 illustrates the midpoint-displacement method for generating a random-walk path in the xy plane. Starting with a straight-line segment, we calculate a displaced y value for the midposition of the line as the average of the endpoint y values plus a random offset:

$$y_{\text{mid}} = \frac{1}{2} [y(a) + y(b)] + r \quad (10-103)$$

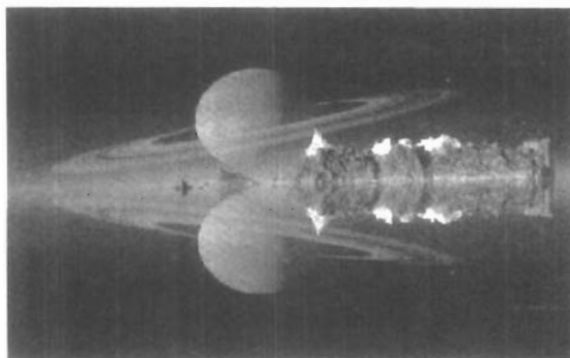
To approximate fractional Brownian motion, we choose a value for r from a Gaussian distribution with a mean of 0 and a variance proportional to $|b - a|^{2H}$, where $H = 2 - D$ and $D > 1$ is the fractal dimension. Another way to obtain a random offset is to take $r = sr_s|b - a|$, with parameter s as a selected "surface-roughness" factor, and r_s as a Gaussian random value with mean 0 and variance 1. Table lookups can be used to obtain the Gaussian values. The process is then repeated by calculating a displaced y value for the midposition of each half of the subdivided line. And we continue the subdivision until the subdivided line sections are less than some preset value. At each step, the value of the random vari-



(a)



(b)



(c)

Figure 10-85

Variations in terrain features modeled with fractional Brownian motion. (Courtesy of (a) R. V. Voss and B. B. Mandelbrot, adapted from *The Fractal Geometry of Nature* by Benoit B. Mandelbrot (New York: W. H. Freeman and Co., 1983); and (b) and (c) Ken Musgrave and Benoit B. Mandelbrot, *Mathematics and Computer Science*, Yale University.)



Figure 10-86
A scene modeled with fractal clouds and mountains.
(Courtesy of Ken Musgrave and Benoit B. Mandelbrot,
Mathematics and Computer Science, Yale University.)

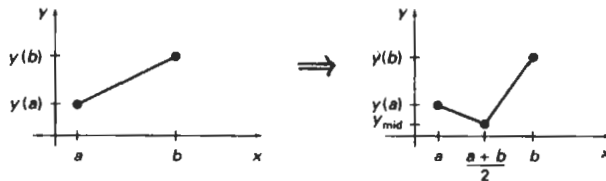


Figure 10-87
Random midpoint-displacement of a straight-line segment.

able r decreases, since it is proportional to the width $|b - a|$ of the line section to be subdivided. Figure 10-88 shows a fractal curve obtained with this method.

Terrain features are generated by applying the random midpoint-displacement procedures to a rectangular ground plane (Fig. 10-89). We begin by assigning an elevation z value to each of the four corners (a, b, c , and d in Fig. 10-89) of the ground plane. Then we divide the ground plane at the midpoint of each edge to obtain the five new grid positions: e, f, g, h , and m . Elevations at midpositions

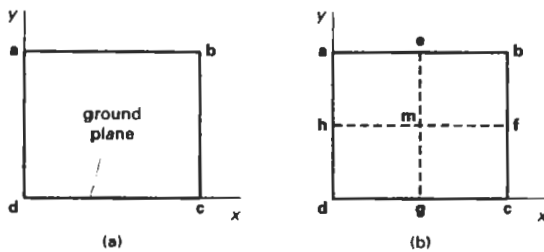


Figure 10-89
A rectangular ground plane (a) is subdivided into four equal grid sections (b) for the first step in a random midpoint-displacement procedure to calculate terrain elevations.

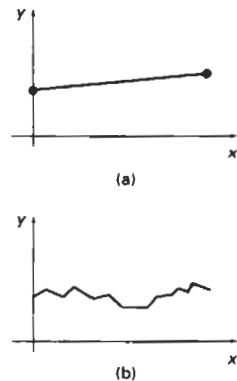


Figure 10-88
A random-walk path generated from a straight-line segment with four iterations of the random midpoint-displacement procedure.

e, f, g, and h of the ground-plane edges can be calculated as the average elevation of the nearest two vertices plus a random offset. For example, elevation z_e at midpoint e is calculated using vertices a and b, and the elevation at midpoint f is calculated using vertices b and c:

$$z_e = (z_a + z_b)/2 + r_e, \quad z_f = (z_b + z_c)/2 + r_f$$

Random values r_e and r_f can be obtained from a Gaussian distribution with mean 0 and variance proportional to the grid separation raised to the $2H$ power, with $H = 3 - D$, and where $D > 2$ is the fractal dimension for the surface. We could also calculate random offsets as the product of a surface-roughness factor times the grid separation times a table lookup value for a Gaussian value with mean 0 and variance 1. The elevation z_m of the ground-plane midpoint m can be calculated using positions e and g, or positions f and h. Alternatively, we could calculate z_m using the assigned elevations of the four ground-plane corners:

$$z_m = (z_a + z_b + z_c + z_d)/4 + r_m$$

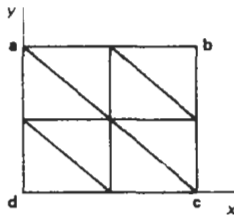


Figure 10-90
Eight surface patches formed
over a ground plane at the
first step of a random
midpoint-displacement
procedure for generating
terrain features.

This process is repeated for each of the four new grid sections at each step until the grid separation becomes smaller than a selected value.

Triangular surface patches can be formed as the elevations are generated. Figure 10-90 shows the eight surface patches formed at the first subdivision step. At each level of recursion, the triangles are successively subdivided into smaller planar patches. When the subdivision process is completed, the patches are rendered according to the position of the light sources, the values for other illumination parameters, and the selected color and surface texture for the terrain.

The random midpoint-displacement method can be applied to generate other components of a scene besides the terrain. For instance, we could use the same methods to obtain surface features for water waves or cloud patterns above a ground plane.

Controlling Terrain Topography

One way to control the placement of peaks and valleys in a fractal terrain scene modeled with a midpoint-displacement method is to constrain the calculated elevations to certain intervals over different regions of the ground plane. We can accomplish this by setting up a set of *control surfaces* over the ground plane, as illustrated in Fig. 10-91. Then we calculate a random elevation at each midpoint grid position on the ground plane that depends on the difference between the control elevation and the average elevation calculated for that position. This procedure constrains elevations to be within a preset interval about the control-surface elevations.

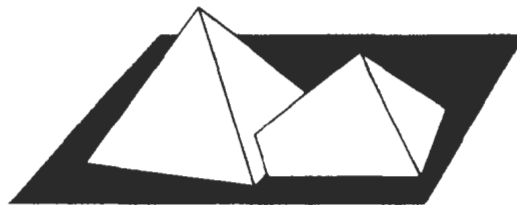


Figure 10-91
Control surfaces over a ground plane.

Control surfaces can be used to model existing terrain features in the Rocky Mountains, or some other region, by constructing the plane facets using the elevations in a contour plot for a particular region. Or we could set the elevations for the vertices of the control polygons to design our own terrain features. Also, control surfaces can have any shape. Planes are easiest to deal with, but we could use spherical surfaces or other curve shapes.

We use the random midpoint-displacement method to calculate grid elevations, but now we select random values from a Gaussian distribution where the mean μ and standard deviation σ are functions of the control elevations. One way to set the values for μ and σ is to make them both proportional to the difference between the calculated average elevation and the predefined control elevation at each grid position. For example, for grid position e in Fig. 10-89, we set the mean and standard deviation as

$$\mu_e = z_c - (z_a + z_b)/2, \quad \sigma_e = s |\mu_e|$$

where z_c is the control elevation for ground-plane position e , and $0 < s < 1$ is a preset scaling factor. Small values for s (say, $s < 0.1$) produce tighter conformity to the terrain envelope, and larger values of s allow greater fluctuations in terrain height.

To determine the values of the control elevations over a plane control surface, we first calculate the plane parameters A , B , C , and D . For any ground-plane position (x, y) , the elevation in the plane containing that control polygon is then calculated as

$$z_c = (-Ax - By - D)/C$$

Incremental methods can then be used to calculate control elevations over positions in the ground-plane grid. To efficiently carry out these calculations, we first subdivide the ground plane into a mesh of xy positions, as shown in Fig. 10-92. Then each polygon control surface is projected onto the ground plane. We can then determine which grid positions are within the projection of the control polygon using procedures similar to those in scan-line area filling. That is, for each y "scan line" in the ground-plane mesh that crosses the polygon edges, we calculate scan-line intersections and determine which grid positions are in the interior of the projection of the control polygon. Calculations for the control elevations at those grid positions can then be performed incrementally as

$$z_{i+1,j} = z_{i,j} - \Delta x(A/C), \quad z_{i,j+1} = z_{i,j} - \Delta y(B/C) \quad (10-104)$$

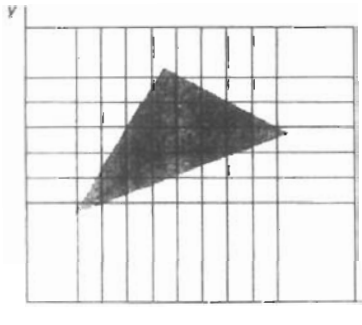


Figure 10-92
Projection of a triangular control surface onto the ground-plane grid.

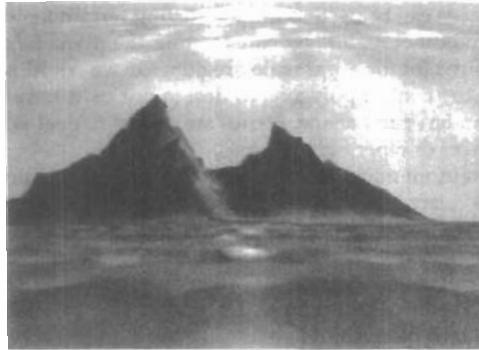


Figure 10-93
A composite scene modeled with a random midpoint-displacement method and planar control surfaces over a ground plane. Surface features for the terrain, water, and clouds were modeled and rendered separately, then combined to form the composite picture. (Courtesy of Eng-Kiat Koh, Information Technology Institute, Republic of Singapore.)

with Δx and Δy as the grid spacing in the x and y directions. This procedure is particularly fast when parallel vector methods are applied to process the control-plane grid positions.

Figure 10-93 shows a scene constructed using control planes to structure the surfaces for the terrain, water, and clouds above a ground plane. Surface-rendering algorithms were then applied to smooth out the polygon edges and to provide the appropriate surface colors.

Self-Squaring Fractals

Another method for generating fractal objects is to repeatedly apply a transformation function to points in complex space. In two dimensions, a complex number can be represented as $z = x + iy$, where x and y are real numbers, and $i^2 = -1$. In three-dimensional and four-dimensional space, points are represented with quaternions. A complex squaring function $f(z)$ is one that involves the calculation of z^2 , and we can use some self-squaring functions to generate fractal shapes.

Depending on the initial position selected for the iteration, repeated application of a self-squaring function will produce one of three possible results (Fig. 10-94):

- The transformed position can diverge to infinity.
- The transformed position can converge to a finite limit point, called an *attractor*.
- The transformed position remains on the boundary of some object.

As an example, the nonfractal squaring operation $f(z) = z^2$ in the complex plane transforms points according to their relation to the unit circle (Fig. 10-95). Any

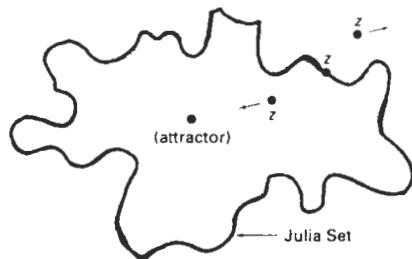


Figure 10-94
Possible outcomes of a self-squaring transformation $f(z)$ in the complex plane, depending on the position of the selected initial position.

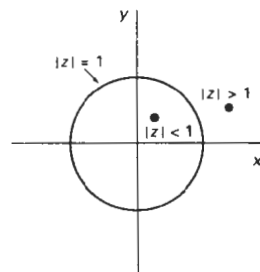


Figure 10-95
A unit circle in the complex plane. The nonfractal, complex squaring function $f(z) = z^2$ moves points that are inside the circle toward the origin, while points outside the circle are moved farther away from the circle. Any initial point on the circle remains on the circle.

point z whose magnitude $|z|$ is greater than 1 is transformed through a sequence of positions that tend to infinity. A point with $|z| < 1$ is transformed toward the coordinate origin. Points on the circle, $|z| = 1$, remain on the circle. For some functions, the boundary between those points that move toward infinity and those that tend toward a finite limit is a fractal. The boundary of the fractal object is called the *Julia set*.

In general, we can locate the fractal boundaries by testing the behavior of selected positions. If a selected position either diverges to infinity or converges to an attractor point, we can try another nearby position. We repeat this process until we eventually locate a position on the fractal boundary. Then, iteration of the squaring transformation generates the fractal shape. For simple transformations in the complex plane, a quicker method for locating positions on the fractal curve is to use the inverse of the transformation function. An initial point chosen on the inside or outside of the curve will then converge to a position on the fractal curve (Fig. 10-96).

A function that is rich in fractals is the squaring transformation

$$z' = f(z) = \lambda z(1 - z) \quad (10-105)$$

where λ is assigned any constant complex value. For this function, we can use the inverse method to locate the fractal curve. We first rearrange terms to obtain the quadratic equation:

$$z^2 - z + z'/\lambda = 0 \quad (10-106)$$

The inverse transformation is then the quadratic formula:

$$z = f^{-1}(z') = \frac{1}{2} \left(1 \pm \sqrt{1 - 4z'/\lambda} \right) \quad (10-107)$$

Using complex arithmetic operations, we solve this equation for the real and imaginary parts of z as

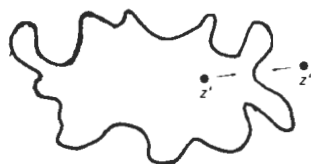


Figure 10-96
Locating the fractal boundary with the inverse, self-squaring function $z' = f^{-1}(z)$.

$$x = \operatorname{Re}(z) = \frac{1}{2} \left(1 \pm \sqrt{\frac{|\operatorname{discr}| + \operatorname{Re}(\operatorname{discr})}{2}} \right) \quad (10-108)$$

$$y = \operatorname{Im}(z) = \pm \frac{1}{2} \sqrt{\frac{|\operatorname{discr}| - \operatorname{Re}(\operatorname{discr})}{2}}$$

with the discriminant of the quadratic formula as $\operatorname{discr} = 1 - 4z'/\lambda$. A few initial values for x and y (say, 10) can be calculated and discarded before we begin to plot the fractal curve. Also, since this function yields two possible transformed (x, y) positions, we can randomly choose either the plus or the minus sign at each step of the iteration as long as $\operatorname{Im}(\operatorname{discr}) \geq 0$. Whenever $\operatorname{Im}(\operatorname{discr}) < 0$, the two possible positions are in the second and fourth quadrants. In this case, x and y must have opposite signs. The following procedure gives an implementation of this self-squaring function, and two example curves are plotted in Fig. 10-97.

```
#include <math.h>
#include <values.h>
#include "graphics.h"

typedef struct {
    float x, y;
} Complex;

void calculatePoint (Complex lambda, Complex * z)
{
    float lambdaMagSq, discrMag;
    Complex discr;
    static Complex fourOverLambda = { 0, 0 };
    static firstPoint = TRUE;

    if (firstPoint) {
        /* Compute 4 divided by lambda */
        lambdaMagSq = lambda.x * lambda.x + lambda.y * lambda.y;
        fourOverLambda.x = 4 * lambda.x / lambdaMagSq;
    }
```



Figure 10-97

Two fractal curves generated with the inverse of the function $f(z) = \lambda z(1-z)$ by procedure selfSquare: (a) $\lambda = 3$ and (b) $\lambda = 2 + i$. Each curve is plotted with 10,000 points.

```

    fourOverLambda.y = -4 * lambda.y / lambdaMagSq;
    firstPoint = FALSE;
}
discr.x = 1.0 - (z->x * fourOverLambda.x - z->y * fourOver-
Lambda.y);
discr.y = z->x * fourOverLambda.y + z->y * fourOverLambda.x;
discrMag = sqrt (discr.x * discr.x + discr.y * discr.y);

/* Update z, checking to avoid the sqrt of a negative number */
if (discrMag + discr.x < 0)
    z->x = 0;
else
    z->x = sqrt ((discrMag + discr.x) / 2.0);
if (discrMag - discr.x < 0)
    z->y = 0;
else
    z->y = 0.5 * sqrt ((discrMag - discr.x) / 2.0);

/* For half the points, use negative root, placing point in quad-
rant 3 */
if (random[] < MAXINT/2) {
    z->x = -z->x;
    z->y = -z->y;
}

/* When imaginary part of discriminant is negative, point
should lie in quadrant 2 or 4, so reverse sign of x */
if (discr.y < 0) z->x = -z->x;

/* Finish up calculation for the real part of z */
z->x = 0.5 * (1 - z->x);
}

void selfSquare (Complex lambda, Complex z, int count)
{
    int k;

    /* Skip the first few points */
    for (k=0; k<10; k++)
        calculatePoint (lambda, &z);
    for (k=0; k<count; k++) {
        calculatePoint (lambda, &z);
        /* Scale point to fit window and draw */
        pPoint (z.x*WINDOW_WIDTH, 0.5*WINDOW_HEIGHT+z.y*WINDOW_HEIGHT);
    }
}

```

A three-dimensional plot in variables x , y , and λ of the self-squaring function $f(z) = \lambda z(1-z)$, with $|\lambda| = 1$, is given in Fig. 10-98. Each cross-sectional slice of this plot is a fractal curve in the complex plane.

A very famous fractal shape is obtained from the **Mandelbrot set**, which is the set of complex values z that do not diverge under the squaring transformation:

$$\begin{aligned}
 z_0 &= z \\
 z_k &= z_{k-1}^2 + z_0, \quad k = 1, 2, 3, \dots
 \end{aligned}
 \tag{10-109}$$

That is, we first select a point z in the complex plane, then we compute the transformed position $z^2 + z$. At the next step, we square this transformed position and add the original z value. We repeat this procedure until we can determine

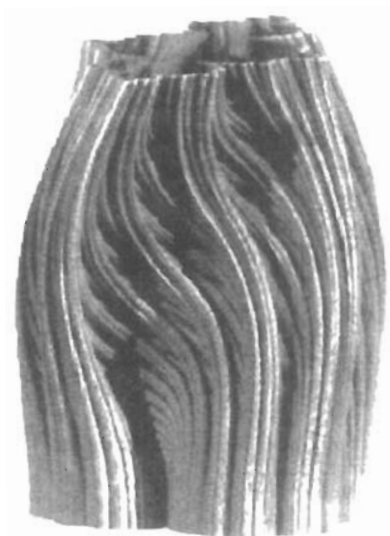


Figure 10-98
The function $f(z) = \lambda z(1-z)$
plotted in three dimensions
with normalized λ values
plotted as the vertical axis.
(Courtesy of Alan Norton, IBM Research.)

whether or not the transformation is diverging. The boundary of the convergence region in the complex plane is a fractal.

To implement transformation 10-109, we first choose a window in the complex plane. Positions in this window are then mapped to color-coded pixel positions in a selected screen viewport (Fig. 10-99). The pixel colors are chosen according to the rate of divergence of the corresponding point in the complex plane under transformation 10-109. If the magnitude of a complex number is greater than 2, then it will quickly diverge under this self-squaring operation. Therefore, we can set up a loop to repeat the squaring operations until either the magnitude of the complex number exceeds 2 or we have reached a preset number of iterations. The maximum number of iterations is usually set to some value between 100 and 1000, although lower values can be used to speed up the calculations. With lower settings for the iteration limit, however, we do tend to lose some detail along the boundary (Julia set) of the convergence region. At the end of the loop, we select a color value according to the number of iterations executed by the loop. For example, we can color the pixel black if the iteration count is at the

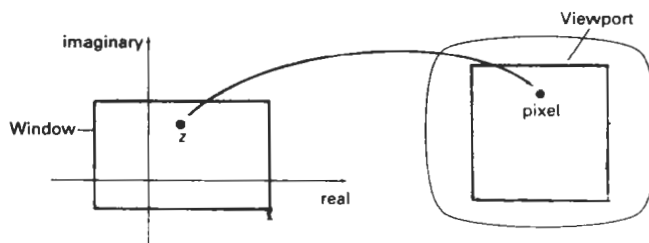


Figure 10-99
Mapping positions in the complex plane to color-coded pixel positions
on a video monitor.

maximum value, and we can color the pixel red if the iteration count is near 0. Other color values can then be chosen according to the value of the iteration count within the interval from 0 to the maximum value. By choosing different color mappings, we can generate a variety of dramatic displays for the Mandelbrot set. One choice of color coding for the set is shown in Fig. 10-100(a).

An algorithm for displaying the Mandelbrot set is given in the following procedure. The major part of the set is contained within the following region of the complex plane:

$$\begin{aligned} -2.25 \leq \operatorname{Re}(z) \leq 0.75 \\ -1.25 \leq \operatorname{Im}(z) \leq 1.25 \end{aligned}$$

We can explore the details along the boundary of the set by choosing successively smaller window regions so that we can zoom in on selected areas of the display. Figure 10-100 shows a color-coded display of the Mandelbrot set and a series of zooms that illustrate some of the features of this remarkable set.

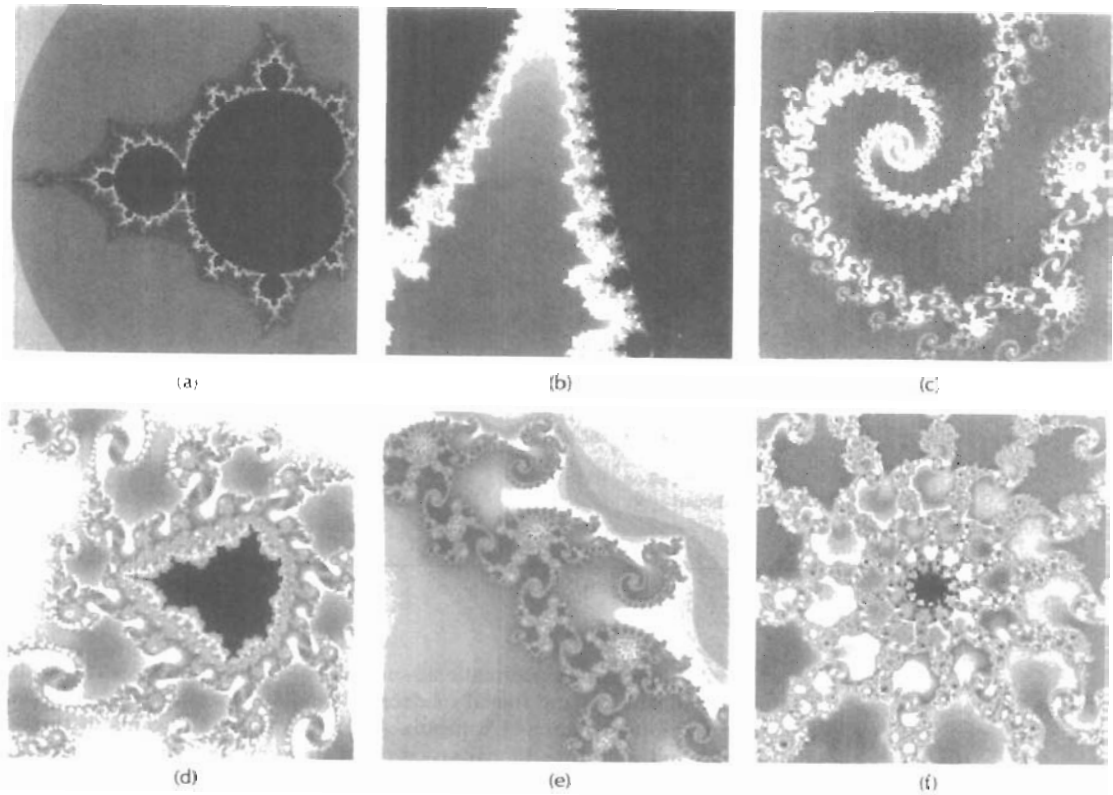


Figure 10-100

Zooming in on the Mandelbrot set. Starting with a display of the Mandelbrot set (a), we zoom in on selected regions (b) through (f). The white box outline shows the window area selected for each successive zoom. (Courtesy of Brian Evans, Vanderbilt University.)

```

#include "graphics.h"

typedef struct { float x, y; } Complex;

Complex complexSquare (Complex c)
{
    Complex cSq;

    cSq.x = c.x * c.x - c.y * c.y;
    cSq.y = 2 * c.x * c.y;
    return (cSq);
}

int iterate (Complex zInit, int maxIter)
{
    Complex z = zInit;
    int cnt = 0;

    /* Quit when z * z > 4 */
    while ((z.x * z.x + z.y * z.y <= 4.0) && (cnt < maxIter)) {
        z = complexSquare (z);
        z.x += zInit.x;
        z.y += zInit.y;
        cnt++;
    }
    return (cnt);
}

void mandelbrot (int nx, int ny, int maxIter, float realMin,
                 float realMax, float imagMin, float imagMax)
{
    float realInc = (realMax - realMin) / nx;
    float imagInc = (imagMax - imagMin) / ny;
    Complex z;
    int x, y;
    int cnt;

    for (x=0, z.x=realMin; x<nx; x++, z.x+=realInc)
        for (y=0, z.y=imagMin; y<ny; y++, z.y+=imagInc) {
            cnt = iterate (z, maxIter);
            if (cnt == maxIter)
                setColor (BLACK);
            else
                setColor (cnt);
            pPoint (x, y);
        }
}

```

Complex-function transformations, such as Eq. 10-105, can be extended to produce fractal surfaces and fractal solids. Methods for generating these objects use *quaternion* representations (Appendix A) for transforming points in three-dimensional and four-dimensional space. A quaternion has four components, one real part and three imaginary parts, and can be represented as an extension of the concept of a number in the complex plane:

$$q = s + ia + jb + kc \quad (10-110)$$

where $i^2 = j^2 = k^2 = -1$. The real part is also referred to as the *scalar part* of the quaternion, and the imaginary terms are called the quaternion *vector part* $\mathbf{v} = (a, b, c)$.

Using the rules for quaternion multiplication and addition discussed in Appendix A, we can apply self-squaring functions and other iteration methods to generate surfaces of fractal objects instead of fractal curves. A basic procedure is to start with a position inside a fractal object and generate successive points from that position until an exterior (diverging) point is identified. The previous interior point is then retained as a surface point. Neighbors of this surface point are then tested to determine whether they are inside (converging) or outside (diverging). Any inside point that connects to an outside point is a surface point. In this way, the procedure threads its way along the fractal boundary without generating points that are too far from the surface. When four-dimensional fractals are generated, three-dimensional slices are projected onto the two-dimensional surface of the video monitor.

Procedures for generating self-squaring fractals in four-dimensional space require considerable computation time for evaluating the iteration function and for testing points. Each point on a surface can be represented as a small cube, giving the inner and outer limits of the surface. Output from such programs for the three-dimensional projections of the fractal typically contain over a million vertices for the surface cubes. Display of the fractal objects is performed by applying illumination models that determine the lighting and color for each surface cube. Hidden-surface methods are then applied so that only visible surfaces of the objects are displayed. Figures 10-101 and 10-102 show examples of self-squaring, four-dimensional fractals with projections into three-dimensions.

Self-Inverse Fractals

Various geometric inversion transformations can be used to create fractal shapes. Again, we start with an initial set of points, and we repeatedly apply nonlinear inversion operations to transform the initial points into a fractal.

As an example, we consider a two-dimensional inversion transformation with respect to a circle with radius r and center at position $P_0 = (x_0, y_0)$. Any point P outside the circle will be inverted to a position P' inside the circle (Fig. 10-103) with the transformation

$$\overline{(P_0P)}\overline{(P_0P')} = r^2 \quad (10-111)$$

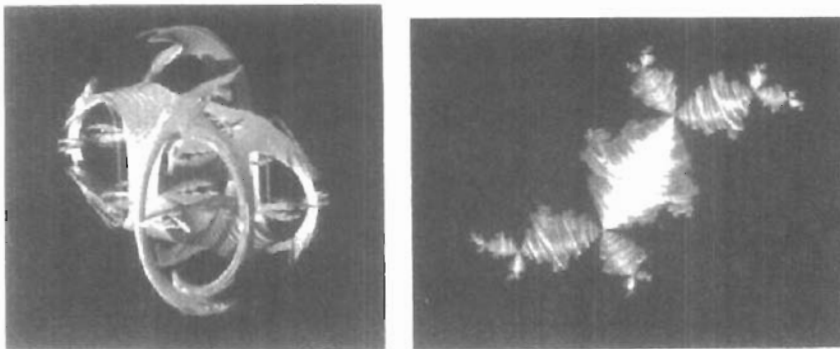


Figure 10-101

Three-dimensional projections of four-dimensional fractals generated with the self-squaring, quaternion function $f(q) = \lambda q(1-q)$: (a) $\lambda = 1.475 + 0.9061i$, and (b) $\lambda = -0.57 + i$. (Courtesy of Alan Norton, IBM Research.)

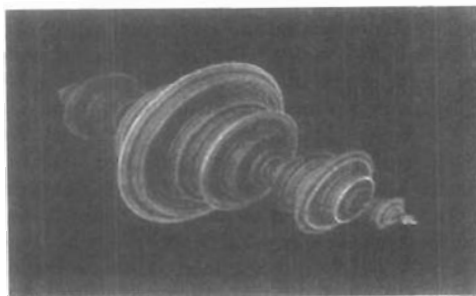


Figure 10-102

A three-dimensional surface projection of a four-dimensional object generated with the self-squaring, quaternion function $f(q) = q^2 - 1$.
(Courtesy of Alan Norton, IBM Research.)

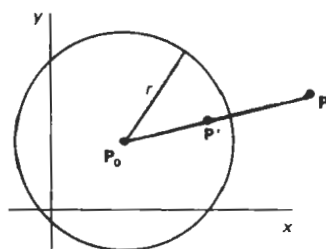


Figure 10-103

Inverting point P to a position P' inside a circle with radius r .

Reciprocally, this transformation inverts any point inside the circle to a point outside the circle. Both P and P' lie on a straight line passing through the circle center P_0 .

If the coordinates of the two points are $P = (x, y)$ and $P' = (x', y')$, we can write Eq. 10-111 as

$$[(x - x_0)^2 + (y - y_0)^2]^{1/2} [(x' - x_0)^2 + (y' - y_0)^2]^{1/2} = r^2$$

Also, since the two points lie along a line passing through the circle center, we have $(y - y_0)/(x - x_0) = (y' - y_0)/(x' - x_0)$. Therefore, the transformed coordinate values are

$$x' = x_0 + \frac{r^2(x - x_0)}{(x - x_0)^2 + (y - y_0)^2}, \quad y' = y_0 + \frac{r^2(y - y_0)}{(x - x_0)^2 + (y - y_0)^2} \quad (10-112)$$

Figure 10-104 illustrates the inversion of points along another circle boundary. As long as the circle to be inverted does not pass through P_0 , it will transform to another circle. But if the circle circumference passes through P_0 , the circle

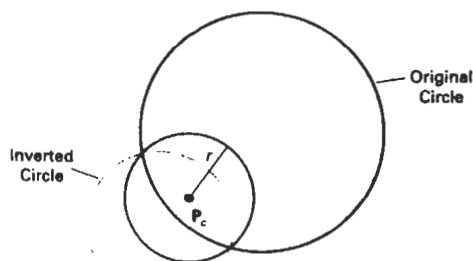


Figure 10-104
Inversion of a circle with respect to another circle.

transforms to a straight line. Conversely, points along a straight line not passing through P_0 invert to a circle. Thus, straight lines are invariant under the inversion transformation. Also invariant under this transformation are circles that are orthogonal to the reference circle. That is, the tangents of the two circles are perpendicular at the intersection points.

We can create various fractal shapes with this inversion transformation by starting with a set of circles and repeatedly applying the transformation using different reference circles. Similarly, we can apply circle inversion to a set of straight lines. Similar inversion methods can be developed for other objects. And, we can generalize the procedure to spheres, planes, or other shapes in three-dimensional space.

10-19

SHAPE GRAMMARS AND OTHER PROCEDURAL METHODS

A number of other procedural methods have been developed for generating object details. **Shape grammars** are sets of production rules that can be applied to an initial object to add layers of detail that are harmonious with the original shape. Transformations can be applied to alter the geometry (shape) of the object, or the transformation rules can be applied to add surface-color or surface-texture detail.

Given a set of production rules, a shape designer can then experiment by applying different rules at each step of the transformation from a given initial object to the final structure. Figure 10-105 shows four geometric substitution rules for altering triangle shapes. The geometry transformations for these rules can be



Figure 10-105
Four geometric substitution rules for subdividing and altering the shape of an equilateral triangle.

written algorithmically by the system based on an input picture drawn with a production-rule editor. That is, each rule can be described graphically by showing the initial and final shapes. Implementations can then be set up in Mathematica or some other programming language with graphics capability.

An application of the geometric substitutions in Fig. 10-105 is given in Fig. 10-106, where Fig. 10-106(d) is obtained by applying the four rules in succession, starting with the initial triangle in Fig. 10-106(a). Figure 10-107 shows another shape created with triangle substitution rules.

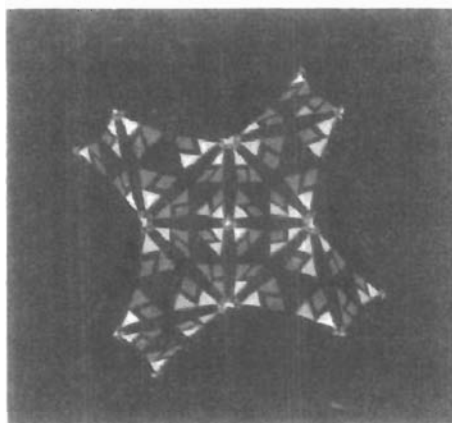
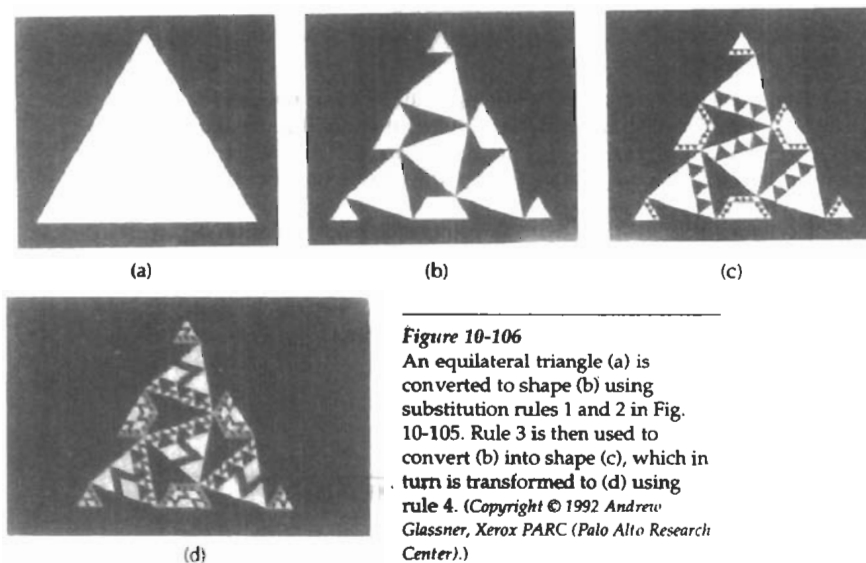


Figure 10-107
A design created with geometric substitution rules for altering triangle shapes. (Copyright © 1992 Andrew Glassner, Xerox PARC (Palo Alto Research Center).)

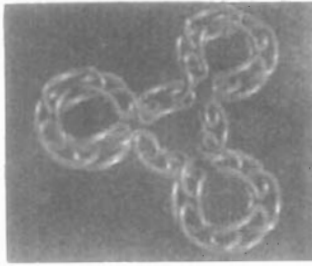


Figure 10-108

A design created with geometric substitution rules for altering prism shapes. The initial shape for this design was a representation of Rubik's Snake. (Copyright © 1992 Andrew Glassner, Xerox PARC (Palo Alto Research Center).)

Three-dimensional shape and surface features are transformed with similar operations. Figure 10-108 shows the results of geometric substitutions applied to polyhedra. The initial shape for the objects shown in Figure 10-109 is an icosahedron, a polyhedron with 20 faces. Geometric substitutions were applied to the plane faces of the icosahedron, and the resulting polygon vertices were projected to the surface of an enclosing sphere.

Another example of using production rules to describe the shape of objects is *L-grammars*, or *graftals*. These rules provide a method for describing plants. For instance, the topology of a tree can be described as a trunk, with some attached branches and leaves. A tree can then be modeled with rules to provide a particular connection of the branches and the leaves on the individual branches. The geometrical description is then given by placing the object structures at particular coordinate positions.

Figure 10-110 shows a scene containing various plants and trees, constructed with a commercial plant-generator package. Procedures in the software for constructing the plants are based on botanical laws.

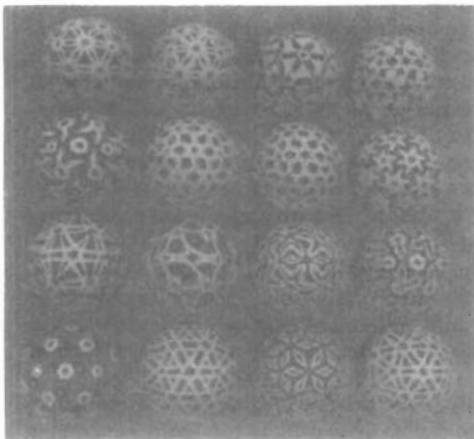


Figure 10-109

Designs created on the surface of a sphere using triangle substitution rules applied to the plane faces of an icosahedron, followed by projections to the sphere surface. (Copyright © 1992 Andrew Glassner, Xerox PARC (Palo Alto Research Center).)



Figure 10-110

Realistic scenery generated with the TDI-AMAP software package, which can generate over 100 varieties of plants and trees using procedures based on botanical laws. (Courtesy of Thomson Digital Image.)

10-20

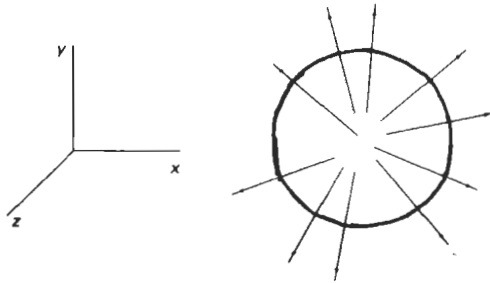
PARTICLE SYSTEMS

A method for modeling natural objects, or other irregularly shaped objects, that exhibit "fluid-like" properties is **particle systems**. This method is particularly good for describing objects that change over time by flowing, billowing, spattering, or expanding. Objects with these characteristics include clouds, smoke, fire, fireworks, waterfalls, water spray, and clumps of grass. For example, particle systems were used to model the planet explosion and expanding wall of fire due to the "genesis bomb" in the motion picture *Star Trek II—The Wrath of Khan*.

Random processes are used to generate objects within some defined region of space and to vary their parameters over time. At some random time, each object is deleted. During the lifetime of a particle, its path and surface characteristics may be color-coded and displayed.

Particle shapes can be small spheres, ellipsoids, boxes, or other shapes. The size and shape of particles may vary randomly over time. Also, other properties such as particle transparency, color, and movement all can vary randomly. In some applications, particle motion may be controlled by specified forces, such as a gravity field.

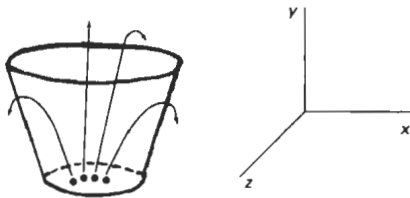
As each particle moves, its path is plotted and displayed in a particular color. For example, a fireworks pattern can be displayed by randomly generating particles within a spherical region of space and allowing them to move radially

**Figure 10-111**

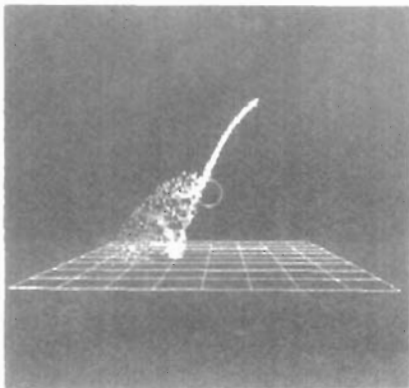
Modeling fireworks as a particle system with particles traveling radially outward from the center of the sphere.

outward, as in Fig. 10-111. The particle paths can be color-coded from red to yellow, for instance, to simulate the temperature of the exploding particles. Similarly, realistic displays of grass clumps have been modeled with “trajectory” particles (Fig. 10-112) that are shot up from the ground and fall back to earth under gravity. In this case, the particle paths can originate within a tapered cylinder, and might be color-coded from green to yellow.

Figure 10-113 illustrates a particle-system simulation of a waterfall. The water particles fall from a fixed elevation, are deflected by an obstacle, and then splash up from the ground. Different colors are used to distinguish the particle

**Figure 10-112**

Modeling a clump of grass by firing particles upward within a tapered cylinder. The particle paths are parabolas due to the downward force of gravity.

**Figure 10-113**

Simulation of the behavior of a waterfall hitting a stone (circle). The water particles are deflected by the stone and then splash up from the ground. (Courtesy of M. Brooks and T. L. J. Howard, Department of Computer Science, University of Manchester.)

paths at each stage. An example of an animation simulating the disintegration of an object is shown in Fig. 10-114. The object on the left disintegrates into the particle distribution on the right. A composite scene formed with a variety of representations is given in Fig. 10-115. The scene is modeled using particle-system grass, fractal mountains, and texture mapping and other surface-rendering procedures.



Figure 10-114
An object disintegrating into a cloud of particles. (Courtesy of Autodesk, Inc.)



Figure 10-115
A scene, entitled *Road to Point Reyes*, showing particle-system grass, fractal mountains, and texture-mapped surfaces. (Courtesy of Pixar. Copyright © 1983 Pixar.)

10-21

PHYSICALLY BASED MODELING

A nonrigid object, such as a rope, a piece of cloth, or a soft rubber ball, can be represented with **physically based modeling** methods that describe the behavior of the object in terms of the interaction of external and internal forces. An accurate description of the shape of a terry cloth towel draped over the back of a chair is obtained by considering the effect of the chair on the fabric loops in the cloth and the interaction between the cloth threads.

A common method for modeling a nonrigid object is to approximate the object with a network of point nodes with flexible connections between the nodes. One simple type of connection is a spring. Figure 10-116 shows a section of a two-dimensional spring network that could be used to approximate the behavior of a sheet of rubber. Similar spring networks can be set up in three dimensions to model a rubber ball or a block of jello. For a homogeneous object, we can use identical springs throughout the network. If we want the object to have different properties in different directions, we can use different spring properties in different directions. When external forces are applied to a spring network, the amount of stretching or compression of the individual springs depends on the value set for the *spring constant* k , also called the *force constant* for the spring.

Horizontal displacement x of a node position under the influence of a force F_x is illustrated in Fig. 10-117. If the spring is not overstretched, we can closely approximate the amount of displacement x from the equilibrium position using Hooke's law:

$$F_s = -F_x = -kx \quad (10-113)$$

where F_s is the equal and opposite restoring force of the spring on the stretched node. This relationship holds also for horizontal compression of a spring by an amount x , and we have similar relationships for displacements and force components in the y and z directions.

If objects are completely flexible, they return to their original configuration when the external forces are removed. But if we want to model putty, or some other deformable object, we need to modify the spring characteristics so that the springs do not return to their original shape when the external forces are removed. Another set of applied forces then can deform the object in some other way.

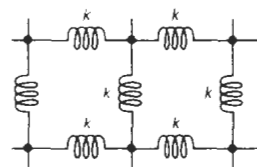


Figure 10-116
A two-dimensional spring network, constructed with identical spring constants k .

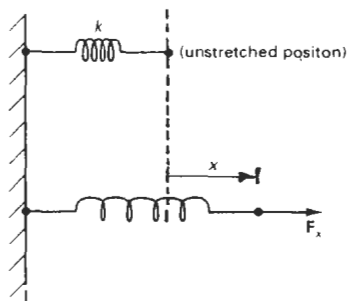


Figure 10-117
An external force F_x pulling on one end of a spring, with the other end rigidly fixed.

Instead of using springs, we can also model the connections between nodes with elastic materials, then we minimize strain-energy functions to determine object shape under the influence of external forces. This method provides a better model for cloth, and various energy functions have been devised to describe the behavior of different cloth materials.

To model a nonrigid object, we first set up the external forces acting on the object. Then we consider the propagation of the forces throughout the network representing the object. This leads to a set of simultaneous equations that we must solve to determine the displacement of the nodes throughout the network.

Figure 10-118 shows a banana peel modeled with a spring network, and the scene in Fig. 10-119 shows examples of cloth modeling using energy functions, with a texture-mapped pattern on one cloth. By adjusting the parameters in a network using energy-function calculations, different kinds of cloth can be modeled. Figure 10-120 illustrates models for cotton, wool, and polyester cotton materials draped over a table.

Physically based modeling methods are also applied in animations to more accurately describe motion paths. In the past, animations were often specified using spline paths and kinematics, where motion parameters are based only on

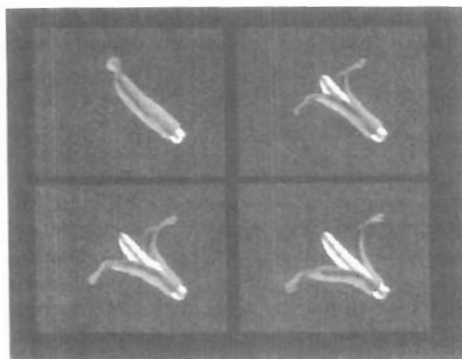


Figure 10-118
Modeling the flexible behavior of a banana peel with a spring network.
(Copyright © 1992 David Laidlaw, John Snyder, Adam Woodbury, and Alan Barr, Computer Graphics Lab, California Institute of Technology.)



Figure 10-119
Modeling the flexible behavior of cloth draped over furniture using energy-function minimization.
(Copyright © 1992 Gene Greger and David E. Breen, Design Research Center, Rensselaer Polytechnic Institute.)

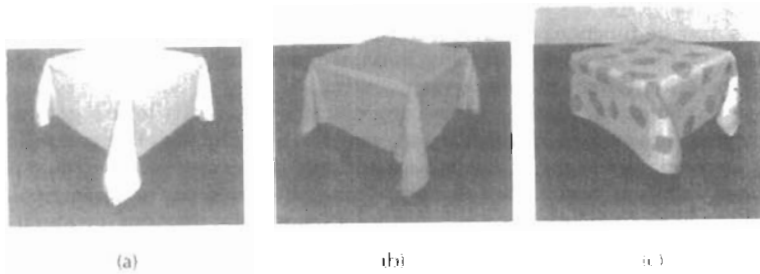


Figure 10-120
Modeling the characteristics of (a) cotton, (b) wool, and (c) polyester cotton using energy-function minimization. (Copyright © 1992 David E. Breen and Donald H. House, Design Research Center, Rensselaer Polytechnic Institute.)

position and velocity. Physically based modeling describes motion using dynamical equations, involving forces and accelerations. Animation descriptions based on the equations of dynamics produce more realistic motions than those based on the equations of kinematics.

10-22 VISUALIZATION OF DATA SETS

The use of graphical methods as an aid in scientific and engineering analysis is commonly referred to as **scientific visualization**. This involves the visualization of data sets and processes that may be difficult or impossible to analyze without graphical methods. For example, visualization techniques are needed to deal with the output of high-volume data sources such as supercomputers, satellite and spacecraft scanners, radio-astronomy telescopes, and medical scanners. Millions of data points are often generated from numerical solutions of computer simulations and from observational equipment, and it is difficult to determine trends and relationships by simply scanning the raw data. Similarly, visualization techniques are useful for analyzing processes that occur over a long time period or that cannot be observed directly, such as quantum-mechanical phenomena and special-relativity effects produced by objects traveling near the speed of light. Scientific visualization uses methods from computer graphics, image processing, computer vision, and other areas to visually display, enhance, and manipulate information to allow better understanding of the data. Similar methods employed by commerce, industry, and other nonscientific areas are sometimes referred to as **business visualization**.

Data sets are classified according to their spatial distribution and according to data type. Two-dimensional data sets have values distributed over a surface, and three-dimensional data sets have values distributed over the interior of a cube, a sphere, or some other region of space. Data types include scalars, vectors, tensors, and multivariate data.

Visual Representations for Scalar Fields

A scalar quantity is one that has a single value. Scalar data sets contain values that may be distributed in time, as well as over spatial positions. Also, the data

values may be functions of other scalar parameters. Some examples of physical scalar quantities are energy, density, mass, temperature, pressure, charge, resistance, reflectivity, frequency, and water content.

A common method for visualizing a scalar data set is to use graphs or charts that show the distribution of data values as a function of other parameters, such as position and time. If the data are distributed over a surface, we could plot the data values as vertical bars rising up from the surface, or we can interpolate the data values to display a smooth surface. Pseudo-color methods are also used to distinguish different values in a scalar data set, and color-coding techniques can be combined with graph and chart methods. To color code a scalar data set, we choose a range of colors and map the range of data values to the color range. For example, blue could be assigned to the lowest scalar value, and red could be assigned to the highest value. Figure 10-121 gives an example of a color-coded surface plot. Color coding a data set can be tricky, because some color combinations can lead to misinterpretations of the data.

Contour plots are used to display *isolines* (lines of constant scalar value) for a data set distributed over a surface. The isolines are spaced at some convenient interval to show the range and variation of the data values over the region of space. A typical application is a contour plot of elevations over a ground plane. Usually, contouring methods are applied to a set of data values that is distributed over a regular grid, as in Fig. 10-122. Regular grids have equally spaced grid lines, and data values are known at the grid intersections. Numerical solutions of computer simulations are usually set up to produce data distributions on a regular grid, while observed data sets are often irregularly spaced. Contouring methods have been devised for various kinds of nonregular grids, but often nonregular data distributions are converted to regular grids. A two-dimensional contouring algorithm traces the isolines from cell to cell within the grid by checking the four corners of grid cells to determine which cell edges are crossed by a

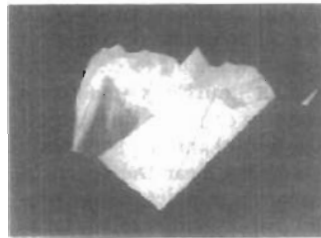


Figure 10-121

A financial surface plot showing stock-growth potential during the October 1987 stock-market crash. Red indicates high returns, and the plot shows that low-growth stocks performed better in the crash.

(Courtesy of Eng-Kiat Koh, Information Technology Institute, Republic of Singapore.)

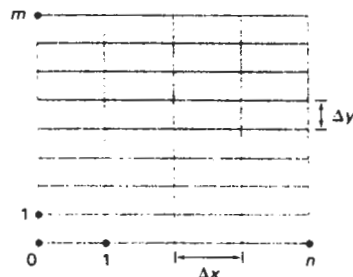


Figure 10-122

A regular, two-dimensional grid with data values at the intersection of the grid lines. The x grid lines have a constant Δx spacing, and the y grid lines have a constant Δy spacing, where the spacing in the x and y directions may not be the same.

particular isoline. The isolines are usually plotted as straight-line sections across each cell, as illustrated in Fig. 10-123. Sometimes isolines are plotted with spline curves, but spline fitting can lead to inconsistencies and misinterpretation of a data set. For example, two spline isolines could cross, or curved isoline paths might not be a true indicator of the data trends since data values are known only at the cell corners. Contouring packages can allow interactive adjustment of isolines by a researcher to correct any inconsistencies. An example of three, overlapping, color-coded contour plots in the xy plane is given in Fig. 10-124, and Fig. 10-125 shows contour lines and color coding for an irregularly shaped space.

For three-dimensional scalar data fields, we can take cross-sectional slices and display the two-dimensional data distributions over the slices. We could either color code the data values over a slice, or we could display isolines. Visualization packages typically provide a slicer routine that allows cross sections to be

Section 10-22

Visualization of Data Sets

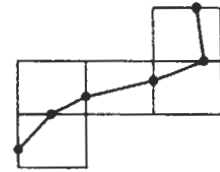


Figure 10-123
The path of an isoline across five grid cells.

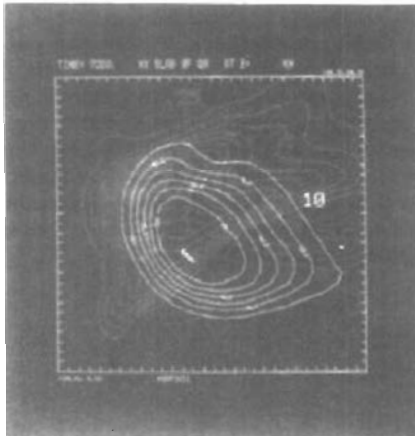


Figure 10-124
Color-coded contour plots for three data sets within the same region of the xy plane. (Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)

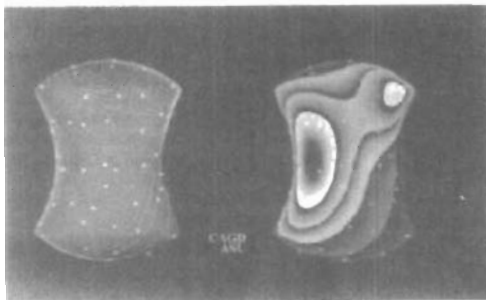


Figure 10-125
Color-coded contour plots over the surface of an apple-core-shaped region of space. (Courtesy of Greg Nielson, Department of Computer Science and Engineering, Arizona State University.)

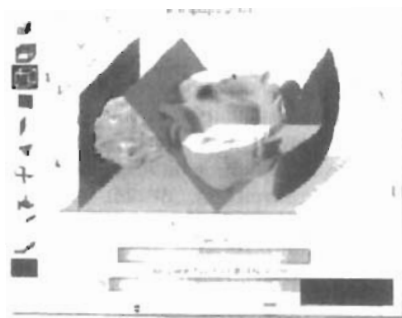


Figure 10-126
Cross-sectional slices of a three-dimensional data set. (Courtesy of Spyglass, Inc.)

taken at any angle. Figure 10-126 shows a display generated by a commercial slicer-dicer package.

Instead of looking at two-dimensional cross sections, we can plot one or more **isosurfaces**, which are simply three-dimensional contour plots (Fig. 10-127). When two overlapping isosurfaces are displayed, the outer surface is made transparent so that we can view the shape of both isosurfaces. Constructing an isosurface is similar to plotting isolines, except now we have three-dimensional grid cells and we need to check the values of the eight corners of a cell to locate sections of an isosurface. Figure 10-128 shows some examples of isosurface intersections with grid cells. Isosurfaces are modeled with triangle meshes, then surface-rendering algorithms are applied to display the final shape.

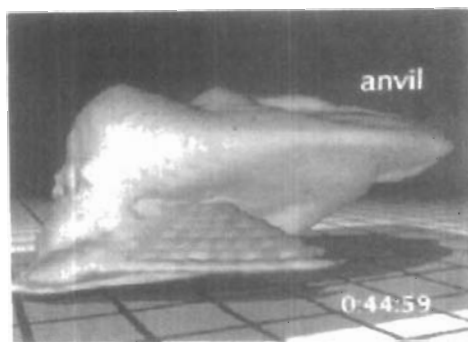


Figure 10-127
An isosurface generated from a set of water-content values obtained from a numerical model of a thunderstorm. (Courtesy of Bob Wilhelmson, Department of Atmospheric Sciences and National Center for Supercomputing Applications, University of Illinois at Urbana Champaign.)

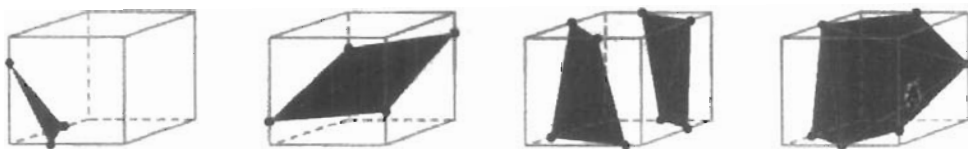


Figure 10-128
Isosurface intersections with grid cells, modeled with triangle patches.

Volume rendering, which is often somewhat like an X-ray picture, is another method for visualizing a three-dimensional data set. The interior information about a data set is projected to a display screen using the ray-casting methods introduced in Section 10-15. Along the ray path from each screen pixel (Fig. 10-129), interior data values are examined and encoded for display. Often, data values at the grid positions are averaged so that one value is stored for each voxel of the data space. How the data are encoded for display depends on the application. Seismic data, for example, is often examined to find the maximum and minimum values along each ray. The values can then be color coded to give information about the width of the interval and the minimum value. In medical applications, the data values are opacity factors in the range from 0 to 1 for the tissue and bone layers. Bone layers are completely opaque, while tissue is somewhat transparent (low opacity). Along each ray, the opacity factors are accumulated until either the total is greater than or equal to 1, or until the ray exits at the back of the three-dimensional data grid. The accumulated opacity value is then displayed as a pixel-intensity level, which can be gray scale or color. Figure 10-130 shows a volume visualization of a medical data set describing the structure of a dog heart. For this volume visualization, a color-coded plot of the distance to the maximum voxel value along each pixel ray was displayed.

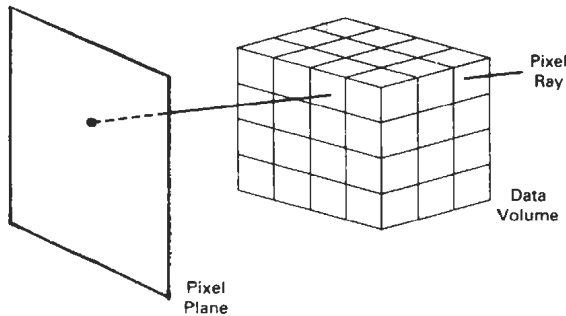


Figure 10-129

Volume visualization of a regular, Cartesian data grid using ray casting to examine interior data values.

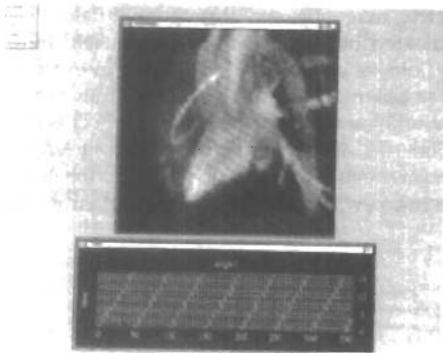


Figure 10-130

Volume visualization of a data set for a dog heart, obtained by plotting the color-coded distance to the maximum voxel value for each pixel. (Courtesy of Patrick Moran and Clinton Potter, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)

Visual Representations for Vector Fields

A vector quantity \mathbf{V} in three-dimensional space has three scalar values (V_x , V_y , V_z), one for each coordinate direction, and a two-dimensional vector has two components (V_x , V_y). Another way to describe a vector quantity is by giving its magnitude $|\mathbf{V}|$ and its direction as a unit vector \mathbf{u} . As with scalars, vector quantities may be functions of position, time, and other parameters. Some examples of physical vector quantities are velocity, acceleration, force, electric fields, magnetic fields, gravitational fields, and electric current.

One way to visualize a vector field is to plot each data point as a small arrow that shows the magnitude and direction of the vector. This method is most often used with cross-sectional slices, as in Fig. 10-131, since it can be difficult to see the data trends in a three-dimensional region cluttered with overlapping arrows. Magnitudes for the vector values can be shown by varying the lengths of the arrows, or we can make all arrows the same size, but make the arrows different colors according to a selected color coding for the vector magnitudes.

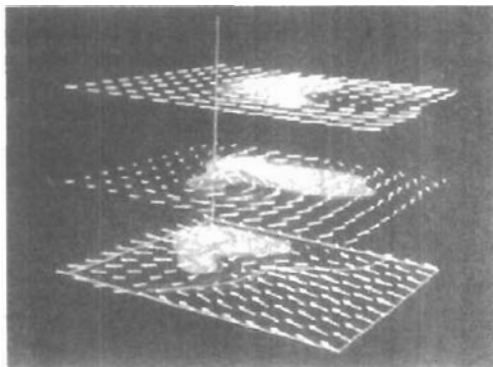


Figure 10-131
Arrow representation for a vector field over cross-sectional slices. (Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)

We can also represent vector values by plotting *field lines* or *streamlines*. Field lines are commonly used for electric, magnetic, and gravitational fields. The magnitude of the vector values is indicated by the spacing between field lines, and the direction is the tangent to the field, as shown in Fig. 10-132. An example of a streamline plot of a vector field is shown in Fig. 10-133. Streamlines can be displayed as wide arrows, particularly when a whirlpool, or vortex, effect is present. An example of this is given in Fig. 10-134, which displays swirling airflow patterns inside a thunderstorm. For animations of fluid flow, the behavior of the vector field can be visualized by tracking particles along the flow direction. An

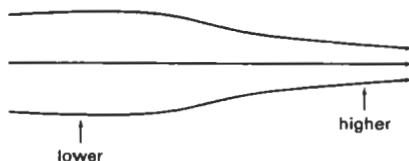


Figure 10-132
Field-line representation for a vector data set.

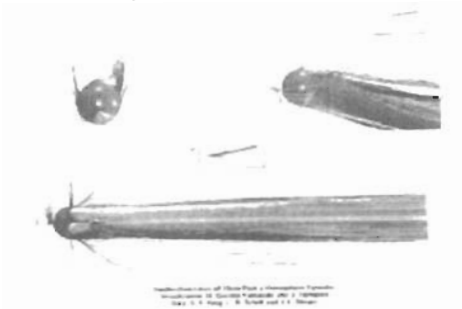


Figure 10-133

Visualizing airflow around a cylinder with a hemispherical cap that is tilted slightly relative to the incoming direction of the airflow.

(Courtesy of M. Gerald-Yamasaki, J. Huilquist, and Sam Useton, NASA Ames Research Center.)

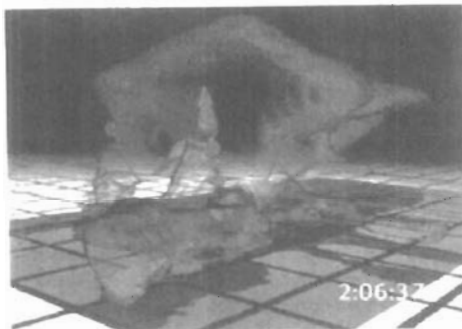


Figure 10-134

Twisting airflow patterns, visualized with wide streamlines inside a transparent isosurface plot of a thunderstorm. (Courtesy of Bob Wilhelmsen, Department of Atmospheric Sciences and National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)

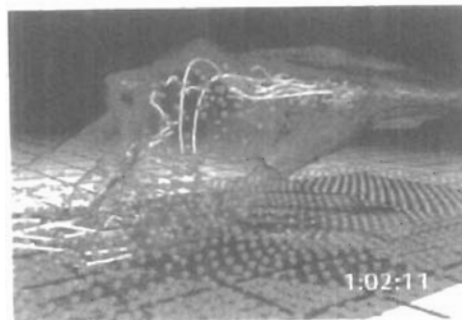


Figure 10-135

Airflow patterns, visualized with both streamlines and particle motion inside a transparent isosurface plot of a thunderstorm. Rising sphere particles are colored orange, and falling sphere particles are blue. (Courtesy of Bob Wilhelmsen, Department of Atmospheric Sciences and National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)

example of a vector-field visualization using both streamlines and particles is shown in Fig. 10-135.

Sometimes, only the magnitudes of the vector quantities are displayed. This is often done when multiple quantities are to be visualized at a single position, or when the directions do not vary much in some region of space, or when vector directions are of less interest.

Visual Representations for Tensor Fields

A tensor quantity in three-dimensional space has nine components and can be represented with a 3 by 3 matrix. Actually, this representation is used for a *second-order tensor*, and higher-order tensors do occur in some applications, particularly general relativity. Some examples of physical, second-order tensors are

stress and strain in a material subjected to external forces, conductivity (or resistivity) of an electrical conductor, and the metric tensor, which gives the properties of a particular coordinate space. The stress tensor in Cartesian coordinates, for example, can be represented as

$$\begin{bmatrix} \sigma_x & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_y & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_z \end{bmatrix} \quad (10-114)$$

Tensor quantities are frequently encountered in anisotropic materials, which have different properties in different directions. The x , xy , and xz elements of the conductivity tensor, for example, describe the contributions of electric field components in the x , y , and z directions to the current in the x direction. Usually, physical tensor quantities are symmetric, so that the tensor has only six distinct values. For instance, the xy and yx components of the stress tensor are the same.

Visualization schemes for representing all six components of a second-order tensor quantity are based on devising shapes that have six parameters. One graphical representation for a tensor is shown in Fig. 10-136. The three diagonal elements of the tensor are used to construct the magnitude and direction of the arrow, and the three off-diagonal terms are used to set the shape and color of the elliptical disk.

Instead of trying to visualize all six components of a tensor quantity, we can reduce the tensor to a vector or a scalar. Using a vector representation, we can simply display a vector representation for the diagonal elements of the tensor. And by applying *tensor-contraction* operations, we can obtain a scalar representation. For example, stress and strain tensors can be contracted to generate a scalar strain-energy density that can be plotted at points in a material subject to external forces (Fig. 10-137).

Visual Representations for Multivariate Data Fields

In some applications, at each grid position over some region of space, we may have multiple data values, which can be a mixture of scalar, vector, and even ten-

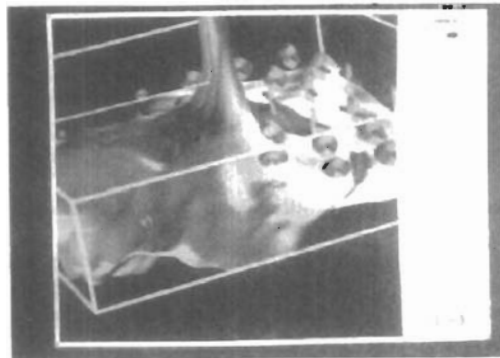


Figure 10-136
Representing stress and strain tensors with an elliptical disk and a rod over the surface of a stressed material. (Courtesy of Bob Haber, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)

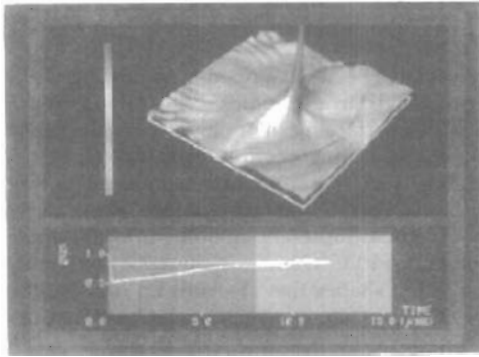


Figure 10-137
Representing stress and strain tensors with a strain-energy density plot in a visualization of crack propagation on the surface of a stressed material.
(Courtesy of Bob Haber, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)

tor values. As an example, for a fluid-flow problem, we may have fluid velocity, temperature, and density values at each three-dimensional position. Thus, we have five scalar values to display at each position, and the situation is similar to displaying a tensor field.

A method for displaying multivariate data fields is to construct graphical objects, sometimes referred to as **glyphs**, with multiple parts. Each part of a glyph represents a physical quantity. The size and color of each part can be used to display information about scalar magnitudes. To give directional information for a vector field, we can use a wedge, a cone, or some other pointing shape for the glyph part representing the vector. An example of the visualization of a multivariate data field using a glyph structure at selected grid positions is shown in Fig. 10-138.

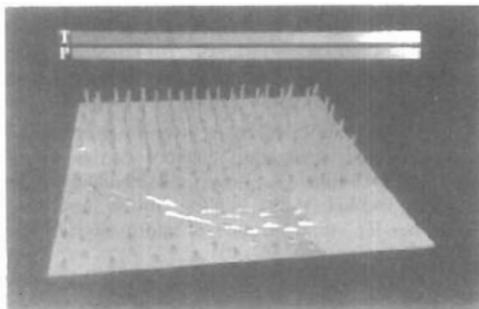


Figure 10-138
One frame from an animated visualization of a multivariate data field using glyphs. The wedge-shaped part of the glyph indicates the direction of a vector quantity at each point. (Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)

SUMMARY

Many representations have been developed for modeling the wide variety of objects that might be displayed in a graphics scene. "Standard graphics objects" are those represented with a surface mesh of polygon facets. Polygon-mesh representations are typically derived from other representations.

Surface functions, such as the quadrics, are used to describe spheres and other smooth surfaces. For design applications, we can use superquadrics, splines, or blobby objects to represent smooth surface shapes. In addition, construction techniques, such as CSG and sweep representations, are useful for designing compound object shapes that are built up from a set of simpler shapes. And interior, as well as surface, information can be stored in octree representations.

Descriptions for natural objects, such as trees and clouds, and other irregularly shaped objects can be specified with fractals, shape grammars, and particle systems. Finally, visualization techniques use graphic representations to display numerical or other types of data sets. The various types of numerical data include scalar, vector, and tensor values. Also many scientific visualizations require methods for representing multivariate data sets, that contain a combination of the various data types.

REFERENCES

- A detailed discussion of superquadrics is contained in Barr (1981). For more information on blobby object modeling, see Blinn (1982). The metaball model is discussed in Nishimura (1985); and the soft-object model is discussed in Wyville, Wyville, and McPheeters (1987).
- Sources of information on parametric curve and surface representations include Bezier (1972), Burt and Adelson (1983), Barsky (1983, 1984), Kochanek and Bartels (1984), Farouki and Hinds (1985), Huitric and Nahas (1985), Mortenson (1985), Farin (1988), and Rogers and Adams (1990).
- Octrees and quadrees are discussed by Doctor (1981), Yamaguchi, Kunii, and Fujimura (1984), and by Carlbom, Chakravarty, and Vanderschel (1985). Sollic-modeling references include Casale and Stanton (1985) and Requicha and Rossignat (1992).
- For further information on fractal representations see Mandelbrot (1977, 1982), Fourrier, Fussell, and Carpenter (1982), Norton (1982), Peitgen and Richter (1986), Peitgen and Saupe (1988), Koh and Hearn (1992), and Barnsley (1993). Shape grammars are discussed in Glassner (1992), and particle systems are discussed in Reeves (1983). A discussion of physically based modeling is given in Barzel (1992).
- A general introduction to visualization methods is given in Hearn and Baker (1991). Additional information on specific visualization methods can be found in Sabin (1985), Lorensen and Cline (1987), Drebin, Carpenter, and Hanrahan (1988), Sabella (1988), Upson and Keeler (1988), Frenkel (1989), Nielson, Shriver, and Rosenblum (1990), and Nielson (1993). Guidelines for visual displays of information are given in Tufte (1983, 1990).

EXERCISES

- 10-1. Set up geometric data tables as in Fig. 10-2 for a unit cube.
- 10-2. Set up geometric data tables for a unit cube using only (a) vertex and polygon tables, and (b) a single polygon table. Compare the two methods for representing the unit cube with a representation using three data tables, and estimate storage requirements for each.

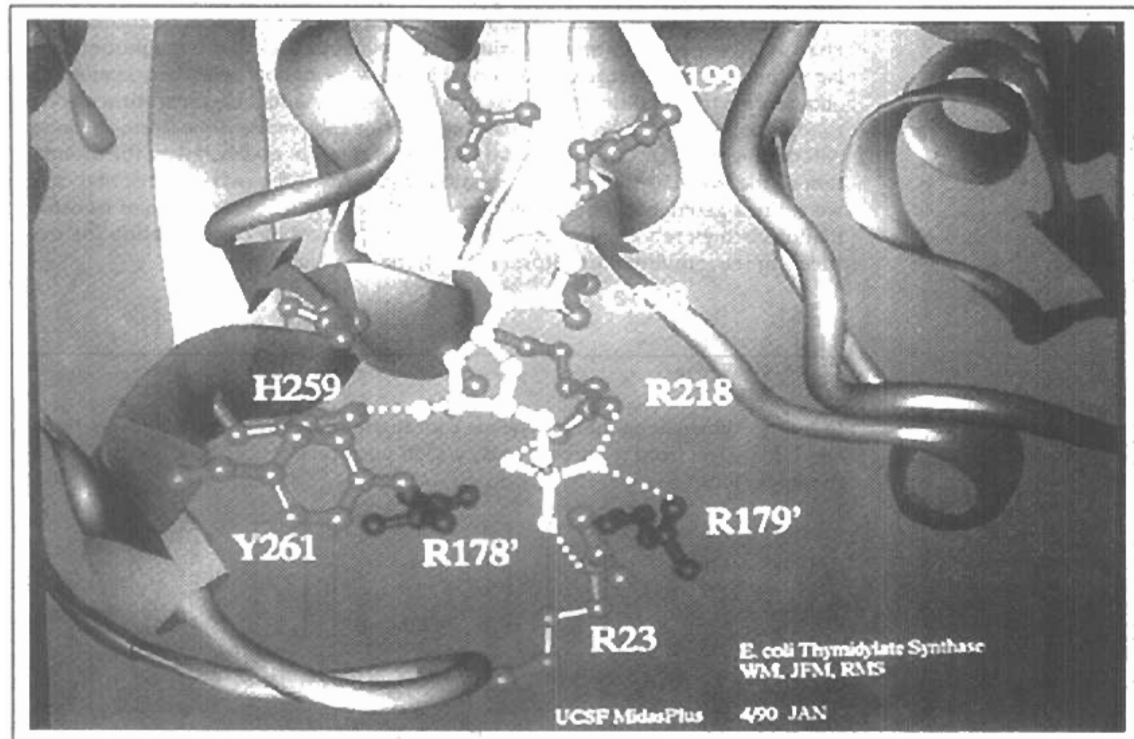
- 10-3. Define an efficient polygon representation for a cylinder. Justify your choice of representation.
- 10-4. Set up a procedure for establishing polygon tables for any input set of data points defining an object.
- 10-5. Devise routines for checking the data tables in Fig. 10-2 for consistency and completeness.
- 10-6. Write a program that calculates parameters A , B , C , and D for any set of three-dimensional plane surfaces defining an object.
- 10-7. Given the plane parameters A , B , C , and D for all surfaces of an object, devise an algorithm to determine whether any specified point is inside or outside the object.
- 10-8. How would the values for parameters A , B , C , and D in the equation of a plane surface have to be altered if the coordinate reference is changed from a right-handed system to a left-handed system?
- 10-9. Set up an algorithm for converting any specified sphere, ellipsoid, or cylinder to a polygon-mesh representation.
- 10-10. Set up an algorithm for converting a specified superellipsoid to a polygon-mesh representation.
- 10-11. Set up an algorithm for converting a metaball representation to a polygon-mesh representation.
- 10-12. Write a routine to display a two-dimensional, cardinal-spline curve, given an input set of control points in the xy plane.
- 10-13. Write a routine to display a two-dimensional, Kochanek-Bartels curve, given an input set of control points in the xy plane.
- 10-14. Determine the quadratic Bézier blending functions for three control points. Plot each function and label the maximum and minimum values.
- 10-15. Determine the Bézier blending functions for five control points. Plot each function and label the maximum and minimum values.
- 10-16. Write an efficient routine to display two-dimensional, cubic Bézier curves, given a set of four control points in the xy plane.
- 10-17. Write a routine to design two-dimensional, cubic Bézier curve shapes that have first-order piecewise continuity. Use an interactive technique for selecting control-point positions in the xy plane for each section of the curve.
- 10-18. Write a routine to design two-dimensional, cubic Bézier curve shapes that have second-order piecewise continuity. Use an interactive technique for selecting control-point positions in the xy plane for each section of the curve.
- 10-19. Write a routine to display a cubic Bézier curve using a subdivision method.
- 10-20. Determine the blending functions for uniform, periodic B-spline curves for $d = 5$.
- 10-21. Determine the blending functions for uniform, periodic B-spline curves for $d = 6$.
- 10-22. Write a program using forward differences to calculate points along a two-dimensional, uniform, periodic, cubic B-spline curve, given an input set of control points.
- 10-23. Write a routine to display any specified conic in the xy plane using a rational Bézier spline representation.
- 10-24. Write a routine to display any specified conic in the xy plane using a rational B-spline representation.
- 10-25. Develop an algorithm for calculating the normal vector to a Bezier surface at the point $P(u, v)$.
- 10-26. Write a program to display any specified quadratic curve using forward differences to calculate points along the curve path.
- 10-27. Write a program to display any specified cubic curve using forward differences to calculate points along the curve path.
- 10-28. Derive expressions for calculating the forward differences for any specified quadratic curve.

- 10-29. Derive expressions for calculating the forward differences for any specified cubic curve.
- 10-30. Set up procedures for generating the description of a three-dimensional object from input parameters that define the object in terms of a translational sweep.
- 10-31. Develop procedures for generating the description of a three-dimensional object using input parameters that define the object in terms of a rotational sweep.
- 10-32. Devise an algorithm for generating solid objects as combinations of three-dimensional primitive shapes, each defined as a set of surfaces, using constructive solid-geometry methods.
- 10-33. Develop an algorithm for performing constructive solid-geometry modeling using a primitive set of solids defined in octree structures.
- 10-34. Develop an algorithm for encoding a two-dimensional scene as a quadtree representation.
- 10-35. Set up an algorithm for loading a quadtree representation of a scene into a frame buffer for display of the scene.
- 10-36. Write a routine to convert the polygon definition of a three-dimensional object into an octree representation.
- 10-37. Using the random, midpoint-displacement method, write a routine to create a mountain outline, starting with a horizontal line in the xy plane.
- 10-38. Write a routine to calculate elevations above a ground plane using the random, midpoint-displacement method.
- 10-39. Write a program for generating a fractal snowflake (Koch curve) for any given number of iterations.
- 10-40. Write a program to generate a fractal curve for a specified number of iterations using one of the generators in Fig. 10-71 or 10-72. What is the fractal dimension of your curve?
- 10-41. Write a program to generate fractal curves using the self-squaring function $f(z) = z^2 + \lambda$, where λ is any selected complex constant.
- 10-42. Write a program to generate fractal curves using the self-squaring function $f(x) = i(z^2 + 1)$, where $i = \sqrt{-1}$.
- 10-43. Write a routine to interactively select different color combinations for displaying the Mandelbrot set.
- 10-44. Write a program to interactively select any rectangular region of the Mandelbrot set and to zoom in on the selected region.
- 10-45. Write a routine to implement point inversion, Eq. 10-112, for any specified circle and any given point position.
- 10-46. Devise a set of geometric-substitution rules for altering the shape of an equilateral triangle.
- 10-47. Write a program to display the stages in the conversion of an equilateral triangle into another shape, given a set of geometric-substitution rules.
- 10-48. Write a program to model an exploding firecracker in the xy plane using a particle system.
- 10-49. Devise an algorithm for modeling a rectangle as a nonrigid body, using identical springs for the four sides of the rectangle.
- 10-50. Write a routine to visualize a two-dimensional, scalar data set using pseudo-color methods.
- 10-51. Write a routine to visualize a two-dimensional, scalar data set using contour lines.
- 10-52. Write a routine to visualize a two-dimensional, vector data set using an arrow representation for the vector values. Make all arrows the same length, but display the arrows with different colors to represent the different vector magnitudes.

CHAPTER

11

Three-Dimensional
Geometric and Modeling
Transformations



Methods for geometric transformations and object modeling in three dimensions are extended from two-dimensional methods by including considerations for the z coordinate. We now translate an object by specifying a three-dimensional translation vector, which determines how much the object is to be moved in each of the three coordinate directions. Similarly, we scale an object with three coordinate scaling factors. The extension for three-dimensional rotation is less straightforward. When we discussed two-dimensional rotations in the xy plane, we needed to consider only rotations about axes that were perpendicular to the xy plane. In three-dimensional space, we can now select any spatial orientation for the rotation axis. Most graphics packages handle three-dimensional rotation as a composite of three rotations, one for each of the three Cartesian axes. Alternatively, a user can easily set up a general rotation matrix, given the orientation of the axis and the required rotation angle. As in the two-dimensional case, we express geometric transformations in matrix form. Any sequence of transformations is then represented as a single matrix, formed by concatenating the matrices for the individual transformations in the sequence.

11-1

TRANSLATION

In a three-dimensional homogeneous coordinate representation, a point is translated (Fig. 11-1) from position $P = (x, y, z)$ to position $P' = (x', y', z')$ with the matrix operation

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (11-1)$$

or

$$P' = T \cdot P \quad (11-2)$$

Parameters t_x , t_y , and t_z , specifying translation distances for the coordinate directions x , y , and z , are assigned any real values. The matrix representation in Eq. 11-1 is equivalent to the three equations

$$x' = x + t_x, \quad y' = y + t_y, \quad z' = z + t_z \quad (11-3)$$

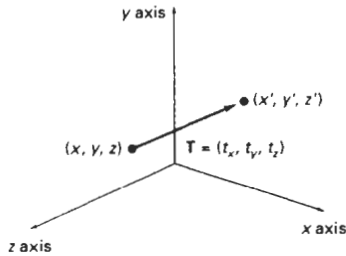


Figure 11-1
Translating a point with translation vector $\mathbf{T} = (t_x, t_y, t_z)$.

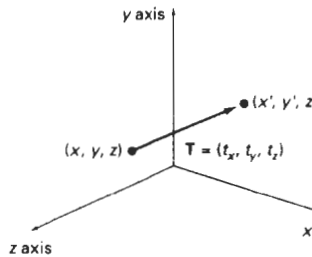


Figure 11-2
Translating an object with translation vector \mathbf{T} .

An object is translated in three dimensions by transforming each of the defining points of the object. For an object represented as a set of polygon surfaces, we translate each vertex of each surface (Fig. 11-2) and redraw the polygon facets in the new position.

We obtain the inverse of the translation matrix in Eq. 11-1 by negating the translation distances t_x , t_y , and t_z . This produces a translation in the opposite direction, and the product of a translation matrix and its inverse produces the identity matrix.

11-2

ROTATION

To generate a rotation transformation for an object, we must designate an axis of rotation (about which the object is to be rotated) and the amount of angular rotation. Unlike two-dimensional applications, where all transformations are carried out in the xy plane, a three-dimensional rotation can be specified around any line in space. The easiest rotation axes to handle are those that are parallel to the coordinate axes. Also, we can use combinations of coordinate-axis rotations (along with appropriate translations) to specify any general rotation.

By convention, positive rotation angles produce counterclockwise rotations about a coordinate axis, if we are looking along the positive half of the axis toward the coordinate origin (Fig. 11-3). This agrees with our earlier discussion of rotation in two dimensions, where positive rotations in the xy plane are counterclockwise about axes parallel to the z axis.

Coordinate-Axes Rotations

The two-dimensional **z-axis rotation** equations are easily extended to three dimensions:

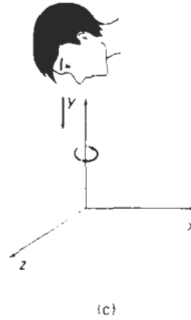
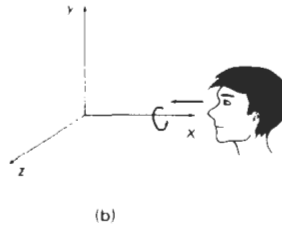
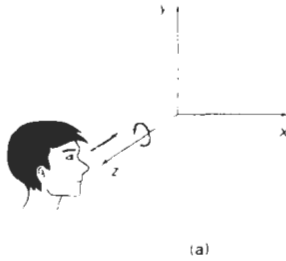


Figure 11-3
Positive rotation directions
about the coordinate axes are
counterclockwise, when looking
toward the origin from a positive
coordinate position on each axis.

$$\begin{aligned}x' &= x \cos \theta - y \sin \theta \\y' &= x \sin \theta + y \cos \theta \\z' &= z\end{aligned}\quad (11-4)$$

Parameter θ specifies the rotation angle. In homogeneous coordinate form, the three-dimensional z-axis rotation equations are expressed as

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (11-5)$$

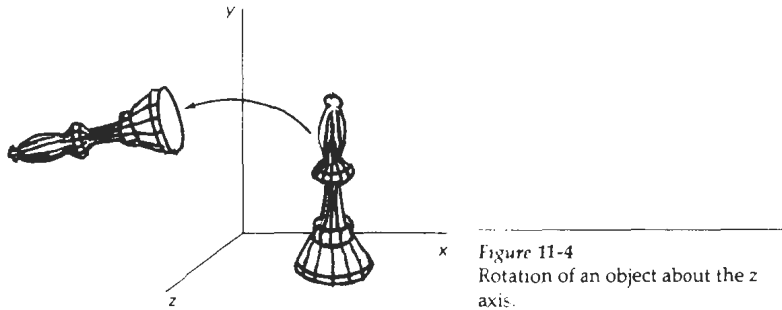


Figure 11-4
Rotation of an object about the z axis.

which we can write more compactly as

$$\mathbf{P}' = \mathbf{R}_z(\theta) \cdot \mathbf{P} \quad (11-6)$$

Figure 11-4 illustrates rotation of an object about the z axis.

Transformation equations for rotations about the other two coordinate axes can be obtained with a cyclic permutation of the coordinate parameters x , y , and z in Eqs. 11-4. That is, we use the replacements

$$x \rightarrow y \rightarrow z \rightarrow x \quad (11-7)$$

as illustrated in Fig. 11-5.

Substituting permutations 11-7 in Eqs. 11-4, we get the equations for an **x -axis rotation**:

$$\begin{aligned} y' &= y \cos \theta - z \sin \theta \\ z' &= y \sin \theta + z \cos \theta \\ x' &= x \end{aligned} \quad (11-8)$$

which can be written in the homogeneous coordinate form

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (11-9)$$

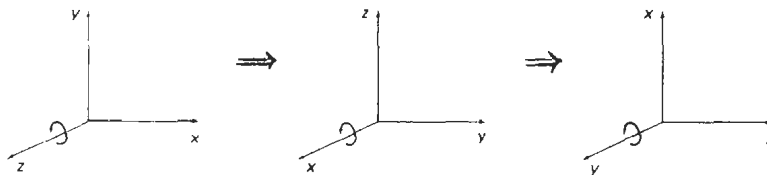


Figure 11-5

Cyclic permutation of the Cartesian-coordinate axes to produce the three sets of coordinate-axis rotation equations.

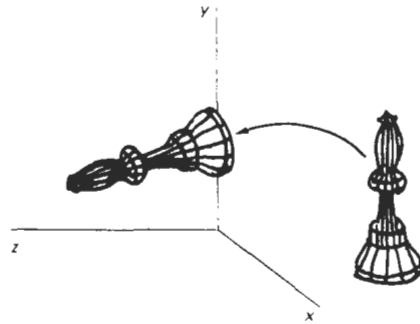


Figure 11-6
Rotation of an object about the
 x axis.

or

$$\mathbf{P}' = \mathbf{R}_x(\theta) \cdot \mathbf{P} \quad (11-10)$$

Rotation of an object around the x axis is demonstrated in Fig. 11.6.

Cyclically permuting coordinates in Eqs. 11-8 give us the transformation equations for a **y -axis rotation**:

$$\begin{aligned} z' &= z \cos \theta - x \sin \theta \\ x' &= z \sin \theta + x \cos \theta \\ y' &= y \end{aligned} \quad (11-11)$$

The matrix representation for y -axis rotation is

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (11-12)$$

or

$$\mathbf{P}' = \mathbf{R}_y(\theta) \cdot \mathbf{P} \quad (11-13)$$

An example of y -axis rotation is shown in Fig. 11-7.

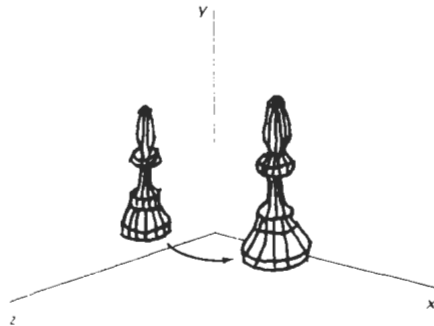


Figure 11-7
Rotation of an object about the
 y axis.

An inverse rotation matrix is formed by replacing the rotation angle θ by $-\theta$. Negative values for rotation angles generate rotations in a clockwise direction, so the identity matrix is produced when any rotation matrix is multiplied by its inverse. Since only the sine function is affected by the change in sign of the rotation angle, the inverse matrix can also be obtained by interchanging rows and columns. That is, we can calculate the inverse of any rotation matrix \mathbf{R} by evaluating its transpose ($\mathbf{R}^{-1} = \mathbf{R}^T$). This method for obtaining an inverse matrix holds also for any composite rotation matrix.

General Three-Dimensional Rotations

A rotation matrix for any axis that does not coincide with a coordinate axis can be set up as a composite transformation involving combinations of translations and the coordinate-axes rotations. We obtain the required composite matrix by first setting up the transformation sequence that moves the selected rotation axis onto one of the coordinate axes. Then we set up the rotation matrix about that coordinate axis for the specified rotation angle. The last step is to obtain the inverse transformation sequence that returns the rotation axis to its original position.

In the special case where an object is to be rotated about an axis that is parallel to one of the coordinate axes, we can attain the desired rotation with the following transformation sequence.

1. Translate the object so that the rotation axis coincides with the parallel coordinate axis.
2. Perform the specified rotation about that axis.
3. Translate the object so that the rotation axis is moved back to its original position.

The steps in this sequence are illustrated in Fig. 11-8. Any coordinate position \mathbf{P} on the object in this figure is transformed with the sequence shown as

$$\mathbf{P}' = \mathbf{T}^{-1} \cdot \mathbf{R}_i(\theta) \cdot \mathbf{T} \cdot \mathbf{P}$$

where the composite matrix for the transformation is

$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{R}_i(\theta) \cdot \mathbf{T}$$

which is of the same form as the two-dimensional transformation sequence for rotation about an arbitrary pivot point.

When an object is to be rotated about an axis that is not parallel to one of the coordinate axes, we need to perform some additional transformations. In this case, we also need rotations to align the axis with a selected coordinate axis and to bring the axis back to its original orientation. Given the specifications for the rotation axis and the rotation angle, we can accomplish the required rotation in five steps:

1. Translate the object so that the rotation axis passes through the coordinate origin.
2. Rotate the object so that the axis of rotation coincides with one of the coordinate axes.
3. Perform the specified rotation about that coordinate axis.

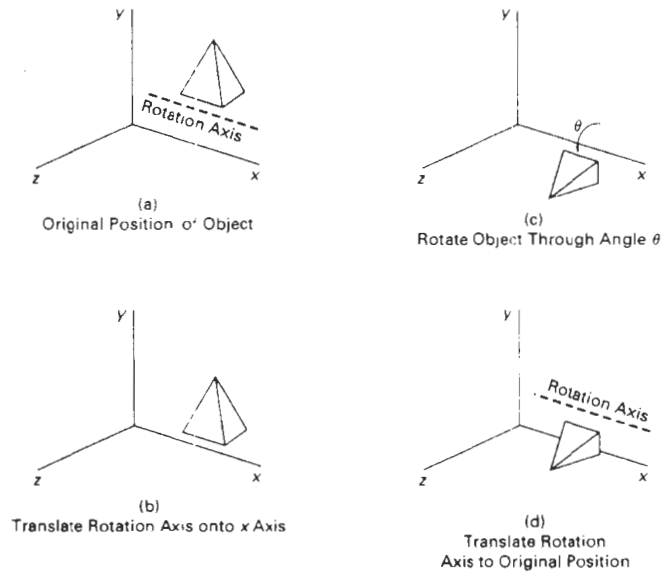


Figure 11-8

Sequence of transformations for rotating an object about an axis that is parallel to the x axis

4. Apply inverse rotations to bring the rotation axis back to its original orientation.
5. Apply the inverse translation to bring the rotation axis back to its original position.

We can transform the rotation axis onto any of the three coordinate axes. The z axis is a reasonable choice, and the following discussion shows how to set up the transformation matrices for getting the rotation axis onto the z axis and returning the rotation axis to its original position (Fig. 11-9).

A rotation axis can be defined with two coordinate positions, as in Fig. 11-10, or with one coordinate point and direction angles (or direction cosines) between the rotation axis and two of the coordinate axes. We will assume that the rotation axis is defined by two points, as illustrated, and that the direction of rotation is to be counterclockwise when looking along the axis from P_2 to P_1 . An axis vector is then defined by the two points as

$$\begin{aligned} \mathbf{V} &= \mathbf{P}_2 - \mathbf{P}_1 \\ &= (x_2 - x_1, y_2 - y_1, z_2 - z_1) \end{aligned} \quad (11-14)$$

A unit vector \mathbf{u} is then defined along the rotation axis as

$$\mathbf{u} = \frac{\mathbf{V}}{|\mathbf{V}|} = (a, b, c) \quad (11-15)$$

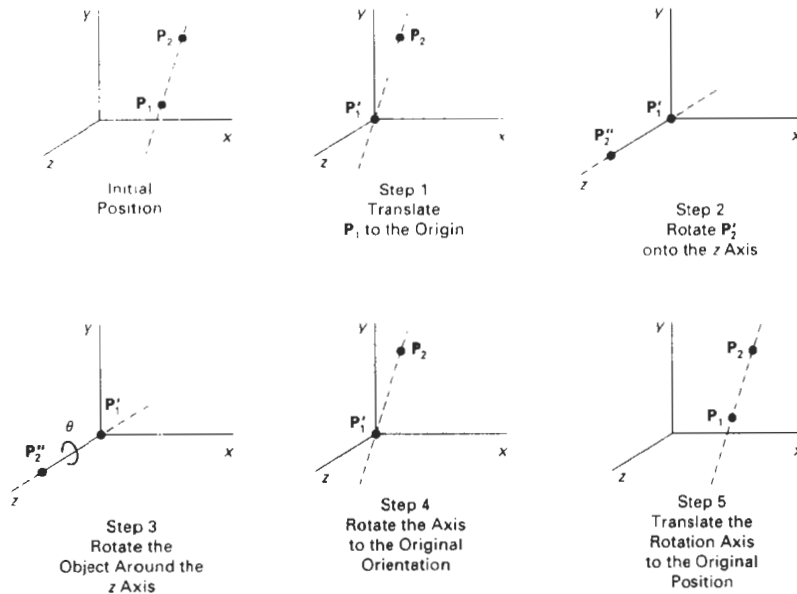


Figure 11-9

Five transformation steps for obtaining a composite matrix for rotation about an arbitrary axis, with the rotation axis projected onto the z axis.

where the components a , b , and c of unit vector \mathbf{u} are the direction cosines for the rotation axis:

$$a = \frac{x_2 - x_1}{|\mathbf{V}|}, \quad b = \frac{y_2 - y_1}{|\mathbf{V}|}, \quad c = \frac{z_2 - z_1}{|\mathbf{V}|} \quad (11-16)$$

If the rotation is to be in the opposite direction (clockwise when viewing from P_2 to P_1), then we would reverse axis vector \mathbf{V} and unit vector \mathbf{u} so that they point from P_2 to P_1 .

The first step in the transformation sequence for the desired rotation is to set up the translation matrix that repositions the rotation axis so that it passes through the coordinate origin. For the desired direction of rotation (Fig. 11-10), we accomplish this by moving point P_1 to the origin. (If the rotation direction had been specified in the opposite direction, we would move P_2 to the origin.) This translation matrix is

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11-17)$$

which repositions the rotation axis and the object, as shown in Fig. 11-11.

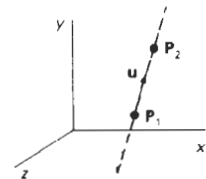


Figure 11-10

An axis of rotation (dashed line) defined with points P_1 and P_2 . The direction for the unit axis vector \mathbf{u} is determined by the specified rotation direction.

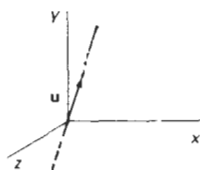


Figure 11-11
Translation of the rotation axis to the coordinate origin.

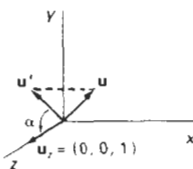


Figure 11-13
Rotation of \mathbf{u} around the x axis into the xz plane is accomplished by rotating \mathbf{u}' (the projection of \mathbf{u} in the yz plane) through angle α onto the z axis.

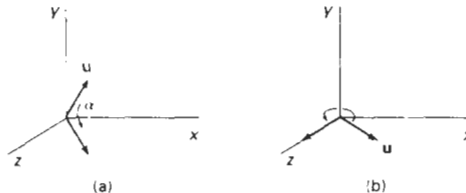


Figure 11-12
Unit vector \mathbf{u} is rotated about the x axis to bring it into the xz plane (a), then it is rotated around the y axis to align it with the z axis (b).

Now we need the transformations that will put the rotation axis on the z axis. We can use the coordinate-axis rotations to accomplish this alignment in two steps. There are a number of ways to perform the two steps. We will first rotate about the x axis to transform vector \mathbf{u} into the xz plane. Then we swing \mathbf{u} around to the z axis using a y -axis rotation. These two rotations are illustrated in Fig. 11-12 for one possible orientation of vector \mathbf{u} .

Since rotation calculations involve sine and cosine functions, we can use standard vector operations (Appendix A) to obtain elements of the two rotation matrices. Dot-product operations allow us to determine the cosine terms, and vector cross products provide a means for obtaining the sine terms.

We establish the transformation matrix for rotation around the x axis by determining the values for the sine and cosine of the rotation angle necessary to get \mathbf{u} into the xz plane. This rotation angle is the angle between the projection of \mathbf{u} in the yz plane and the positive z axis (Fig. 11-13). If we designate the projection of \mathbf{u} in the yz plane as the vector $\mathbf{u}' = (0, b, c)$, then the cosine of the rotation angle α can be determined from the dot product of \mathbf{u}' and the unit vector \mathbf{u}_z along the z axis:

$$\cos \alpha = \frac{\mathbf{u}' \cdot \mathbf{u}_z}{|\mathbf{u}'| |\mathbf{u}_z|} = \frac{d}{d} \quad (11-18)$$

where d is the magnitude of \mathbf{u}' :

$$d = \sqrt{b^2 + c^2} \quad (11-19)$$

Similarly, we can determine the sine of α from the cross product of \mathbf{u}' and \mathbf{u}_z . The coordinate-independent form of this cross product is

$$\mathbf{u}' \times \mathbf{u}_z = \mathbf{u}_x |\mathbf{u}'| |\mathbf{u}_z| \sin \alpha \quad (11-20)$$

and the Cartesian form for the cross product gives us

$$\mathbf{u}' \times \mathbf{u}_z = \mathbf{u}_x \cdot b \quad (11-21)$$

Equating the right sides of Eqs. 11-20 and 11-21, and noting that $|\mathbf{u}_z| = 1$ and $|\mathbf{u}'| = d$, we have

$$d \sin \alpha = b$$

or

$$\sin \alpha = \frac{b}{d} \quad (11-22)$$

Now that we have determined the values for $\cos \alpha$ and $\sin \alpha$ in terms of the components of vector \mathbf{u} , we can set up the matrix for rotation of \mathbf{u} about the x axis:

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c/d & -b/d & 0 \\ 0 & b/d & c/d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11-23)$$

This matrix rotates unit vector \mathbf{u} about the x axis into the xz plane.

Next we need to determine the form of the transformation matrix that will swing the unit vector in the xz plane counterclockwise around the y axis onto the positive z axis. The orientation of the unit vector in the xz plane (after rotation about the x axis) is shown in Fig. 11-14. This vector, labeled \mathbf{u}'' , has the value a for its x component, since rotation about the x axis leaves the x component unchanged. Its z component is d (the magnitude of \mathbf{u}'), because vector \mathbf{u}' has been rotated onto the z axis. And the y component of \mathbf{u}'' is 0, because it now lies in the xz plane. Again, we can determine the cosine of rotation angle β from expressions for the dot product of unit vectors \mathbf{u}'' and \mathbf{u}_z :

$$\cos \beta = \frac{\mathbf{u}'' \cdot \mathbf{u}_z}{|\mathbf{u}''| |\mathbf{u}_z|} = d \quad (11-24)$$

since $|\mathbf{u}_z| = |\mathbf{u}''| = 1$. Comparing the coordinate-independent form of the cross product

$$\mathbf{u}'' \times \mathbf{u}_z = \mathbf{u}_y |\mathbf{u}''| |\mathbf{u}_z| \sin \beta \quad (11-25)$$

with the Cartesian form

$$\mathbf{u}'' \times \mathbf{u}_z = \mathbf{u}_y \cdot (-a) \quad (11-26)$$

we find that

$$\sin \beta = -a \quad (11-27)$$

Thus, the transformation matrix for rotation of \mathbf{u}'' about the y axis is

$$\mathbf{R}_y(\beta) = \begin{bmatrix} d & 0 & -a & 0 \\ 0 & 1 & 0 & 0 \\ a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11-28)$$

With transformation matrices 11-17, 11-23, and 11-28, we have aligned the rotation axis with the positive z axis. The specified rotation angle θ can now be applied as a rotation about the z axis:

Section 11-2

Rotation

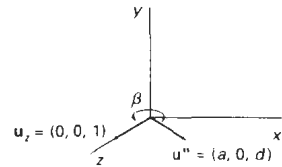


Figure 11-14

Rotation of unit vector \mathbf{u}'' (vector \mathbf{u} after rotation into the xz plane) about the y axis. Positive rotation angle β aligns \mathbf{u}'' with vector \mathbf{u}_z .

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11-29)$$

To complete the required rotation about the given axis, we need to transform the rotation axis back to its original position. This is done by applying the inverse of transformations 11-17, 11-23, and 11-28. The transformation matrix for rotation about an arbitrary axis then can be expressed as the composition of these seven individual transformations:

$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{R}_x^{-1}(\alpha) \cdot \mathbf{R}_y^{-1}(\beta) \cdot \mathbf{R}_z(\theta) \cdot \mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha) \cdot \mathbf{T} \quad (11-30)$$

A somewhat quicker, but perhaps less intuitive, method for obtaining the composite rotation matrix $\mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha)$ is to take advantage of the form of the composite matrix for any sequence of three-dimensional rotations:

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11-31)$$

The upper left 3 by 3 submatrix of this matrix is orthogonal. This means that the rows (or the columns) of this submatrix form a set of orthogonal unit vectors that are rotated by matrix \mathbf{R} onto the x , y , and z axes, respectively:

$$\mathbf{R} \cdot \begin{bmatrix} r_{11} \\ r_{12} \\ r_{13} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{R} \cdot \begin{bmatrix} r_{21} \\ r_{22} \\ r_{23} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{R} \cdot \begin{bmatrix} r_{31} \\ r_{32} \\ r_{33} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad (11-32)$$

Therefore, we can consider a local coordinate system defined by the rotation axis and simply form a matrix whose columns are the local unit coordinate vectors. Assuming that the rotation axis is not parallel to any coordinate axis, we can form the following local set of unit vectors (Fig. 11-15):

$$\begin{aligned} \mathbf{u}'_z &= \mathbf{u} \\ \mathbf{u}'_y &= \frac{\mathbf{u} \times \mathbf{u}_x}{|\mathbf{u} \times \mathbf{u}_x|} \\ \mathbf{u}'_x &= \mathbf{u}'_y \times \mathbf{u}'_z \end{aligned} \quad (11-33)$$

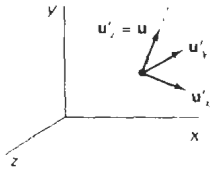


Figure 11-15

Local coordinate system for a rotation axis defined by unit vector \mathbf{u} .

If we express the elements of the local unit vectors for the rotation axis as

$$\begin{aligned} \mathbf{u}'_x &= (u'_{x1}, u'_{x2}, u'_{x3}) \\ \mathbf{u}'_y &= (u'_{y1}, u'_{y2}, u'_{y3}) \\ \mathbf{u}'_z &= (u'_{z1}, u'_{z2}, u'_{z3}) \end{aligned} \quad (11-34)$$

then the required composite matrix, equal to the product $\mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha)$, is

$$\mathbf{R} = \begin{bmatrix} u'_{x1} & u'_{x2} & u'_{x3} & 0 \\ u'_{y1} & u'_{y2} & u'_{y3} & 0 \\ u'_{z1} & u'_{z2} & u'_{z3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11-35)$$

This matrix transforms the unit vectors \mathbf{u}'_x , \mathbf{u}'_y , and \mathbf{u}'_z onto the x , y , and z axes, respectively. Thus, the rotation axis is aligned with the z axis, since $\mathbf{u}'_z = \mathbf{u}$.

Rotations with Quaternions

A more efficient method for obtaining rotation about a specified axis is to use a quaternion representation for the rotation transformation. In Chapter 10, we discussed the usefulness of quaternions for generating three-dimensional fractals using self-squaring procedures. Quaternions are useful also in a number of other computer graphics procedures, including three-dimensional rotation calculations. They require less storage space than 4-by-4 matrices, and it is simpler to write quaternion procedures for transformation sequences. This is particularly important in animations that require complicated motion sequences and motion interpolations between two given positions of an object.

One way to characterize a quaternion (Appendix A) is as an ordered pair, consisting of a *scalar part* and a *vector part*:

$$q = (s, \mathbf{v})$$

We can also think of a quaternion as a higher-order complex number with one real part (the scalar part) and three complex parts (the elements of vector \mathbf{v}). A rotation about any axis passing through the coordinate origin is performed by first setting up a unit quaternion with the following scalar and vector parts:

$$s = \cos \frac{\theta}{2}, \quad \mathbf{v} = \mathbf{u} \sin \frac{\theta}{2} \quad (11-36)$$

where \mathbf{u} is a unit vector along the selected rotation axis, and θ is the specified rotation angle about this axis (Fig. 11-16). Any point position \mathbf{P} to be rotated by this quaternion can be represented in quaternion notation as

$$\mathbf{P} = (0, \mathbf{p})$$

with the coordinates of the point as the vector part $\mathbf{p} = (x, y, z)$. The rotation of the point is then carried out with the quaternion operation

$$\mathbf{P}' = q\mathbf{P}q^{-1} \quad (11-37)$$

where $q^{-1} = (s, -\mathbf{v})$ is the inverse of the unit quaternion q with the scalar and vector parts given in Eqs. 11-36. This transformation produces the new quaternion with scalar part equal to 0:

$$\mathbf{P}' = (0, \mathbf{p}') \quad (11-38)$$

and the vector part is calculated with dot and cross products as

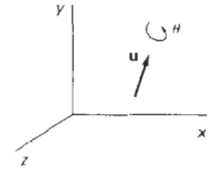


Figure 11-16
Unit quaternion parameters θ and \mathbf{u} for rotation about a specified axis.

$$\mathbf{p}' = s\mathbf{p} + \mathbf{v}(\mathbf{p} \cdot \mathbf{v}) + 2s(\mathbf{v} \times \mathbf{p}) + \mathbf{v} \times (\mathbf{v} \times \mathbf{p}) \quad (11-39)$$

Parameters s and \mathbf{v} have the rotation values given in Eqs. 11-36. Many computer graphics systems use efficient hardware implementations of these vector calculations to perform rapid three-dimensional object rotations.

Transformation 11-37 is equivalent to rotation about an axis that passes through the coordinate origin. This is the same as the sequence of rotation transformations in Eq. 11-30 that aligns the rotation axis with the z axis, rotates about z , and then returns the rotation axis to its original position.

Using the definition for quaternion multiplication given in Appendix A, and designating the components of the vector part of q as $\mathbf{v} = (a, b, c)$, we can evaluate the terms in Eq. 11-39 to obtain the elements for the composite rotation matrix $\mathbf{R}_x^{-1}(\alpha) \cdot \mathbf{R}_y^{-1}(\beta) \cdot \mathbf{R}_z(\theta) \cdot \mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha)$ in a 3 by 3 form as

$$\mathbf{M}_R(\theta) = \begin{bmatrix} 1 - 2b^2 - 2c^2 & 2ab - 2sc & 2ac + 2sb \\ 2ab + 2sc & 1 - 2a^2 - 2c^2 & 2bc - 2sa \\ 2ac - 2sb & 2bc + 2sa & 1 - 2a^2 - 2b^2 \end{bmatrix} \quad (11-40)$$

To obtain the complete general rotation equation 11-30, we need to include the translations that move the rotation axis to the coordinate axis and back to its original position. That is,

$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{M}_R \cdot \mathbf{T} \quad (11-41)$$

As an example, we can perform a rotation about the z axis by setting the unit quaternion parameters as

$$s = \cos \frac{\theta}{2}, \quad \mathbf{v} = (0, 0, 1) \sin \frac{\theta}{2}$$

where the quaternion vector elements are $a = b = 0$ and $c = \sin(\theta/2)$. Substituting these values into matrix 11-40, and using the following trigonometric identities

$$\cos^2 \frac{\theta}{2} - \sin^2 \frac{\theta}{2} = 1 - 2\sin^2 \frac{\theta}{2} = \cos \theta, \quad 2\cos \frac{\theta}{2} \sin \frac{\theta}{2} = \sin \theta$$

we get the 3 by 3 version of the z -axis rotation matrix $\mathbf{R}_z(\theta)$ in transformation equation 11-5. Similarly, substituting the unit quaternion rotation values into the transformation equation 11-37 produces the rotated coordinate values in Eqs. 11-4.

11-3

SCALING

The matrix expression for the scaling transformation of a position $\mathbf{P} = (x, y, z)$ relative to the coordinate origin can be written as

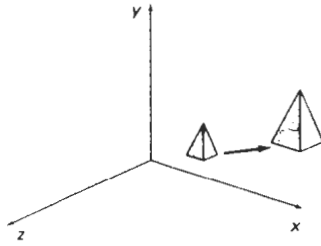


Figure 11-17
Doubling the size of an object with transformation 11-42 also moves the object farther from the origin.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (11-42)$$

or

$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P} \quad (11-43)$$

where scaling parameters s_x , s_y , and s_z are assigned any positive values. Explicit expressions for the coordinate transformations for scaling relative to the origin are

$$x' = x \cdot s_x, \quad y' = y \cdot s_y, \quad z' = z \cdot s_z \quad (11-44)$$

Scaling an object with transformation 11-42 changes the size of the object and repositions the object relative to the coordinate origin. Also, if the transformation parameters are not all equal, relative dimensions in the object are changed. We preserve the original shape of an object with a uniform scaling ($s_x = s_y = s_z$). The result of scaling an object uniformly with each scaling parameter set to 2 is shown in Fig. 11-17.

Scaling with respect to a selected fixed position (x_f, y_f, z_f) can be represented with the following transformation sequence:

1. Translate the fixed point to the origin.
2. Scale the object relative to the coordinate origin using Eq. 11-42.
3. Translate the fixed point back to its original position.

This sequence of transformations is demonstrated in Fig. 11-18. The matrix representation for an arbitrary fixed-point scaling can then be expressed as the concatenation of these translate-scale-translate transformations as

$$\mathbf{T}(x_f, y_f, z_f) \cdot \mathbf{S}(s_x, s_y, s_z) \cdot \mathbf{T}(-x_f, -y_f, -z_f) = \begin{bmatrix} s_x & 0 & 0 & (1-s_x)x_f \\ 0 & s_y & 0 & (1-s_y)y_f \\ 0 & 0 & s_z & (1-s_z)z_f \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11-45)$$

We form the inverse scaling matrix for either Eq. 11-42 or Eq. 11-45 by replacing the scaling parameters s_x , s_y , and s_z with their reciprocals. The inverse ma-

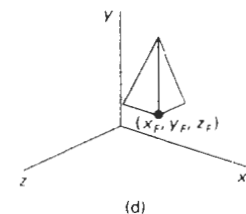
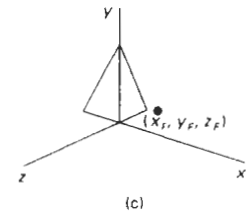
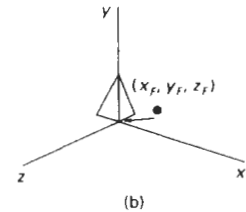
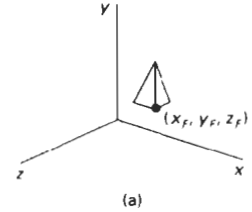


Figure 11-18
Scaling an object relative to a selected fixed point is equivalent to the sequence of transformations shown.

trix generates an opposite scaling transformation, so the concatenation of any scaling matrix and its inverse produces the identity matrix.

11-4 OTHER TRANSFORMATIONS

In addition to translation, rotation, and scaling, there are various additional transformations that are often useful in three-dimensional graphics applications. Two of these are reflection and shear.

Reflections

A three-dimensional reflection can be performed relative to a selected *reflection axis* or with respect to a selected *reflection plane*. In general, three-dimensional reflection matrices are set up similarly to those for two dimensions. Reflections relative to a given axis are equivalent to 180° rotations about that axis. Reflections with respect to a plane are equivalent to 180° rotations in four-dimensional space. When the reflection plane is a coordinate plane (either xy , xz , or yz), we can think of the transformation as a conversion between left-handed and right-handed systems.

An example of a reflection that converts coordinate specifications from a right-handed system to a left-handed system (or vice versa) is shown in Fig. 11-19. This transformation changes the sign of the z coordinates, leaving the x - and y -coordinate values unchanged. The matrix representation for this reflection of points relative to the xy plane is

$$RF_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11-46)$$

Transformation matrices for inverting x and y values are defined similarly, as reflections relative to the yz plane and xz plane, respectively. Reflections about other planes can be obtained as a combination of rotations and coordinate-plane reflections.

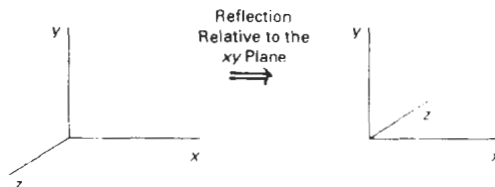


Figure 11-19

Conversion of coordinate specifications from a right-handed to a left-handed system can be carried out with the reflection transformation 11-46.

Shears

Shearing transformations can be used to modify object shapes. They are also useful in three-dimensional viewing for obtaining general projection transformations. In two dimensions, we discussed transformations relative to the x or y axes to produce distortions in the shapes of objects. In three dimensions, we can also generate shears relative to the z axis.

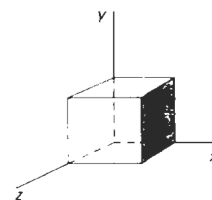
As an example of three-dimensional shearing, the following transformation produces a z -axis shear:

$$SH_z = \begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11-47)$$

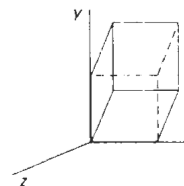
Parameters a and b can be assigned any real values. The effect of this transformation matrix is to alter x - and y -coordinate values by an amount that is proportional to the z value, while leaving the z coordinate unchanged. Boundaries of planes that are perpendicular to the z axis are thus shifted by an amount proportional to z . An example of the effect of this shearing matrix on a unit cube is shown in Fig. 11-20, for shearing values $a = b = 1$. Shearing matrices for the x axis and y axis are defined similarly.

Section 11-5

Composite Transformations



(a)



(b)

11-5

COMPOSITE TRANSFORMATIONS

As with two-dimensional transformations, we form a composite three-dimensional transformation by multiplying the matrix representations for the individual operations in the transformation sequence. This concatenation is carried out from right to left, where the rightmost matrix is the first transformation to be applied to an object and the leftmost matrix is the last transformation. The following program provides an example for implementing a composite transformation. A sequence of basic, three-dimensional geometric transformations are combined to produce a single composite transformation, which is then applied to the coordinate definition of an object.

Figure 11-20

A unit cube (a) is sheared (b) by transformation matrix 11-47, with $a = b = 1$.

```
#include <math.h>
#include "graphics.h"

#define PI 3.14159

typedef float Matrix4x4[4][4];

Matrix4x4 theMatrix;

void matrix4x4SetIdentity (Matrix4x4 m)
{
    int i,c;

    for (i=0; i<4; i++)
        for (c=0; c<4; c++)
```

```

        m[r][c] = (r == c);
    }

/* Multiplies matrix a times b, putting result in b */
void matrix4x4PreMultiply (Matrix4x4 a, Matrix4x4 b)
{
    int r,c;
    Matrix4x4 tmp;

    for (r=0; r<4; r++)
        for (c=0; c<4; c++)
            tmp[r][c] = a[r][0]*b[0][c] + a[r][1]*b[1][c] +
                a[r][2]*b[2][c] + a[r][3]*b[3][c];
    for (r=0; r<4; r++)
        for (c=0; c<4; c++)
            b[r][c] = tmp[r][c];
}

void translate3 (float tx, float ty, float tz)
{
    Matrix4x4 m;

    matrix4x4SetIdentity (m);
    m[0][3] = tx; m[1][3] = ty; m[2][3] = tz;
    matrix4x4PreMultiply (m, theMatrix);
}

void scale3 (float sx, float sy, float sz, wcPt3 center)
{
    Matrix4x4 m;

    matrix4x4SetIdentity (m);
    m[0][0] = sx;
    m[0][3] = (1 - sx) * center.x;
    m[1][1] = sy;
    m[1][3] = (1 - sy) * center.y;
    m[2][2] = sz;
    m[2][3] = (1 - sz) * center.z;
    matrix4x4PreMultiply (m, theMatrix);
}

void rotate3 (wcPt3 p1, wcPt3 p2, float radianAngle)
{
    float length = sqrt ((p2.x - p1.x) * (p2.x - p1.x) +
        (p2.y - p1.y) * (p2.y - p1.y) +
        (p2.z - p1.z) * (p2.z - p1.z));

    float cosA2 = cosf (radianAngle / 2.0);
    float sinA2 = sinf (radianAngle / 2.0);
    float a = sinA2 * (p2.x - p1.x) / length;
    float b = sinA2 * (p2.y - p1.y) / length;
    float c = sinA2 * (p2.z - p1.z) / length;
    Matrix4x4 m;

    translate3 (-p1.x, -p1.y, -p1.z);
    matrix4x4SetIdentity (m);
    m[0][0] = 1.0 - 2*b*b - 2*c*c;
    m[0][1] = 2*a*b - 2*cosA2*c;
    m[0][2] = 2*a*c + 2*cosA2*b;
    m[1][0] = 2*a*b + 2*cosA2*c;
    m[1][1] = 1.0 - 2*a*a - 2*c*c;
    m[1][2] = 2*b*c - 2*cosA2*a;
    m[2][0] = 2*a*c - 2*cosA2*b;

```

```

    m[2][1] = 2*b*c + 2*cosA2*a;
    m[2][2] = 1.0 - 2*a*a - 2*b*b;
    matrix4x4PreMultiply (m, theMatrix);
    translate3 (p1.x, p1.y, p1.z);
}

void transformPoints3 (int nPts, wcPt3 * pts)
{
    int k, j;
    float tmp[3];

    for (k=0; k<nPts; k++) {
        for (j=0; j<3; j++)
            tmp[j] = theMatrix[j][0] * pts[k].x + theMatrix[j][1] * pts[k].y +
                    theMatrix[j][2] * pts[k].z + theMatrix[j][3];
        setWcPt3 (&pts[k], tmp[0], tmp[1], tmp[2]);
    }
}

void main (int argc, char ** argv)
{
    wcPt3 pts[5] = { 10,10,0, 100,10,0, 125,50,0, 35.50,0, 10,10,0 };
    wcPt3 p1 = { 10,10,0 }, p2 = { 10,10,10 };
    wcPt3 refPt = { 68.0,30.0,0.0 };

    long windowID = openGraphics (*argv, 200, 200);
    setBackground (WHITE);
    setColor (BLUE);
    pPolyline3 (5, pts);

    matrix4x4SetIdentity (theMatrix);
    rotate3 (p1, p2, PI/4.0);
    scale3 (0.75, 0.75, 1.0, refPt);
    translate3 (25, 40, 0);
    transformPoints3 (5, pts);
    setColor (RED);
    pPolyline3 (5, pts);

    sleep (10);
    closeGraphics (windowID);
}

```

11-6

THREE-DIMENSIONAL TRANSFORMATION FUNCTIONS

We set up matrices for modeling and other transformations with functions similar to those given in Chapter 5 for two-dimensional transformations. The major difference is that we can now specify rotations around any coordinate axis. These functions are

```

translate3 (translateVector, matrixTranslate);
rotateX (thetaX, xMatrixRotate)
rotateY (thetaY, yMatrixRotate)
rotateZ (thetaZ, zMatrixRotate)
scale3 (scaleVector, matrixScale)

```

Each of these functions produces a 4 by 4 transformation matrix that can then be used to transform coordinate positions expressed as homogeneous column vectors. Parameter `translateVector` is a pointer to the list of translation distances t_x , t_y , and t_z . Similarly, parameter `scaleVector` specifies the three scaling parameters s_x , s_y , and s_z . Rotate and scale matrices transform objects with respect to the coordinate origin.

And we can construct composite transformations with the functions

```
composeMatrix3  
buildTransformationMatrix3  
composeTransformationMatrix3
```

which have parameters similar to two-dimensional transformation functions for setting up composite matrices, except we can now specify three rotation angles. The order of the transformation sequence for the `buildTransformationMatrix3` and `composeTransformationMatrix3` functions is the same as in two dimensions: (1) scale, (2) rotate, and (3) translate.

Once we have specified a transformation matrix, we can apply the matrix to specified points with

```
transformPoint3 (inPoint, matrix, outPoint)
```

In addition, we can set the transformations for hierarchical constructions using structures with the function

```
setLocalTransformation3 (matrix, type)
```

where parameter `matrix` specifies the elements of a 4 by 4 transformation matrix, and parameter `type` can be assigned one of the following three values: *pre-concatenate*, *postconcatenate*, or *replace*.

11-7

MODELING AND COORDINATE TRANSFORMATIONS

So far, we have discussed three-dimensional transformations as operations that move objects from one position to another within a single reference frame. There are many times, however, when we are interested in switching coordinates from one system to another. General three-dimensional viewing procedures, for example, involve an initial transformation of world-coordinate descriptions to a viewing-coordinate system. Then viewing coordinates are transformed to device coordinates. And in modeling, objects are often described in a local (modeling) coordinate reference frame, then the objects are repositioned into a world-coordinate scene. For example, tables, chairs, and other furniture, each defined in a local (modeling) coordinate system, can be placed into the description of a room, defined in another reference frame, by transforming the furniture coordinates to room coordinates. Then the room might be transformed into a larger scene, constructed in world coordinates.

An example of the use of multiple coordinate systems and hierarchical modeling with three-dimensional objects is given in Fig. 11-21. This figure illustrates simulation of tractor movement. As the tractor moves, the tractor coordinate system and front-wheel coordinate system move in the world-coordinate

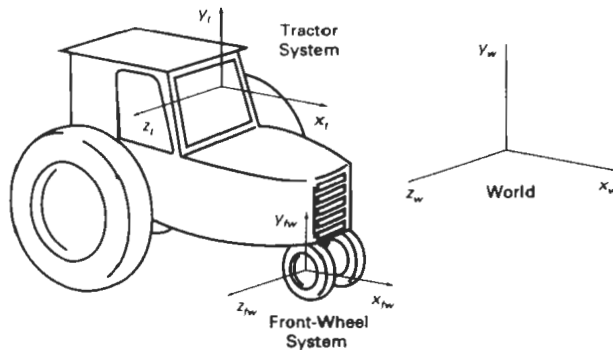


Figure 11-21

Possible coordinate systems used in simulating tractor movement. Wheel rotations are described in the front-wheel system. Turning of the tractor is described by a rotation of the front-wheel system in the tractor system. Both the wheel and tractor reference frames move in the world-coordinate system.

system. The front wheels rotate in the wheel system, and the wheel system rotates in the tractor system when the tractor turns.

Three-dimensional objects and scenes are constructed using structure (or segment) operations similar to those discussed in Chapter 7. Modeling transformation functions can be applied to create hierarchical representation for three-dimensional objects. We can define three-dimensional object shapes in local (modeling) coordinates, then we construct a scene or a hierarchical representation with instances of the individual objects. That is, we transform object descriptions from modeling coordinates to world coordinates or to another system in the hierarchy. An example of a PHIGS structure hierarchy is shown in Fig. 11-22. This display was generated by the PHIGS Toolkit software, developed at the University of

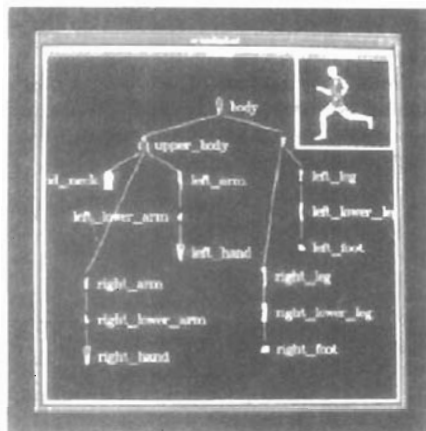


Figure 11-22

Displaying an object hierarchy using the PHIGS Toolkit package developed at the University of Manchester. The displayed object tree is itself a PHIGS structure.

(Courtesy of T. L. J. Howard, J. G. Williams, and W. T. Hewitt, Department of Computer Science, University of Manchester, United Kingdom.)

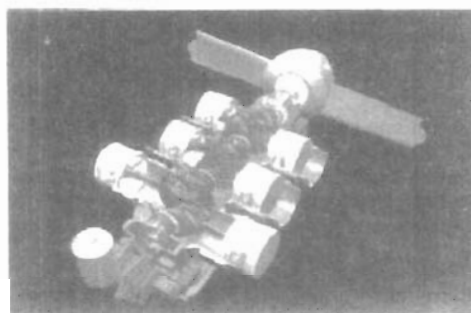
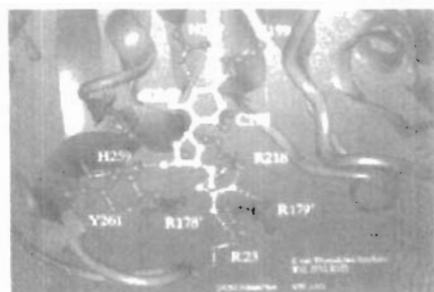


Figure 11-23

Three-dimensional modeling: (a) A ball-and-stick representation for key amino acid residues interacting with the natural substrate of Thymidylate Synthase, modeled and rendered by Julie Newdell, UCSF Computer Graphics Lab. (b) A CAD model showing individual engine components, rendered by Ted Malone, FTI/3D-Magic. (Courtesy of Silicon Graphics, Inc.)

Manchester, to provide an editor, windows, menus, and other interface tools for PHIGS applications. Figure 11-23 shows two example applications of three-dimensional modeling.

Coordinate descriptions of objects are transferred from one system to another with the same procedures used to obtain two-dimensional coordinate transformations. We need to set up the transformation matrix that brings the two coordinate systems into alignment. First, we set up a translation that brings the new coordinate origin to the position of the other coordinate origin. This is followed by a sequence of rotations that corresponding coordinate axes. If different scales are used in the two coordinate systems, a scaling transformation may also be necessary to compensate for the differences in coordinate intervals.

If a second coordinate system is defined with origin (x_0, y_0, z_0) and unit axis vectors as shown in Fig. 11-24, relative to an existing Cartesian reference frame, we first construct the translation matrix $T(-x_0, -y_0, -z_0)$. Next, we can use the unit axis vectors to form the coordinate rotation matrix

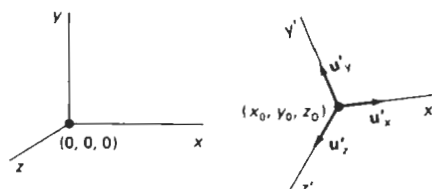


Figure 11-24

Transformation of an object description from one coordinate system to another.

$$\mathbf{R} = \begin{bmatrix} u'_{x1} & u'_{x2} & u'_{x3} & 0 \\ u'_{y1} & u'_{y2} & u'_{y3} & 0 \\ u'_{z1} & u'_{z2} & u'_{z3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11-48) \quad \text{Summary}$$

which transforms unit vectors u'_x , u'_y , and u'_z onto the x , y , and z axes, respectively. The complete coordinate-transformation sequence is then given by the composite matrix $\mathbf{R} \cdot \mathbf{T}$. This matrix correctly transforms coordinate descriptions from one Cartesian system to another even if one system is left-handed and the other is right-handed.

SUMMARY

Three-dimensional transformations useful in computer graphics applications include geometric transformations within a single coordinate system and transformations between different coordinate systems. The basic geometric transformations are translation, rotation, and scaling. Two additional object transformations are reflections and shears. Transformations between different coordinate systems are common elements of modeling and viewing routines. In three dimensions, transformation operations are represented with 4 by 4 matrices. As in two-dimensional graphics methods, a composite transformation in three-dimensions is obtained by concatenating the matrix representations for the individual components of the overall transformation.

Representations for translation and scaling are straightforward extensions of two-dimensional transformation representations. For rotations, however, we need more general representations, since objects can be rotated about any specified axis in space. Any three-dimensional rotation can be represented as a combination of basic rotations around the x , y , and z axes. And many graphics packages provide functions for these three rotations. In general, however, it is more efficient to set up a three-dimensional rotation using either a local rotation-axis reference frame or a quaternion representation. Quaternions are particularly useful for fast generation of repeated rotations that are often required in animation sequences.

Reflections and shears in three dimensions can be carried out relative to any reference axis in space. Thus, these transformations are also more involved than the corresponding transformations in two dimensions. Transforming object descriptions from one coordinate system to another is equivalent to a transformation that brings the two reference frames into coincidence. Finally, object modeling often requires a hierarchical transformation structure that ensures that the individual components of an object move in harmony with the overall structure.

REFERENCES

For additional techniques involving matrices, modeling, and three-dimensional transformations, see Glassner (1990), Arvo (1991), and Kirk (1992). A detailed discussion of quaternion rotations is given in Shoemake (1985). Three-dimensional PHIGS and PHIGS + transformation functions are discussed in Howard et al. (1991), Gaskins (1992), and Blake (1993).

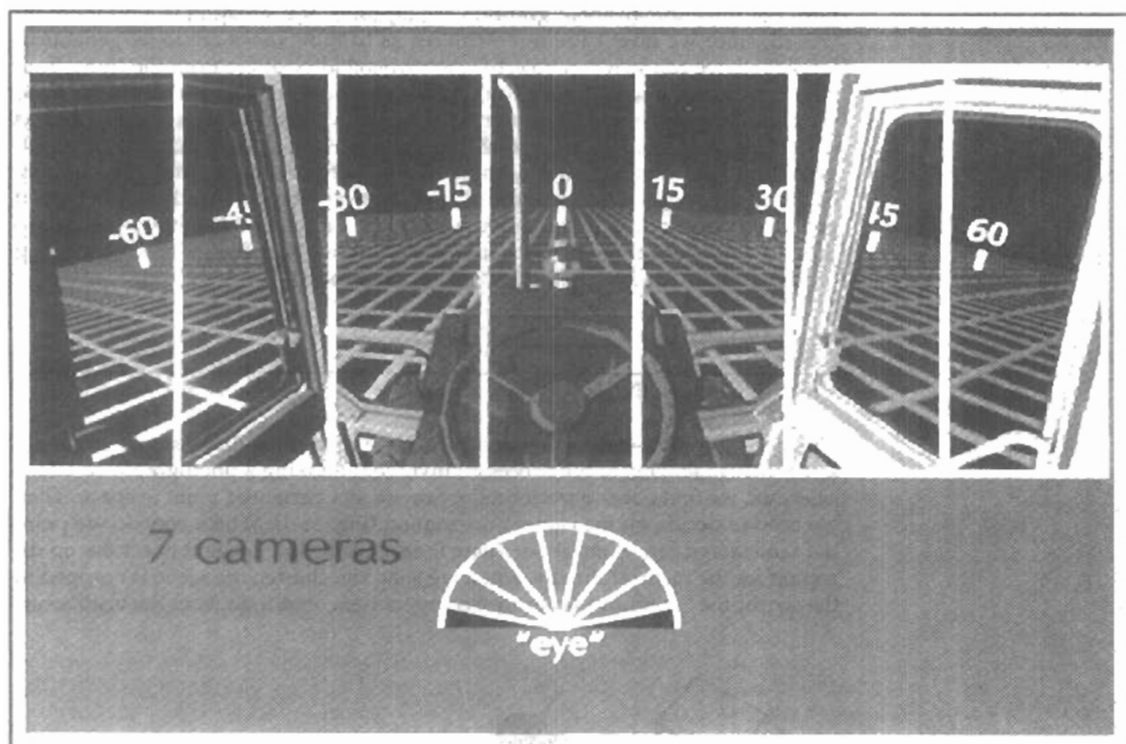
EXERCISES

- 11-1. Prove that the multiplication of three-dimensional transformation matrices for each of the following sequence of operations is commutative:
 - (a) Any two successive translations.
 - (b) Any two successive scaling operations.
 - (c) Any two successive rotations about any one of the coordinate axes.
- 11-2. Using either Eq. 11-30 or Eq. 11-41, prove that any two successive rotations about a given rotation axis is commutative.
- 11-3. By evaluating the terms in Eq. 11-39, derive the elements for general rotation matrix given in Eq. 11-40.
- 11-4. Show that rotation matrix 11-35 is equal to the composite matrix $R_y(\beta) \cdot R_x(\alpha)$.
- 11-5. Prove that the quaternion rotation matrix Eq. 11-40 reduces to the matrix representation in Eq. 11-5 when the rotation axis is the coordinate z axis.
- 11-6. Prove that Eq. 11-41 is equivalent to the general rotation transformation given in Eq. 11-30.
- 11-7. Write a procedure to implement general rotation transformations using the rotation matrix 11-35.
- 11-8. Write a routine to implement quaternion rotations, Eq. 11-41, for any specified axis.
- 11-9. Derive the transformation matrix for scaling an object by a scaling factor s in a direction defined by the direction angles α , β , and γ .
- 11-10. Develop an algorithm for scaling an object defined in an octree representation.
- 11-11. Develop a procedure for animating an object by incrementally rotating it about any specified axis. Use appropriate approximations to the trigonometric equations to speed up the calculations, and reset the object to its initial position after each complete revolution about the axis.
- 11-12. Devise a procedure for rotating an object that is represented in an octree structure.
- 11-13. Develop a routine to reflect an object about an arbitrarily selected plane.
- 11-14. Write a program to shear an object with respect to any of the three coordinate axes, using input values for the shearing parameters.
- 11-15. Develop a procedure for converting an object definition in one coordinate reference to any other coordinate system defined relative to the first system.
- 11-16. Develop a complete algorithm for implementing the procedures for constructive solid modeling by combining three-dimensional primitives to generate new shapes. Initially, the primitives can be combined to form subassemblies, then the subassemblies can be combined with each other and with primitive shapes to form the final assembly. Interactive input of translation and rotation parameters can be used to position the objects. Output of the algorithm is to be the sequence of operations needed to produce the final CSG object.

CHAPTER

12

Three-Dimensional Viewing



In two-dimensional graphics applications, viewing operations transfer positions from the world-coordinate plane to pixel positions in the plane of the output device. Using the rectangular boundaries for the world-coordinate window and the device viewport, a two-dimensional package maps the world scene to device coordinates and clips the scene against the four boundaries of the viewport. For three-dimensional graphics applications, the situation is a bit more involved, since we now have more choices as to how views are to be generated. First of all, we can view an object from any spatial position: from the front, from above, or from the back. Or we could generate a view of what we would see if we were standing in the middle of a group of objects or inside a single object, such as a building. Additionally, three-dimensional descriptions of objects must be projected onto the flat viewing surface of the output device. And the clipping boundaries now enclose a volume of space, whose shape depends on the type of projection we select. In this chapter, we explore the general operations needed to produce views of a three-dimensional scene, and we also discuss specific viewing procedures provided in packages such as PHIGS and GL.

12-1

VIEWING PIPELINE

The steps for computer generation of a view of a three-dimensional scene are somewhat analogous to the processes involved in taking a photograph. To take a snapshot, we first need to position the camera at a particular point in space. Then we need to decide on the camera orientation (Fig. 12-1): Which way do we point the camera and how should we rotate it around the line of sight to set the up direction for the picture? Finally, when we snap the shutter, the scene is cropped to the size of the "window" (aperture) of the camera, and light from the visible sur-

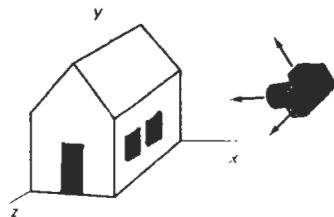


Figure 12-1
Photographing a scene involves
selection of a camera position and
orientation.

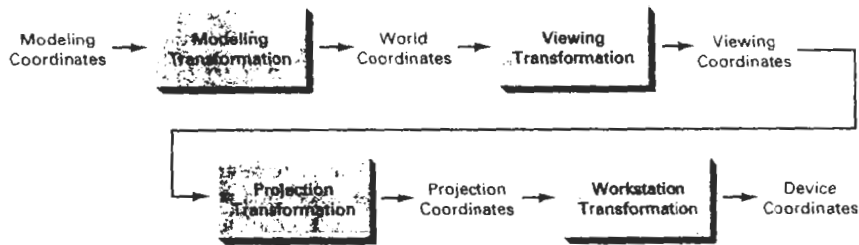


Figure 12-2
General three-dimensional transformation pipeline, from modeling coordinates to final device coordinates.

faces is projected onto the camera film. We need to keep in mind, however, that the camera analogy can be carried only so far, since we have more flexibility and many more options for generating views of a scene with a graphics package than we do with a camera.

Figure 12-2 shows the general processing steps for modeling and converting a world-coordinate description of a scene to device coordinates. Once the scene has been modeled, world-coordinate positions are converted to viewing coordinates. The viewing-coordinate system is used in graphics packages as a reference for specifying the observer viewing position and the position of the projection plane, which we can think of in analogy with the camera film plane. Next, projection operations are performed to convert the viewing-coordinate description of the scene to coordinate positions on the projection plane, which will then be mapped to the output device. Objects outside the specified viewing limits are clipped from further consideration, and the remaining objects are processed through visible-surface identification and surface-rendering procedures to produce the display within the device viewport.

12-2

VIEWING COORDINATES

Generating a view of an object in three dimensions is similar to photographing the object. We can walk around and take its picture from any angle, at various distances, and with varying camera orientations. Whatever appears in the viewfinder is projected onto the flat film surface. The type and size of the camera lens determines which parts of the scene appear in the final picture. These ideas are incorporated into three-dimensional graphics packages so that views of a scene can be generated, given the spatial position, orientation, and aperture size of the "camera".

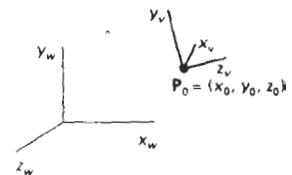


Figure 12-3
A right-handed viewing-coordinate system, with axes x_v , y_v , and z_v , relative to a world-coordinate scene.

Specifying the View Plane

We choose a particular view for a scene by first establishing the **viewing-coordinate system**, also called the **view reference coordinate system**, as shown in Fig. 12-3. A **view plane**, or **projection plane**, is then set up perpendicular to the

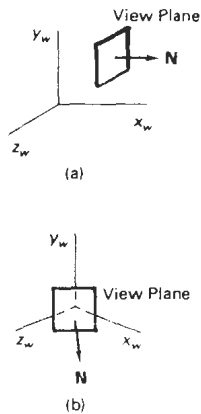


Figure 12-4
Orientations of the view plane for specified normal vector coordinates relative to the world origin. Position (1, 0, 0) orients the view plane as in (a), while (1, 0, 1) gives the orientation in (b).

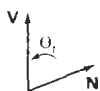


Figure 12-6
Specifying the view-up vector with a twist angle θ_i .

viewing z_v axis. We can think of the view plane as the film plane in a camera that has been positioned and oriented for a particular shot of the scene. World-coordinate positions in the scene are transformed to viewing coordinates, then viewing coordinates are projected onto the view plane.

To establish the viewing-coordinate reference frame, we first pick a world-coordinate position called the **view reference point**. This point is the origin of our viewing-coordinate system. The view reference point is often chosen to be close to or on the surface of some object in a scene. But we could also choose a point that is at the center of an object, or at the center of a group of objects, or somewhere out in front of the scene to be displayed. If we choose a point that is near to or on some object, we can think of this point as the position where we might want to aim a camera to take a picture of the object. Alternatively, if we choose a point that is at some distance from a scene, we could think of this as the camera position.

Next, we select the positive direction for the viewing z_v axis, and the orientation of the view plane, by specifying the **view-plane normal vector**, N . We choose a world-coordinate position, and this point establishes the direction for N relative either to the world origin or to the viewing-coordinate origin. Graphics packages such as GKS and PHIGS, for example, orient N relative to the world-coordinate origin, as shown in Fig. 12-4. The view-plane normal N is then the directed line segment from the world origin to the selected coordinate position. In other words, N is simply specified as a world-coordinate vector. Some other packages (GL from Silicon Graphics, for instance) establish the direction for N using the selected coordinate position as a *look-at point* relative to the view reference point (viewing-coordinate origin). Figure 12-5 illustrates this method for defining the direction of N , which is from the look-at point to the view reference point. Another possibility is to set up a left-handed viewing system and take N and the positive z_v axis from the viewing origin to the look-at point. Only the direction of N is needed to establish the z_v direction; the magnitude is irrelevant, because N will be normalized to a unit vector by the viewing calculations.

Finally, we choose the up direction for the view by specifying a vector V , called the **view-up vector**. This vector is used to establish the positive direction for the y_v axis. Vector V also can be defined as a world-coordinate vector, or in some packages, it is specified with a *twist angle* θ_i about the z_v axis, as shown in Fig. 12-6. For a general orientation of the normal vector, it can be difficult (or at least time consuming) to determine the direction for V that is precisely perpendicular to N . Therefore, viewing procedures typically adjust the user-defined orientation of vector V , as shown in Fig. 12-7, so that V is projected into a plane that is perpendicular to the normal vector. We can choose the view-up vector V to be in any convenient direction, as long as it is not parallel to N . As an example, con-

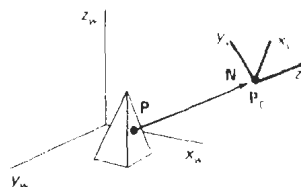


Figure 12-5
Orientation of the view plane for a specified look-at point P , relative to the viewing-coordinate origin P_v .

sider an interactive specification of viewing reference coordinates using PHIGS, where the view reference point is often set at the center of an object to be viewed. If we then want to view the object at the angled direction shown in Fig. 12-8, we can simply choose V as the world vector $(0, 1, 0)$, and this vector will be projected into the plane perpendicular to N to establish the y_v axis. This is much easier than trying to input a vector that is exactly perpendicular to N .

Using vectors N and V , the graphics package can compute a third vector U , perpendicular to both N and V , to define the direction for the x_v axis. Then the direction of V can be adjusted so that it is perpendicular to both N and U to establish the viewing y_v direction. As we will see in the next section (Transformation from World to Viewing Coordinates), these computations are conveniently carried out with unit axis vectors, which are also used to obtain the elements of the world-to-viewing-coordinate transformation matrix. The viewing system is then often described as a uvn system (Fig. 12-9).

Generally, graphics packages allow users to choose the position of the view plane (with some restrictions) along the z_v axis by specifying the *view-plane distance* from the viewing origin. The view plane is always parallel to the $x_v y_v$ plane, and the projection of objects to the view plane correspond to the view of the scene that will be displayed on the output device. Figure 12-10 gives examples of view-plane positioning. If we set the view-plane distance to the value 0, the $x_v y_v$ plane (or uv plane) of viewing coordinates becomes the view plane for the projection transformation. Occasionally, the term " uv plane" is used in reference to the viewing plane, no matter where it is positioned in relation to the $x_v y_v$ plane. But we will only use the term " uv plane" to mean the $x_v y_v$ plane, which is not necessarily the view plane.

Left-handed viewing coordinates are sometimes used in graphics packages so that the viewing direction is in the positive z_v direction. But right-handed viewing systems are more common, because they have the same orientation as the world-reference frame. This allows graphics systems to deal with only one coordinate orientation for both world and viewing references. We will follow the convention of PHIGS and GL and use a right-handed viewing system for all algorithm development.

To obtain a series of views of a scene, we can keep the view reference point fixed and change the direction of N , as shown in Fig. 12-11. This corresponds to generating views as we move around the viewing-coordinate origin. In interac-

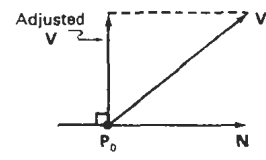


Figure 12-7

Adjusting the input position of the view-up vector V to a position perpendicular to the normal vector N .

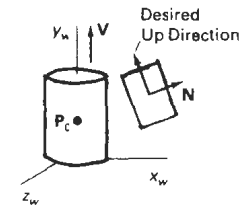


Figure 12-8

Choosing V along the y_w axis sets the up orientation for the view plane in the desired direction.

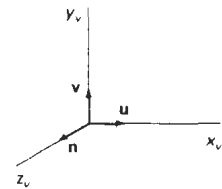


Figure 12-9

A right-handed viewing system defined with unit vectors u , v , and n .

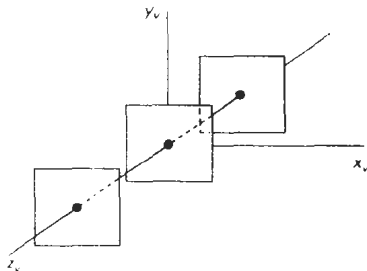


Figure 12-10

View-plane positioning along the z_v axis.

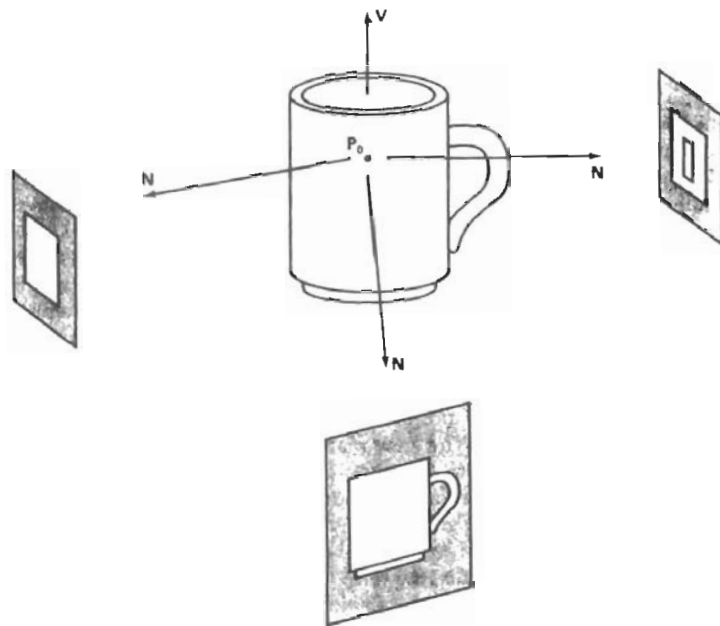


Figure 12-11
Viewing a scene from different directions with a fixed view-reference point.

tive applications, the normal vector N is the viewing parameter that is most often changed. By changing only the direction of N , we can view a scene from any direction except along the line of V . To obtain either of the two possible views along the line of V , we would need to change the direction of V . If we want to simulate camera motion through a scene, we can keep N fixed and move the view reference point around (Fig. 12-12).

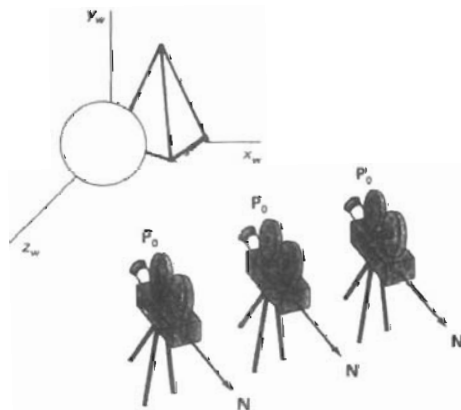


Figure 12-12
Moving around in a scene by changing the position of the view reference point.

Before object descriptions can be projected to the view plane, they must be transferred to viewing coordinates. Conversion of object descriptions from world to viewing coordinates is equivalent to a transformation that superimposes the viewing reference frame onto the world frame using the basic geometric translate-rotate operations discussed in Section 11-7. This transformation sequence is

1. Translate the view reference point to the origin of the world-coordinate system.
2. Apply rotations to align the x_v , y_v , and z_v axes with the world x_w , y_w , and z_w axes, respectively.

If the view reference point is specified at world position (x_0, y_0, z_0) , this point is translated to the world origin with the matrix transformation

$$T = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (12-1)$$

The rotation sequence can require up to three coordinate-axis rotations, depending on the direction we choose for N . In general, if N is not aligned with any world'-coordinate axis, we can superimpose the viewing and world systems with the transformation sequence $R_z \cdot R_y \cdot R_x$. That is, we first rotate around the world x_w axis to bring z_v into the $x_w z_w$ plane. Then, we rotate around the world y_w axis to align the z_w and z_v axes. The final rotation is about the z_w axis to align the y_w and y_v axes. Further, if the view reference system is left-handed, a reflection of one of the viewing axes (for example, the z_v axis) is also necessary. Figure 12-13 illustrates the general sequence of translate-rotate transformations. The composite transformation matrix is then applied to world-coordinate descriptions to transfer them to viewing coordinates.

Another method for generating the rotation-transformation matrix is to calculate unit *uvn* vectors and form the composite rotation matrix directly, as dis-

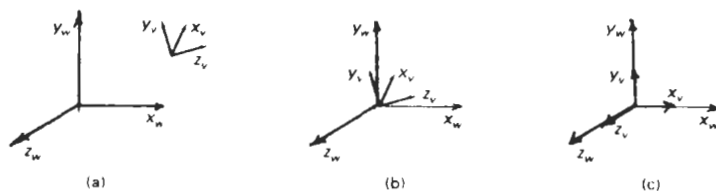


Figure 12-13

Aligning a viewing system with the world-coordinate axes using a sequence of translate-rotate transformations.

cussed in Section 11-7. Given vectors \mathbf{N} and \mathbf{V} , these unit vectors are calculated as

$$\begin{aligned}\mathbf{n} &= \frac{\mathbf{N}}{|\mathbf{N}|} = (n_1, n_2, n_3) \\ \mathbf{u} &= \frac{\mathbf{V} \times \mathbf{N}}{|\mathbf{V} \times \mathbf{N}|} = (u_1, u_2, u_3) \\ \mathbf{v} &= \mathbf{n} \times \mathbf{u} = (v_1, v_2, v_3)\end{aligned}\quad (12-2)$$

This method also automatically adjusts the direction for \mathbf{V} so that \mathbf{v} is perpendicular to \mathbf{n} . The composite rotation matrix for the viewing transformation is then

$$\mathbf{R} = \begin{bmatrix} u_1 & u_2 & u_3 & 0 \\ v_1 & v_2 & v_3 & 0 \\ n_1 & n_2 & n_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\quad (12-3)$$

which transforms \mathbf{u} onto the world x_w axis, \mathbf{v} onto the y_w axis, and \mathbf{n} onto the z_w axis. In addition, this matrix automatically performs the reflection necessary to transform a left-handed viewing system onto the right-handed world system.

The complete world-to-viewing coordinate transformation matrix is obtained as the matrix product

$$\mathbf{M}_{WC,VC} = \mathbf{R} \cdot \mathbf{T}\quad (12-4)$$

This transformation is then applied to coordinate descriptions of objects in the scene to transfer them to the viewing reference frame.

12-3 PROJECTIONS

Once world-coordinate descriptions of the objects in a scene are converted to viewing coordinates, we can project the three-dimensional objects onto the two-dimensional view plane. There are two basic projection methods. In a **parallel projection**, coordinate positions are transformed to the view plane along parallel lines, as shown in the example of Fig. 12-14. For a **perspective projection** (Fig. 12-15), object positions are transformed to the view plane along lines that converge to a point called the **projection reference point** (or **center of projection**). The projected view of an object is determined by calculating the intersection of the projection lines with the view plane.

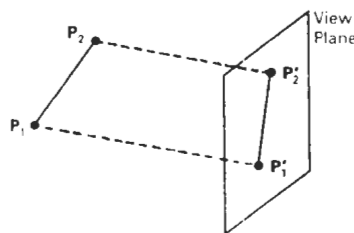


Figure 12-14
Parallel projection of an object to the view plane

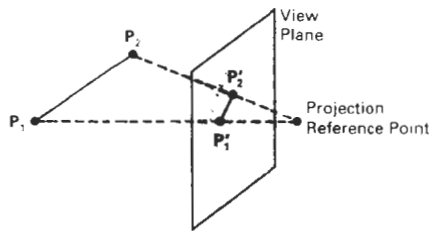


Figure 12-15
Perspective projection of an object
to the view plane.

A parallel projection preserves relative proportions of objects, and this is the method used in drafting to produce scale drawings of three-dimensional objects. Accurate views of the various sides of an object are obtained with a parallel projection, but this does not give us a realistic representation of the appearance of a three-dimensional object. A perspective projection, on the other hand, produces realistic views but does not preserve relative proportions. Projections of distant objects are smaller than the projections of objects of the same size that are closer to the projection plane (Fig. 12-16).

Parallel Projections

We can specify a parallel projection with a **projection vector** that defines the direction for the projection lines. When the projection is perpendicular to the view plane, we have an **orthographic parallel projection**. Otherwise, we have an **oblique parallel projection**. Figure 12-17 illustrates the two types of parallel projections. Some graphics packages, such as GL on Silicon Graphics workstations, do not provide for oblique projections. In this package, for example, a parallel projection is specified by simply giving the boundary edges of a rectangular parallelepiped.

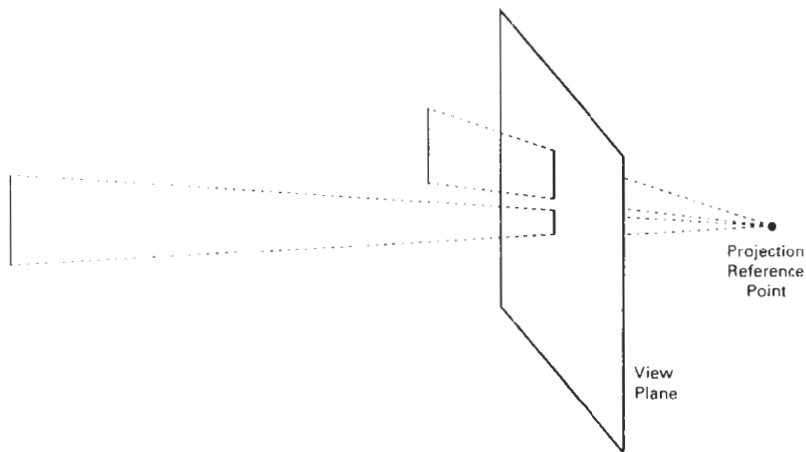


Figure 12-16
Perspective projection of equal-sized objects at different distances from the view plane.



Figure 12-17
Orientation of the projection vector V_p to produce an orthographic projection (a) and an oblique projection (b).

Orthographic projections are most often used to produce the front, side, and top views of an object, as shown in Fig. 12-18. Front, side, and rear orthographic projections of an object are called *elevations*; and a top orthographic projection is called a *plan view*. Engineering and architectural drawings commonly employ these orthographic projections, because lengths and angles are accurately depicted and can be measured from the drawings.

We can also form orthographic projections that display more than one face of an object. Such views are called **axonometric** orthographic projections. The most commonly used axonometric projection is the **isometric** projection. We generate an isometric projection by aligning the projection plane so that it intersects each coordinate axis in which the object is defined (called the *principal axes*) at the same distance from the origin. Figure 12-19 shows an isometric projection for a

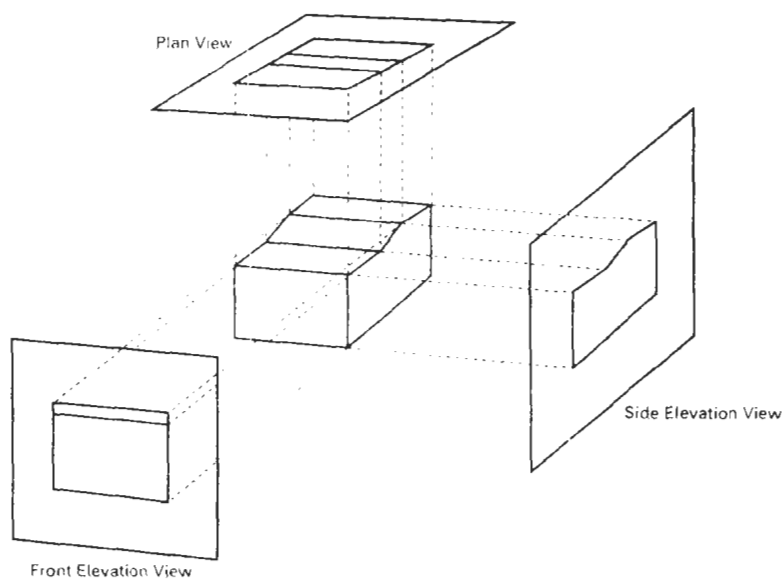


Figure 12-18
Orthographic projections of an object, displaying plan and elevation views.

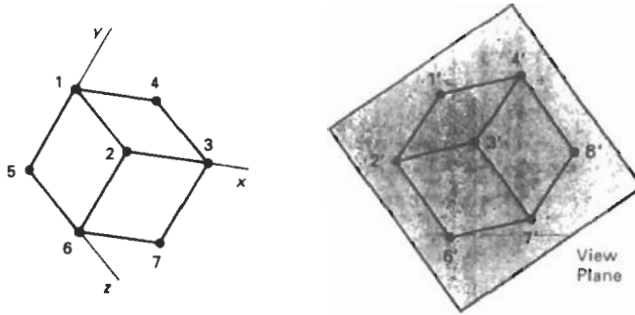


Figure 12-19
Isometric projection for a cube.

cube. The isometric projection is obtained by aligning the projection vector with the cube diagonal. There are eight positions, one in each octant, for obtaining an isometric view. All three principal axes are foreshortened equally in an isometric projection so that relative proportions are maintained. This is not the case in a general axonometric projection, where scaling factors may be different for the three principal directions.

Transformation equations for an orthographic parallel projection are straightforward. If the view plane is placed at position z_{vp} along the z_v axis (Fig. 12-20), then any point (x, y, z) in viewing coordinates is transformed to projection coordinates as

$$x_p = x, \quad y_p = y \quad (12-5)$$

where the original z -coordinate value is preserved for the depth information needed in depth cueing and visible-surface determination procedures.

An oblique projection is obtained by projecting points along parallel lines that are not perpendicular to the projection plane. In some applications packages, an oblique projection vector is specified with two angles, α and ϕ , as shown in Fig. 12-21. Point (x, y, z) is projected to position (x_p, y_p) on the view plane. Orthographic projection coordinates on the plane are (x, y) . The oblique projection line from (x, y, z) to (x_p, y_p) makes an angle α with the line on the projection plane that joins (x_p, y_p) and (x, y) . This line, of length L , is at an angle ϕ with the horizontal direction in the projection plane. We can express the projection coordinates in terms of x, y, L , and ϕ as

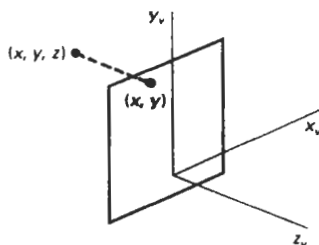


Figure 12-20
Orthographic projection of a point
onto a viewing plane.

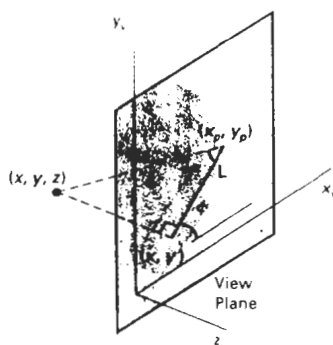


Figure 12-21
Oblique projection of coordinate position (x, y, z) to position (x_p, y_p) on the view plane.

$$\begin{aligned}x_p &= x + L \cos \phi \\y_p &= y + L \sin \phi\end{aligned}\quad (12-6)$$

Length L depends on the angle α and the z coordinate of the point to be projected:

$$\tan \alpha = \frac{z}{L} \quad (12-7)$$

Thus,

$$\begin{aligned}L &= \frac{z}{\tan \alpha} \\&= zL_1\end{aligned}\quad (12-8)$$

where L_1 is the inverse of $\tan \alpha$, which is also the value of L when $z = 1$. We can then write the oblique projection equations 12-6 as

$$\begin{aligned}x_p &= x + z(L_1 \cos \phi) \\y_p &= y + z(L_1 \sin \phi)\end{aligned}\quad (12-9)$$

The transformation matrix for producing any parallel projection onto the $x_v y_v$ plane can be written as

$$\mathbf{M}_{\text{parallel}} = \begin{bmatrix} 1 & 0 & L_1 \cos \phi & 0 \\ 0 & 1 & L_1 \sin \phi & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (12-10)$$

An orthographic projection is obtained when $L_1 = 0$ (which occurs at a projection angle α of 90°). Oblique projections are generated with nonzero values for L_1 . Projection matrix 12-10 has a structure similar to that of a z -axis shear matrix. In fact, the effect of this projection matrix is to shear planes of constant z and project them onto the view plane. The x - and y -coordinate values within each plane of constant z are shifted by an amount proportional to the z value of the plane so that angles, distances, and parallel lines in the plane are projected accurately. This

effect is shown in Fig. 12-22, where the back plane of the box is sheared and overlapped with the front plane in the projection to the viewing surface. An edge of the box connecting the front and back planes is projected into a line of length L_1 that makes an angle ϕ with a horizontal line in the projection plane.

Common choices for angle ϕ are 30° and 45° , which display a combination view of the front, side, and top (or front, side, and bottom) of an object. Two commonly used values for α are those for which $\tan \alpha = 1$ and $\tan \alpha = 2$. For the first case, $\alpha = 45^\circ$ and the views obtained are called **cavalier** projections. All lines perpendicular to the projection plane are projected with no change in length. Examples of cavalier projections for a cube are given in Fig. 12-23.

When the projection angle α is chosen so that $\tan \alpha = 2$, the resulting view is called a **cabinet** projection. For this angle ($\approx 63.4^\circ$), lines perpendicular to the viewing surface are projected at one-half their length. Cabinet projections appear more realistic than cavalier projections because of this reduction in the length of perpendiculars. Figure 12-24 shows examples of cabinet projections for a cube.

Perspective Projections

To obtain a perspective projection of a three-dimensional object, we transform points along projection lines that meet at the projection reference point. Suppose we set the projection reference point at position z_{prp} along the z_v axis, and we

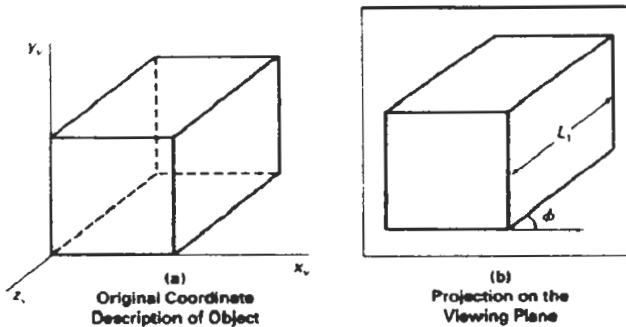


Figure 12-22
Oblique projection of a box onto the $z_v = 0$ plane.

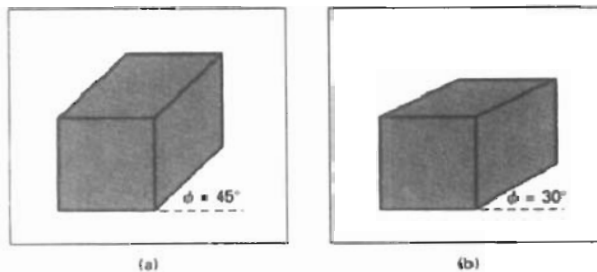


Figure 12-23
Cavalier projections of a cube onto a view plane for two values of angle ϕ .
Note: Depth of the cube is projected equal to the width and height.

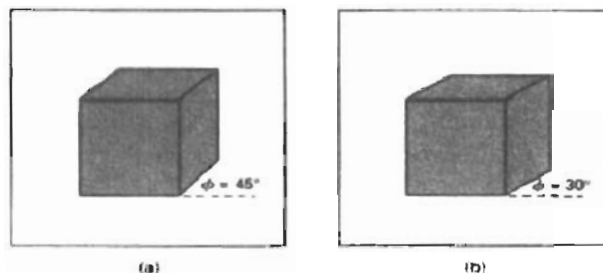


Figure 12-24
Cabinet projections of a cube onto a view plane for two values of angle ϕ . Depth is projected as one-half that of the width and height.

place the view plane at z_{vp} , as shown in Fig. 12-25. We can write equations describing coordinate positions along this perspective projection line in parametric form as

$$\begin{aligned}x' &= x - xu \\y' &= y - yu \\z' &= z - (z - z_{prp})u\end{aligned}\tag{12-11}$$

Parameter u takes values from 0 to 1, and coordinate position (x', y', z') represents any point along the projection line. When $u = 0$, we are at position $P = (x, y, z)$. At the other end of the line, $u = 1$ and we have the projection reference point coordinates $(0, 0, z_{prp})$. On the view plane, $z' = z_{vp}$ and we can solve the z' equation for parameter u at this position along the projection line:

$$u = \frac{z_{vp} - z}{z_{prp} - z}\tag{12-12}$$

Substituting this value of u into the equations for x' and y' , we obtain the perspective transformation equations

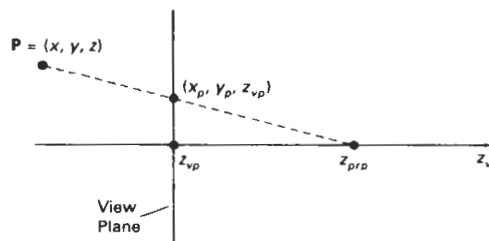


Figure 12-25
Perspective projection of a point P with coordinates (x, y, z) to position (x_p, y_p, z_{vp}) on the view plane.

$$\begin{aligned}x_p &= x \left(\frac{z_{prp} - z_{vp}}{z_{prp} - z} \right) = x \left(\frac{d_p}{z_{prp} - z} \right) \\y_p &= y \left(\frac{z_{prp} - z_{vp}}{z_{prp} - z} \right) = y \left(\frac{d_p}{z_{prp} - z} \right)\end{aligned}\quad (12-13)$$

where $d_p = z_{prp} - z_{vp}$ is the distance of the view plane from the projection reference point.

Using a three-dimensional homogeneous-coordinate representation, we can write the perspective-projection transformation 12-13 in matrix form as

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -z_{vp}/d_p & z_{vp}(z_{prp}/d_p) \\ 0 & 0 & -1/d_p & z_{prp}/d_p \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (12-14)$$

In this representation, the homogeneous factor is

$$h = \frac{z_{prp} - z}{d_p} \quad (12-15)$$

and the projection coordinates on the view plane are calculated from the homogeneous coordinates as

$$x_p = x_h/h, \quad y_p = y_h/h \quad (12-16)$$

where the original z -coordinate value would be retained in projection coordinates for visible-surface and other depth processing.

In general, the projection reference point does not have to be along the z_v axis. We can select any coordinate position $(x_{prp}, y_{prp}, z_{prp})$ on either side of the view plane for the projection reference point, and we discuss this generalization in the next section.

There are a number of special cases for the perspective transformation equations 12-13. If the view plane is taken to be the uv plane, then $z_{vp} = 0$ and the projection coordinates are

$$\begin{aligned}x_p &= x \left(\frac{z_{prp}}{z_{prp} - z} \right) = x \left(\frac{1}{1 - z/z_{prp}} \right) \\y_p &= y \left(\frac{z_{prp}}{z_{prp} - z} \right) = y \left(\frac{1}{1 - z/z_{prp}} \right)\end{aligned}\quad (12-17)$$

And, in some graphics packages, the projection reference point is always taken to be at the viewing-coordinate origin. In this case, $z_{prp} = 0$ and the projection coordinates on the viewing plane are

$$\begin{aligned}x_p &= x \left(\frac{z_{vp}}{z} \right) = x \left(\frac{1}{z/z_{vp}} \right) \\y_p &= y \left(\frac{z_{vp}}{z} \right) = y \left(\frac{1}{z/z_{vp}} \right)\end{aligned}\quad (12-18)$$

When a three-dimensional object is projected onto a view plane using perspective transformation equations, any set of parallel lines in the object that are not parallel to the plane are projected into converging lines. Parallel lines that are parallel to the view plane will be projected as parallel lines. The point at which a set of projected parallel lines appears to converge is called a **vanishing point**. Each such set of projected parallel lines will have a separate vanishing point; and in general, a scene can have any number of vanishing points, depending on how many sets of parallel lines there are in the scene.

The vanishing point for any set of lines that are parallel to one of the principal axes of an object is referred to as a **principal vanishing point**. We control the number of principal vanishing points (one, two, or three) with the orientation of the projection plane, and perspective projections are accordingly classified as one-point, two-point, or three-point projections. The number of principal vanishing points in a projection is determined by the number of principal axes intersecting the view plane. Figure 12-26 illustrates the appearance of one-point and two-point perspective projections for a cube. In Fig. 12-26(b), the view plane is aligned parallel to the xy object plane so that only the object z axis is intersected.

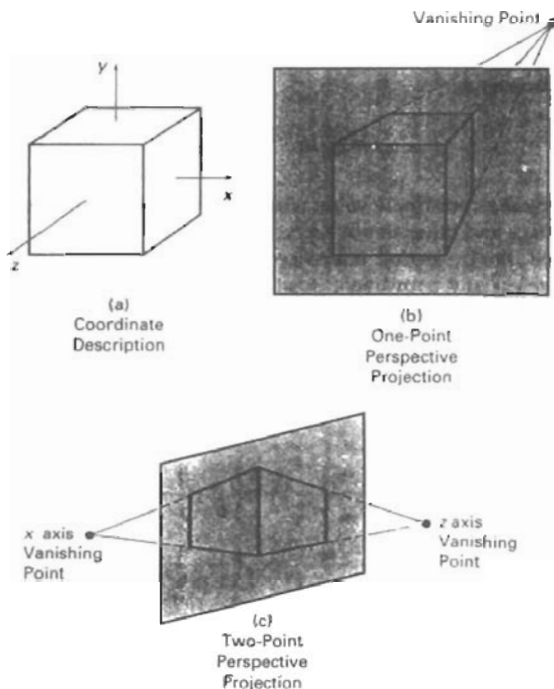


Figure 12-26

Perspective views and principal vanishing points of a cube for various orientations of the view plane relative to the principal axes of the object.

This orientation produces a one-point perspective projection with a z -axis vanishing point. For the view shown in Fig. 12-26(c), the projection plane intersects both the x and z axes but not the y axis. The resulting two-point perspective projection contains both x -axis and z -axis vanishing points.

12-4

VIEW VOLUMES AND GENERAL PROJECTION TRANSFORMATIONS

In the camera analogy, the type of lens used on the camera is one factor that determines how much of the scene is caught on film. A wide-angle lens takes in more of the scene than a regular lens. In three-dimensional viewing, a rectangular **view window**, or **projection window**, in the view plane is used to the same effect. Edges of the view window are parallel to the x_v, y_v axes, and the window boundary positions are specified in viewing coordinates, as shown in Fig. 12-27. The view window can be placed anywhere on the view plane.

Given the specification of the view window, we can set up a **view volume** using the window boundaries. Only those objects within the view volume will appear in the generated display on an output device; all others are clipped from the display. The size of the view volume depends on the size of the window, while the shape of the view volume depends on the type of projection to be used to generate the display. In any case, four sides of the volume are planes that pass through the edges of the window. For a parallel projection, these four sides of the view volume form an infinite parallelepiped, as in Fig. 12-28. For a perspective projection, the view volume is a pyramid with apex at the projection reference point (Fig. 12-29).

A finite view volume is obtained by limiting the extent of the volume in the z_v direction. This is done by specifying positions for one or two additional boundary planes. These z_v -boundary planes are referred to as the **front plane** and **back plane**, or the **near plane** and the **far plane**, of the viewing volume. The front and back planes are parallel to the view plane at specified positions z_{front} and z_{back} . Both planes must be on the same side of the projection reference point, and the back plane must be farther from the projection point than the front plane. Including the front and back planes produces a view volume bounded by six planes, as shown in Fig. 12-30. With an orthographic parallel projection, the six planes form a rectangular parallelepiped, while an oblique parallel projection produces an oblique parallelepiped view volume. With a perspective projection, the front and back clipping planes truncate the infinite pyramidal view volume to form a **frustum**.

Front and back clipping planes allow us to eliminate parts of the scene from the viewing operations based on depth. We can then pick out parts of a scene that we would like to view and exclude objects that are in front of or behind the part that we want to look at. Also, in a perspective projection, we can use the front clipping plane to take out large objects close to the view plane that can project into unrecognizable sections within the view window. Similarly, the back clipping plane can be used to cut out objects far from the projection reference point that can project to small blots on the output device.

Relative placement of the view plane and the front and back clipping planes depends on the type of view we want to generate and the limitations of a particular graphics package. With PHIGS, the view plane can be positioned anywhere along the z_v axis except that it cannot contain the projection reference point. And

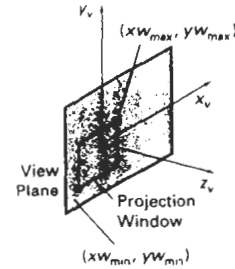


Figure 12-27

Window specification on the view plane, with minimum and maximum coordinates given in the viewing reference system.

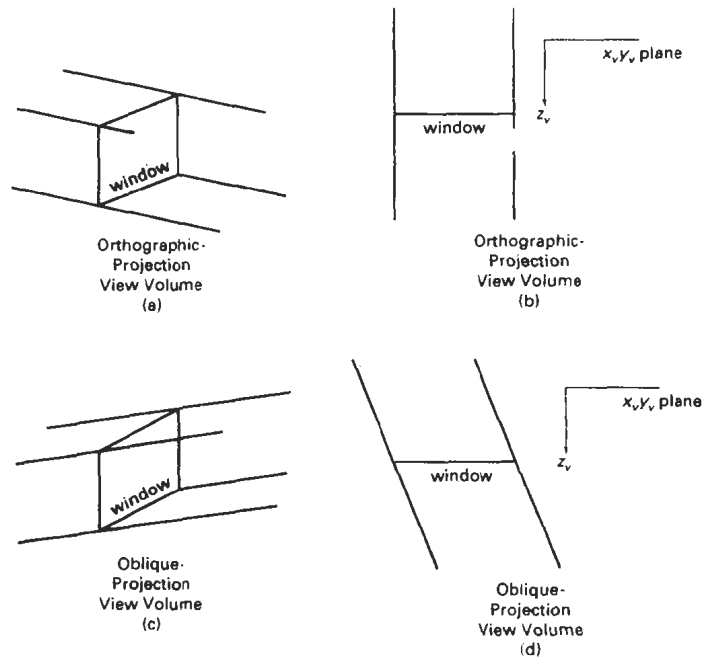


Figure 12-28

View volume for a parallel projection. In (a) and (b), the side and top views of the view volume for an orthographic projection are shown; and in (c) and (d), the side and top views of an oblique view volume are shown.

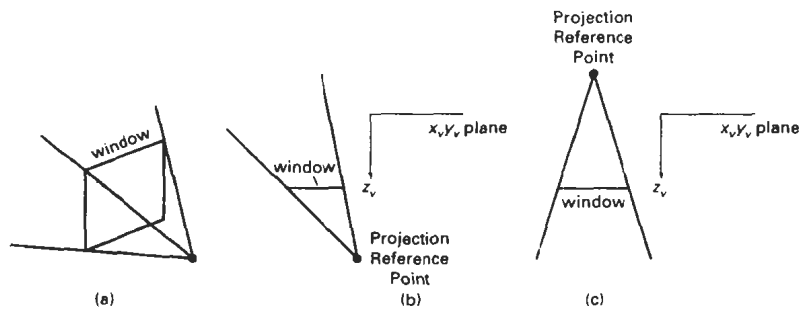


Figure 12-29

Examples of a perspective-projection view volume for various positions of the projection reference point.

Section 12-4

View Volumes and General Projection Transformations

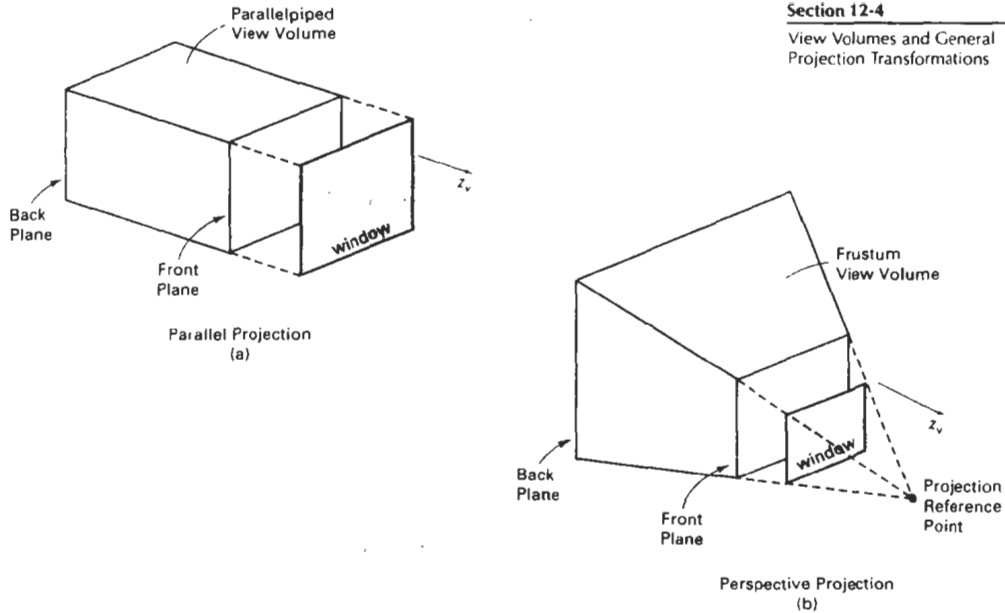


Figure 12-30

View volumes bounded by front and back planes, and by top, bottom, and side planes. Front and back planes are parallel to the view plane at positions z_{front} and z_{back} along the z_v axis.

the front and back planes can be in any position relative to the view plane as long as the projection reference point is not between the front and back planes. Figure 12-31 illustrates possible arrangements of the front and back planes in relation to the view plane. The default view volume in PHIGS is formed as a unit cube using a parallel projection with $z_{\text{front}} = 1$, $z_{\text{back}} = 0$, the view plane coincident with the back plane, and the projection reference point at position (0.5, 0.5, 1.0) on the front plane.

Orthographic parallel projections are not affected by view-plane positioning, because the projection lines are perpendicular to the view plane regardless of

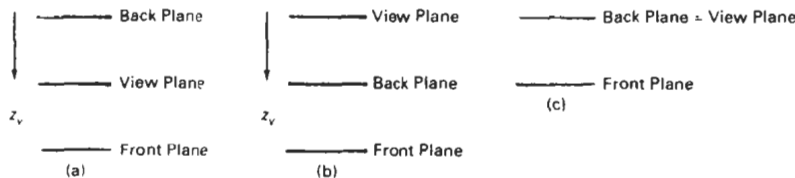


Figure 12-31

Possible arrangements of the front and back clipping planes relative to the view plane.

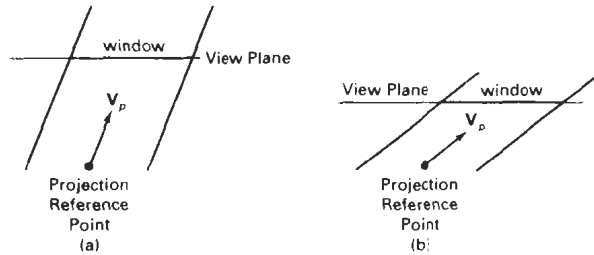


Figure 12-32
Changing the shape of the oblique-projection view volume by moving the window position, when the projection vector V_P is determined by the projection reference point and the window position.

its location. Oblique projections may be affected by view-plane positioning, depending on how the projection direction is to be specified. In PHIGS, the oblique projection direction is parallel to the line from the projection reference point to the center of the window. Therefore, moving the position of the view plane without moving the projection reference point changes the skewness of the sides of the view volume, as shown in Fig. 12-32. Often, the view plane is positioned at the view reference point or on the front clipping plane when generating a parallel projection.

Perspective effects depend on the positioning of the projection reference point relative to the view plane, as shown in Figure 12-33. If we place the projec-

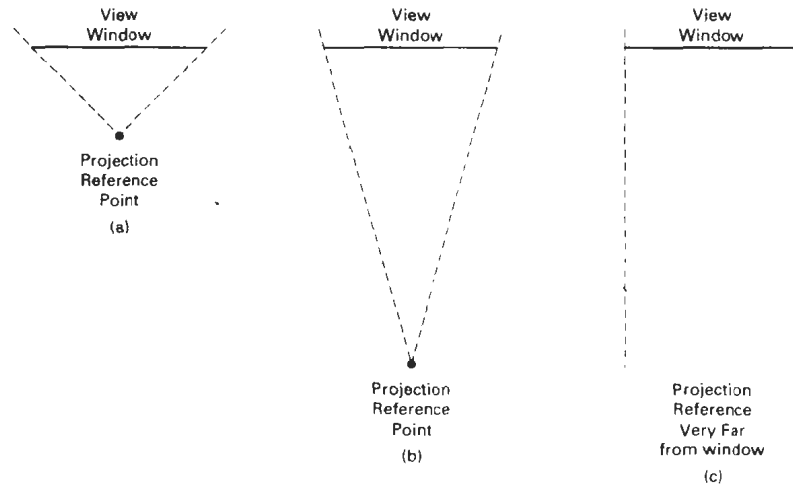


Figure 12-33
Changing perspective effects by moving the projection reference point away from the view plane

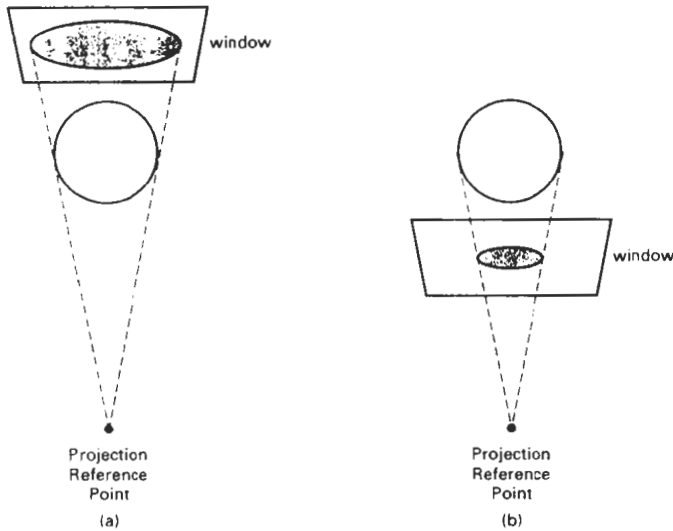


Figure 12-34

Projected object size depends on whether the view plane is positioned in front of the object or behind it, relative to the position of the projection reference point.

tion reference point close to the view plane, perspective effects are emphasized; that is, closer objects will appear much larger than more distant objects of the same size. Similarly, as we move the projection reference point farther from the view plane, the difference in the size of near and far objects decreases. In the limit, as we move the projection reference point infinitely far from the view plane, a perspective projection approaches a parallel projection.

The projected size of an object in a perspective view is also affected by the relative position of the object and the view plane (Fig. 12-34). If the view plane is in front of the object (nearer the projection reference point), the projected size is smaller. Conversely, object size is increased when we project onto a view plane in back of the object.

View-plane positioning for a perspective projection also depends on whether we want to generate a static view or an animation sequence. For a static view of a scene, the view plane is usually placed at the viewing-coordinate origin, which is at some convenient point in the scene. Then it is easy to adjust the size of the window to include all parts of the scene that we want to view. The projection reference point is positioned to obtain the amount of perspective desired. In an animation sequence, we can place the projection reference point at the viewing-coordinate origin and put the view plane in front of the scene (Fig. 12-35). This placement simulates a camera reference frame. We set the field of view (lens angle) by adjusting the size of the window relative to the distance of the view plane from the projection reference point. We move through the scene by moving the viewing reference frame, and the projection reference point will move with the view reference point.

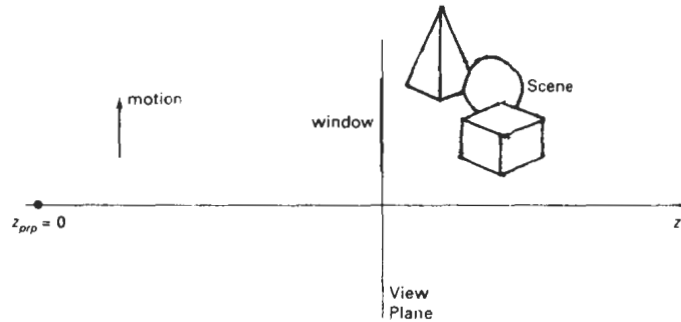


Figure 12-35
View-plane positioning to simulate a camera reference frame for an animation sequence.

General Parallel-Projection Transformations

In PHIGS, the direction of a parallel projection is specified with a projection vector from the projection reference point to the center of the view window. Figure 12-36 shows the general shape of a finite view volume for a given projection vector and projection window in the view plane. We obtain the oblique-projection transformation with a shear operation that converts the view volume in Fig. 12-36 to the regular parallelepiped shown in Fig. 12-37.

The elements of the shearing transformation needed to generate the view volume shown in Fig. 12-37 are obtained by considering the shear transformation of the projection vector. If the projection vector is specified in world coordinates, it must first be transformed to viewing coordinates using the rotation matrix discussed in Section 12-2. (The projection vector is unaffected by the translation, since it is simply a direction with no fixed position.) For graphics packages that allow specification of the projection vector in viewing coordinates, we apply the shear directly to the input elements of the projection vector.

Suppose the elements of the projection vector in viewing coordinates are

$$\mathbf{V}_p = (p_x, p_y, p_z) \quad (12-19)$$

We need to determine the elements of a shear matrix that will align the projection vector \mathbf{V}_p with the view plane normal vector \mathbf{N} (Fig. 12-37). This transformation can be expressed as

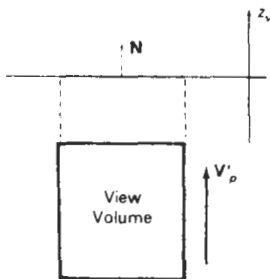


Figure 12-37
Regular parallelepiped view volume obtained by shearing the view volume in Fig. 12-36.

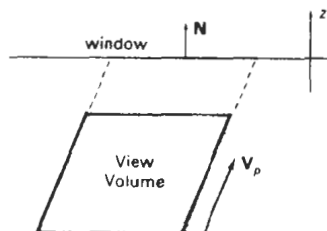


Figure 12-36
Oblique projection vector and associated view volume.

$$\begin{aligned} \mathbf{V}'_p &= \mathbf{M}_{\text{parallel}} \cdot \mathbf{V}_p \\ &= \begin{bmatrix} 0 \\ 0 \\ p_z \\ 0 \end{bmatrix} \end{aligned} \quad (12-20)$$

where $\mathbf{M}_{\text{parallel}}$ is equivalent to the parallel projection matrix 12-10 and represents a z-axis shear of the form

$$\mathbf{M}_{\text{parallel}} = \begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (12-21)$$

The explicit transformation equations from 12-20 in terms of shear parameters a and b are

$$\begin{aligned} 0 &= p_x + a p_z \\ 0 &= p_y + b p_z \end{aligned} \quad (12-22)$$

so that the values for the shear parameters are

$$a = -\frac{p_x}{p_z}, \quad b = -\frac{p_y}{p_z} \quad (12-23)$$

Thus, we have the general parallel-projection matrix in terms of the elements of the projection vector as

$$\mathbf{M}_{\text{parallel}} = \begin{bmatrix} 1 & 0 & -p_x/p_z & 0 \\ 0 & 1 & -p_y/p_z & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (12-24)$$

This matrix is then concatenated with transformation $\mathbf{R} \cdot \mathbf{T}$, from Section 12-2, to produce the transformation from world coordinates to parallel-projection coordinates. For an orthographic parallel projection, $p_x = p_y = 0$, and $\mathbf{M}_{\text{parallel}}$ is the identity matrix. From Fig. 12-38, we can relate the components of the projection vector to parameters L , α , and ϕ (Section 12-3). By similar triangles, we see that

$$\begin{aligned} \frac{L \cos \phi}{z} &= -\frac{p_x}{p_z} \\ \frac{L \sin \phi}{z} &= -\frac{p_y}{p_z} \end{aligned} \quad (12-25)$$

which illustrates the equivalence of the elements of transformation matrices 12-10 and 12-24. In Eqs. 12-25, z and p_z are of opposite signs, and for the positions illustrated in Fig. 12-38, $z < 0$.

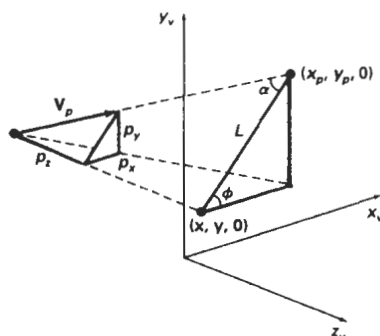


Figure 12-38
Relationship between the parallel-projection vector V_p and parameters L , α , and ϕ .

General Perspective-Projection Transformations

With the PHIGS programming standard, the projection reference point can be located at any position in the viewing system, except on the view plane or between the front and back clipping planes. Figure 12-39 shows the shape of a finite view volume for an arbitrary position of the projection reference point. We can obtain the general perspective-projection transformation with the following two operations:

1. Shear the view volume so that the centerline of the frustum is perpendicular to the view plane.
2. Scale the view volume with a scaling factor that depends on $1/z$.

The second step (scaling the view volume) is equivalent to the perspective transformation discussed in Section 12-3.

A shear operation to align a general perspective view volume with the pro-

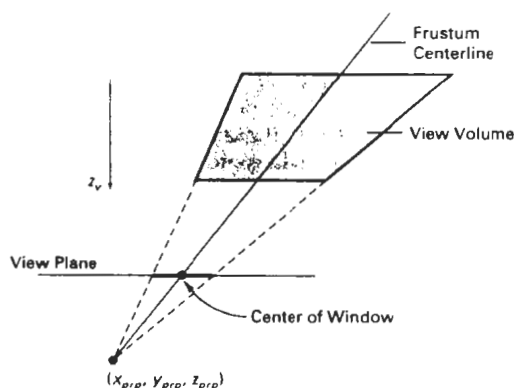


Figure 12-39
General shape for the perspective view volume with a projection reference point that is not on the z_v axis

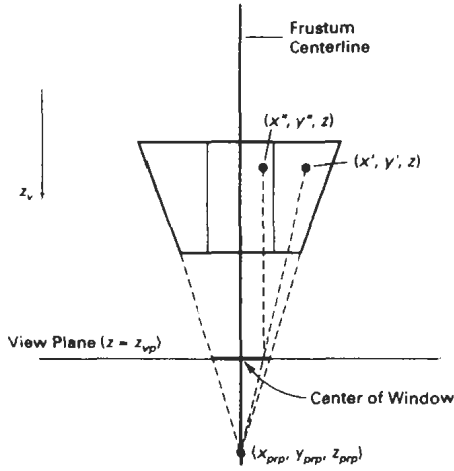


Figure 12-40

Shearing a general perspective view volume to center it on the projection window.

jection window is shown in Fig. 12-40. This transformation has the effect of shifting all positions that lie along the frustum centerline, including the window center, to a line perpendicular to the view plane. With the projection reference point at a general position $(x_{prp}, y_{prp}, z_{prp})$, the transformation involves a combination z -axis shear and a translation:

$$M_{\text{shear}} = \begin{bmatrix} 1 & 0 & a & -az_{prp} \\ 0 & 1 & b & -bz_{prp} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (12-26)$$

where the shear parameters are

$$a = -\frac{x_{prp} - (xw_{\min} + xw_{\max})/2}{z_{prp}} \quad (12-27)$$

$$b = -\frac{y_{prp} - (yw_{\min} + yw_{\max})/2}{z_{prp}}$$

Points within the view volume are transformed by this operation as

$$\begin{aligned} x' &= x + a(z - z_{prp}) \\ y' &= y + b(z - z_{prp}) \\ z' &= z \end{aligned} \quad (12-28)$$

When the projection reference point is on the z_v axis, $x_{prp} = y_{prp} = 0$.

Once we have converted a position (x, y, z) in the original view volume to position (x', y', z') in the sheared frustum, we then apply a scaling transformation to produce a regular parallelepiped (Fig. 12-40). The transformation for this conversion is

$$x'' = x' \left(\frac{z_{prp} - z_{vp}}{z_{prp} - z} \right) + x_{prp} \left(\frac{z_{vp} - z}{z_{prp} - z} \right) \quad (12-29)$$

$$y'' = y' \left(\frac{z_{prp} - z_{vp}}{z_{prp} - z} \right) + y_{prp} \left(\frac{z_{vp} - z}{z_{prp} - z} \right)$$

and the homogeneous matrix representation is

$$\mathbf{M}_{\text{scale}} = \begin{bmatrix} 1 & 0 & \frac{-x_{prp}}{z_{prp} - z_{vp}} & \frac{x_{prp} z_{vp}}{z_{prp} - z_{vp}} \\ 0 & 1 & \frac{-y_{prp}}{z_{prp} - z_{vp}} & \frac{y_{prp} z_{vp}}{z_{prp} - z_{vp}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{-1}{z_{prp} - z_{vp}} & \frac{z_{prp}}{z_{prp} - z_{vp}} \end{bmatrix} \quad (12-30)$$

Therefore, the general perspective-projection transformation can be expressed in matrix form as

$$\mathbf{M}_{\text{perspective}} = \mathbf{M}_{\text{scale}} \cdot \mathbf{M}_{\text{shear}} \quad (12-31)$$

The complete transformation from world coordinates to perspective-projection coordinates is obtained by right concatenating $\mathbf{M}_{\text{perspective}}$ with the composite viewing transformation $\mathbf{R} \cdot \mathbf{T}$ from Section 12-2.

12-5

CLIPPING

In this section, we first explore the general ideas involved in three-dimensional clipping by considering how clipping could be performed using the view-volume clipping planes directly. Then we discuss more efficient methods using normalized view volumes and homogeneous coordinates.

An algorithm for three-dimensional clipping identifies and saves all surface segments within the view volume for display on the output device. All parts of objects that are outside the view volume are discarded. Clipping in three dimensions can be accomplished using extensions of two-dimensional clipping methods. Instead of clipping against straight-line window boundaries, we now clip objects against the boundary planes of the view volume.

To clip a line segment against the view volume, we would need to test the relative position of the line using the view volume's boundary plane equations. By substituting the line endpoint coordinates into the plane equation of each boundary in turn, we could determine whether the endpoint is inside or outside that boundary. An endpoint (x, y, z) of a line segment is outside a boundary plane if $Ax + By + Cz + D > 0$, where A , B , C , and D are the plane parameters for that boundary. Similarly, the point is inside the boundary if $Ax + By + Cz + D < 0$. Lines with both endpoints outside a boundary plane are discarded, and those with both endpoints inside all boundary planes are saved. The intersection of a line with a boundary is found using the line equations along with the plane equation. Intersection coordinates (x_i, y_i, z_i) are values that are on the line and that satisfy the plane equation $Ax_i + By_i + Cz_i + D = 0$.

To clip a polygon surface, we can clip the individual polygon edges. First, we could test the coordinate extents against each boundary of the view volume to determine whether the object is completely inside or completely outside that

boundary. If the coordinate extents of the object are inside all boundaries, we save it. If the coordinate extents are outside all boundaries, we discard it. Otherwise, we need to apply the intersection calculations. We could do this by determining the polygon edge-intersection positions with the boundary planes of the view volume, as described in the previous paragraph.

As in two-dimensional viewing, the projection operations can take place before the view-volume clipping or after clipping. All objects within the view volume map to the interior of the specified projection window. The last step is to transform the window contents to a two-dimensional viewport, which specifies the location of the display on the output device.

Clipping in two dimensions is generally performed against an upright rectangle; that is, the clip window is aligned with the x and y axes. This greatly simplifies the clipping calculations, because each window boundary is defined by one coordinate value. For example, the intersections of all lines crossing the left boundary of the window have an x coordinate equal to the left boundary.

View-volume clipping boundaries are planes whose orientations depend on the type of projection, the projection window, and the position of the projection reference point. Since the front and back clipping planes are parallel to the view plane, each has a constant z -coordinate value. The z coordinate of the intersections of lines with these planes is simply the z coordinate of the corresponding plane. But the other four sides of the view volume can have arbitrary spatial orientations. To find the intersection of a line with one of the view volume boundaries means that we must obtain the equation for the plane containing that boundary polygon. This process is simplified if we convert the view volume before clipping to a rectangular parallelepiped. In other words, we first perform the projection transformation, which converts coordinate values in the view volume to orthographic parallel coordinates, then we carry out the clipping calculations.

Clipping against a regular parallelepiped is much simpler because each surface is now perpendicular to one of the coordinate axes. As seen in Fig. 12-41, the top and bottom of the view volume are now planes of constant y , the sides are planes of constant x , and the front and back are planes of constant z . A line cutting through the top plane of the parallelepiped, for example, has an intersection point whose y -coordinate value is that of the top plane.

In the case of an orthographic parallel projection, the view volume is already a rectangular parallelepiped. As we have seen in Section 12-3, oblique-projection view volumes are converted to a rectangular parallelepiped by the shearing operation, and perspective view volumes are converted, in general, with a combination shear-scale transformation.

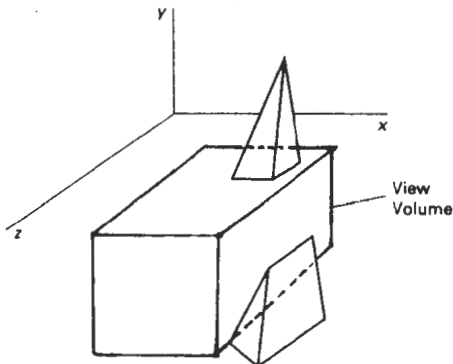


Figure 12-41
An object intersecting a rectangular parallelepiped view volume.

Normalized View Volumes

Figure 12-42 shows the expanded PHIGS transformation pipeline. At the first step, a scene is constructed by transforming object descriptions from modeling coordinates to world coordinates. Next, a view mapping converts the world descriptions to viewing coordinates. At the projection stage, the viewing coordinates are transformed to projection coordinates, which effectively converts the view volume into a rectangular parallelepiped. Then, the parallelepiped is mapped into the unit cube, a **normalized view volume** called the **normalized projection coordinate system**. The mapping to normalized projection coordinates is accomplished by transforming points within the rectangular parallelepiped into a position within a specified three-dimensional viewport, which occupies part or all of the unit cube. Finally, at the workstation stage, normalized projection coordinates are converted to device coordinates for display.

The normalized view volume is a region defined by the planes

$$x = 0, \quad x = 1, \quad y = 0, \quad y = 1, \quad z = 0, \quad z = 1 \quad (12-32)$$

A similar transformation sequence is used in other graphics packages, with individual variations depending on the system. The GL package, for example, maps the rectangular parallelepiped into the interior of a cube with boundary planes at positions ± 1 in each coordinate direction.

There are several advantages to clipping against the unit cube instead of the original view volume or even the rectangular parallelepiped in projection coordinates. First, the normalized view volume provides a standard shape for representing any sized view volume. This separates the viewing transformations from any workstation considerations, and the unit cube then can be mapped to a workstation of any size. Second, clipping procedures are simplified and standardized with unit clipping planes or the viewport planes, and additional clipping planes can be specified within the normalized space before transforming to

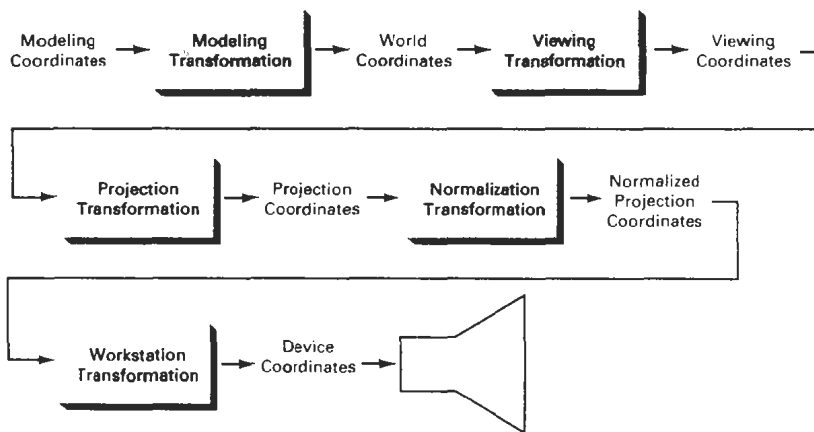


Figure 12-42
Expanded PHIGS transformation pipeline

device coordinates. Third, depth cueing and visible-surface determination are simplified, since the z axis always points toward the viewer (the projection reference point has now been transformed to the z axis). Front faces of objects are those with normal vectors having a component along the positive z direction; and back surfaces are facing in the negative z direction.

Mapping positions within a rectangular view volume to a three-dimensional rectangular viewport is accomplished with a combination of scaling and translation, similar to the operations needed for a two-dimensional window-to-viewport mapping. We can express the three-dimensional transformation matrix for these operations in the form

$$\begin{bmatrix} D_x & 0 & 0 & K_x \\ 0 & D_y & 0 & K_y \\ 0 & 0 & D_z & K_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (12-33)$$

Factors D_x , D_y , and D_z are the ratios of the dimensions of the viewport and regular parallelepiped view volume in the x , y , and z directions (Fig. 12-43):

$$\begin{aligned} D_x &= \frac{xv_{\max} - xv_{\min}}{xw_{\max} - xw_{\min}} \\ D_y &= \frac{yv_{\max} - yv_{\min}}{yw_{\max} - yw_{\min}} \\ D_z &= \frac{zv_{\max} - zv_{\min}}{z_{\text{back}} - z_{\text{front}}} \end{aligned} \quad (12-34)$$

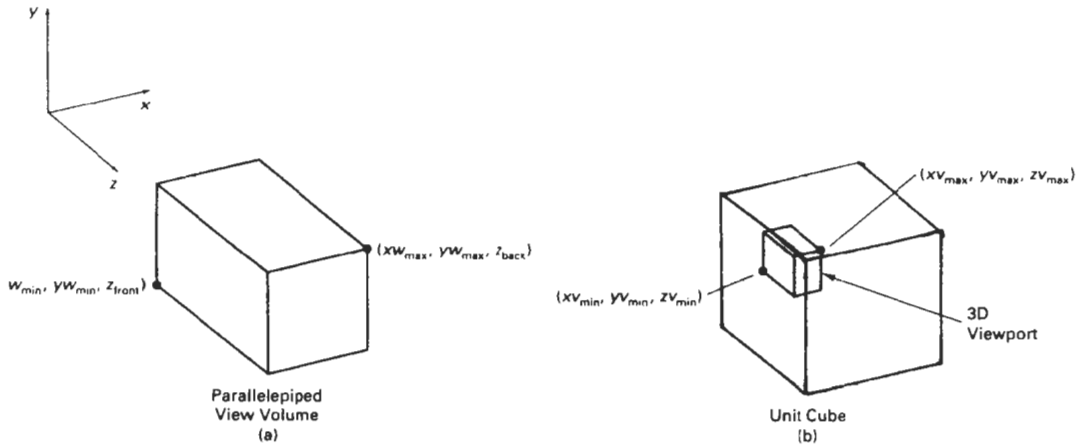


Figure 12-43

Dimensions of the view volume and three-dimensional viewport.

where the view-volume boundaries are established by the window limits (xw_{\min} , xw_{\max} , yw_{\min} , yw_{\max}) and the positions z_{front} and z_{back} of the front and back planes. Viewport boundaries are set with the coordinate values xv_{\min} , xv_{\max} , yv_{\min} , yv_{\max} , zv_{\min} , and zv_{\max} . The additive translation factors K_x , K_y , and K_z in the transformation are

$$\begin{aligned}K_x &= xv_{\min} - xw_{\min}D_x \\K_y &= yv_{\min} - yw_{\min}D_y \\K_z &= zv_{\min} - z_{\text{front}}D_z\end{aligned}\tag{12-35}$$

Viewport Clipping

Lines and polygon surfaces in a scene can be clipped against the viewport boundaries with procedures similar to those used for two dimensions, except that objects are now processed against clipping planes instead of clipping edges. Curved surfaces are processed using the defining equations for the surface boundary and locating the intersection lines with the parallelepiped planes.

The two-dimensional concept of region codes can be extended to three dimensions by considering positions in front and in back of the three-dimensional viewport, as well as positions that are left, right, below, or above the volume. For two-dimensional clipping, we used a four-digit binary region code to identify the position of a line endpoint relative to the viewport boundaries. For three-dimensional points, we need to expand the region code to six bits. Each point in the description of a scene is then assigned a six-bit region code that identifies the relative position of the point with respect to the viewport. For a line endpoint at position (x , y , z), we assign the bit positions in the region code from right to left as

$$\begin{aligned}\text{bit 1} &= 1, & \text{if } x < xv_{\min}(\text{left}) \\ \text{bit 2} &= 1, & \text{if } x > xv_{\max}(\text{right}) \\ \text{bit 3} &= 1, & \text{if } y < yv_{\min}(\text{below}) \\ \text{bit 4} &= 1, & \text{if } y > yv_{\max}(\text{above}) \\ \text{bit 5} &= 1, & \text{if } z < zv_{\min}(\text{front}) \\ \text{bit 6} &= 1, & \text{if } z > zv_{\max}(\text{back})\end{aligned}$$

For example, a region code of 101000 identifies a point as above and behind the viewport, and the region code 000000 indicates a point within the volume.

A line segment can be immediately identified as completely within the viewport if both endpoints have a region code of 000000. If either endpoint of a line segment does not have a region code of 000000, we perform the logical *and* operation on the two endpoint codes. The result of this *and* operation will be nonzero for any line segment that has both endpoints in one of the six outside regions. For example, a nonzero value will be generated if both endpoints are behind the viewport, or both endpoints are above the viewport. If we cannot identify a line segment as completely inside or completely outside the volume, we test for intersections with the bounding planes of the volume.

As in two-dimensional line clipping, we use the calculated intersection of a line with a viewport plane to determine how much of the line can be thrown

away. The remaining part of the line is checked against the other planes, and we continue until either the line is totally discarded or a section is found inside the volume.

Equations for three-dimensional line segments are conveniently expressed in parametric form. The two-dimensional parametric clipping methods of Cyrus–Beck or Liang–Barsky can be extended to three-dimensional scenes. For a line segment with endpoints $P_1 = (x_1, y_1, z_1)$ and $P_2 = (x_2, y_2, z_2)$, we can write the parametric line equations as

$$\begin{aligned}x &= x_1 + (x_2 - x_1)u, & 0 \leq u \leq 1 \\y &= y_1 + (y_2 - y_1)u \\z &= z_1 + (z_2 - z_1)u\end{aligned}\tag{12-36}$$

Coordinates (x, y, z) represent any point on the line between the two endpoints. At $u = 0$, we have the point P_1 , and $u = 1$ puts us at P_2 .

To find the intersection of a line with a plane of the viewport, we substitute the coordinate value for that plane into the appropriate parametric expression of Eq. 12-36 and solve for u . For instance, suppose we are testing a line against the $z_{v_{\min}}$ plane of the viewport. Then

$$u = \frac{z_{v_{\min}} - z_1}{z_2 - z_1}\tag{12-37}$$

When the calculated value for u is not in the range from 0 to 1, the line segment does not intersect the plane under consideration at any point between endpoints P_1 and P_2 (line *A* in Fig. 12-44). If the calculated value for u in Eq. 12-37 is in the interval from 0 to 1, we calculate the intersection's x and y coordinates as

$$\begin{aligned}x_I &= x_1 + (x_2 - x_1) \left(\frac{z_{v_{\min}} - z_1}{z_2 - z_1} \right) \\y_I &= y_1 + (y_2 - y_1) \left(\frac{z_{v_{\min}} - z_1}{z_2 - z_1} \right)\end{aligned}\tag{12-38}$$

If either x_I or y_I is not in the range of the boundaries of the viewport, then this line intersects the front plane beyond the boundaries of the volume (line *B* in Fig. 12-44).

Clipping in Homogeneous Coordinates

Although we have discussed the clipping procedures in terms of three-dimensional coordinates, PHIGS and other packages actually represent coordinate positions in homogeneous coordinates. This allows the various transformations to be represented as 4 by 4 matrices, which can be concatenated for efficiency. After all viewing and other transformations are complete, the homogeneous-coordinate positions are converted back to three-dimensional points.

As each coordinate position enters the transformation pipeline, it is converted to a homogeneous-coordinate representation:

$$(x, y, z) \rightarrow (x, y, z, 1)$$

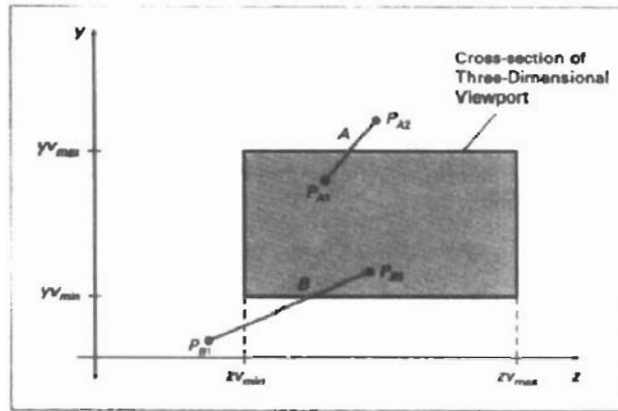


Figure 12-44

Side view of two line segments that are to be clipped against the $z_{v_{\min}}$ plane of the viewport. For line A, Eq. 12-37 produces a value of u that is outside the range from 0 to 1. For line B, Eqs. 12-38 produce intersection coordinates that are outside the range from $y_{v_{\min}}$ to $y_{v_{\max}}$.

The various transformations are applied and we obtain the final homogeneous point:

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (12-39)$$

where the homogeneous parameter h may not be 1. In fact, h can have any real value. Clipping is then performed in homogeneous coordinates, and clipped homogeneous positions are converted to nonhomogeneous coordinates in three-dimensional normalized-projection coordinates:

$$x' = \frac{x_h}{h}, \quad y' = \frac{y_h}{h}, \quad z' = \frac{z_h}{h} \quad (12-40)$$

We will, of course, have a problem if the magnitude of parameter h is very small or has the value 0; but normally this will not occur, if the transformations are carried out properly. At the final stage in the transformation pipeline, the normalized point is transformed to a three-dimensional device coordinate point. The xy position is plotted on the device, and the z component is used for depth-information processing.

Setting up clipping procedures in homogeneous coordinates allows hardware viewing implementations to use a single procedure for both parallel and perspective projection transformations. Objects viewed with a parallel projection could be correctly clipped in three-dimensional normalized coordinates, pro-

vided the value $h = 1$ has not been altered by other operations. But perspective projections, in general, produce a homogeneous parameter that no longer has the value 1. Converting the sheared frustum to a rectangular parallelepiped can change the value of the homogeneous parameter. So we must clip in homogeneous coordinates to be sure that the clipping is carried out correctly. Also, rational spline representations are set up in homogeneous coordinates with arbitrary values for the homogeneous parameter, including $h < 1$. Negative values for the homogeneous parameter can also be generated in perspective projections when coordinate positions are behind the projection reference point. This can occur in applications where we might want to move inside of a building or other object to view its interior.

To determine homogeneous viewport clipping boundaries, we note that any homogeneous-coordinate position (x_h, y_h, z_h, h) is inside the viewport if it satisfies the inequalities

$$xv_{\min} \leq \frac{x_h}{h} \leq xv_{\max}, \quad yv_{\min} \leq \frac{y_h}{h} \leq yv_{\max}, \quad zv_{\min} < \frac{z_h}{h} \leq zv_{\max} \quad (12-41)$$

Thus, the homogeneous clipping limits are

$$\begin{aligned} hxv_{\min} \leq x_h \leq hxv_{\max}, \quad hyv_{\min} \leq y_h \leq hyv_{\max}, \quad hzv_{\min} \leq z_h \leq hzv_{\max}, & \quad \text{if } h > 0 \\ hxv_{\max} \leq x_h \leq hxv_{\min}, \quad hyv_{\max} \leq y_h \leq hyv_{\min}, \quad hzv_{\max} \leq z_h \leq hzv_{\min}, & \quad \text{if } h < 0 \end{aligned} \quad (12-42)$$

And clipping is carried out with procedures similar to those discussed in the previous section. To avoid applying both sets of inequalities in 12-42, we can simply negate the coordinates for any point with $h < 0$ and use the clipping inequalities for $h > 0$.

12-6

HARDWARE IMPLEMENTATIONS

Most graphics processes are now implemented in hardware. Typically, the viewing, visible-surface identification, and shading algorithms are available as graphics chip sets, employing VLSI (very large-scale integration) circuitry techniques. Hardware systems are now designed to transform, clip, and project objects to the output device for either three-dimensional or two-dimensional applications.

Figure 12-45 illustrates an arrangement of components in a graphics chip set to implement the viewing operations we have discussed in this chapter. The chips are organized into a pipeline for accomplishing geometric transformations, coordinate-system transformations, projections, and clipping. Four initial chips are provided for matrix operations involving scaling, translation, rotation, and the transformations needed for converting world coordinates to projection coordinates. Each of the next six chips performs clipping against one of the viewport boundaries. Four of these chips are used in two-dimensional applications, and the other two are needed for clipping against the front and back planes of the three-dimensional viewport. The last two chips in the pipeline convert viewport coordinates to output device coordinates. Components for implementation of visible-surface identification and surface-shading algorithms can be added to this set to provide a complete three-dimensional graphics system.

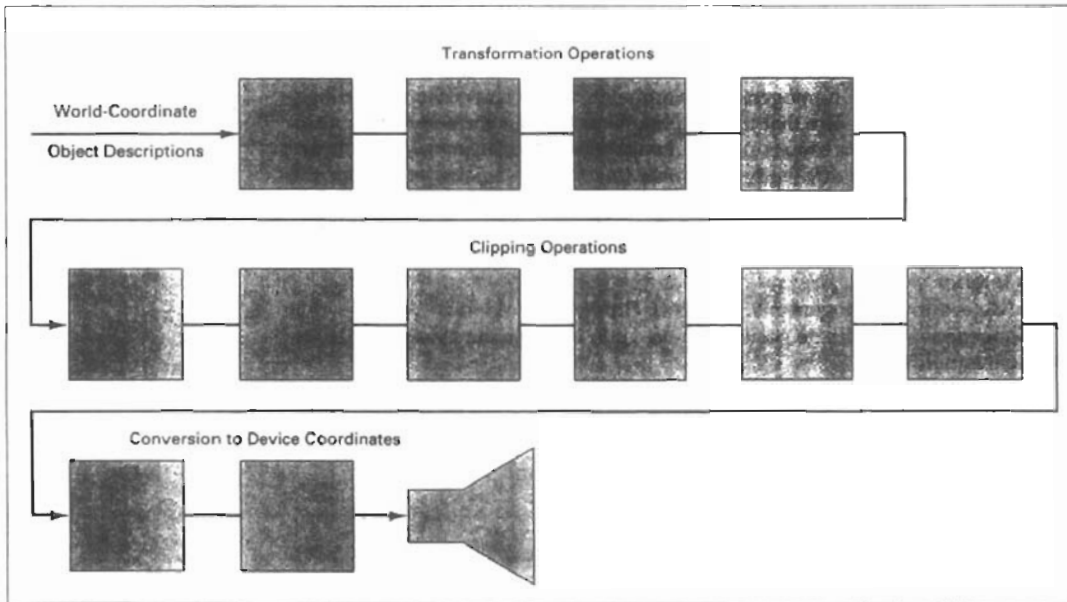


Figure 12-45

A hardware implementation of three-dimensional viewing operations using 12 chips for the coordinate transformations and clipping operations.

Other specialized hardware implementations have been developed. These include hardware systems for processing octree representations and for displaying three-dimensional scenes using ray-tracing algorithms (Chapter 14).

12-7

THREE-DIMENSIONAL VIEWING FUNCTIONS

Several procedures are usually provided in a three-dimensional graphics library to enable an application program to set the parameters for viewing transformations. There are, of course, a number of different methods for structuring these procedures. Here, we discuss the PHIGS functions for three-dimensional viewing.

With parameters specified in world coordinates, elements of the matrix for transforming world-coordinate descriptions to the viewing reference frame are calculated using the function

```
evaluateViewOrientationMatrix3 (x0, y0, z0, xN, yN, zN,  
                               xV, yV, zV, error, viewMatrix)
```

This function creates the `viewMatrix` from input coordinates defining the viewing system, as discussed in Section 12-2. Parameters `x0`, `y0`, and `z0` specify the

origin (view reference point) of the viewing system. World-coordinate vector (x_N , y_N , z_N) defines the normal to the view plane and the direction of the positive z_v viewing axis. And world-coordinate vector (x_V , y_V , z_V) gives the elements of the view-up vector. The projection of this vector perpendicular to (x_N , y_N , z_N) establishes the direction for the positive y_v axis of the viewing system. An integer error code is generated in parameter *error* if input values are not specified correctly. For example, an error will be generated if we set (x_V , y_V , z_V) parallel to (x_N , y_N , z_N).

To specify a second viewing-coordinate system, we can redefine some or all of the coordinate parameters and invoke `evaluateViewOrientationMatrix3` with a new matrix designation. In this way, we can set up any number of world-to-viewing-coordinate matrix transformations.

The matrix `projMatrix` for transforming viewing coordinates to normalized projection coordinates is created with the function

```
evaluateViewMappingMatrix3 (xwmin, xwmax, ywmin, ywmax,
                           xvmin, xvmax, yvmin, yvmax, zvmin, zvmax,
                           projType, xprojRef, yprojRef, zprojRef, zview,
                           zback, zfront, error, projMatrix)
```

Window limits on the view plane are given in viewing coordinates with parameters `xwmin`, `xwmax`, `ywmin`, and `ywmax`. Limits of the three-dimensional viewport within the unit cube are set with normalized coordinates `xvmin`, `xvmax`, `yvmin`, `yvmax`, `zvmin`, and `zvmax`. Parameter `projType` is used to choose the projection type as either *parallel* or *perspective*. Coordinate position (`xprojRef`, `yprojRef`, `zprojRef`) sets the projection reference point. This point is used as the center of projection if `projType` is set to *perspective*; otherwise, this point and the center of the view-plane window define the parallel-projection vector. The position of the view plane along the viewing z_v axis is set with parameter `zview`. Positions along the viewing z_v axis for the front and back planes of the view volume are given with parameters `zfront` and `zback`. And the *error* parameter returns an integer error code indicating erroneous input data. Any number of projection matrix transformations can be created with this function to obtain various three-dimensional views and projections.

A particular combination of viewing and projection matrices is selected on a specified workstation with

```
setViewRepresentation3 (ws, viewIndex, viewMatrix, projMatrix,
                       xclipmin, xclipmax, yclipmin, yclipmax, zclipmin,
                       zclipmax, clipxy, clipback, clipfront)
```

Parameter `ws` is used to select the workstation, and parameters `viewMatrix` and `projMatrix` select the combination of viewing and projection matrices to be used. The concatenation of these matrices is then placed in the workstation *view table* and referenced with an integer value assigned to parameter `viewIndex`. Limits, given in normalized projection coordinates, for clipping a scene are set with parameters `xclipmin`, `xclipmax`, `yclipmin`, `yclipmax`, `zclipmin`, and `zclipmax`. These limits can be set to any values, but they are usually set to the limits of the viewport. Values of *clip* or *noclip* are assigned to parameters `clipxy`, `clipfront`, and `clipback` to turn the clipping routines on or off for the *xy* planes or for the front or back planes of the view volume (or the defined clipping limits).

There are several times when it is convenient to bypass the clipping routines. For initial constructions of a scene, we can disable clipping so that trial placements of objects can be displayed quickly. Also, we can eliminate one or more of the clipping planes if we know that all objects are inside those planes.

Once the view tables have been set up, we select a particular view representation on each workstation with the function

```
setViewIndex (viewIndex)
```

The view index number identifies the set of viewing-transformation parameters that are to be applied to subsequently specified output primitives, for each of the active workstations.

Finally, we can use the **workstation transformation** functions to select sections of the projection window for display on different workstations. These operations are similar to those discussed for two-dimensional viewing, except now our window and viewport regions are three-dimensional regions. The window function selects a region of the unit cube, and the viewport function selects a display region for the output device. Limits, in normalized projection coordinates, for the window are set with

```
setWorkstationWindow3 (ws, xwsWindmin, xwsWindmax,  
                      ywsWindmin, ywsWindmax, zwsWindmin, zwsWindmax)
```

and limits, in device coordinates, for the viewport are set with

```
setWorkstationViewport3 (ws, xwsVPortmin, xwsVPortmax,  
                        ywsVPortmin, ywsVPortmax, zwsVPortmin, zwsVPortmax)
```

Figure 12-46 shows an example of interactive selection of viewing parameters in the PHIGS viewing pipeline, using the PHIGS Toolkit software. This software was developed at the University of Manchester to provide an interface to PHIGS with a viewing editor, windows, menus, and other interface tools.

For some applications, composite methods are used to create a display consisting of multiple views using different camera orientations. Figure 12-47 shows

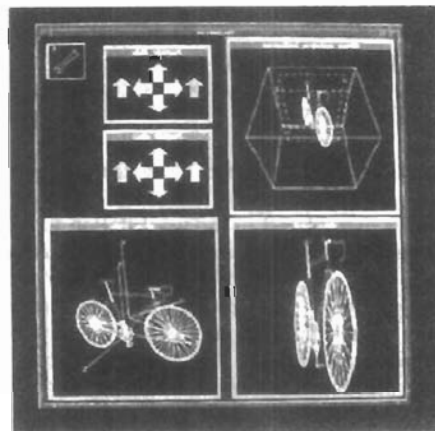


Figure 12-46
Using the PHIGS Toolkit,
developed at the University of
Manchester, to interactively control
parameters in the viewing pipeline.
(Courtesy of T. L. J. Howard, J. G. Williams,
and W. T. Hewitt, Department of Computer
Science, University of Manchester, United
Kingdom.)

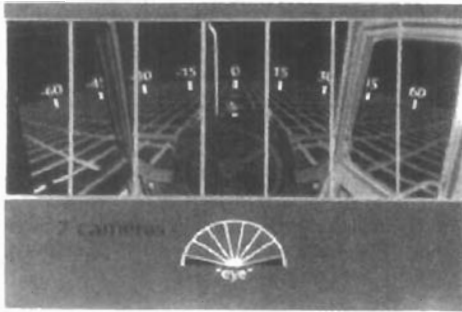


Figure 12-47

A wide-angle view for a virtual-reality display generated with seven sections, each from a slightly different viewing direction. (Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)

a wide-angle perspective display produced for a virtual-reality environment. The wide viewing angle is attained by generating seven views of the scene from the same viewing position, but with slight shifts in the viewing direction.

SUMMARY

Viewing procedures for three-dimensional scenes follow the general approach used in two-dimensional viewing. That is, we first create a world-coordinate scene from the definitions of objects in modeling coordinates. Then we set up a viewing-coordinate reference frame and transfer object descriptions from world coordinates to viewing coordinates. Finally, viewing-coordinate descriptions are transformed to device-coordinates.

Unlike two-dimensional viewing, however, three-dimensional viewing requires projection routines to transform object descriptions to a viewing plane before the transformation to device coordinates. Also, three-dimensional viewing operations involve more spatial parameters. We can use the camera analogy to describe three-dimensional viewing parameters, which include camera position and orientation. A viewing-coordinate reference frame is established with a view reference point, a view-plane normal vector N , and a view-up vector V . View-plane position is then established along the viewing z axis, and object descriptions are projected to this plane. Either perspective-projection or parallel-projection methods can be used to transfer object descriptions to the view plane.

Parallel projections are either orthographic or oblique and can be specified with a projection vector. Orthographic parallel projections that display more than one face of an object are called axonometric projections. An isometric view of an object is obtained with an axonometric projection that foreshortens each principal axis by the same amount. Commonly used oblique projections are the cavalier projection and the cabinet projection. Perspective projections of objects are obtained with projection lines that meet at the projection reference point.

Objects in three-dimensional scenes are clipped against a view volume. The top, bottom, and sides of the view volume are formed with planes that are parallel to the projection lines and that pass through the view-plane window edges. Front and back planes are used to create a closed view volume. For a parallel projection, the view volume is a parallelepiped, and for a perspective projection, the view volume is a frustum. Objects are clipped in three-dimensional viewing by testing object coordinates against the bounding planes of the view volume. Clipping is generally carried out in graphics packages in homogeneous coordinates

after all viewing and other transformations are complete. Then, homogeneous coordinates are converted to three-dimensional Cartesian coordinates.

REFERENCES

For additional information on three-dimensional viewing and clipping operations in PHIGS and PHIGS+, see Howard et al. (1991), Gaskins (1992), and Blake (1993). Discussions of three-dimensional clipping and viewing algorithms can be found in Blinn and Newell (1978), Cyrus and Beck (1978), Riesenfeld (1981), Liang and Barsky (1984), Arvo (1991), and Blinn (1993).

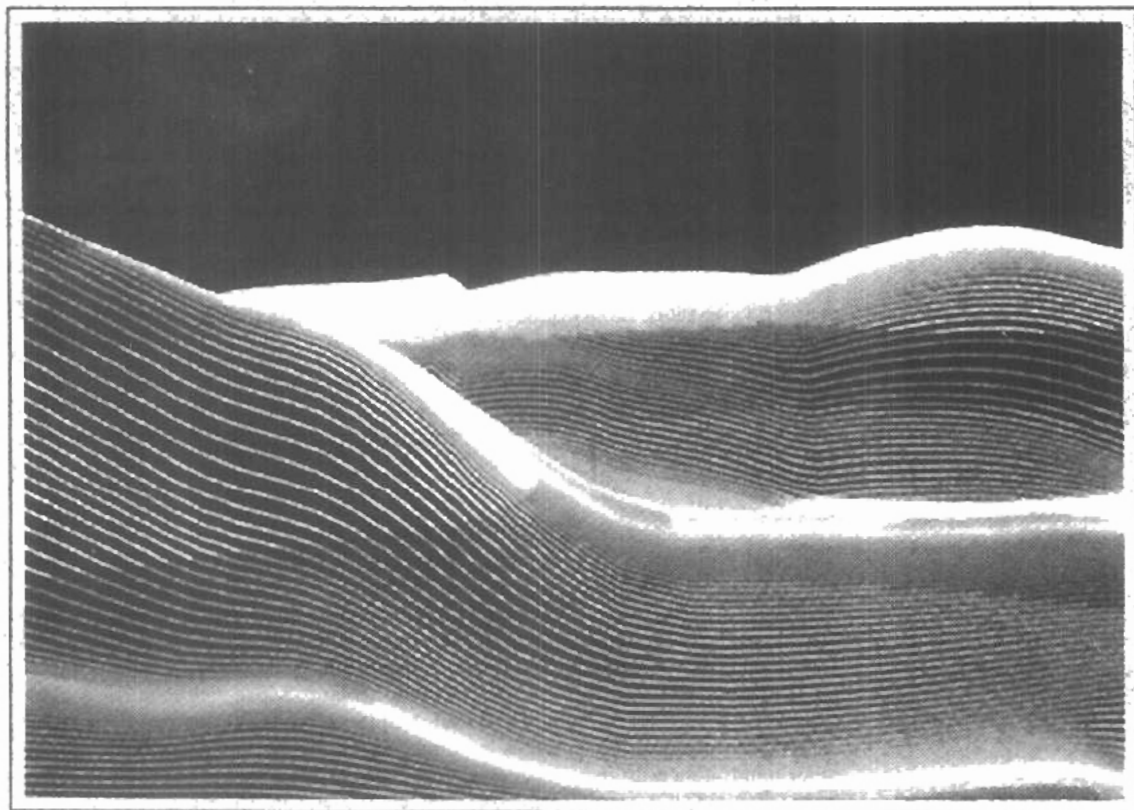
EXERCISES

- 12-1. Write a procedure to implement the `evaluateViewOrientationMatrix3` function using Eqs. 12-2 through 12-4.
- 12-2. Write routines to implement the `setViewRepresentation3` and `setViewIndex` functions.
- 12-3. Write a procedure to transform the vertices of a polyhedron to projection coordinates using a parallel projection with a specified projection vector.
- 12-4. Write a procedure to obtain different parallel-projection views of a polyhedron by first applying a specified rotation.
- 12-5. Write a procedure to perform a one-point perspective projection of an object.
- 12-6. Write a procedure to perform a two-point perspective projection of an object.
- 12-7. Develop a routine to perform a three-point perspective projection of an object.
- 12-8. Write a routine to convert a perspective projection frustum to a regular parallelepiped.
- 12-9. Extend the Sutherland-Hodgman polygon clipping algorithm to clip three-dimensional planes against a regular parallelepiped.
- 12-10. Devise an algorithm to clip objects in a scene against a defined frustum. Compare the operations needed in this algorithm to those needed in an algorithm that clips against a regular parallelepiped.
- 12-11. Modify the two-dimensional Liang-Barsky line-clipping algorithm to clip three-dimensional lines against a specified regular parallelepiped.
- 12-12. Modify the two-dimensional Liang-Barsky line-clipping algorithm to clip a given polyhedron against a specified regular parallelepiped.
- 12-13. Set up an algorithm for clipping a polyhedron against a parallelepiped.
- 12-14. Write a routine to perform clipping in homogeneous coordinates.
- 12-15. Using any clipping procedure and orthographic parallel projections, write a program to perform a complete viewing transformation from world coordinates to device coordinates.
- 12-16. Using any clipping procedure, write a program to perform a complete viewing transformation from world coordinates to device coordinates for any specified parallel-projection vector.
- 12-17. Write a program to perform all steps in the viewing pipeline for a perspective transformation.

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

$$Ax + By + Cz + D < 0 \quad (13-1)$$

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 \mathbf{N} to a polygon surface, which has Cartesian components (A, B, C) . In general, if \mathbf{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

$$\mathbf{V} \cdot \mathbf{N} > 0 \quad (13-2)$$

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 \mathbf{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:

$$C \leq 0 \quad (13-3)$$

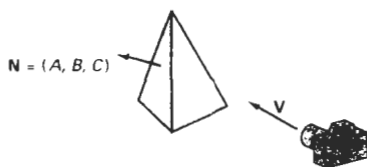


Figure 13-1
Vector \mathbf{V} in the viewing direction
and a back-face normal vector \mathbf{N} of
a polyhedron

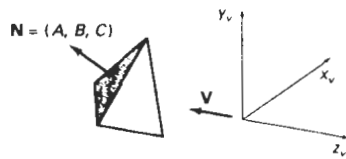


Figure 13-2
A polygon surface with plane parameter $C < 0$ in a right-handed viewing coordinate system is identified as a back face when the viewing direction is along the negative z_v axis.

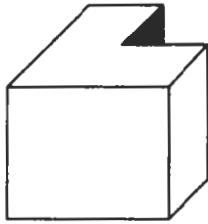


Figure 13-3
View of a concave polyhedron with one face partially hidden by other faces.

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.

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-

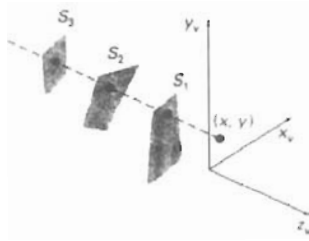


Figure 13-4

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) ,

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

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

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

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:

$$z = \frac{-Ax - By - D}{C} \quad (13-4)$$

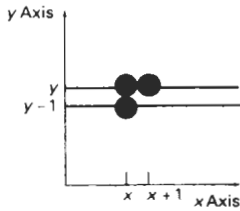


Figure 13-5
From position (x, y) on a scan line, the next position across the line has coordinates $(x + 1, y)$, and the position immediately below on the next line has coordinates $(x, y - 1)$.

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

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

or

$$z' = z - \frac{A}{C} \quad (13-6)$$

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

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

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-

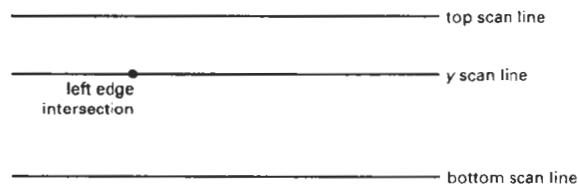


Figure 13-6
Scan lines intersecting a polygon surface.

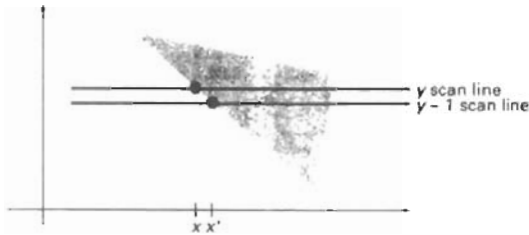


Figure 13-7

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.

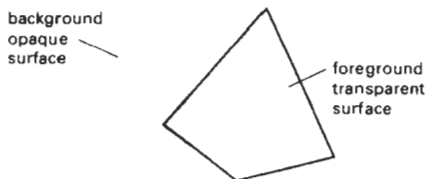


Figure 13-8

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

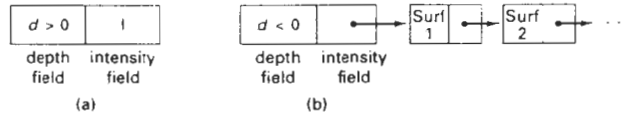


Figure 13-9
Organization of an A-buffer pixel position: (a) single-surface overlap of the corresponding pixel area, and (b) multiple-surface overlap.

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-

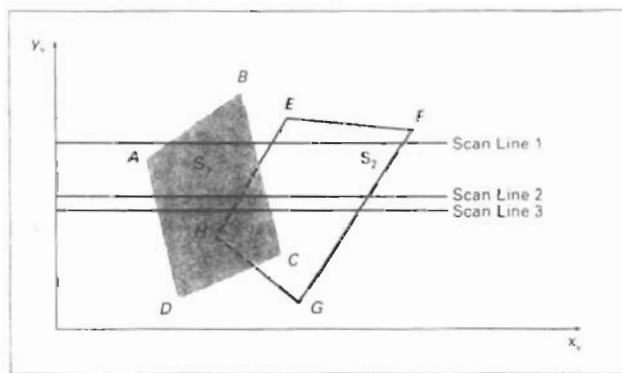


Figure 13-10

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.

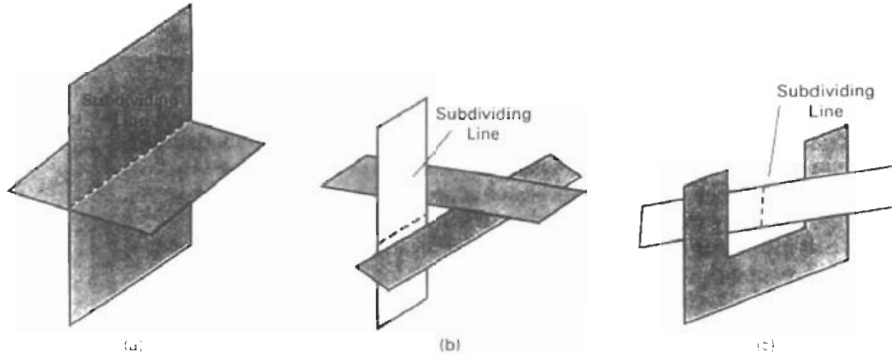


Figure 13-11

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

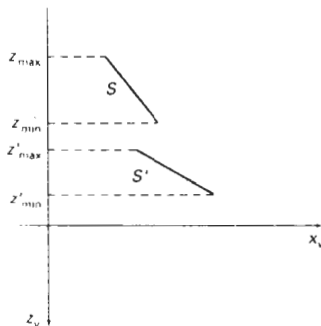


Figure 13-12
Two surfaces with no depth overlap.

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

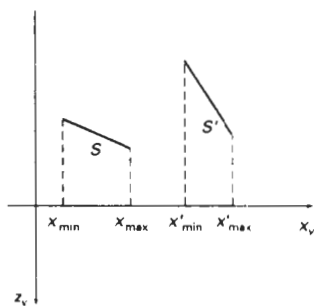


Figure 13-13
Two surfaces with depth overlap but no overlap in the x direction.

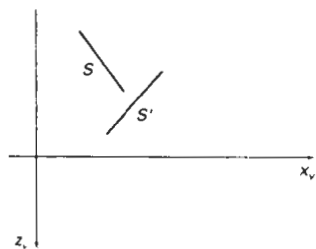


Figure 13-14
Surface S is completely behind ("inside") the overlapping surface S' .

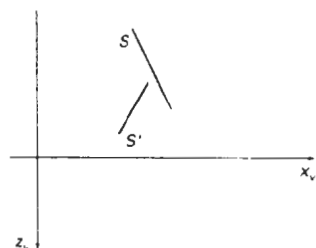


Figure 13-15
Overlapping surface S' is completely in front ("outside") of surface S , but S is not completely behind S' .

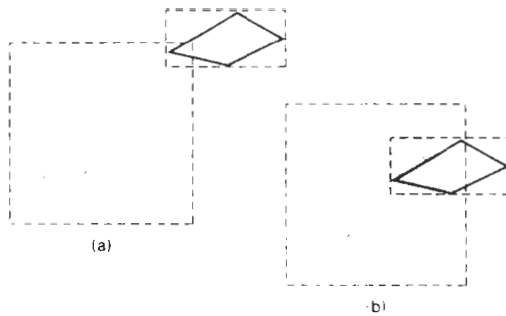


Figure 13-16
Two surfaces with overlapping bounding rectangles in the xy plane.

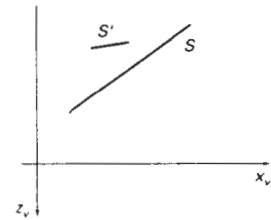


Figure 13-17
Surface S has greater depth but obscures surface S' .

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.

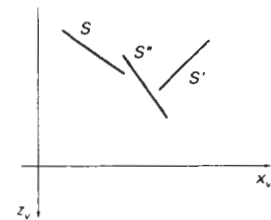


Figure 13-18
Three surfaces entered into the sorted surface list in the order S, S', S'' should be reordered S', S'', S .

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.

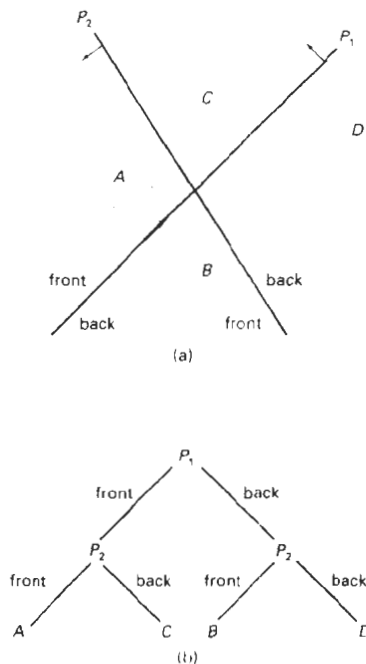


Figure 13-19
A region of space (a) is partitioned with two planes P_1 and P_2 to form the BSP tree representation in (b).

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 that 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.

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-

Section 13-8

Area-Subdivision Method

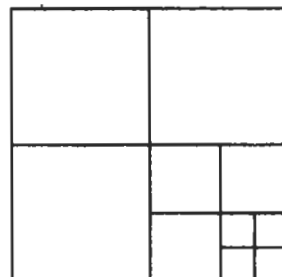


Figure 13-20

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

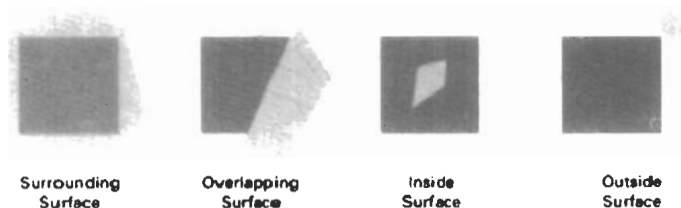


Figure 13-21

Possible relationships between polygon surfaces and a rectangular area.

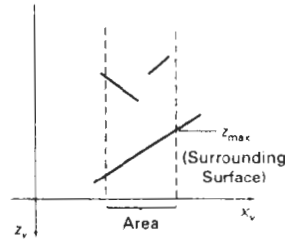


Figure 13-22
Within a specified area, a surrounding surface with a maximum depth of z_{max} obscures all surfaces that have a minimum depth beyond z_{max} .

imum 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.

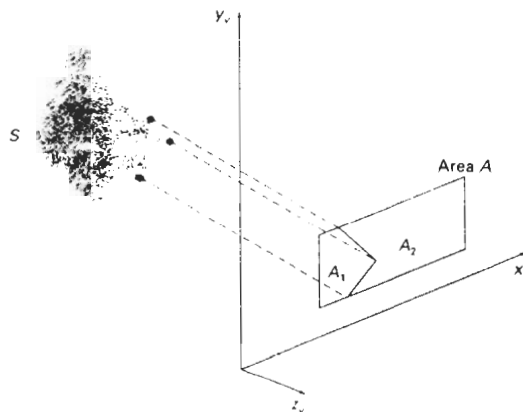


Figure 13-23
Area A is subdivided into A_1 and A_2 using the boundary of surface S on the view plane.

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

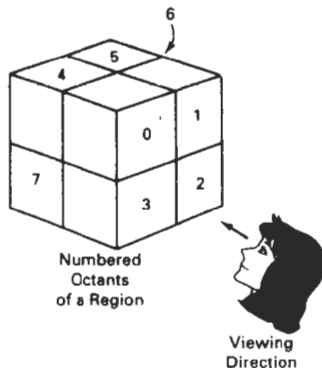
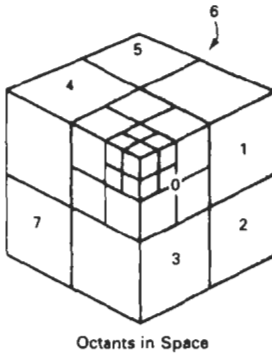
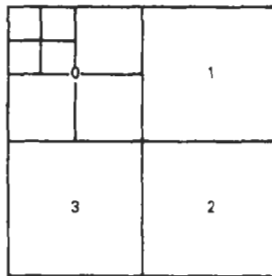


Figure 13-24
Objects in octants 0, 1, 2, and 3 obscure objects in the back octants (4, 5, 6, 7) when the viewing direction is as shown.



Octants in Space



Quadrants for
the View Plane

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

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.

```
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

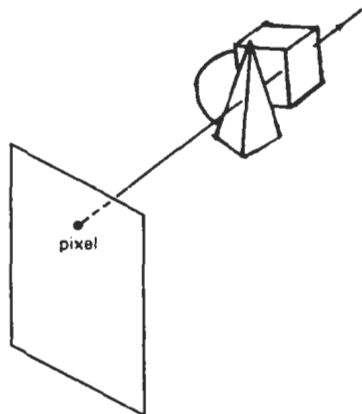


Figure 13-26

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.

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

$$y = f(x, z) \quad (13-8)$$

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

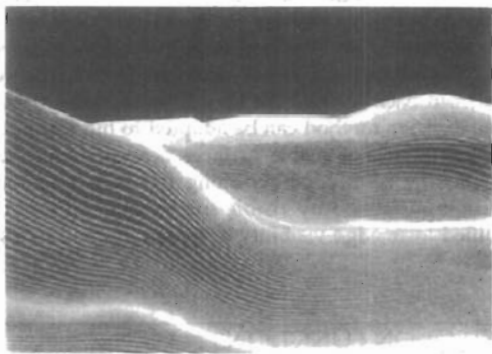


Figure 13-27
A color-coded surface contour plot. (Courtesy of Los Alamos National Laboratory.)

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.

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.

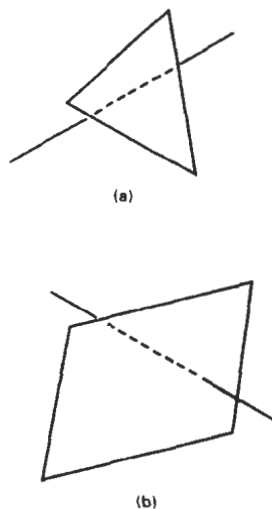


Figure 13-28
Hidden-line sections (dashed)
for a line that (a) passes behind
a surface and (b) penetrates a
surface.

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 `back-Face` 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:

Summary

```
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 perfor-

mance 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).

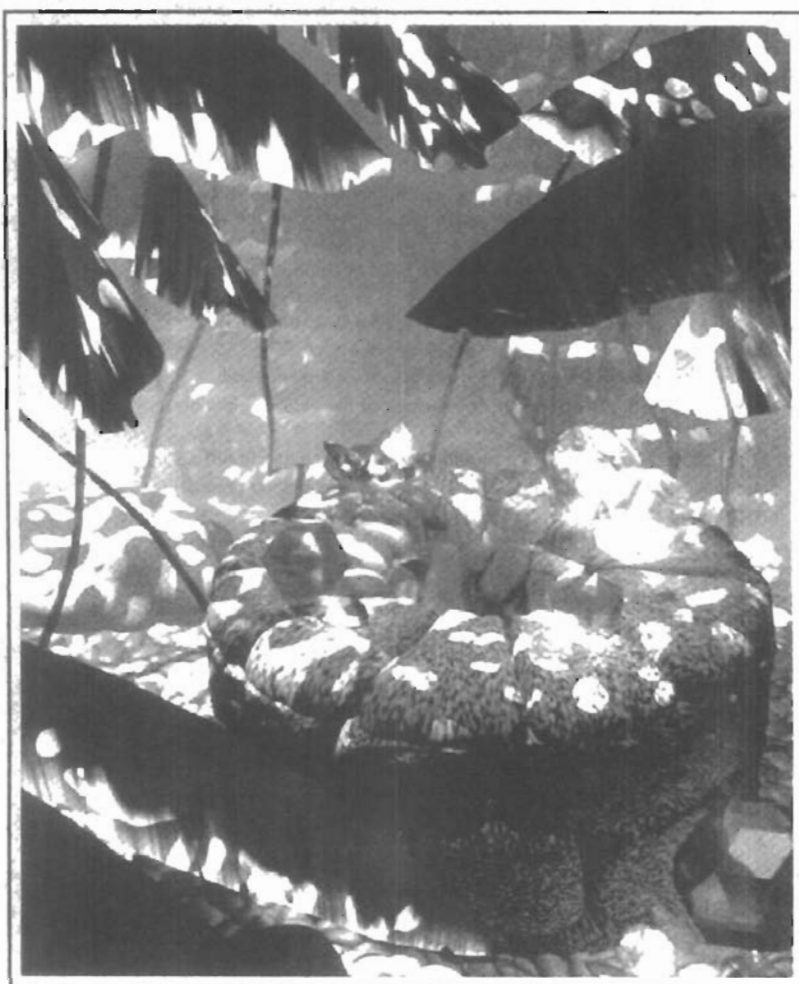
EXERCISES

- 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.

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 cal-

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

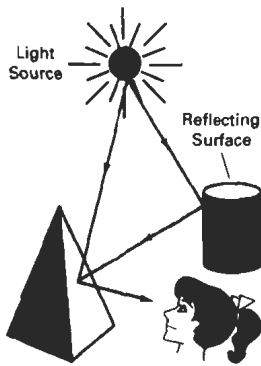


Figure 14-1
Light viewed from an opaque nonluminous surface is in general a combination of reflected light from a light source and reflections of light reflections from other surfaces.



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

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

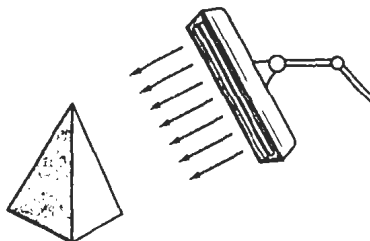


Figure 14-3
An object illuminated with a distributed light source.

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.

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-

Section 14-2

Basic Illumination Models



Figure 14-4
Diffuse reflections from a surface.

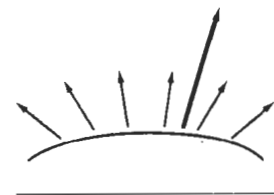


Figure 14-5
Specular reflection
superimposed on diffuse
reflection vectors.

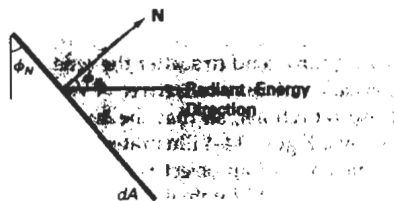


Figure 14-6
Radiant energy from a surface area dA in direction ϕ_N relative to the surface normal direction.

lected 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

$$I_{\text{ambdiff}} = k_d I_a \quad (14-1)$$

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.

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-

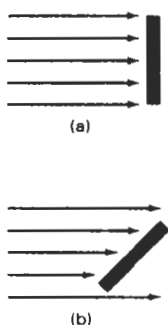


Figure 14-7
A surface perpendicular to the direction of the incident light (a) is more illuminated than an equal-sized surface at an oblique angle (b) to the incoming light direction.

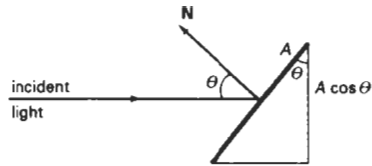


Figure 14-8
An illuminated area projected perpendicular to the path of the incoming light rays.

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_i is the intensity of the point light source, then the diffuse reflection equation for a point on the surface can be written as

$$I_{i,\text{diff}} = k_d I_i \cos \theta \quad (14-2)$$

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 $\cos \theta = N \cdot L$ and the diffuse reflection equation for single point-source illumination is

$$I_{i,\text{diff}} = k_d I_i (N \cdot L) \quad (14-3)$$

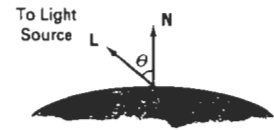


Figure 14-9
Angle of incidence θ between the unit light-source direction vector L and the unit surface normal N .

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

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

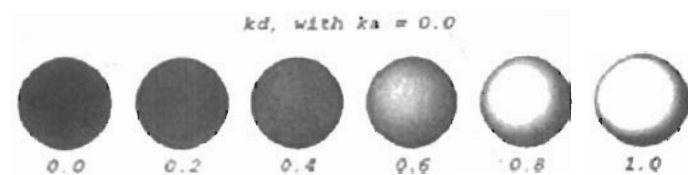


Figure 14-10
 Diffuse reflections from a spherical surface illuminated by a point light source for values of the diffuse reflectivity coefficient in the interval $0 \leq k_d \leq 1$.

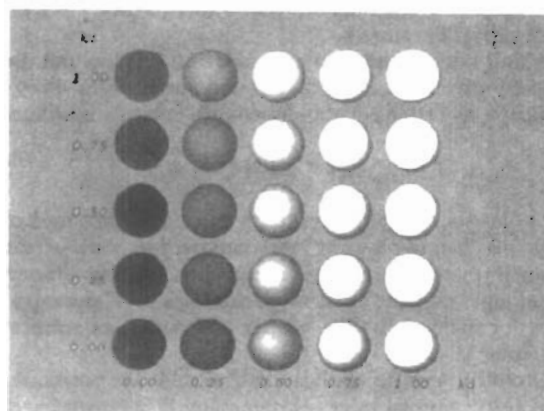


Figure 14-11
 Diffuse reflections from a spherical surface illuminated with ambient light and a single point source for values of k_a and k_d in the interval $(0, 1)$.

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 intensities 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 di-

reflections. 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 \mathbf{N} . In this figure, we use \mathbf{R} to represent the unit vector in the direction of ideal specular reflection; \mathbf{L} to represent the unit vector directed toward the point light source; and \mathbf{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 \mathbf{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 \mathbf{V} and \mathbf{R} coincide ($\phi = 0$).

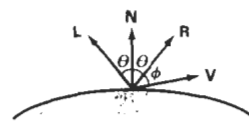


Figure 14-12
Specular-reflection angle equals angle of incidence θ .

Objects other than ideal reflectors exhibit specular reflections over a finite range of viewing positions around vector \mathbf{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 $\cos^n \phi$. 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 $\theta = 0^\circ$ to $\theta = 90^\circ$ for a few materials. In general, $W(\theta)$ tends to increase as the angle of incidence increases. At $\theta = 90^\circ$, $W(\theta) = 1$ 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

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

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

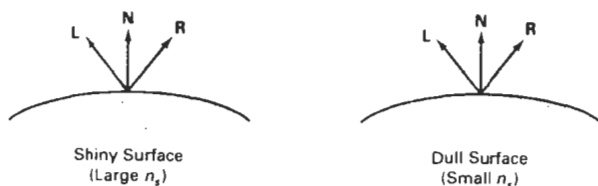


Figure 14-13
Modeling specular reflections (shaded area) with parameter n_s .

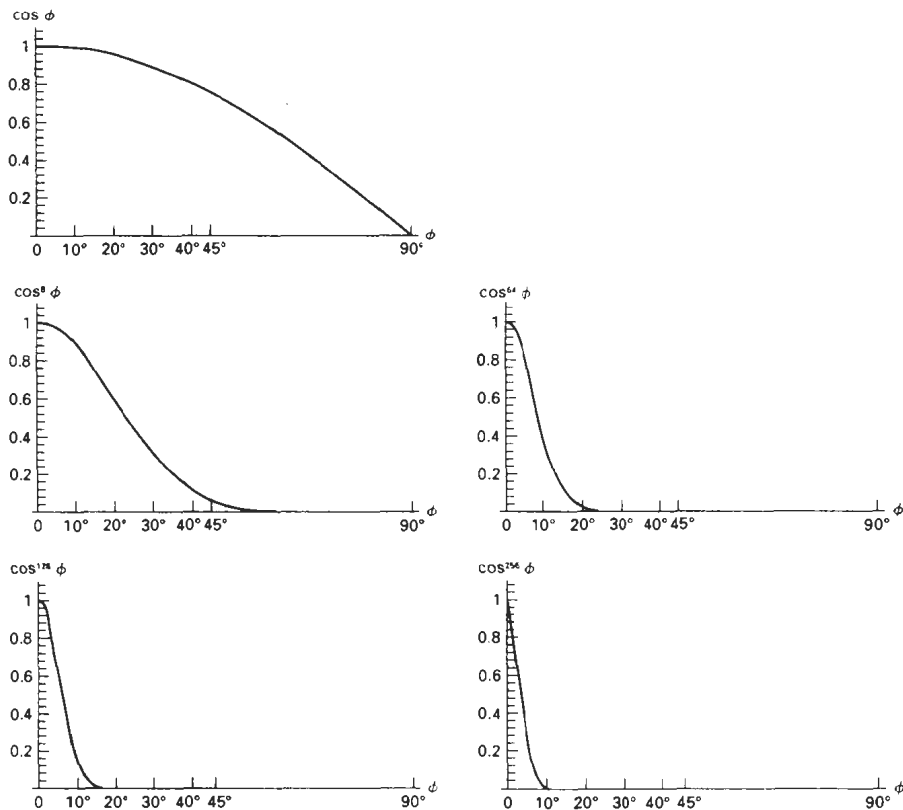


Figure 14-14

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 $\theta = 0^\circ$, 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 \mathbf{V} and \mathbf{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

$$I_{\text{spec}} = k_s I_i (\mathbf{V} \cdot \mathbf{R})^{n_s} \quad (14-10)$$

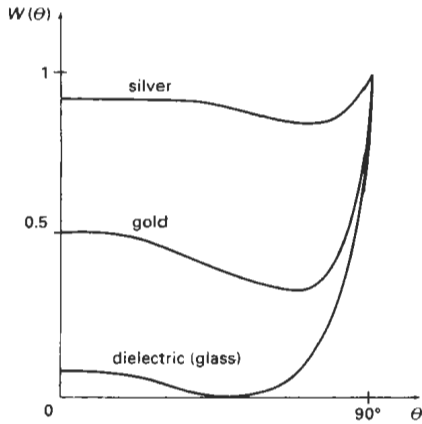


Figure 14-15
Approximate variation of the specular-reflection coefficient as a function of angle of incidence for different materials.

Vector \mathbf{R} in this expression can be calculated in terms of vectors \mathbf{L} and \mathbf{N} . As seen in Fig. 14-16, the projection of \mathbf{L} onto the direction of the normal vector is obtained with the dot product $\mathbf{N} \cdot \mathbf{L}$. 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

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

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* \mathbf{H} between \mathbf{L} and \mathbf{V} to calculate the range of specular reflections. If we replace $\mathbf{V} \cdot \mathbf{R}$ in the Phong model with the dot product $\mathbf{N} \cdot \mathbf{H}$, this simply replaces the empirical $\cos \phi$ calculation with the empirical $\cos \alpha$ calculation (Fig. 14-18). The halfway vector is obtained as

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{|\mathbf{L} + \mathbf{V}|} \quad (14-8)$$

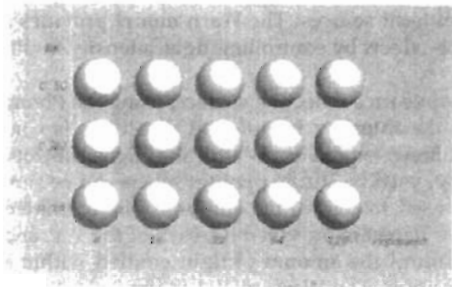


Figure 14-17
Specular reflections from a spherical surface for varying specular parameter values and a single light source.

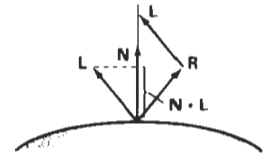


Figure 14-16
Calculation of vector \mathbf{R} by considering projections onto the direction of the normal vector \mathbf{N} .

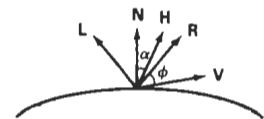


Figure 14-18
Halfway vector \mathbf{H} along the bisector of the angle between \mathbf{L} and \mathbf{V} .

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

For given light-source and viewer positions, vector \mathbf{H} is the orientation direction for the surface that would produce maximum specular reflection in the viewing direction. For this reason, \mathbf{H} is sometimes referred to as the surface orientation direction for maximum highlights. Also, if vector \mathbf{V} is coplanar with vectors \mathbf{L} and \mathbf{R} (and thus \mathbf{N}), angle α has the value $\phi/2$. When \mathbf{V} , \mathbf{L} , and \mathbf{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

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

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:

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

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

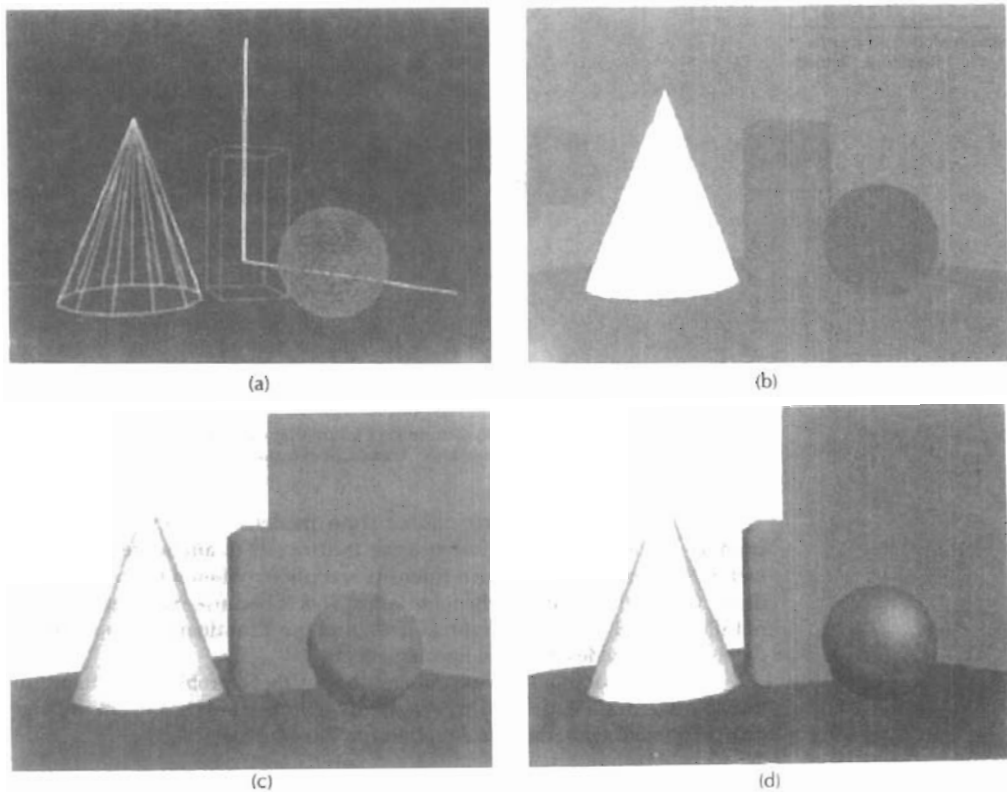


Figure 14-19

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

$$f(d) = \frac{1}{a_0 + a_1d + a_2d^2} \quad (14-11)$$

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

$$f(d) = \min\left(1, \frac{1}{a_0 + a_1d + a_2d^2}\right) \quad (14-12)$$

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

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

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

$$I_B = k_{dB}I_{dB} + \sum_{i=1}^n f_i(d)I_{Bi}[k_{dB}(N \cdot L_i) + k_{sB}(N \cdot H_i)^{n_s}] \quad (14-14)$$

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

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

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

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

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-consuming, 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*:

$$\sin \theta_r = \frac{\eta_i}{\eta_r} \sin \theta_i \quad (14-17)$$



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.)

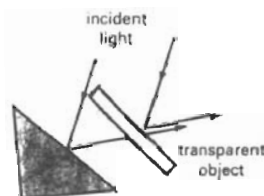


Figure 14-25
Light emission from a transparent surface is in general a combination of reflected and transmitted light.

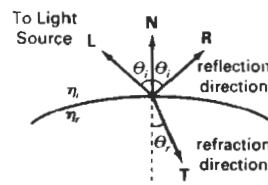


Figure 14-27
Reflection direction **R** and refraction direction **T** for a ray of light incident upon a surface with index of refraction η_r .

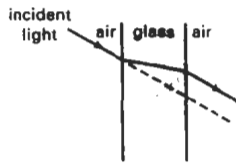


Figure 14-28
Refraction of light through a glass object. The emerging refracted ray travels along a path that is parallel to the incident light path (dashed line).

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.

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

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

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

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

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 stored 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

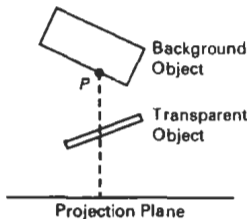


Figure 14-29
The intensity of a background object at point P can be combined with the reflected intensity off the surface of a transparent object along a perpendicular projection line (dashed).

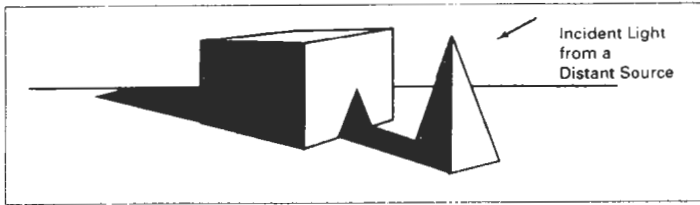


Figure 14-30
Objects modeled with shadow regions.

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:

$$\frac{I_1}{I_0} = \frac{I_2}{I_1} = \dots = \frac{I_n}{I_{n-1}} = r \quad (14-20)$$

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

$$I_k = r^k I_0 \quad (14-21)$$

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

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

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

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

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:

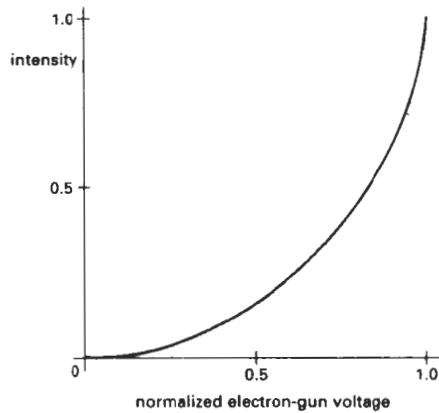


Figure 14-31

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

$$I_{Bk} = r_B^k I_{B0} \quad (14-24)$$

where

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

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 non-linearity 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

$$I = aV^\gamma \quad (14-26)$$

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

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

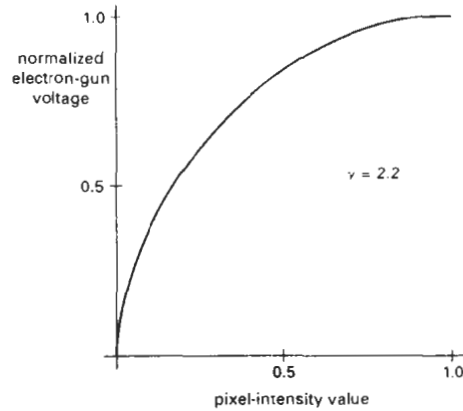


Figure 14-32

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

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

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:

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

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.

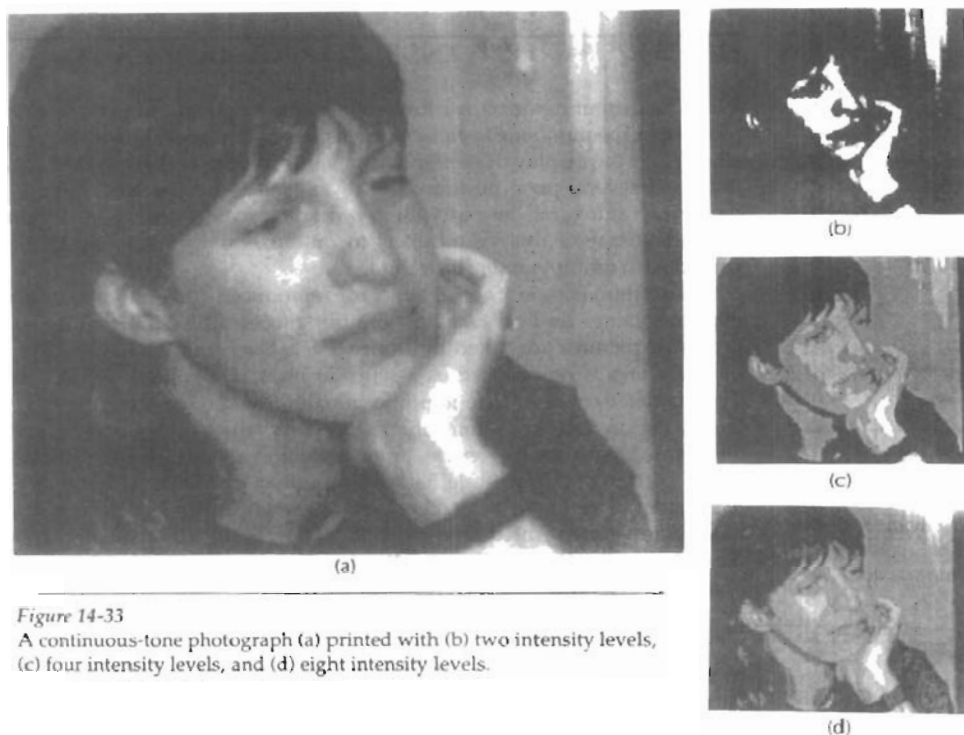


Figure 14-33

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.

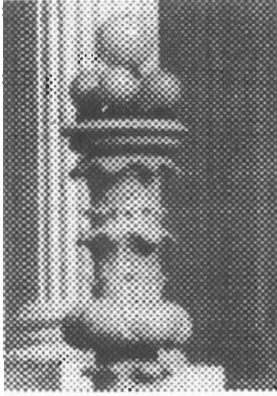


Figure 14-34

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

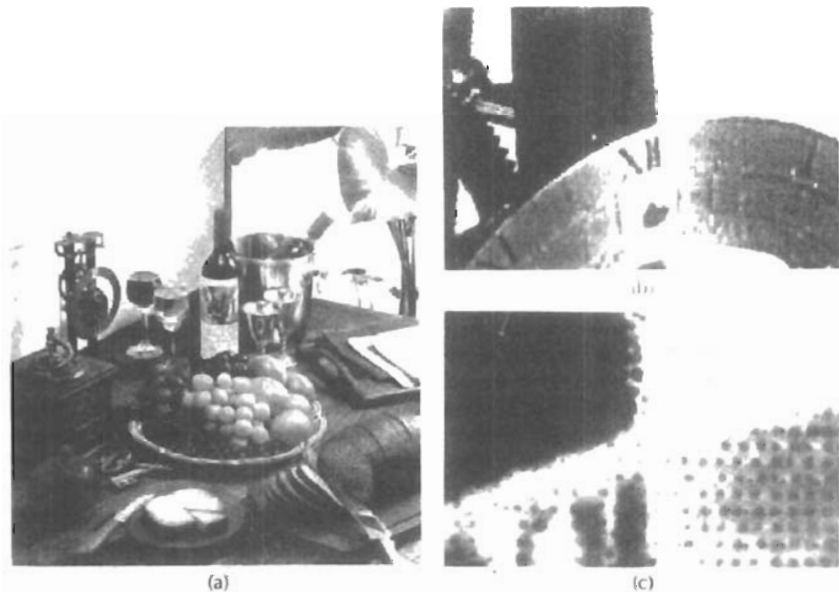


Figure 14-35

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).

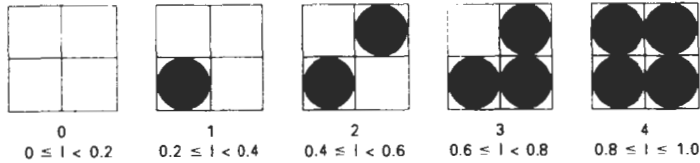


Figure 14-36

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.

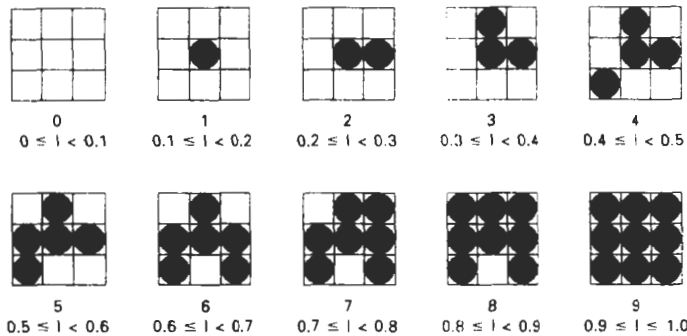


Figure 14-37

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 $0 \leq I < 0.2$, pattern 1 for $0.2 \leq I < 0.4$, and pattern 4 is used for $0.8 \leq I \leq 1.0$.

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.

$$\begin{bmatrix} 8 & 3 & 7 \\ 5 & 1 & 2 \\ 4 & 9 & 6 \end{bmatrix} \quad (14.30)$$

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.

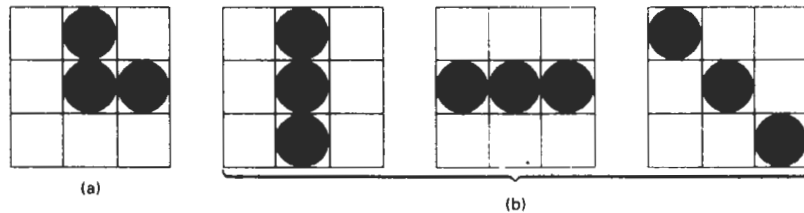


Figure 14-38

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.

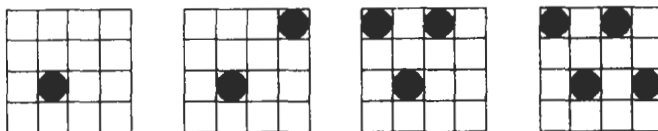


Figure 14-39
Halftone grid patterns with isolated pixels that cannot be effectively reproduced on some hard-copy devices.

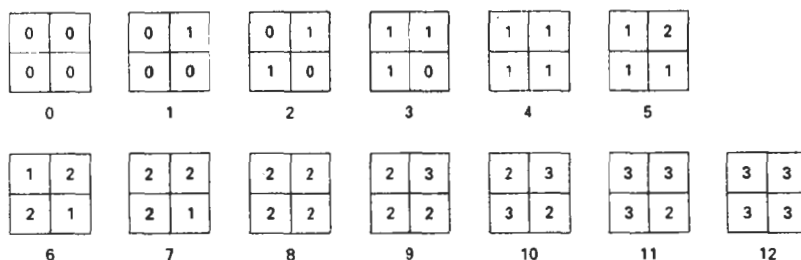


Figure 14-40
Intensity levels 0 through 12 obtained with halftone approximations using 2 by 2 pixel grids on a four-level system.

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.

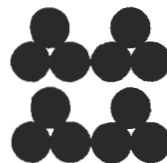


Figure 14-41
An RGB 2 by 2 pixel-grid pattern.

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 ran-

dom 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

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

and we can generate nine intensity levels with

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

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

$$i = (x \bmod n) + 1, \quad j = (y \bmod n) + 1 \quad (14-33)$$

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:

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

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

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

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  $M_{ij}$ . */
     $I_{ij} := I_k$ ;
     $err := M_{ij} - I_{ij}$ ;
     $M_{ij+1} := M_{ij+1} + \alpha \cdot err$ ;
     $M_{i+1,j-1} := M_{i+1,j-1} + \beta \cdot err$ ;
     $M_{i+1,j} := M_{i+1,j} + \gamma \cdot err$ ;
     $M_{i+1,j+1} := M_{i+1,j+1} + \delta \cdot err$ ;
  }

```

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

$$\alpha + \beta + \gamma + \delta \leq 1 \quad (14-36)$$

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-

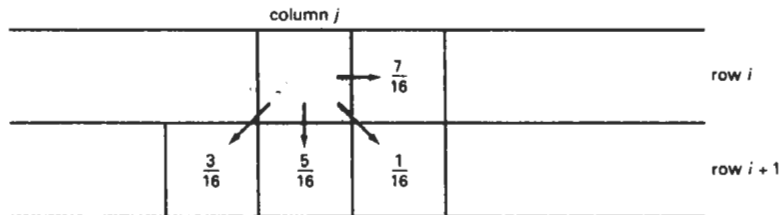


Figure 14-42

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.

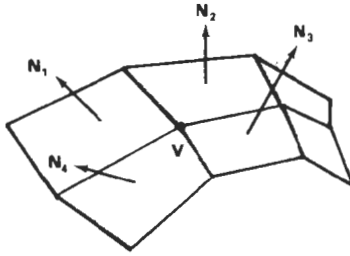


Figure 14-44

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 $N \cdot L$ and the attenuation function are constant over the surface.
- The viewing position is sufficiently far from the surface so that $V \cdot R$ 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

$$N_V = \frac{\sum_{k=1}^n N_k}{\left| \sum_{k=1}^n N_k \right|} \quad (14-37)$$

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:

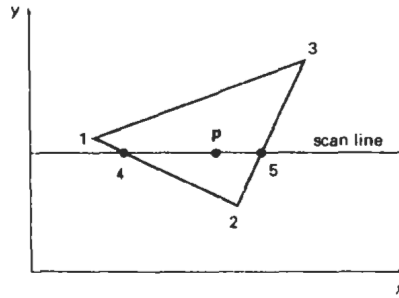


Figure 14-45

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.

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

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

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

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

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

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

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

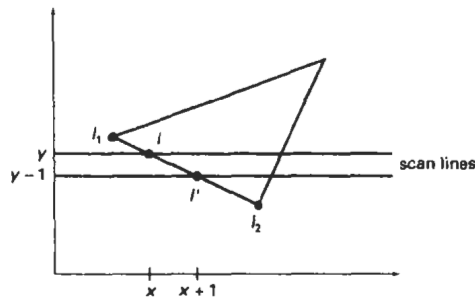


Figure 14-46

Incremental interpolation of intensity values along a polygon edge for successive scan lines.

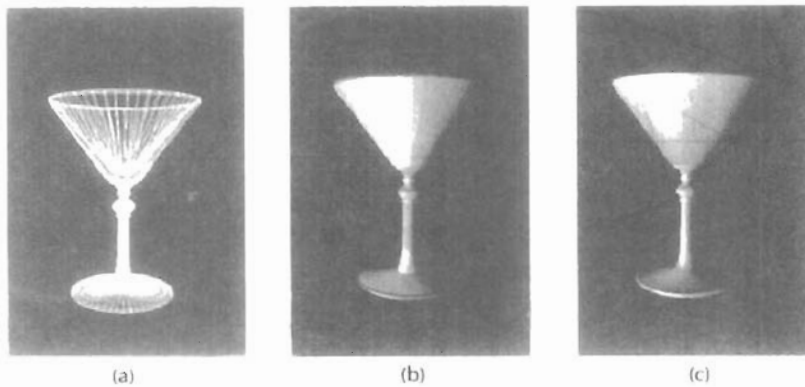


Figure 14-47

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:

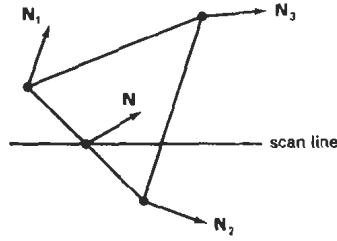


Figure 14-48
Interpolation of surface normals
along a polygon edge

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

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 \mathbf{N} at any point (x, y) over a triangle as

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

where vectors \mathbf{A} , \mathbf{B} , and \mathbf{C} are determined from the three vertex equations:

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

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

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

We can rewrite this expression in the form

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

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

$$a = \frac{\mathbf{L} \cdot \mathbf{A}}{|\mathbf{L}|} \quad (14-47)$$

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

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

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 $(\mathbf{N} \cdot \mathbf{H})^n$ 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.

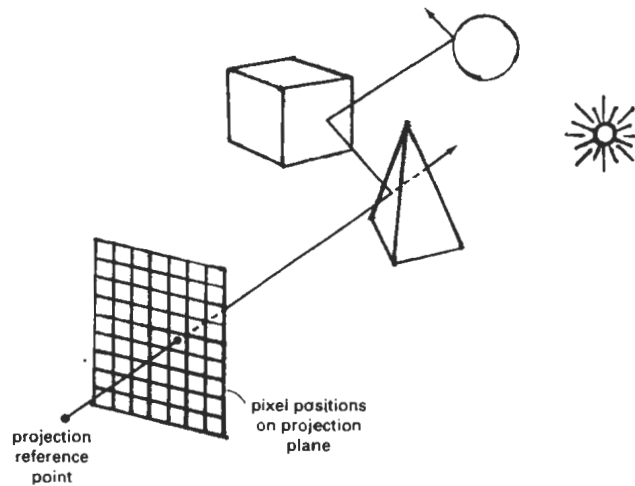


Figure 14-49

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.

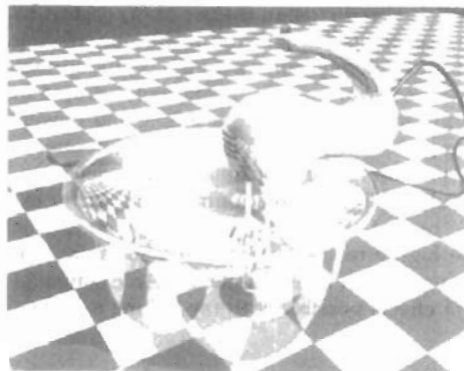


Figure 14-50

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

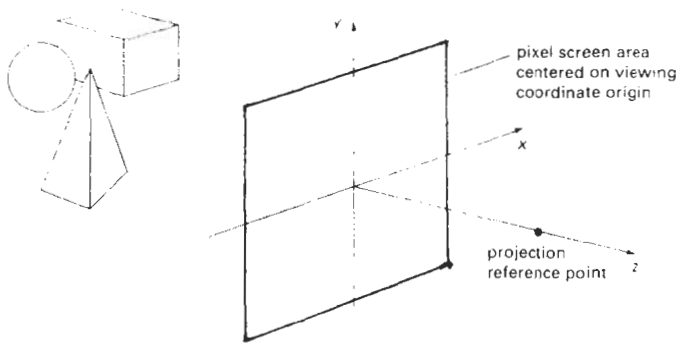


Figure 14-51
Ray-tracing coordinate-reference frame.

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 \mathbf{u} is in the direction of the ray path. \mathbf{N} is the unit surface normal, \mathbf{R} is the unit reflection vector, \mathbf{L} is the unit vector pointing to the light source, and \mathbf{H} is the unit vector halfway between \mathbf{V} (opposite to \mathbf{u}) and \mathbf{L} . The path along \mathbf{L} is referred to as the **shadow ray**. If any object intersects the shadow ray between the surface and the point light

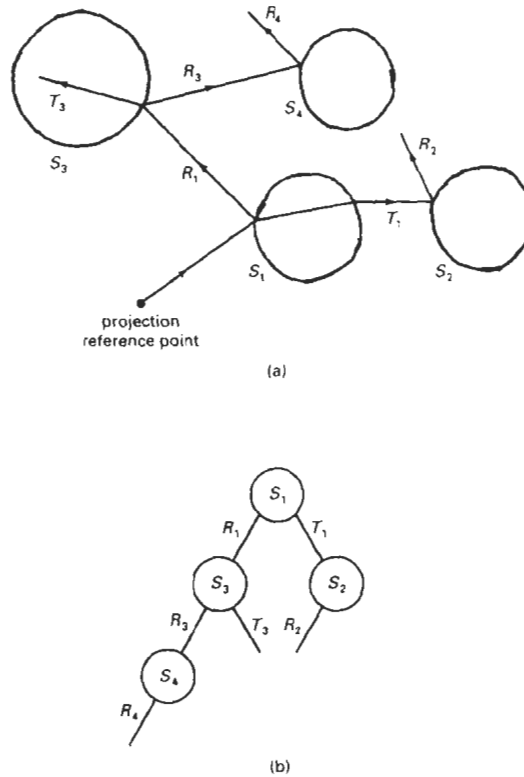


Figure 14-52
(a) Reflection and refraction ray paths through a scene for a screen pixel. (b) Binary ray-tracing tree for the paths shown in (a).

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 $k_s(\mathbf{H} \cdot \mathbf{N})^n$. As discussed in Section 14-2, the specular-reflection direction for the secondary ray path \mathbf{R} depends on the surface normal and the incoming ray direction:

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

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 \mathbf{T} , as shown in Fig. 14-54. The unit transmission vector can be obtained from vectors \mathbf{u} and \mathbf{N} as

$$\mathbf{T} = \frac{\eta_i}{\eta_r} \mathbf{u} - (\cos \theta_i - \frac{\eta_i}{\eta_r} \cos \theta_i) \mathbf{N} \quad (14-50)$$

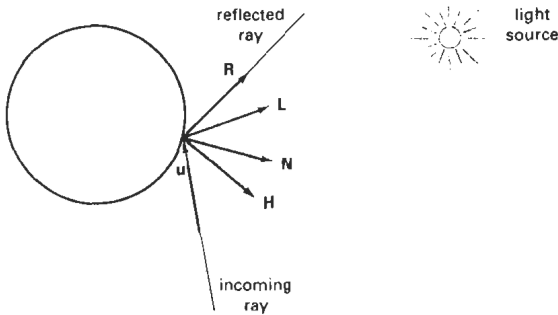


Figure 14-53
Unit vectors at the surface of an object intersected by an incoming ray along direction \mathbf{u} .

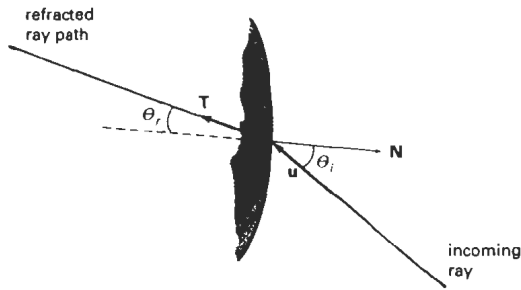


Figure 14-54
Refracted ray path T through a transparent material.

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:

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

Ray-Surface Intersection Calculations

A ray can be described with an initial position P_0 and unit direction vector \mathbf{u} , as illustrated in Fig. 14-55. The coordinates of any point P along the ray at a distance s from P_0 is computed from the ray equation:

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

Initially, 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 \mathbf{u} is initially ob-

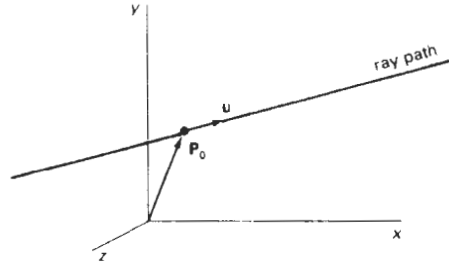


Figure 14-55
Describing a ray with an initial-
position vector \mathbf{P}_0 and unit
direction vector \mathbf{u} .

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

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

At each intersected surface, vectors \mathbf{P}_0 and \mathbf{u} are updated for the secondary rays at the ray-surface intersection point. For the secondary rays, reflection direction for \mathbf{u} is \mathbf{R} and the transmission direction is \mathbf{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 \mathbf{P} on the surface must satisfy the sphere equation:

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

Substituting the ray equation 14-52, we have

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

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

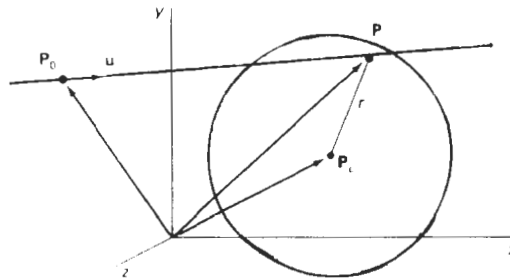


Figure 14-56
A ray intersecting a sphere with radius r centered on
position \mathbf{P}_c .

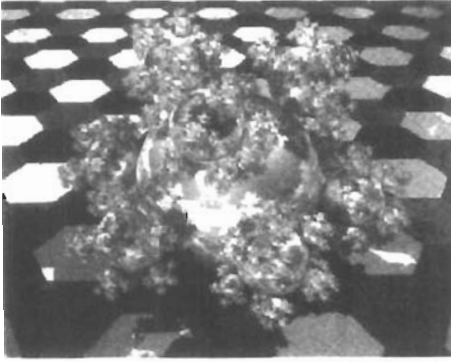


Figure 14-57
A "sphereflake" rendered with ray tracing using 7381 spheres and 3 light sources. (Courtesy of Eric Haines, 3D/EYE Inc.)

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

whose solution is

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

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

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

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

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

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

$$\mathbf{u} \cdot \mathbf{N} < 0 \quad (14-59)$$

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

$$\mathbf{N} \cdot \mathbf{P} = -D \quad (14-60)$$

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

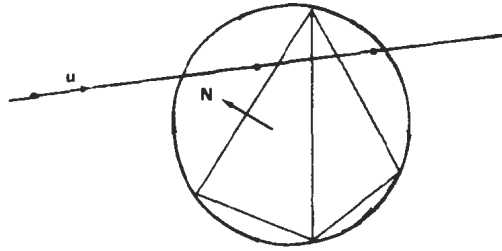


Figure 14-58
Polyhedron enclosed by a bounding sphere.

and D is the fourth plane parameter. Position P is both on the plane and on the ray path if

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

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

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

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

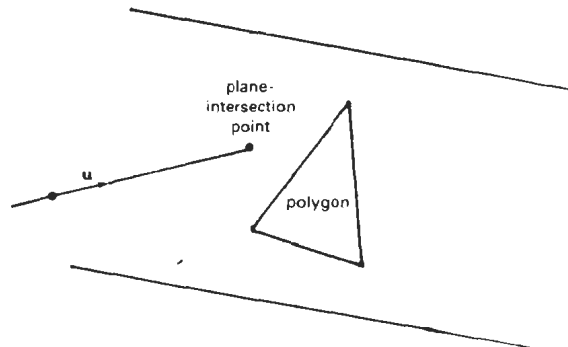


Figure 14-59
Ray intersection with the plane of a polygon.

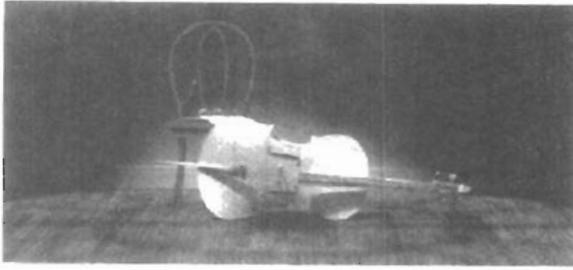


Figure 14-60
A ray-traced scene showing global reflection of surface-texture patterns. (Courtesy of Sun Microsystems.)

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

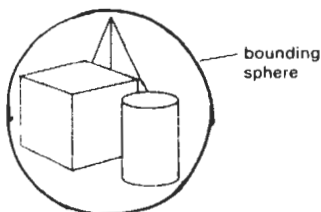


Figure 14-61
A group of objects enclosed within a bounding sphere.

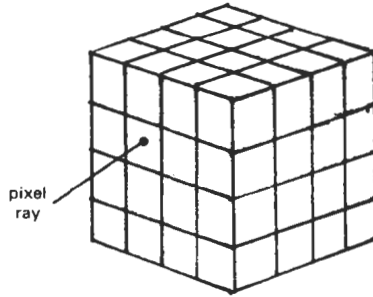


Figure 14-62
Ray intersection with a cube
enclosing all objects in a scene.

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}_{in} for a cell, the potential exit faces are those for which

$$\mathbf{u} \cdot \mathbf{N}_k > 0 \quad (14-63)$$

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

$$\mathbf{N}_k = \begin{cases} (\pm 1, 0, 0) \\ (0, \pm 1, 0) \\ (0, 0, \pm 1) \end{cases}$$

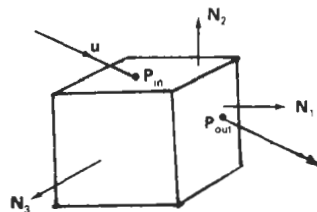


Figure 14-63
Ray traversal through a subregion
(cell) of a cube enclosing a scene.

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

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

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

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

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

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

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

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

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 \mathbf{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.

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.

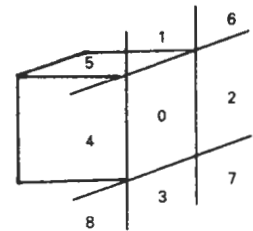


Figure 14-64
Sectors of the trial exit plane.

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,

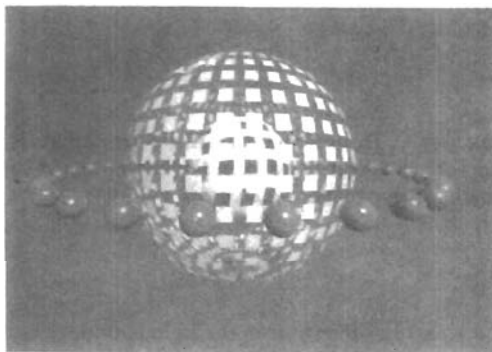


Figure 14-65

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. Hubbard, Department of Computer Science, University of Manchester.)

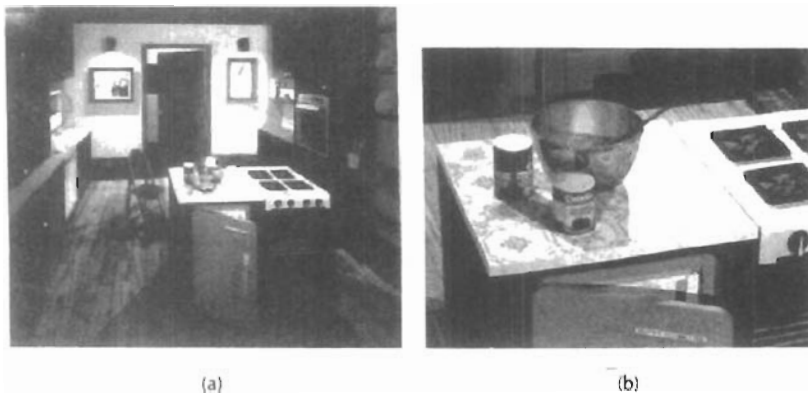


Figure 14-67

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.)

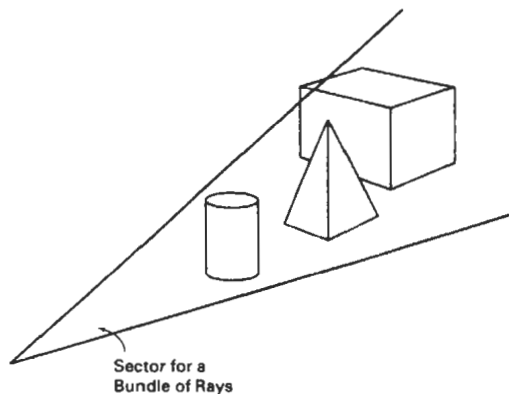


Figure 14-68

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

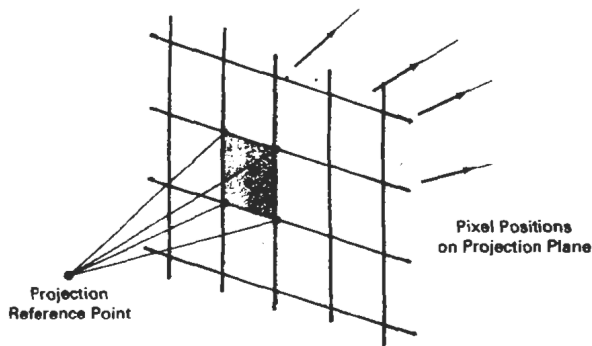


Figure 14-69
Supersampling with four rays per pixel, one at each pixel corner.

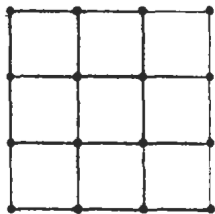


Figure 14-70
Subdividing a pixel into nine subpixels with one ray at each subpixel corner.

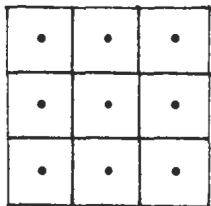


Figure 14-71
Ray positions centered on subpixel areas.

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-

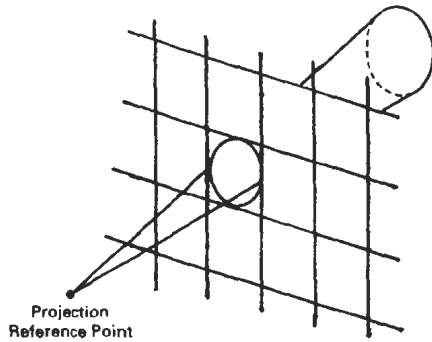


Figure 14-72
A pixel ray cone.

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 $(x + \delta_x, y + \delta_y)$.

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 \mathbf{R} according to the assigned ray codes (Fig. 14-75). The

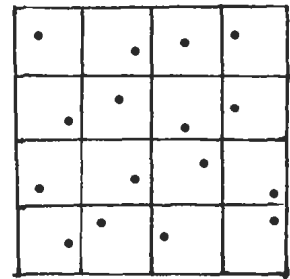


Figure 14-73
Pixel sampling using 16 subpixel areas and a jittered ray position from the center coordinates for each subarea.

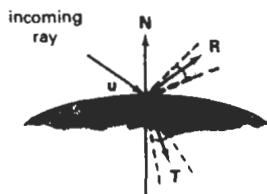


Figure 14-75
Distributing subpixel rays
about the reflection direction
R and the transmission
direction T.

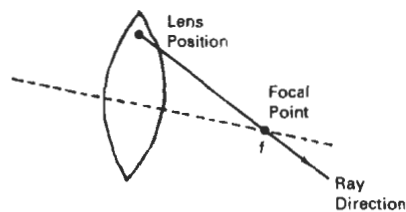


Figure 14-74
Distributing subpixel rays over a
camera lens of focal length f .

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, $\cos^n \phi$, 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

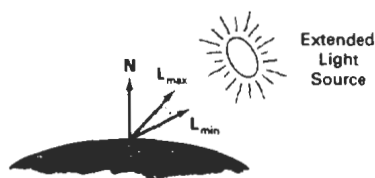


Figure 14-76
Distributing shadow rays over a
finite-sized light source.

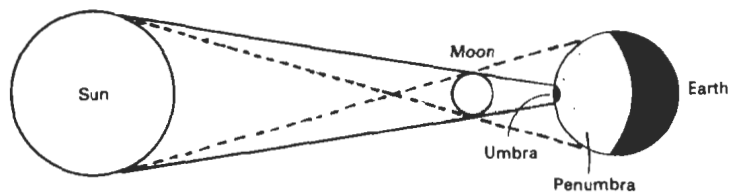


Figure 14-77
Umbra and penumbra regions created by a solar eclipse on the surface
of the earth.

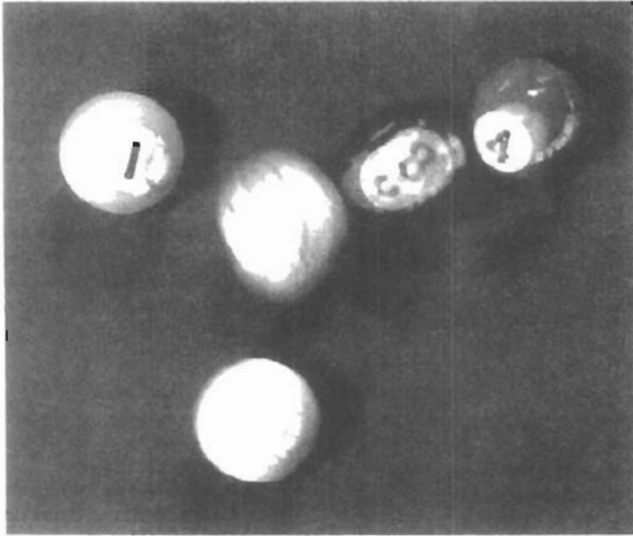


Figure 14-78
A scene, entitled *1984*, rendered with distributed ray tracing, illustrating motion-blur and penumbra effects. (Courtesy of Pixar. © 1984 Pixar. All rights reserved.)

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 bounding 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.)

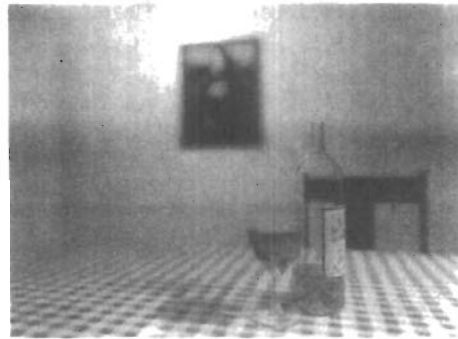


Figure 14-81

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 $\text{joules}/(\text{second} \cdot \text{meter}^2)$, or $\text{watts}/\text{meter}^2$.

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 $\text{watts}/(\text{meter}^2 \cdot \text{steradians})$:

$$I = \frac{dB}{d\omega \cos \phi} \quad (14-68)$$

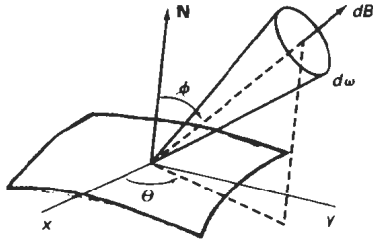


Figure 14-82
Visible radiant energy emitted from a surface point in direction (θ, ϕ) within solid angle $d\omega$.

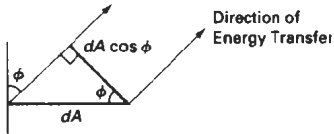


Figure 14-83
For a unit surface element, the projected area perpendicular to the direction of energy transfer is equal to $\cos \phi$.

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:

$$B = \int_{\text{hemi}} dB \quad (14-69)$$

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

$$B = I \int_{\text{hemi}} \cos \phi d\omega \quad (14-70)$$

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

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

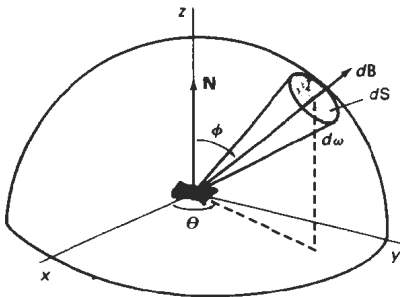


Figure 14-84
Total radiant energy from a surface point is the sum of the contributions in all directions over a hemisphere centered on the surface point.

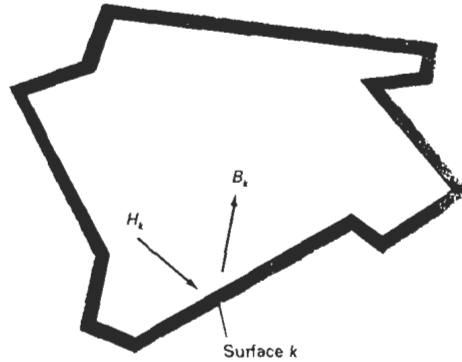


Figure 14-85
An enclosure of surfaces for the radiosity model.

so that

$$B = I \int_0^{2\pi} \int_0^{\pi/2} \cos \phi \sin \phi \, d\phi \, d\theta = I\pi \quad (14-71)$$

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,

$$H_k = \sum_j B_j F_{jk} \quad (14-72)$$

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**:

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

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/meter²*). 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

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

or

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

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

$$dB_j dA_k = (I_j \cos \phi_j d\omega) dA_k, \quad (14-76)$$

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

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

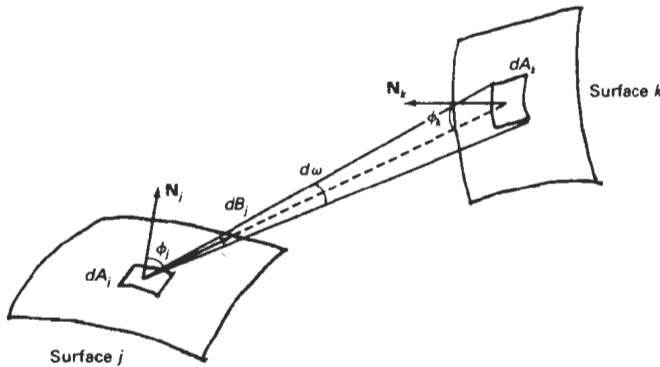


Figure 14-86
Rate of energy transfer dB_j from a surface element with area dA_j to surface element dA_k .

so we can express Eq. 14-76 as

$$dB_j dA_j = \frac{I_j \cos \phi_j \cos \phi_k dA_j dA_k}{r^2} \quad (14-78)$$

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

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

Also $B_j = \pi I_j$, so that

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

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

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

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:

$$F_{ik} = \frac{1}{A_i} \int_{\text{surf}_i} \int_{\text{surf}_k} \frac{\cos \phi_i \cos \phi_k}{\pi r^2} dA_k dA_i \quad (14-82)$$

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)
- $F_{ij} = 0$, 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 lin-

ear 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 renderings, 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

$$B_k \text{ due to } B_i = \rho_k B_i F_{ik} \quad (14-83)$$

Reciprocally,

$$B_i \text{ due to } B_k = \rho_i B_k F_{ki}, \quad \text{for all } j \quad (14-84)$$

which we can rewrite as

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

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  $F_{ik}$  */

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

 $\Delta B_k = 0$ ;

```

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**

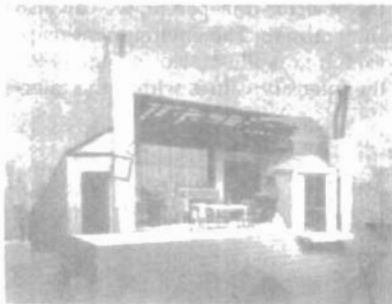
Image of a constructivist museum rendered with a progressive-refinement radiosity method.

(Courtesy of Shenchang Eric Chen, Stuart I. Feldman, and Julie Dorsey, Program of Computer Graphics, Cornell University. © 1988, Cornell University, Program of Computer Graphics.)

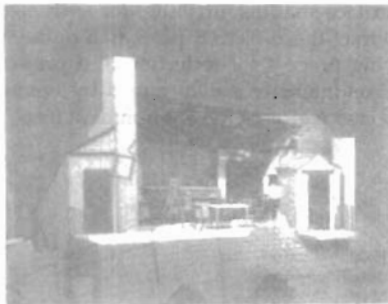
**Figure 14-89**

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.)



(a)



(b)

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.)

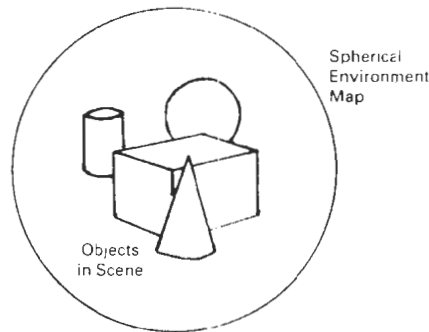


Figure 14-91
A spherical enclosing universe
containing the environment map

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.

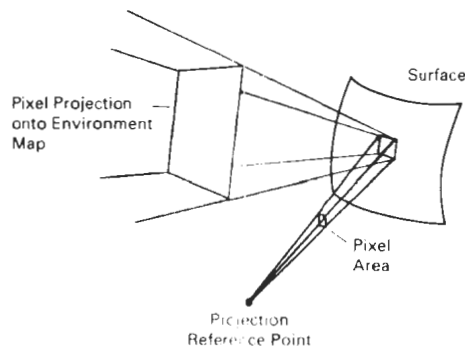


Figure 14-92
Projecting a pixel area to a surface,
then reflecting the area to the
environment map.

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.

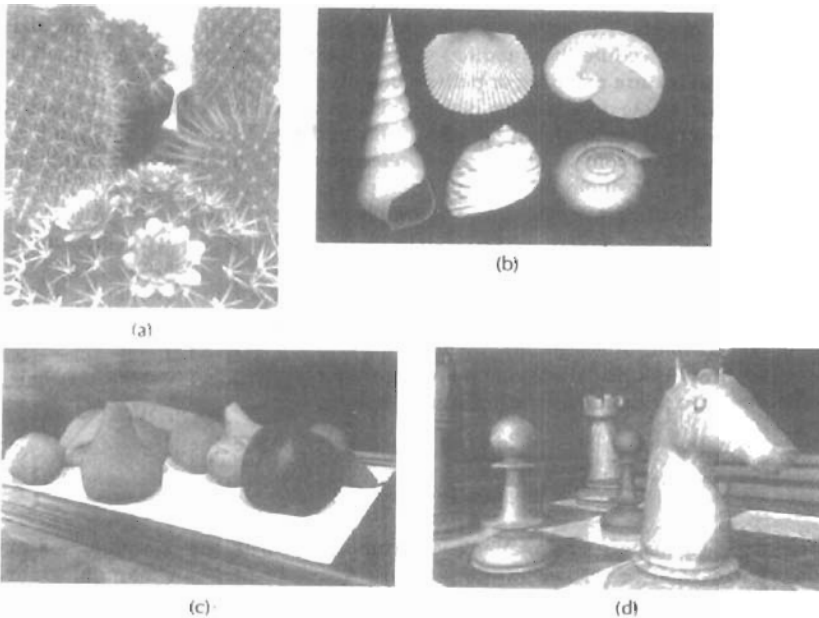


Figure 14-93
Scenes illustrating computer graphics generation of surface detail.
((a) © 1992 Deborah R. Fowler, Przemyslaw Prusinkiewicz, and Johannes Battjes;
(b) © 1992 Deborah R. Fowler, Hans Meinhardt, and Przemyslaw Prusinkiewicz,
University of Calgary; (c) and (d) Courtesy of SOFTIMAGE, Inc.)

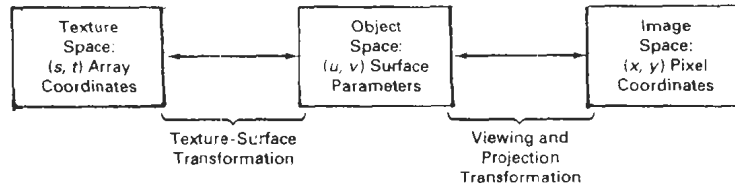


Figure 14-94
Coordinate reference systems for texture space, object space, and image space.

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

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

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-

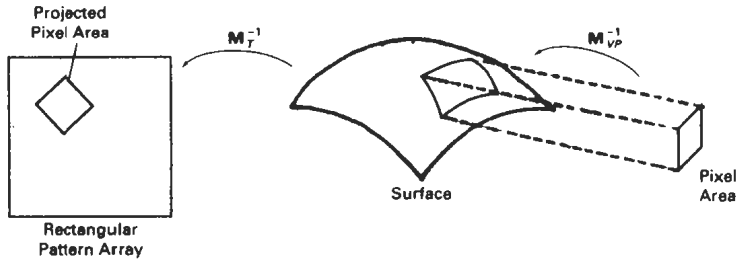


Figure 14-95
Texture mapping by projecting pixel areas to texture space.

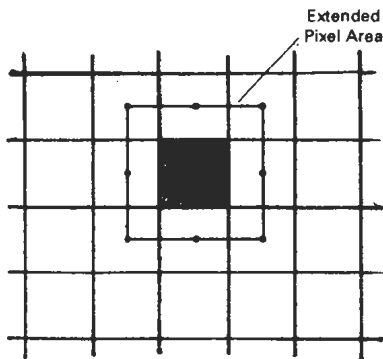


Figure 14-96
Extended area for a pixel that includes centers of adjacent pixels.

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 M_{VP}^{-1} and the inverse texture-map transformation 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

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

with

$$0 \leq \theta \leq \pi/2, \quad 0 \leq z \leq 1$$

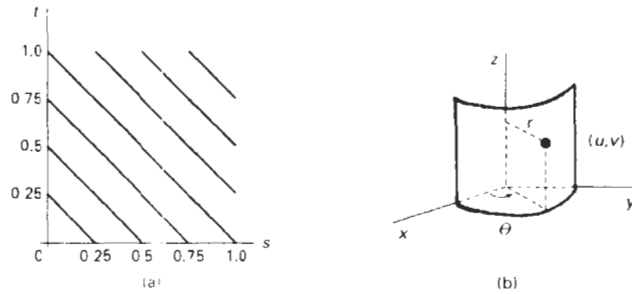


Figure 14-97
Mapping a texture pattern defined on a unit square (a) to a cylindrical surface (b).

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

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

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.

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

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

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

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

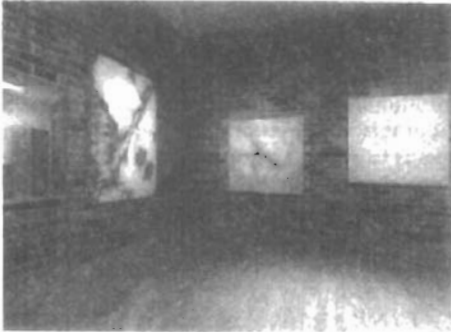
$$s = 2u/\pi, \quad t = v$$

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 technique 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

$$\mathbf{N} = \mathbf{P}_u \times \mathbf{P}_v \quad (14-87)$$

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*:

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

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

$$\mathbf{N}' = \mathbf{P}'_u \times \mathbf{P}'_v \quad (14-89)$$

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

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

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

$$\mathbf{P}'_u \approx \mathbf{P}_u + b_u\mathbf{n} \quad (14-91)$$

Similarly,

$$\mathbf{P}'_v \approx \mathbf{P}_v + b_v\mathbf{n} \quad (14-92)$$

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

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

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

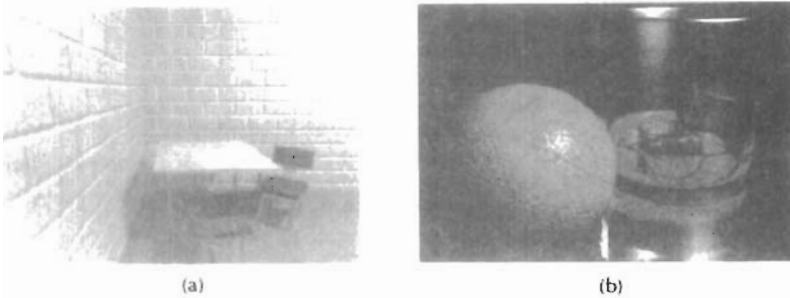


Figure 14-100

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 antialias, 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

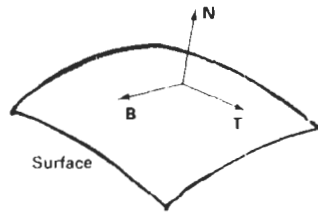


Figure 14-102
A local coordinate system at a
surface point.

N The local coordinates are defined with a surface-tangent vector **T** and a binormal vector **B** $\mathbf{B} = \mathbf{T} \times \mathbf{N}$

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 display 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 con-

stant-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

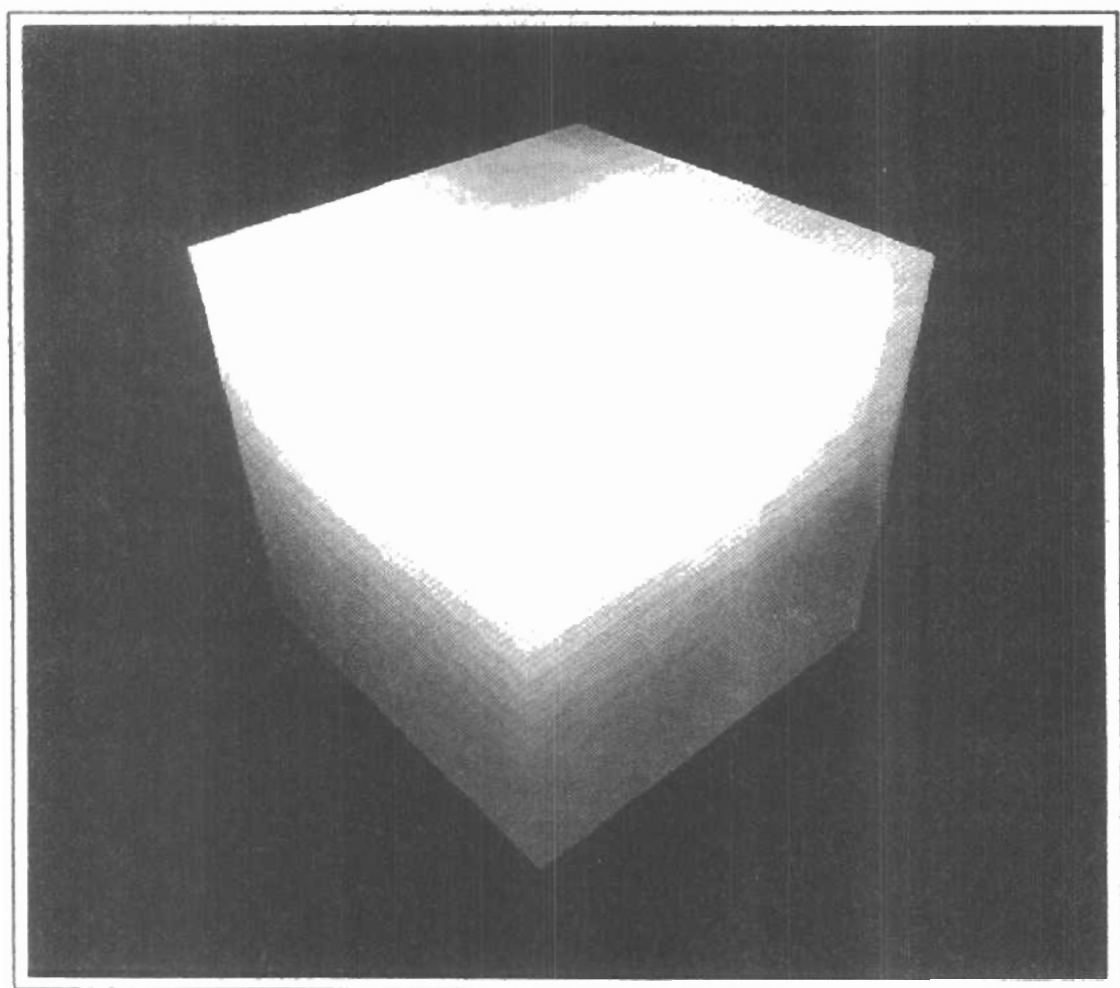
REFERENCES

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 $(\mathbf{N} \cdot \mathbf{H})^n$ compared to specular reflections modeled with $(\mathbf{V} \cdot \mathbf{R})^n$.
- 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.



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.

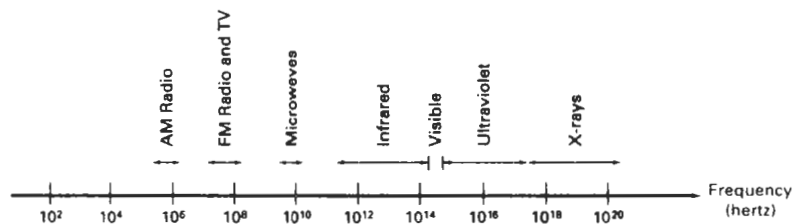


Figure 15-1
Electromagnetic spectrum.

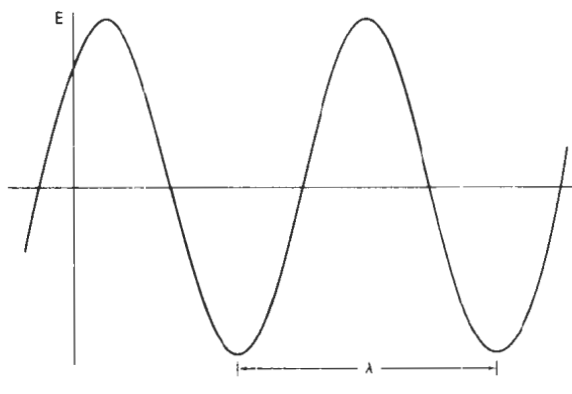


Figure 15-2
Time variations for one electric frequency component of a plane-polarized electromagnetic wave.

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 :

$$c = \lambda f \quad (15-1)$$

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 ($1\text{\AA} = 10^{-8}$ cm) or nanometers ($1\text{ nm} = 10^{-7}$ 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

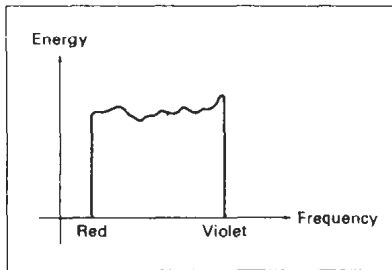


Figure 15-3
Energy distribution of a white-light source.

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-

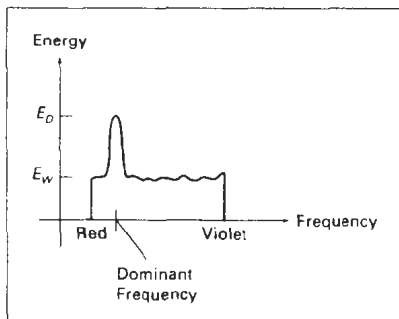


Figure 15-4
Energy distribution of a light source with a dominant frequency near the red end of the frequency range.

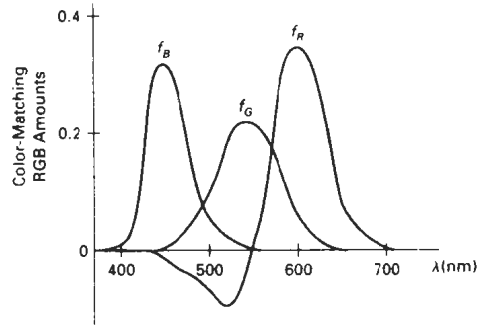


Figure 15-5
Amounts of RGB primaries needed to display
spectral colors.

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

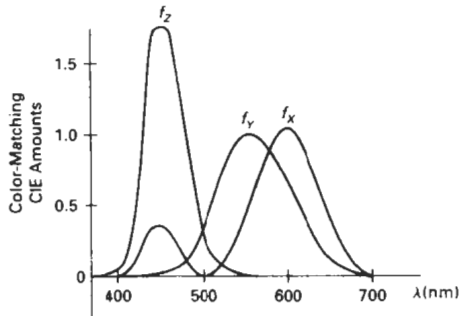


Figure 15-6
Amounts of CIE primaries needed
to display spectral colors.

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

$$C_\lambda = XX + YY + ZZ \quad (15-2)$$

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

$$x = \frac{X}{X + Y + Z}, \quad y = \frac{Y}{X + Y + Z}, \quad z = \frac{Z}{X + Y + Z} \quad (15-3)$$

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

$$X = \frac{x}{y}Y, \quad Z = \frac{z}{y}Y \quad (15-4)$$

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

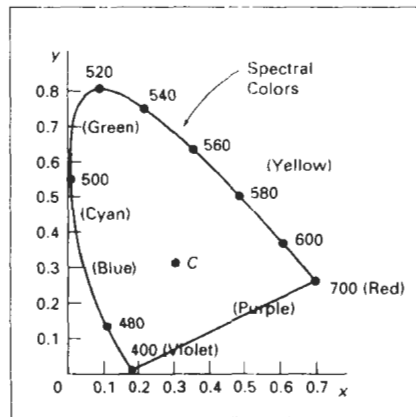


Figure 15-7
CIE chromaticity diagram. Spectral
color positions along the curve are
labeled in wavelength units (nm).

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

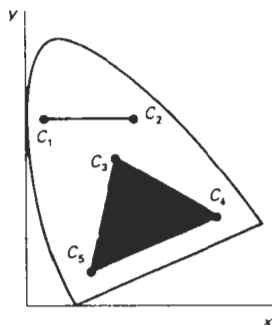


Figure 15-8
Color gamuts defined on the chromaticity diagram for a two-color and a three-color system of primaries.

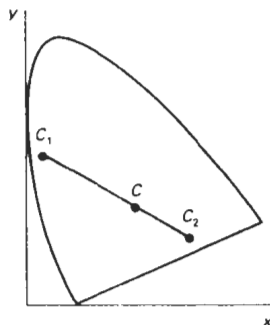


Figure 15-9
Representing complementary colors on the chromaticity diagram.

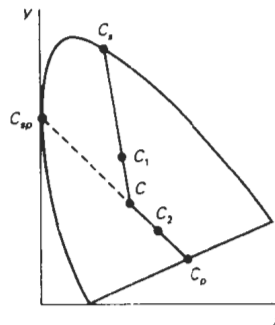


Figure 15-10
Determining dominant wavelength and purity with the chromaticity diagram.

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 complement 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_i is expressed in RGB components as

$$C_i = RR + GG + BB \quad (15-5)$$

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

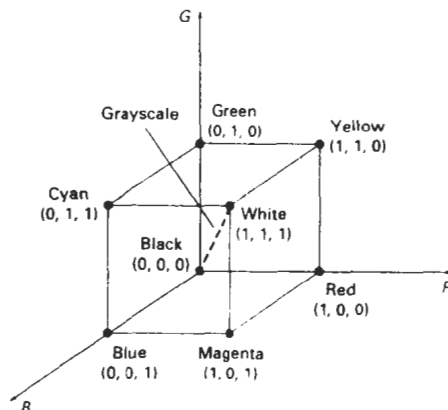


Figure 15-11
The RGB color model, defining colors with an additive process within the unit cube.

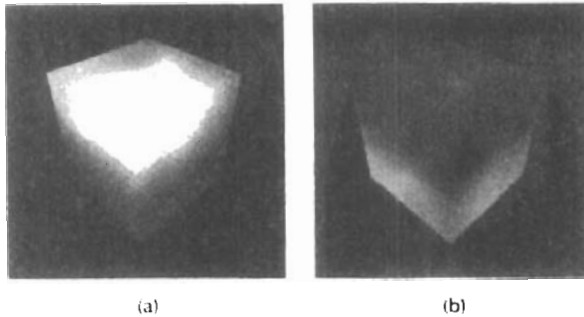


Figure 15-12
Two views of the RGB color cube: (a) along the grayscale diagonal from white to black and (b) along the grayscale diagonal from black to white.

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)

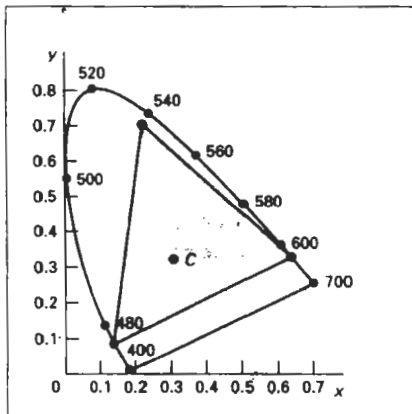


Figure 15-13
RGB color gamut.

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

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

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:

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

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.

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

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

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

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

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

Section 15-7

HSV Color Model

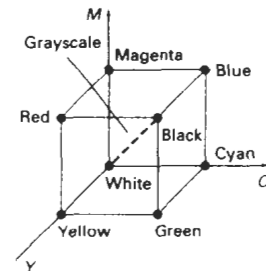


Figure 15-14

The CMY color model, defining colors with a subtractive process inside a unit cube.

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).

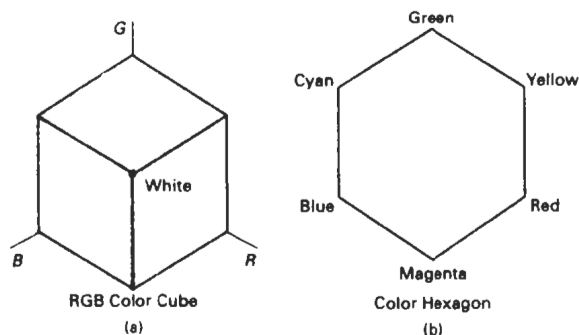


Figure 15-15
When the RGB color cube (a) is viewed along the diagonal from white to black, the color-cube outline is a hexagon (b).

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 $H = 180^\circ$. Complementary colors are 180° apart.

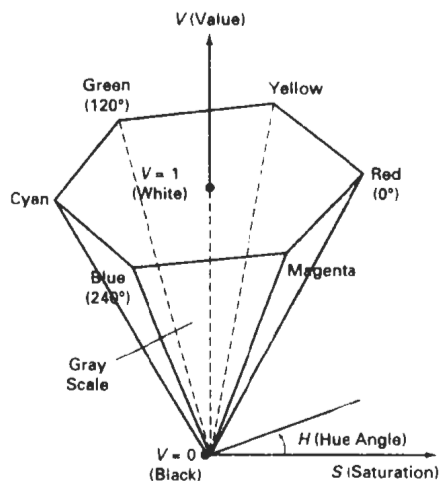


Figure 15-16
The HSV hexcone.

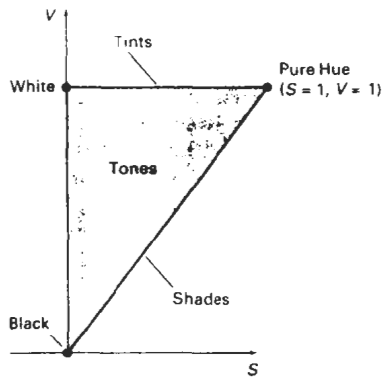


Figure 15-17
Cross section of the HSV hexcone,
showing regions for shades, tints,
and tones.

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 $H = 240^\circ$. 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 $H = 240^\circ$. 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.

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, $H = 0^\circ$ 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 $H = 180^\circ$. 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.

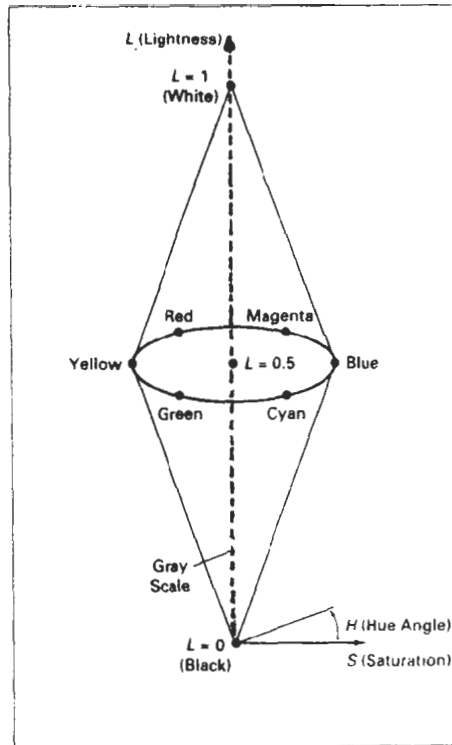


Figure 15-18
The HLS double cone.

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.

References

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.

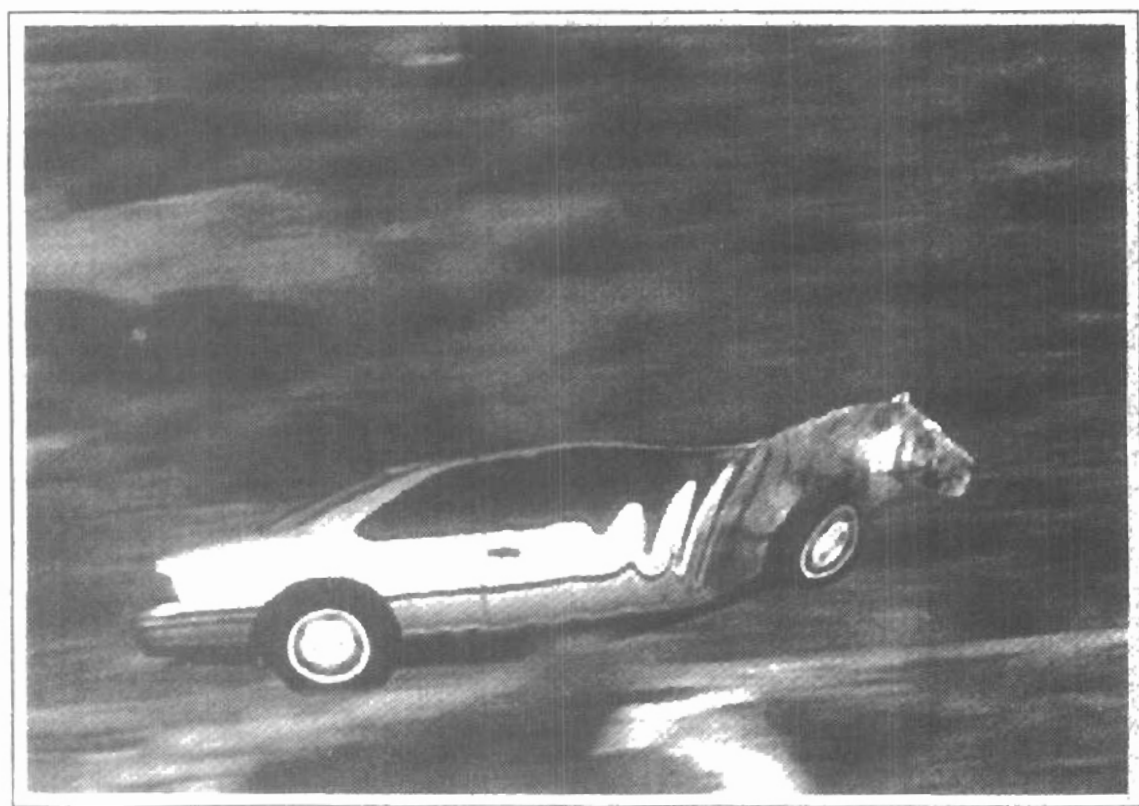
An important consideration in the generation of a color display is the selection of harmonious color combinations. We can do this by following a few simple rules. Coordinating colors usually can be selected from within a small subspace of a color model. Also, we should avoid displaying adjacent colors 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.



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 varying 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.

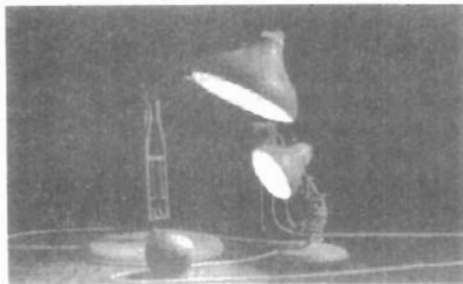


Figure 16-1
One frame from the award-winning computer-animated short film *Luxo Jr.* The film was designed using a key-frame animation system and cartoon animation techniques to provide lifelike actions of the lamps. Final images were rendered with multiple light sources and procedural texturing techniques. (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 rectangular 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

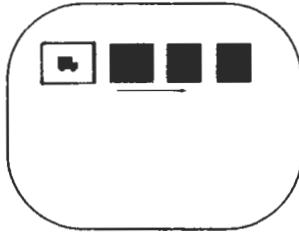


Figure 16-3
Real-time raster color-table
animation.

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.

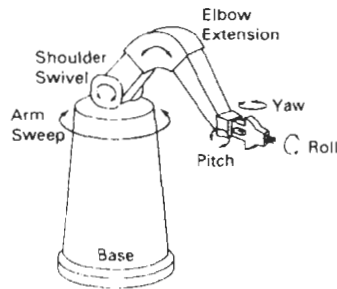


Figure 16-4
Degrees of freedom for a stationary,
single-arm robot

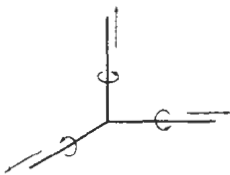


Figure 16-5
Translational and rotational
degrees of freedom for the
base of the robot arm.

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.

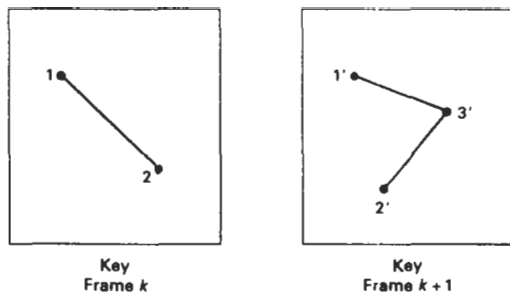


Figure 16-6
An edge with vertex positions 1 and 2 in key frame k evolves into two connected edges in key frame $k + 1$.

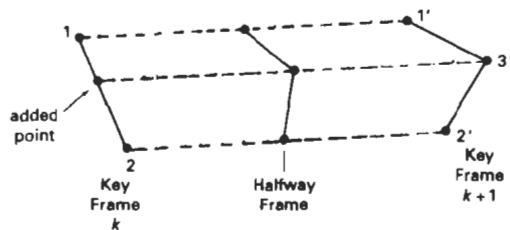


Figure 16-7
Linear interpolation for transforming a line segment in key frame k into two connected line segments in key frame $k + 1$.

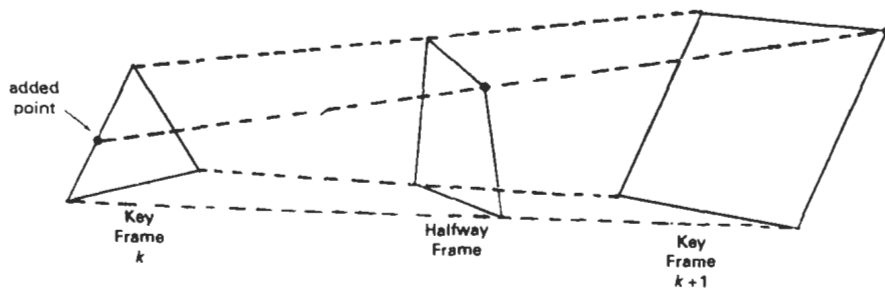


Figure 16-8
Linear interpolation for transforming a triangle into a quadrilateral.

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

$$L_{\max} = \max(L_k, L_{k+1}), \quad L_{\min} = \min(L_k, L_{k+1}) \quad (16-1)$$

and

$$N_e = L_{\max} \bmod L_{\min}$$

$$N_s = \text{int}\left(\frac{L_{\max}}{L_{\min}}\right) \quad (16-2)$$



(a)



(b)



(c)



(d)



(e)

Figure 16-9
Transformation of an STP oil can
into an engine block. (Courtesy of
Silicon Graphics, Inc.)



(a)



(b)



(c)



(d)

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

$$V_{\max} = \max(V_k, V_{k+1}), \quad V_{\min} = \min(V_k, V_{k+1}) \quad (16-3)$$

and

$$\begin{aligned} N_s &= (V_{\max} - 1) \bmod (V_{\min} - 1) \\ N_p &= \text{int}\left(\frac{V_{\max} - 1}{V_{\min} - 1}\right) \end{aligned} \quad (16-4)$$

Preprocessing using vertex count is performed by

1. adding N_p points to N_s 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_s 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

$$\Delta t = \frac{t_2 - t_1}{n + 1} \quad (16-5)$$

We can calculate the time for any in-between as

$$tB_j = t_1 + j \Delta t, \quad j = 1, 2, \dots, n \quad (16-6)$$

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

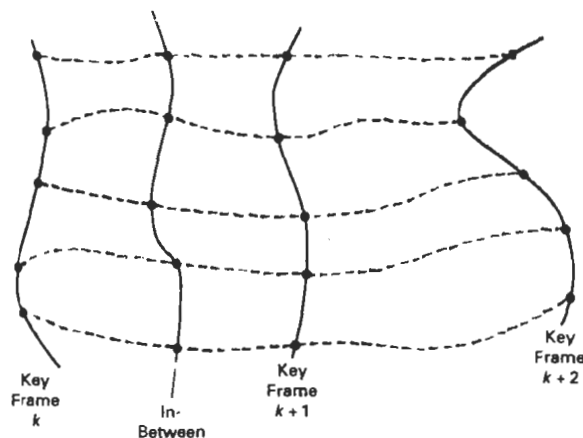


Figure 16-11
Fitting key-frame vertex positions with nonlinear splines.

trigonometric functions. Parabolic and cubic time functions have been applied to acceleration modeling, but trigonometric 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

$$1 - \cos \theta, \quad 0 < \theta < \pi/2$$

For n in-betweens, the time for the j th in-between would then be calculated as

$$tB_j = t_1 + \Delta t \left[1 - \cos \frac{j\pi}{2(n+1)} \right], \quad j = 1, 2, \dots, n \quad (16-7)$$

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

$$tB_j = t_1 + \Delta t \sin \frac{j\pi}{2(n+1)}, \quad j = 1, 2, \dots, n \quad (16-8)$$

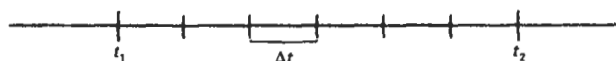


Figure 16-12
In-between positions for motion at constant speed.

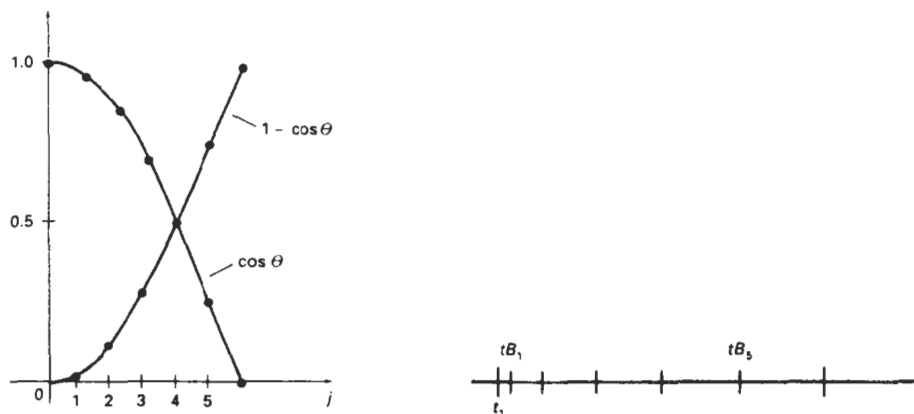


Figure 16-13

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

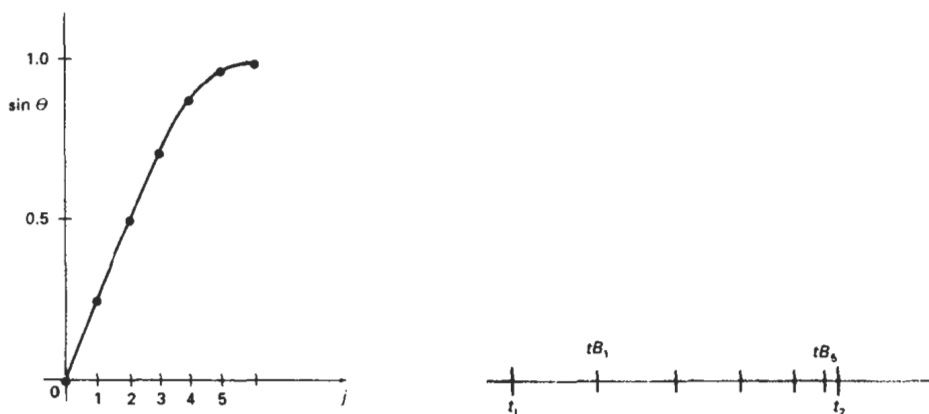


Figure 16-14

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.

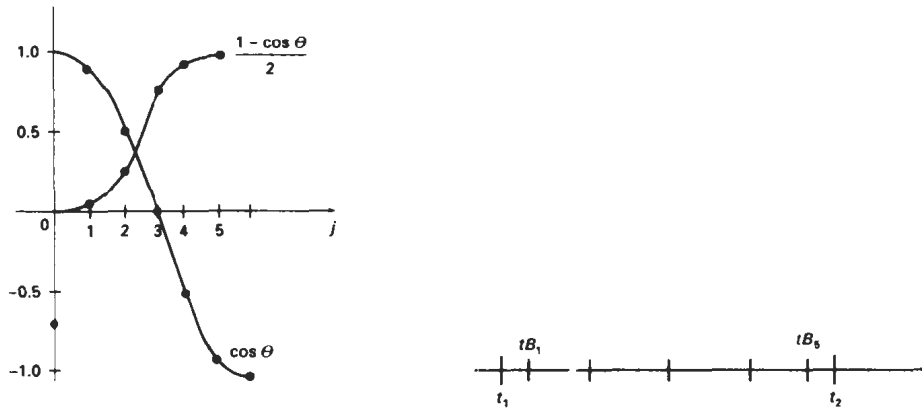


Figure 16-15
A trigonometric accelerate-decelerate function and the corresponding in-between spacing for $n = 5$ in Eq. 16-9.

$$\frac{1}{2}(1 - \cos \theta), \quad 0 < \theta < \pi/2$$

The time for the j th in-between is now calculated as

$$tB_j = t_1 + \Delta t \left\{ \frac{1 - \cos[j\pi/(n+1)]}{2} \right\}, \quad j = 1, 2, \dots, n \quad (16-9)$$

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

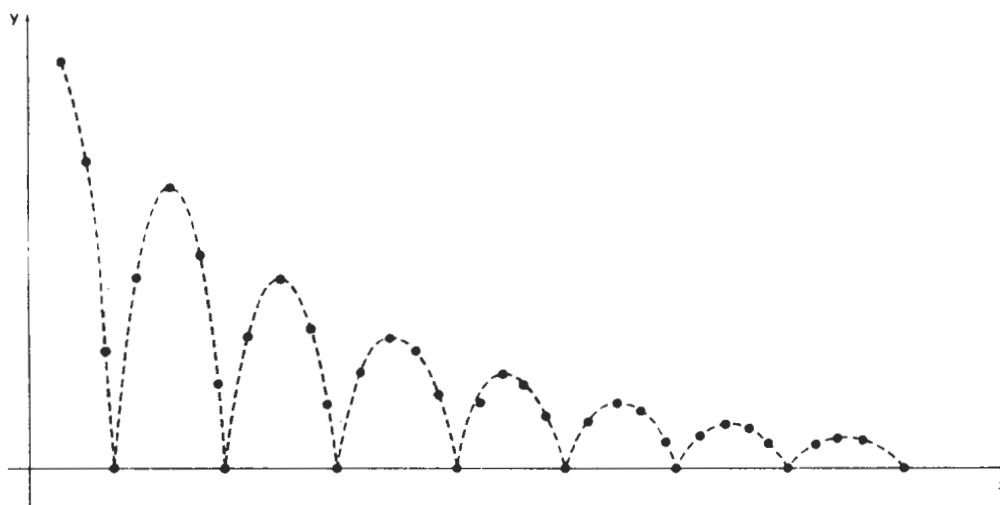


Figure 16-16

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):

$$y(x) = A |\sin(\omega x + \theta_0)| e^{-kx} \quad (16-10)$$

where A is the initial amplitude, ω is the angular frequency, 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 sub-motions 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

$$\mathbf{F} = \frac{d}{dt}(m\mathbf{v}) \quad (16-11)$$

with \mathbf{F} as the force vector, and \mathbf{v} as the velocity vector. If mass is constant, we solve the equation $\mathbf{F} = m\mathbf{a}$, where \mathbf{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

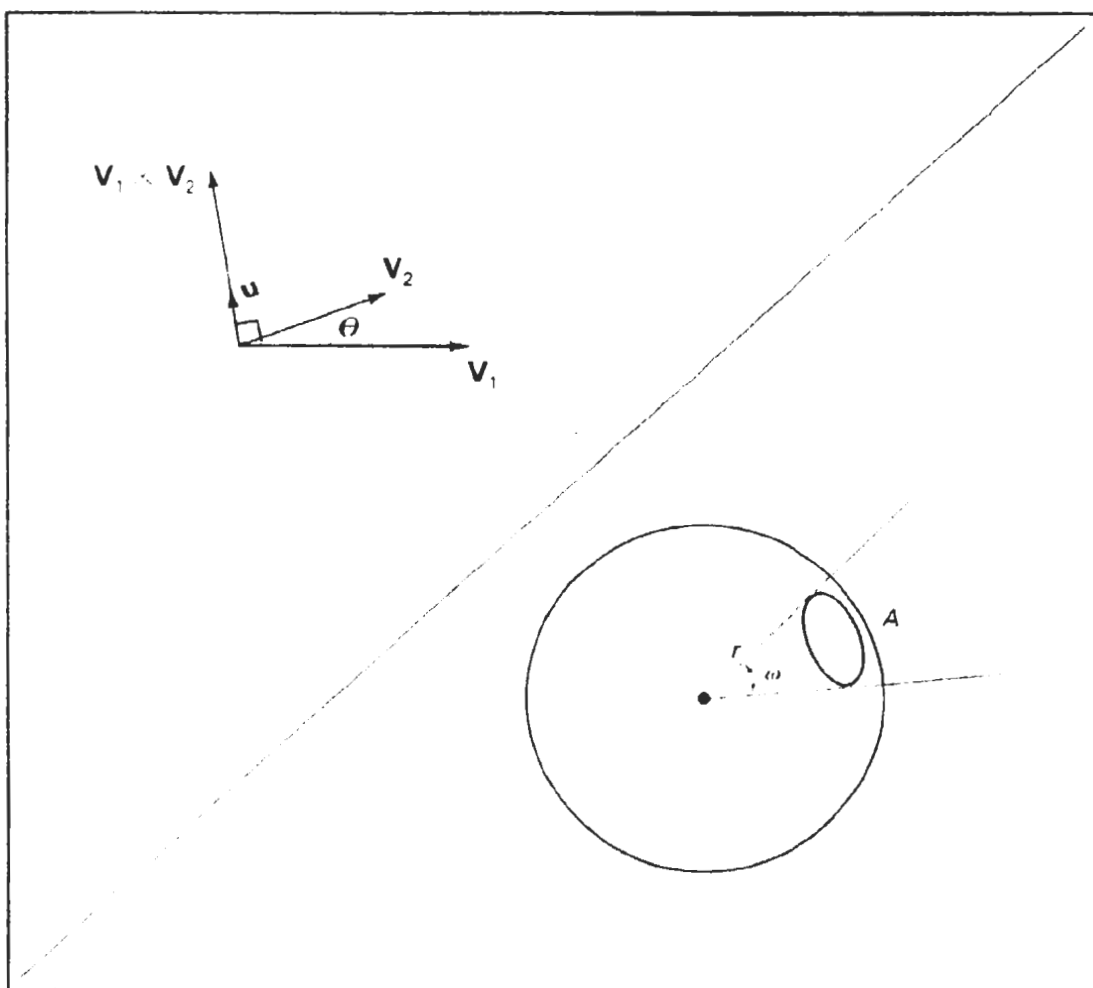
- 16-1. Design a storyboard layout and accompanying key frames for an animation of a single polyhedron.
- 16-2. Write a program to generate the in-betweens for the key frames specified in Exercise 16-1 using linear interpolation.
- 16-3. Expand the animation sequence in Exercise 16-1 to include two or 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.)

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

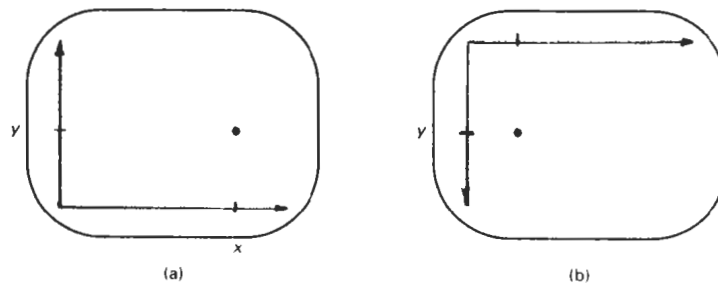


Figure A-1
Screen Cartesian reference systems: (a) coordinate origin at the lower-left screen corner and (b) coordinate origin in the upper-left corner.

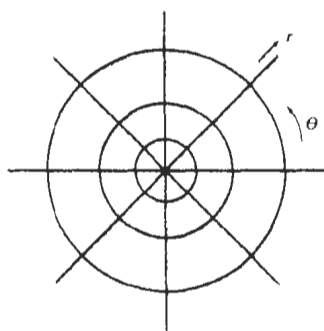


Figure A-2
A polar coordinate reference frame, formed with concentric circles and radial lines.

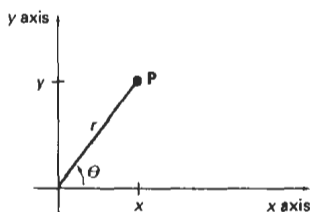


Figure A-3
Relationship between polar and Cartesian coordinates.

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

$$x = r \cos \theta, \quad y = r \sin \theta \quad (\text{A-1})$$

The inverse transformation from Cartesian to polar coordinates is

$$r = \sqrt{x^2 + y^2}, \quad \theta = \tan^{-1} \left(\frac{y}{x} \right) \quad (\text{A-2})$$

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.

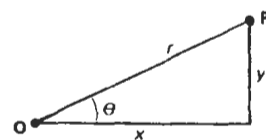


Figure A-4
Right triangle with hypotenuse r and sides x and y .

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

$$\theta = \frac{s}{r} \quad (A-3)$$

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 (2π) divided by r , or 2π radians.

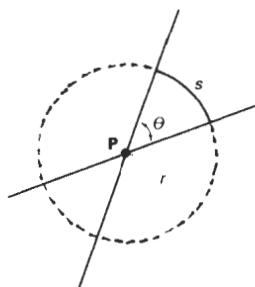


Figure A-5
An angle θ subtended by a circular arc of length s and radius r .

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-

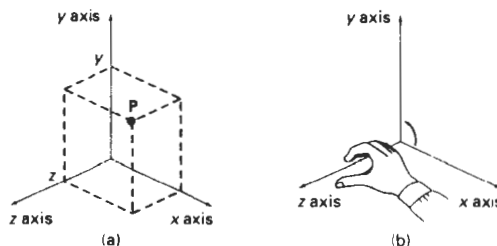


Figure A-6
Coordinate representation of a point **P** at position (x, y, z) in a right-handed Cartesian reference system.

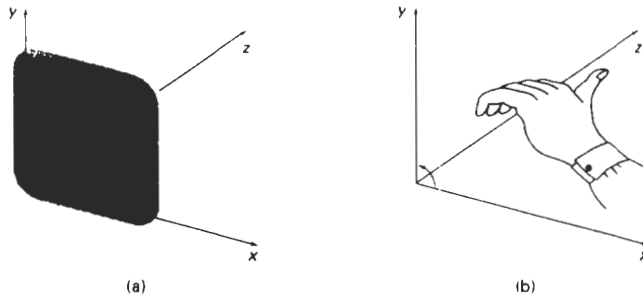


Figure A-7
Left-handed Cartesian coordinate system superimposed on the surface of a video monitor.

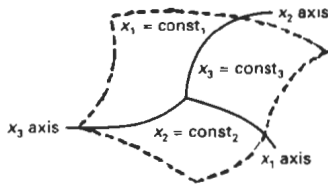


Figure A-8
A general curvilinear coordinate reference frame.

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

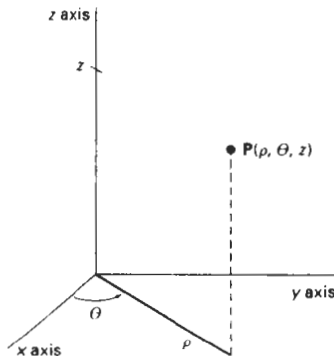


Figure A-9
Cylindrical coordinates: ρ , θ , z .

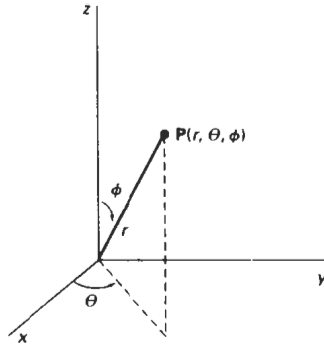


Figure A-10
Spherical coordinates: r , θ , ϕ .

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

$$x = \rho \cos \theta, \quad y = \rho \sin \theta, \quad z = z \quad (A-4)$$

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 $\phi < 90^\circ$, the cone is above the xy plane. If $\phi > 90^\circ$, the cone is below the xy plane. We transform from a spherical-coordinate specification to a Cartesian reference frame with the calculations

$$x = r \cos \theta \sin \phi, \quad y = r \sin \theta \sin \phi, \quad z = r \cos \phi \quad (A-5)$$

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

$$\omega = \frac{A}{r^2} \quad (A-6)$$

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.

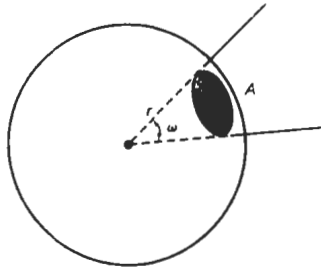


Figure A-11
A solid angle ω subtended by a spherical surface patch of area A with radius r .

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

$$\begin{aligned} \mathbf{V} &= \mathbf{P}_2 - \mathbf{P}_1 \\ &= (x_2 - x_1, y_2 - y_1) \\ &= (V_x, V_y) \end{aligned} \quad (\text{A-7})$$

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:

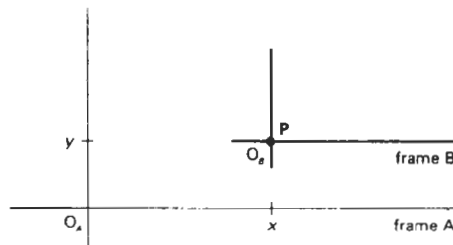


Figure A-12
Position of point P with respect to two different Cartesian reference frames.

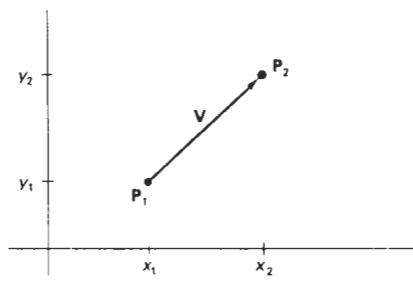


Figure A-13
Vector V in the xy plane of a Cartesian reference frame.

$$|V| = \sqrt{V_x^2 + V_y^2} \quad (A-8)$$

The direction for this two-dimensional vector can be given in terms of the angular displacement from the x axis as

$$\alpha = \tan^{-1}\left(\frac{V_y}{V_x}\right) \quad (A-9)$$

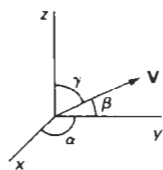


Figure A-14
Direction angles α , β , and γ .

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

$$|V| = \sqrt{V_x^2 + V_y^2 + V_z^2} \quad (A-10)$$

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

$$\cos\alpha = \frac{V_x}{|V|}, \quad \cos\beta = \frac{V_y}{|V|}, \quad \cos\gamma = \frac{V_z}{|V|} \quad (A-11)$$

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

$$\cos^2\alpha + \cos^2\beta + \cos^2\gamma = 1 \quad (A-12)$$

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-

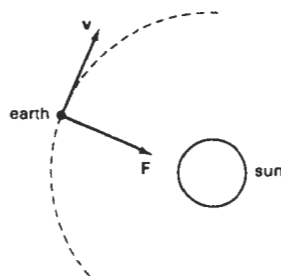


Figure A-15
A gravitational force vector F and a velocity vector v .

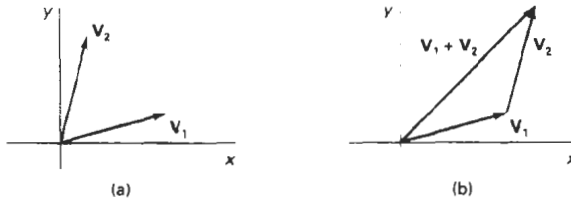


Figure A-16

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:

$$\mathbf{V}_1 + \mathbf{V}_2 = (V_{1x} + V_{2x}, V_{1y} + V_{2y}, V_{1z} + V_{2z}) \quad (\text{A-13})$$

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

$$a\mathbf{V} = (aV_x, aV_y, aV_z) \quad (\text{A-14})$$

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

$$\mathbf{V}_1 \cdot \mathbf{V}_2 = |\mathbf{V}_1| |\mathbf{V}_2| \cos \theta, \quad 0 \leq \theta \leq \pi \quad (\text{A-15})$$

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.

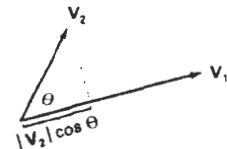


Figure A-17

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

$$\mathbf{V}_1 \cdot \mathbf{V}_2 = V_{1x}V_{2x} + V_{1y}V_{2y} + V_{1z}V_{2z} \quad (\text{A-16})$$

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

$$\mathbf{V}_1 \cdot \mathbf{V}_2 = \mathbf{V}_2 \cdot \mathbf{V}_1 \quad (\text{A-17})$$

because this operation produces a scalar, and dot products are distributive with respect to vector addition

$$\mathbf{V}_1 \cdot (\mathbf{V}_2 + \mathbf{V}_3) = \mathbf{V}_1 \cdot \mathbf{V}_2 + \mathbf{V}_1 \cdot \mathbf{V}_3 \quad (\text{A-18})$$

Vector Product of Two Vectors

Multiplication of two vectors to produce another vector is defined as

$$\mathbf{V}_1 \times \mathbf{V}_2 = u |\mathbf{V}_1| |\mathbf{V}_2| \sin \theta, \quad 0 \leq \theta \leq \pi \quad (\text{A-19})$$

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

$$\mathbf{V}_1 \times \mathbf{V}_2 = (V_{1y}V_{2z} - V_{1z}V_{2y}, V_{1z}V_{2x} - V_{1x}V_{2z}, V_{1x}V_{2y} - V_{1y}V_{2x}) \quad (\text{A-20})$$

If we let u_x , u_y , and 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:

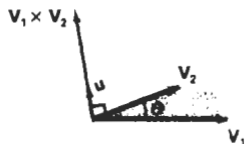


Figure A-18

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.

$$\mathbf{V}_1 \times \mathbf{V}_2 = \begin{vmatrix} \mathbf{u}_x & \mathbf{u}_y & \mathbf{u}_z \\ V_{1x} & V_{1y} & V_{1z} \\ V_{2x} & V_{2y} & V_{2z} \end{vmatrix} \quad (\text{A-21})$$

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:

$$\mathbf{V}_1 \times \mathbf{V}_2 = -(\mathbf{V}_2 \times \mathbf{V}_1) \quad (\text{A-22})$$

And the cross product is not associative:

$$\mathbf{V}_1 \times (\mathbf{V}_2 \times \mathbf{V}_3) \neq (\mathbf{V}_1 \times \mathbf{V}_2) \times \mathbf{V}_3 \quad (\text{A-23})$$

But the cross product is distributive with respect to vector addition; that is,

$$\mathbf{V}_1 \times (\mathbf{V}_2 + \mathbf{V}_3) = (\mathbf{V}_1 \times \mathbf{V}_2) + (\mathbf{V}_1 \times \mathbf{V}_3) \quad (\text{A-24})$$

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.

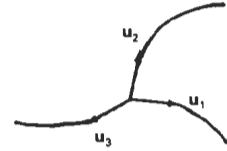


Figure A-19
Curvilinear coordinate-axis vectors.

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:

$$\begin{aligned} \mathbf{u}_k \cdot \mathbf{u}_k &= 1, & \text{for all } k \\ \mathbf{u}_j \cdot \mathbf{u}_k &= 0, & \text{for all } j \neq k \end{aligned} \quad (\text{A-25})$$

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

$$\mathbf{u}_x = (1, 0), \quad \mathbf{u}_y = (0, 1) \quad (\text{A-26})$$

And the orthonormal basis for a three-dimensional Cartesian reference frame is

$$\mathbf{u}_x = (1, 0, 0), \quad \mathbf{u}_y = (0, 1, 0), \quad \mathbf{u}_z = (0, 0, 1) \quad (\text{A-27})$$

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

$$g_{jk} = \mathbf{u}_j \cdot \mathbf{u}_k \quad (\text{A-28})$$

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:

$$g_{jk} = 0, \quad \text{for } j \neq k \quad (\text{A-29})$$

And in a Cartesian coordinate system (assuming unit base vectors):

$$g_{jk} = \begin{cases} 1, & \text{if } j = k \\ 0, & \text{otherwise} \end{cases} \quad (\text{A-30})$$

The unit base vectors in polar coordinates can be expressed in terms of Cartesian base vectors as

$$\mathbf{u}_r = \mathbf{u}_x \cos \theta + \mathbf{u}_y \sin \theta, \quad \mathbf{u}_\theta = -\mathbf{u}_x r \sin \theta + \mathbf{u}_y r \cos \theta \quad (\text{A-31})$$

Substituting these expressions into Eq. A-28, we obtain the elements of the metric tensor, which can be written in the matrix form:

$$\mathcal{G} = \begin{bmatrix} 1 & 0 \\ 0 & r^2 \end{bmatrix} \quad (\text{A-32})$$

For a cylindrical coordinate reference frame, the base vectors are

$$\mathbf{u}_\rho = \mathbf{u}_x \cos \theta + \mathbf{u}_y \sin \theta, \quad \mathbf{u}_\theta = -\mathbf{u}_x \rho \sin \theta + \mathbf{u}_y \rho \cos \theta, \quad \mathbf{u}_z \quad (\text{A-33})$$

And the matrix representation for the metric tensor in cylindrical coordinates is

$$\mathbf{g} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \rho & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (\text{A-34})$$

We can write the base vectors in spherical coordinates as

$$\begin{aligned} \mathbf{u}_r &= \mathbf{u}_x \cos\theta \sin\phi + \mathbf{u}_y \sin\theta \sin\phi + \mathbf{u}_z \cos\phi \\ \mathbf{u}_\theta &= -\mathbf{u}_x r \sin\theta \sin\phi + \mathbf{u}_y r \cos\theta \sin\phi \\ \mathbf{u}_\phi &= \mathbf{u}_x r \cos\theta \cos\phi + \mathbf{u}_y r \sin\theta \cos\phi - \mathbf{u}_z r \sin\phi \end{aligned} \quad (\text{A-35})$$

Then the matrix representation for the metric tensor in spherical coordinates is

$$\mathbf{g} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & r^2 \sin^2\phi & 0 \\ 0 & 0 & r^2 \end{bmatrix} \quad (\text{A-36})$$

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

$$\begin{bmatrix} 3.60 & -0.01 & 2.00 \\ -5.46 & 0.00 & 1.63 \end{bmatrix}, \quad \begin{bmatrix} e^x & x \\ e^{2x} & x^2 \end{bmatrix}, \quad [a_1 \ a_2 \ a_3], \quad \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (\text{A-37})$$

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

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \quad (\text{A-38})$$

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

$$\mathbf{V} = \begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix} \quad (\text{A-39})$$

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 \mathbf{A} by a scalar value s , we multiply each element a_{ik} 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 \mathbf{A} by a p by q matrix \mathbf{B} to form the matrix product \mathbf{AB} , providing that the number of columns in \mathbf{A} is equal to the number of rows in \mathbf{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 \mathbf{A} with the corresponding elements in the column vectors of \mathbf{B} . Thus, for the following product

$$\mathbf{C} = \mathbf{AB} \quad (\text{A-40})$$

we obtain an m by q matrix \mathbf{C} whose elements are calculated as

$$c_{ij} = \sum_{k=1}^n a_{ik}b_{kj} \quad (\text{A-41})$$

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,

$$\mathbf{AB} \neq \mathbf{BA} \quad (\text{A-42})$$

But matrix multiplication is distributive with respect to matrix addition:

$$\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC} \quad (\text{A-43})$$

Matrix Transpose

The **transpose** \mathbf{A}^T of a matrix is obtained by interchanging rows and columns. For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}, \quad [a \ b \ c]^T = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \quad (\text{A-44})$$

For a matrix product, the transpose is

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T \quad (\text{A-45})$$

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

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21} \quad (\text{A-46})$$

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

$$\det \mathbf{A} = \sum_{j=1}^n (-1)^{j+k} a_{jk} \det \mathbf{A}_{jk} \quad (\text{A-47})$$

where $\det \mathbf{A}_{jk}$ is the $(n-1)$ by $(n-1)$ determinant of the submatrix obtained from \mathbf{A} by deleting the j th row and the k th column. Alternatively, we can select any row j and calculate the determinant as

$$\det \mathbf{A} = \sum_{k=1}^n (-1)^{j+k} a_{jk} \det \mathbf{A}_{jk} \quad (\text{A-48})$$

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:

$$\det(\mathbf{AB}) = (\det \mathbf{A})(\det \mathbf{B}) \quad (\text{A-49})$$

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 **non-singular 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

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I} \quad (\text{A-50})$$

where \mathbf{I} is the identity 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

$$a_{jk}^{-1} = \frac{(-1)^{j+k} \det \mathbf{A}_{kj}}{\det \mathbf{A}} \quad (\text{A-51})$$

where a_{jk}^{-1} is the element in the j th row and k th column of \mathbf{A}^{-1} , and \mathbf{A}_{kj} is the $(n-1)$ by $(n-1)$ submatrix obtained by deleting the k th row and j th 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 .

By definition, a **complex number** z is an ordered pair of real numbers:

$$z = (x, y) \quad (A-52)$$

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

$$x = \operatorname{Re}(z), \quad y = \operatorname{Im}(z) \quad (A-53)$$

Geometrically, a complex number is represented in the *complex plane*, as in Fig. A-20.

Complex numbers arise from solutions of equations such as

$$x^2 + 1 = 0, \quad x^2 - 2x + 5 = 0$$

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

$$(x_1, y_1)(x_2, y_2) = (x_1x_2 - y_1y_2, x_1y_2 + x_2y_1) \quad (A-54)$$

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

$$i = (0, 1) \quad (A-55)$$

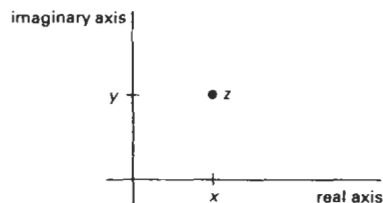


Figure A-20
Position of a point z in the complex plane.

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

$$i = \sqrt{-1} \quad (\text{A-56})$$

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

$$z = x + iy \quad (\text{A-57})$$

which is the usual form used in practical applications.

Another concept associated with a complex number is the *complex conjugate*:

$$\bar{z} = x - iy \quad (\text{A-58})$$

Modulus, or *absolute value*, of a complex number is defined to be

$$|z| = z\bar{z} = \sqrt{x^2 + y^2} \quad (\text{A-59})$$

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

$$\begin{aligned} \frac{z_1}{z_2} &= \frac{z_1 \bar{z}_2}{z_2 \bar{z}_2} \\ &= \frac{(x_1, y_1)(x_2, -y_2)}{x_2^2 + y_2^2}, \\ &= \left(\frac{x_1 x_2 + y_1 y_2}{x_2^2 + y_2^2}, \frac{x_2 y_1 - x_1 y_2}{x_2^2 + y_2^2} \right) \end{aligned} \quad (\text{A-60})$$

A particularly useful representation for complex numbers is to express the real and imaginary parts in terms of polar coordinates (Fig. A-21):

$$z = r(\cos\theta + i\sin\theta) \quad (\text{A-61})$$

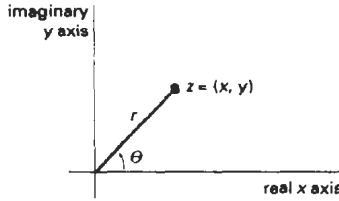


Figure A-21
Polar coordinate position of a complex number z .

We can also write the polar form of z as

$$z = r e^{i\theta} \quad (A-62)$$

where e is the base of the natural logarithms ($e = 2.718281828 \dots$), and

$$e^{i\theta} = \cos\theta + i\sin\theta \quad (A-63)$$

which is *Euler's formula*. Complex multiplications and divisions are easily obtained as

$$z_1 z_2 = r_1 r_2 e^{i(\theta_1 + \theta_2)}, \quad \frac{z_1}{z_2} = r_1 r_2 e^{i(\theta_1 - \theta_2)}$$

And the n th roots of a complex number are calculated as

$$\sqrt[n]{z} = \sqrt[n]{r} \left[\cos\left(\frac{\theta + 2k\pi}{n}\right) + i\sin\left(\frac{\theta + 2k\pi}{n}\right) \right], \quad k = 0, 1, 2, \dots, n-1 \quad (A-64)$$

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

$$q = s + ia + jb + kc \quad (A-65)$$

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

$$i^2 = j^2 = k^2 = -1, \quad ij = -ji = k \quad (A-66)$$

From these properties, it follows that

$$jk = -kj = i, \quad ki = -ik = j \quad (A-67)$$

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

$$q_1 + q_2 = (s_1 + s_2) + i(a_1 + a_2) + j(b_1 + b_2) + k(c_1 + c_2) \quad (A-68)$$

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:

$$q = (s, \mathbf{v}) \quad (A-69)$$

where \mathbf{v} is the vector (a, b, c) . In this notation, quaternion addition is expressed as

$$q_1 + q_2 = (s_1 + s_2, \mathbf{v}_1 + \mathbf{v}_2) \quad (A-70)$$

Quaternion multiplication can then be expressed in terms of vector dot and cross products as

$$q_1 q_2 = (s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, s_1 \mathbf{v}_2 + s_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2) \quad (A-71)$$

As an extension of complex operations, the magnitude squared of a quaternion is defined using the vector dot product as

$$|q|^2 = s^2 + \mathbf{v} \cdot \mathbf{v} \quad (A-72)$$

And the inverse of a quaternion is

$$q^{-1} = \frac{1}{|q|^2} (s, -\mathbf{v}) \quad (A-73)$$

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 representation is called **nonparametric**. For example, we can represent a surface with either of the following Cartesian functions:

$$f(x, y, z) = 0, \quad \text{or} \quad z = f(x, y) \quad (A-74)$$

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

$$y = f(x), \quad z = g(x) \quad (A-75)$$

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):

$$\mathbf{P}(u) = \langle x(u), y(u), z(u) \rangle \quad (A-76)$$

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

$$x(u) = r \cos(2\pi u), \quad y(u) = r \sin(2\pi u), \quad z(u) = 0, \quad 0 \leq u \leq 1 \quad (A-77)$$

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

$$\mathbf{P}(u, v) = \langle x(u, v), y(u, v), z(u, v) \rangle \quad (A-78)$$

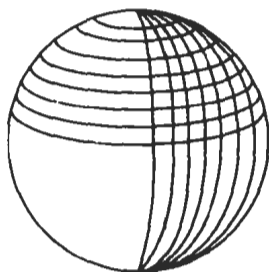


Figure A-22
Section of a spherical surface
described by lines of constant
 u and lines of constant v in
Eqs. A-79.

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

$$\begin{aligned}x(u,v) &= r \sin(\pi u) \cos(2\pi v) \\y(u,v) &= r \sin(\pi u) \sin(2\pi v) \\z(u,v) &= r \cos(\pi u)\end{aligned}\tag{A-79}$$

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 , $k = 1, 2, \dots, n$, we can write a system of n linear equations as

$$\begin{aligned}a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1 \\a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2 \\&\vdots \\a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n &= b_n\end{aligned}\tag{A-80}$$

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*:

$$x_k = \frac{\det \mathbf{A}_k}{\det \mathbf{A}} \quad (\text{A-83})$$

where \mathbf{A}_k is the matrix \mathbf{A} with the k th column replaced with the elements of \mathbf{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 than $n!$ 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

$$\begin{aligned} x_1 + 2x_2 &= -4 \\ 3x_1 + 4x_2 &= 1 \end{aligned} \quad (\text{A-84})$$

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 susceptible 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

$$\begin{aligned} x_1 &= \frac{b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n}{a_{11}} \\ x_2 &= \frac{b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n}{a_{22}} \\ &\vdots \end{aligned} \quad (\text{A-85})$$

If we can rearrange matrix \mathbf{A} so that each diagonal element has a magnitude greater than the sum of the magnitudes of the other elements across that row, then 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-

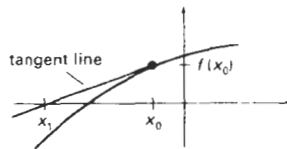


Figure A-23
Approximating a curve at an initial value x_0 with a straight line that is tangent to the curve at that point.

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

$$\frac{df}{dx} = \frac{f(x_0)}{x_0 - x_1} \quad (\text{A-86})$$

Thus, the next approximation to the root is

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \quad (\text{A-87})$$

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

$$\int_a^b f(x) dx \approx \sum_{k=1}^n f_k(x) \Delta x_k \quad (\text{A-88})$$

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

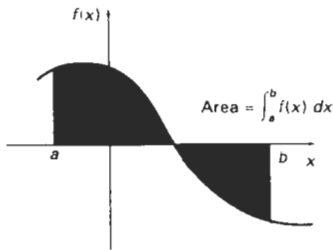


Figure A-24
The integral of $f(x)$ is equal to the amount of area between the function and the x axis over the interval from a to b .

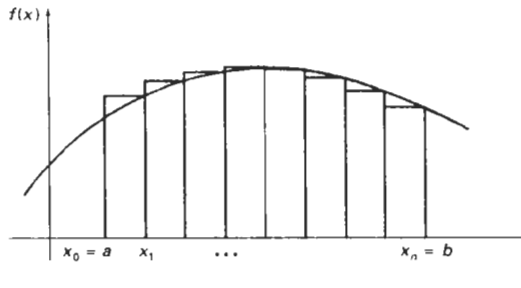


Figure A-25
Approximating an integral as the sum of the areas of small rectangles.

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

$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} \left[f(a) + f(b) + 4 \sum_{\text{odd } k=1}^{n-1} f(x_k) + 2 \sum_{\text{even } k=2}^{n-2} f(x_k) \right] \quad (\text{A-89})$$

where the interval from a to b is divided into n equal-width intervals:

$$\Delta x = \frac{b - a}{n} \quad (\text{A-90})$$

where n is a multiple of 2, and with

$$x_0 = a, \quad x_k = x_{k-1} + \Delta x, \quad k = 1, 2, \dots, n$$

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-

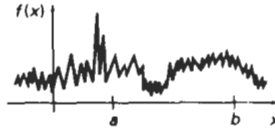


Figure A-26
A function with high-frequency oscillations.

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

$$\int_a^b f(x) dx \approx h(b-a) \frac{n_{\text{count}}}{n} \quad (\text{A-91})$$

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

$$h = y_{\text{max}} - y_{\text{min}}, \quad x = a + r_1(b-a), \quad y = y_{\text{min}} + r_2 h \quad (\text{A-92})$$

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*:

$$i_k = ai_{k-1} + c \pmod{m}, \quad k = 1, 2, 3, \dots \quad (\text{A-93})$$

$$r_k = \frac{i_k}{m}$$

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 $c = 1013904223$.

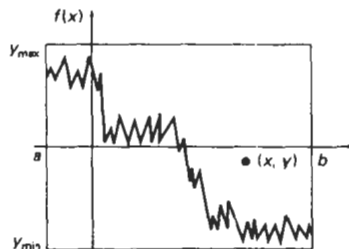


Figure A-27
A rectangular area enclosing a function $f(x)$ over the interval (a, b) .

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) , $k = 1, 2, \dots$, 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:

$$E = \sum_{k=1}^n [y_k - f(x_k)]^2 \quad (A-94)$$

Parameters in the function $f(x)$ are determined by minimizing the expression for E . For example, for the linear function

$$f(x) = a_0 + a_1x$$

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:

$$\frac{\partial E}{\partial a_0} = 0, \quad \frac{\partial E}{\partial a_1} = 0$$

Similar calculations are carried out for other functions. For the polynomial

$$f(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

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 $f(x_1, x_2, \dots, x_m)$ that can be linear or nonlinear in each of the variables.

Bibliography

- AKELEY, K. AND T. JERMOLUK (1988). "High-Performance Polygon Rendering", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 239-246.
- AKELEY, K. (1993). "RealityEngine Graphics", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 109-116.
- AMANATIDES, J. (1984). "Ray Tracing with Cones", in proceedings of SIGGRAPH '84, *Computer Graphics*, 18(3), pp. 129-135.
- AMBURN, P., E. GRANT AND T. WHITTED (1986). "Managing Geometric Complexity with Enhanced Procedural Models", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 189-196.
- ANJO, K., Y. USAMI AND T. KURIHARA (1992). "A Simple Method for Extracting the Natural Beauty of Hair", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 111-120.
- APPLE COMPUTER, INC. (1985). *Inside Macintosh*, Volume 1, Addison-Wesley, Reading, MA.
- APPLE COMPUTER, INC. (1987). *Human Interface Guidelines: The Apple Desktop Interface*, Addison-Wesley, Reading, MA.
- ARVO, J. AND D. KIRK (1987). "Fast Ray Tracing by Ray Classification", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 55-64.
- ARVO, J. AND D. KIRK (1990). "Particle Transport and Image Synthesis", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 63-66.
- ARVO, J., ED. (1991). *Graphics Gems II*, Academic Press, Inc., San Diego, CA.
- ATHERTON, P. R. (1983). "A Scan-Line Hidden Surface Removal Procedure for Constructive Solid Geometry", in proceedings of SIGGRAPH '83, *Computer Graphics*, 17(3), pp. 73-82.
- BARAFF, D. (1989). "Analytical Methods for Dynamic Simulation of Non-Penetrating Rigid Bodies", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 223-232.
- BARAFF, D. AND A. WITKIN (1992). "Dynamic Simulation of Non-Penetrating Flexible Bodies", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 303-308.
- BARKANS, A. C. (1990). "High-Speed, High-Quality, Antialiased Vector Generation", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 319-326.
- BARNESLEY, M. F., A. JACQUIN, F. MALASSENT, ET AL. (1988). "Harnessing Chaos for Image Synthesis", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 131-140.
- BARNESLEY, M. (1993). *Fractals Everywhere*, Second Edition, Academic Press, Inc., San Diego, CA.
- BARR, A. H. (1981). "Superquadrics and Angle-Preserving Transformations", *IEEE Computer Graphics and Applications*, 1(1), pp. 11-23.
- BARR, A. H. (1986). "Ray Tracing Deformed Surfaces", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 287-296.
- BARSKY, B. A. AND J. C. BEATTY (1983). "Local Control of Bias and Tension in Beta-Splines", *ACM Transactions on Graphics*, 2(2), pp. 109-134.
- BARSKY, B. A. (1984). "A Description and Evaluation of Various 3-D Models", *IEEE Computer Graphics and Applications*, 4(1), pp. 38-52.
- BARZEL, R. AND A. H. BARR (1988). "A Modeling System Based on Dynamic Constraints", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 179-188.
- BARZEL, R. (1992). *Physically-Based Modeling for Computer Graphics*, Academic Press, Inc., San Diego, CA.
- BAUM, D. R., S. MANN, K. P. SMITH, ET AL. (1991). "Making Radiosity Usable: Automatic Preprocessing and Meshing Techniques for the Generation of Accurate Radiosity Solutions", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 51-61.
- BECKER, S. C., W. A. BARRETT, AND D. R. OLSEN JR. (1991). "Interactive Measurement of Three-Dimensional Objects Using a Depth Buffer and Linear Probe", *ACM Transactions on Graphics*, 10(2), pp. 201-207.
- BECKER, B. G. AND N. L. MAX (1993). "Smooth Transitions between Bump-Rendering Algorithms", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 183-190.
- BEIER, T. AND S. NEELY (1992). "Feature-Based Image Metamorphosis", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 35-42.

- BERGMAN, L., H. FUCHS, E. GRANT, ET AL. (1986). "Image Rendering by Adaptive Refinement", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 29-38.
- BERGMAN, L. D., J. S. RICHARDSON, D. C. RICHARDSON, ET AL. (1993). "VIEW—an Exploratory Molecular Visualization System with User-Definable Interaction Sequences", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 117-126.
- BEZIER, P. (1972). *Numerical Control: Mathematics and Applications*, translated by A. R. Forrest and A. F. Pankhurst, John Wiley & Sons, London.
- BIER, E. A., S. A. MACKEY, D. A. STEWART, ET AL. (1986). "Snap-Dragging", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 241-248.
- BIER, E. A., M. C. STONE, K. PIER, ET AL. (1993). "Toolglass and Magic Lenses: The See-Through Interface", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 73-80.
- BISHOP, G. AND D. M. WIEMER (1986). "Fast Phong Shading", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 103-106.
- BLAKE, J. W. (1993). *PHIGS and PHIGS Plus*, Academic Press, London.
- BLESER, T. (1988). "TAE Plus Styleguide User Interface Description", NASA Goddard Space Flight Center, Greenbelt, MD.
- BLINN, J. F. AND M. E. NEWELL (1976). "Texture and Reflection in Computer-Generated Images", *CACM*, 19(10), pp. 542-547.
- BLINN, J. F. (1977). "Models of Light Reflection for Computer-Synthesized Pictures", *Computer Graphics*, 11(2), pp. 192-198.
- BLINN, J. F. AND M. E. NEWELL (1978). "Clipping Using Homogeneous Coordinates", *Computer Graphics*, 12(3), pp. 245-251.
- BLINN, J. F. (1978). "Simulation of Wrinkled Surfaces", *Computer Graphics*, 12(3), pp. 286-292.
- BLINN, J. F. (1982). "A Generalization of Algebraic Surface Drawing", *ACM Transactions on Graphics*, 1(3), pp. 235-256.
- BLINN, J. F. (1982). "Light Reflection Functions for Simulation of Clouds and Dusty Surfaces", in proceedings of SIGGRAPH '82, *Computer Graphics*, 16(3), pp. 21-29.
- BLINN, J. F. (1993). "A Trip Down the Graphics Pipeline: The Homogeneous Perspective Transform", *IEEE Computer Graphics and Applications*, 13(3), pp. 75-80.
- BLOOMENTHAL, J. (1985). "Modeling the Mighty Maple", in proceedings of SIGGRAPH '85, *Computer Graphics*, 19(3), pp. 305-312.
- BONO, P. R., J. L. ENCARNACAO, F. R. A. HOPGOOD, ET AL. (1982). "GKS: The First Graphics Standard", *IEEE Computer Graphics and Applications*, 2(5), pp. 9-23.
- BOOTH, K. S., M. P. BRYDEN, W. B. COWAN, ET AL. (1987). "On the Parameters of Human Visual Performance: An Investigation of the Benefits of Antialiasing", *IEEE Computer Graphics and Applications*, 7(9), pp. 34-41.
- BRESENHAM, J. E. (1965). "Algorithm for Computer Control of A Digital Plotter", *IBM Systems Journal*, 4(1), pp. 25-30.
- BRESENHAM, J. E. (1977). "A Linear Algorithm for Incremental Digital Display of Circular Arcs", *CACM*, 20(2), pp. 100-106.
- BROOKS, F. P., JR. (1986). "Walkthrough: A Dynamic Graphics System for Simulating Virtual Buildings", *Interactive 3D* 1986.
- BROOKS, F. P., JR. (1988). "Grasping Reality Through Illusion: Interactive Graphics Serving Science", *CHI '88*, pp. 1-11.
- BROOKS, J., P. FREDERICK, M. OUI-YOUNG, J. J. BATTER, ET AL. (1990). "Project GROPE - Haptic Display for Scientific Visualization", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 177-185.
- BROWN, M. H. AND R. SEDGEWICK (1984). "A System for Algorithm Animation", in proceedings of SIGGRAPH '84, *Computer Graphics*, 18(3), pp. 177-186.
- BROWN, J. R. AND S. CUNNINGHAM (1989). *Programming the User Interface*, John Wiley & Sons, New York.
- BRUDERLIN, A. AND T. W. CALVERT (1989). "Goal-Directed, Dynamic Animation of Human Walking", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 233-242.
- BRUNET, P. AND I. NAVAZO (1990). "Solid Representation and Operation Using Extended Octrees", *ACM Transactions on Graphics*, 9(2), pp. 170-197.
- BRYSON, S. AND C. LEVIT (1992). "The Virtual Wind Tunnel", *IEEE Computer Graphics and Applications*, 12(4), pp. 25-34.
- BURT, P. J. AND E. H. ADELSON (1983). "A Multiresolution Spline with Application to Image Mosaics", *ACM Transactions on Graphics*, 2(4), pp. 217-236.
- BUXTON, W., M. R. LAMB, D. SHERMAN, ET AL. (1983). "Towards a Comprehensive User Interface Management System", in proceedings of SIGGRAPH '83, *Computer Graphics*, 17(3), pp. 35-42.
- BUXTON, W., R. HILL, AND P. ROWLEY (1985). "Issues and Techniques in Touch-Sensitive Tablet Input", in proceedings of SIGGRAPH '85, *Computer Graphics*, 19(3), pp. 215-224.
- CALVERT, T., A. BRUDERLIN, J. DILL, ET AL. (1993). "Desktop Animation of Multiple Human Figures", *IEEE Computer Graphics and Applications*, 13(3), pp. 18-26.
- CAMBELL, G., T. A. DEFANTI, J. FREDERIKSEN, ET AL. (1986). "Two Bit/Pixel Full-Color Encoding", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 215-224.
- CAMPBELL, III, A. T. AND D. S. FUSSELL (1990). "Adaptive Mesh Generation for Global Diffuse Illumination", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 155-164.
- CARD, S. K., J. D. MACKINLAY, AND G. G. ROBERTSON (1991). "The Information Visualizer, an Information Workspace", *CHI '91*, pp. 181-188.

- CARIGNAN, M., Y. YANG, N. M. THALMANN, ET AL. (1992). "Dressing Animated Synthetic Actors with Complex Deformable Clothes", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 99-104.
- CARBOM, I., I. CHAKRAVARTY, AND D. VANDERSCHEL (1985). "A Hierarchical Data Structure for Representing the Spatial Decomposition of 3-D Objects", *IEEE Computer Graphics and Applications*, 5(4), pp. 24-31.
- CARPENTER, L. (1984). "The A-Buffer: An Antialiased Hidden-Surface Method", in proceedings of SIGGRAPH '84, *Computer Graphics*, 18(3), pp. 103-108.
- CARROLL, J. M. AND C. CARRITHERS (1984). "Training Wheels in a User Interface", *CACM*, 27(8), pp. 800-806.
- CASALE M. S. AND E. L. STANTON (1985). "An Overview of Analytic Solid Modeling", *IEEE Computer Graphics and Applications*, 5(2), pp. 45-56.
- CATMULL, E. (1975). "Computer Display of Curved Surfaces", in proceedings of the IEEE Conference on Computer Graphics, *Pattern Recognition and Data Structures Also in Freeman* (1980), pp. 309-315.
- CATMULL, E. (1984). "An Analytic Visible Surface Algorithm for Independent Pixel Processing", in proceedings of SIGGRAPH '84, *Computer Graphics*, 18(3), pp. 109-115.
- CHAZELLE, B. AND J. INCERPI (1984). "Triangulation and Shape Complexity", *ACM Transactions on Graphics*, 3(2), pp. 135-152.
- CHEN, M., S. J. MOUNTFORD, AND A. SELLEN (1988). "A Study in Interactive 3D Rotation Using 2D Control Devices", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 121-130.
- CHEN, S. E., H. E. RUSHMEIER, G. MILLER, ET AL. (1991). "A Progressive Multi-Pass Method for Global Illumination", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 165-174.
- CHIN, N. AND S. FEINER (1989). "Near Real-Time Shadow Generation Using BSP Trees", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 99-106.
- CHUANG, R. AND G. ENTIS (1983). "3-D Shaded Computer Animation—Step by Step", *IEEE Computer Graphics and Applications*, 3(3), pp. 18-25.
- CHUNG, J. C., ET AL. (1989). "Exploring Virtual Worlds with Head-Mounted Visual Displays", *Proceedings of SPIE Meeting on Non-Holographic True 3-Dimensional Display Technologies*, 1083, January 1989, pp. 15-20.
- CLARK, J. H. (1982). "The Geometry Engine: A VLSI Geometry System for Graphics", in proceedings of SIGGRAPH '82, *Computer Graphics*, 16(3), pp. 127-133.
- COHEN, M. F. AND D. P. GREENBERG (1985). "The Hemisphere: A Radiosity Solution for Complex Environments", in proceedings of SIGGRAPH '85, *Computer Graphics*, 19(3), pp. 31-40.
- COHEN, M. F., S. E. CHEN, J. R. WALLACE, ET AL. (1988). "A Progressive Refinement Approach to Fast Radiosity Image Generation", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 75-84.
- COHEN, M. F. AND J. R. WALLACE (1993). *Radiosity and Realistic Image Synthesis*, Academic Press, Boston, MA.
- COOK, R. L. AND K. E. TORRANCE (1982). "A Reflectance Model for Computer Graphics", *ACM Transactions on Graphics*, 1(1), pp. 7-24.
- COOK, R. L., T. PORTER, AND L. CARPENTER (1984). "Distributed Ray Tracing", in proceedings of SIGGRAPH '84, *Computer Graphics*, 18(3), pp. 137-145.
- COOK, R. L. (1984). "Shade Trees", in proceedings of SIGGRAPH '84, *Computer Graphics*, 18(3), pp. 223-231.
- COOK, R. L. (1986). "Stochastic Sampling in Computer Graphics", *ACM Transactions on Graphics*, 6(1), pp. 51-72.
- COOK, R. L., L. CARPENTER, AND E. CATMULL (1987). "The Reyes Image Rendering Architecture", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 95-102.
- COQUILLART, S. AND P. JANCENE (1991). "Animated Free-Form Deformation: An Interactive Animation Technique", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 23-26.
- CROW, F. C. (1977). "The Aliasing Problem in Computer-Synthesized Shaded Images", *CACM*, 20(11), pp. 799-805.
- CROW, F. C. (1977). "Shadow Algorithms for Computer Graphics", in proceedings of SIGGRAPH '77, *Computer Graphics*, 11(2), pp. 242-248.
- CROW, F. C. (1978). "The Use of Grayscale for Improved Raster Display of Vectors and Characters", in proceedings of SIGGRAPH '78, *Computer Graphics*, 12(3), pp. 1-5.
- CROW, F. C. (1981). "A Comparison of Antialiasing Techniques", *IEEE Computer Graphics and Applications*, 1(1), pp. 40-49.
- CROW, F. C. (1982). "A More Flexible Image Generation Environment", in proceedings of SIGGRAPH '82, *Computer Graphics*, 16(3), pp. 9-18.
- CRUZ-NEIRA, C., D. J. SANDIN, AND T. A. DEFANTI (1993). "Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 135-142.
- CUNNINGHAM, S., N. K. CRAIGHILL, M. W. FONG, ET AL., ED. (1992). *Computer Graphics Using Object-Oriented Programming*, John Wiley & Sons, New York.
- CUTLER, E., D. GILLY, AND T. O'REILLY, ED. (1992). *The X Window System in a Nutshell*, Second Edition, O'Reilly & Assoc., Inc., Sebastopol, CA.
- CYRUS, M. AND J. BECK (1978). "Generalized Two- and Three-Dimensional Clipping", *Computers and Graphics*, 3(1), pp. 23-28.
- DAY, A. M. (1990). "The Implementation of an Algorithm to Find the Convex Hull of a Set of Three-Dimensional Points", *ACM Transactions on Graphics*, 9(1), pp. 105-132.
- DE REFFYE, P., C. EDELIN, J. FRANCON, ET AL. (1988). "Plant Models Faithful to Botanical Structure and Development", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 151-158.

- DEERING, M. (1992). "High Resolution Virtual Reality", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 195-202.
- DEERING, M. F. AND S. R. NELSON (1993). "Leo: A System for Cost-Effective 3D Shaded Graphics", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 101-108.
- DEMKO, S., L. HODGES, AND B. NAYLOR (1985). "Construction of Fractal Objects with Iterated Function Systems", in proceedings of SIGGRAPH '85, *Computer Graphics*, 19(3), pp. 271-278.
- DEPP, S. W. AND W. E. HOWARD (1993). "Flat-Panel Displays", *Scientific American*, 266(3), pp. 90-97.
- DEROSE, T. D. (1988). "Geometric Continuity, Shape Parameters, and Geometric Constructions for Catmull-Rom Splines", *ACM Transactions on Graphics*, 7(1), pp. 1-41.
- DIGITAL EQUIPMENT CORP. (1989). "Digital Equipment Corporation UI Style Guide", Maynard, MA.
- DIPPE, M. AND J. SWENSEN (1984). "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis", in proceedings of SIGGRAPH '84, *Computer Graphics*, 18(3), pp. 149-158.
- DOBKIN, D., L. GUIBAS, J. HERSHBERGER, ET AL. (1988). "An Efficient Algorithm for Finding the CSG Representation of a Simple Polygon", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 31-40.
- DOCTOR, L. J. AND J. G. TORBERG (1981). "Display Techniques for Octree-Encoded Objects", *IEEE Computer Graphics and Applications*, 1(3), pp. 29-38.
- DORSEY, J. O., F. X. SILLION, AND D. P. GREENBERG (1991). "Design and Simulation of Opera Lighting and Projection Effects", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 41-50.
- DREBIN, R. A., L. CARPENTER, AND P. HANRAHAN (1988). "Volume Rendering", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 65-74.
- DUFF, T. (1985). "Compositing 3D Rendered Images", in proceedings of SIGGRAPH '85, *Computer Graphics*, 19(3), pp. 41-44.
- DURRETT, H. J., ED. (1987). *Color and the Computer*, Academic Press, Boston.
- DUVANENKO, V. (1990). "Improved Line Segment Clipping", *Dr. Dobb's Journal*, July 1990.
- DYER, S. AND S. WHITMAN (1987). "A Vectorized Scan-Line Z-Buffer Rendering Algorithm", *IEEE Computer Graphics and Applications*, 7(7), pp. 34-45.
- DYER, S. (1990). "A Dataflow Toolkit for Visualization", *IEEE Computer Graphics and Applications*, 10(4), pp. 60-69.
- EARNSHAW, R. A., ED. (1985). *Fundamental Algorithms for Computer Graphics*, Springer-Verlag, Berlin.
- EDFELSBRUNNER, H. (1987). *Algorithms in Computational Geometry*, Springer-Verlag, Berlin.
- EDELSBRUNNER, H. AND E. P. MÜCKE (1990). "Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms", *ACM Transactions on Graphics*, 9(1), pp. 66-104.
- ELBER, G. AND E. COHEN (1990). "Hidden Curve Removal for Free Form Surfaces", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 95-104.
- ENDERLE, G., K. KANSY, AND G. PFAFF (1984). *Computer Graphics Programming: GKS—The Graphics Standard*, Springer-Verlag, Berlin.
- FARIN, G. (1988). *Curves and Surfaces for Computer Aided Geometric Design*, Academic Press, Boston, MA.
- FAROUKI, R. T. AND J. K. HINDS (1985). "A Hierarchy of Geometric Forms", *IEEE Computer Graphics and Applications*, 5(5), pp. 51-78.
- FEDER, J. (1988). *Fractals*, Plenum Press, New York.
- FEINER, S., S. NAGY, AND A. VAN DAM (1982). "An Experimental System for Creating and Presenting Interactive Graphical Documents", *ACM Transactions on Graphics*, 1(1), pp. 59-77.
- FERWERDA, J. A. AND D. P. GREENBERG (1988). "A Psychophysical Approach to Assessing the Quality of Antialiased Images", *IEEE Computer Graphics and Applications*, 8(5), pp. 85-95.
- FISHKIN, K. P. AND B. A. BARSKY (1984). "A Family of New Algorithms for Soft Filling", in proceedings of SIGGRAPH '84, *Computer Graphics*, 18(3), pp. 235-244.
- FIUME, E. L. (1989). *The Mathematical Structure of Raster Graphics*, Academic Press, Boston.
- FOLEY, J. D., V. L. WALLACE, AND P. CHAN (1984). "The Human Factors of Computer Graphics Interaction Techniques", *IEEE Computer Graphics and Applications*, 4(11), pp. 13-48.
- FOLEY, J. D. (1987). "Interfaces for Advanced Computing", *Scientific American*, 257(4), pp. 126-135.
- FOLEY, J. D., A. VAN DAM, S. K. FEINER, ET AL. (1990). *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, MA.
- FOURNIER, A., D. FUSSELL, AND L. CARPENTER (1982). "Computer Rendering of Stochastic Models", *CACM*, 25(6), pp. 371-384.
- FOURNIER, A. AND D. Y. MONTUNO (1984). "Triangulating Simple Polygons and Equivalent Problems", *ACM Transactions on Graphics*, 3(2), pp. 153-174.
- FOURNIER, A. AND W. T. REEVES (1986). "A Simple Model of Ocean Waves", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 75-84.
- FOURNIER, A. AND D. FUSSELL (1988). "On the Power of the Frame Buffer", *ACM Transactions on Graphics*, 7(2), pp. 103-128.
- FOURNIER, A. AND E. FIUME (1988). "Constant-Time Filtering with Space-Variant Kernels", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 229-238.
- FOWLER, D. R., H. MEINHARDT, AND P. PRUSINKIEWICZ (1992). "Modeling Seashells", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 379-387.
- FOX, D. AND M. WAITE (1984). *Computer Animation Primer*, McGraw-Hill, New York.
- FRANCIS, G. K. (1987). *A Topological Picturebook*, Springer-Verlag, New York.

- FRANKLIN, W. R. AND M. S. KANKANHALLI (1990). "Parallel Object-Space Hidden Surface Removal", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 87-94.
- FREEMAN, H. ED. (1980). *Tutorial and Selected readings in Interactive Computer Graphics*, IEEE Computer Society Press, Silver Springs, MD.
- FRENKEL, K. A. (1989). "Volume Rendering", *CACM*, 32(4), pp. 426-435.
- FRIEDER, G., D. GORDON, AND R. A. REYNOLD (1985). "Back-to-Front Display of Voxel-Based Objects", *IEEE Computer Graphics and Applications*, 5(1), pp. 52-60.
- FRIEDHOFF, R. M. AND W. BENZON (1989). *The Second Computer Revolution: Visualization*, Harry N. Abrams, Inc., New York.
- FUCHS, H., S. M. PIZER, E. R. HEINZ, S. H. BLOOMBER, L. TSAI, AND D. C. STRICKLAND (1982). "Design of and Image Editing with a Space-Filling Three-Dimensional Display Based on a Standard Raster Graphics System", *Proceedings of SPIE*, 367, August 1982, pp. 117-127.
- FUCHS, H., J. POULTON, J. EYLES, ET AL. (1989). "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 79-88.
- FUJIMOTO, A. AND K. IWATA (1983). "Jag-Free Images on Raster Displays", *IEEE Computer Graphics and Applications*, 3(9), pp. 26-34.
- FUNKHOUSER, T. A. AND C. H. SEQUIN (1993). "Adaptive Display Algorithms for Interactive Frame Rates During Visualization Complex Virtual Environments", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 247-254.
- GALYEAN, T. A. AND J. F. HUGHES (1991). "Sculpting: An Interactive Volumetric Modeling Technique", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 267-274.
- GARDNER, G. Y. (1985). "Visual Simulation of Clouds", in proceedings of SIGGRAPH '85, *Computer Graphics*, 19(3), pp. 297-304.
- GASCUEL, M.-P. (1993). "An Implicit Formulation for Precise Contact Modeling between Flexible Solids", in proceedings of SIGGRAPH '93, *Computer Graphics*, pp. 313-320.
- GASKINS, T. (1992). *PHIGS Programming Manual*, O'Reilly & Associates, Sebastopol, CA.
- GHARACHORLOO, N., S. GUPTA, R. F. SPROULL, ET AL. (1989). "A Characterization of Ten Rasterization Algorithms", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 355-368.
- GIRARD, M. (1987). "Interactive Design of 3D Computer-Animated Legged Animal Motion", *IEEE Computer Graphics and Applications*, 7(6), pp. 39-51.
- GLASSNER, A. S. (1984). "Space Subdivision for Fast Ray Tracing", *IEEE Computer Graphics and Applications*, 4(10), pp. 15-22.
- GLASSNER, A. S. (1986). "Adaptive Precision in Texture Mapping", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 297-306.
- GLASSNER, A. S. (1988). "Spacetime Ray Tracing for Animation", *IEEE Computer Graphics and Applications*, 8(2), pp. 60-70.
- GLASSNER, A. S., ED. (1989). *An Introduction to Ray Tracing*, Academic Press, San Diego, CA.
- GLASSNER, A. S., ED. (1990). *Graphics Gems*, Academic Press, San Diego, CA.
- GLASSNER, A. S. (1992). "Geometric Substitution: A Tutorial", *IEEE Computer Graphics and Applications*, 12(1), pp. 22-36.
- GLASSNER, A. S. (1994). *Principles of Digital Image Synthesis*, Morgan-Kaufmann, Inc., New York.
- GLEICHER, M. AND A. WITKIN (1992). "Through-the-Lens Camera Control", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 331-340.
- GOLDSMITH, J. AND J. SALMON (1987). "Automatic Creation of Object Hierarchies for Ray Tracing", *IEEE Computer Graphics and Applications*, 7(5), pp. 14-20.
- GONZALEZ, R. C. AND P. WINTZ (1987). *Digital Image Processing*, Addison-Wesley, Reading, MA.
- GOOD, D. M., J. A. WHITESIDE, D. R. WIXON, AND S. J. JONES (1984). "Building A User-Derived Interface", *CACM*, 27(10), pp. 1032-1042.
- GOODMAN, T. AND R. SPENCE (1978). "The Effect of System Response Time on Interactive Computer-Aided Problem Solving", in proceedings of SIGGRAPH '78, *Computer Graphics*, 12(3), pp. 100-104.
- GORAL, C. M., K. E. TORRANCE, D. P. GREENBERG, ET AL. (1984). "Modeling the Interaction of Light Between Diffuse Surfaces", in proceedings of SIGGRAPH '84, *Computer Graphics*, 18(3), pp. 213-222.
- GORDON, D. AND S. CHEN (1991). "Front-to-Back Display of BSP Trees", *IEEE Computer Graphics and Applications*, 11(5), pp. 79-85.
- GORTLER, S. J., P. SCHRODER, M. F. COHEN, ET AL. (1993). "Wavelet Radiosity", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 221-230.
- GREEN, M. (1985). "The University of Alberta User Interface Management System", in proceedings of SIGGRAPH '85, *Computer Graphics*, 19(3), pp. 205-214.
- GREENE, N., M. KASS, AND G. MILLER (1993). "Hierarchical Z-Buffer Visibility", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 231-238.
- HAEBERLI, P. AND K. AKELLY (1990). "The Accumulation Buffer: Hardware Support for High-Quality Rendering", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 309-318.
- HAHN, J. K. (1988). "Realistic Animation of Rigid Bodies", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 299-308.
- HALL, R. A. AND D. P. GREENBERG (1983). "A Testbed for Realistic Image Synthesis", *IEEE Computer Graphics and Applications*, 3(8), pp. 10-20.
- HALL, R. (1989). *Illumination and Color in Computer Generated Imagery*, Springer-Verlag, New York.

- HANRAHAN, P. (1982). "Creating Volume Models from Edge-Vertex Graphs", in proceedings of SIGGRAPH '82, *Computer Graphics*, 16(3), pp. 77-84.
- HANRAHAN, P. AND J. LAWSON (1990). "A Language for Shading and Lighting Calculations", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 289-298.
- HART, J. C., D. J. SANDIN, AND L. H. KAUFFMAN (1989). "Ray Tracing Deterministic 3D Fractals", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 289-296.
- HART, J. C. AND T. A. DEFANTI (1991). "Efficient Antialiased Rendering of 3-D Linear Fractals", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 91-100.
- HE, X. D., P. O. HEYNEN, R. L. PHILLIPS, ET AL. (1992). "A Fast and Accurate Light Reflection Model", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 253-254.
- HEARN, D. AND P. BAKER (1991). "Scientific Visualization: An Introduction", *Eurographics '91 Technical Report Series*, Tutorial Lecture 6.
- HECKBERT, P. (1982). "Color Image Quantization for Frame Buffer Display", in proceedings of SIGGRAPH '82, *Computer Graphics*, 16(3), pp. 297-307.
- HECKBERT, P. AND P. HANRAHAN (1984). "Beam Tracing Polygonal Objects", in proceedings of SIGGRAPH '84, *Computer Graphics*, 18(3), pp. 119-127.
- HOPGOOD, F. R. A., D. A. DUCE, J. R. GALLOP, ET AL. (1983) *Introduction to the Graphical Kernel System (GKS)*, Academic Press, London.
- HOPGOOD, F. R. A. AND D. A. DUCE (1991). *A Primer for PHIGS*, John Wiley & Sons, Chichester, England.
- HOPPE, H., T. DEROSE, T. McDONALD, ET AL. (1993). "Mesh Optimization", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 19-26.
- HOWARD, T. L. J., W. T. HEWITT, R. J. HUBBOLD, ET AL. (1991). *A Practical Introduction to PHIGS and PHIGS Plus*, Addison-Wesley, Wokingham, England.
- HUGHES, J. F. (1992). "Scheduled Fourier Volume Morphing", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 43-46.
- HUITRIC, H. AND M. NAHAS (1985). "B-Spline Surfaces: A Tool for Computer Painting", *IEEE Computer Graphics and Applications*, 5(3), pp. 39-47.
- IKEDO, T. (1984). "High-Speed Techniques for a 3-D Color Graphics Terminal", *IEEE Computer Graphics and Applications*, 4(5), pp. 46-58.
- IMMEL, D. S., M. F. COHEN, AND D. P. GREENBERG (1986). "A Radiosity Method for Non-Diffuse Environments", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 133-142.
- ISAACS, P. M. AND M. F. COHEN (1987). "Controlling Dynamic Simulation with Kinematic Constraints, Behavior Functions, and Inverse Dynamics", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 215-224.
- JARVIS, J. F., C. N. JUDICE, AND W. H. NINKE (1976). "A Survey of Techniques for the Image Display of Continuous Tone Pictures on Bilevel Displays", *Computer Graphics and Image Processing*, 5(1), pp. 13-40.
- JOHNSON, S. A. (1982). "Clinical Varifocal Mirror Display System at the University of Utah", *Proceedings of SPIE*, 367, August 1982, pp. 145-148.
- KAJIYA, J. T. (1983). "New Techniques for Ray Tracing Procedurally Defined Objects", *ACM Transactions on Graphics*, 2(3), pp. 161-181.
- KAJIYA, J. T. (1986). "The Rendering Equation", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 143-150.
- KAJIYA, J. T. AND T. L. KAY (1989). "Rendering Fur with Three-Dimensional Textures", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 271-280.
- KAPPEL, M. R. (1985). "An Ellipse-Drawing Algorithm for Faster Displays", in *Fundamental Algorithms for Computer Graphics*, Springer-Verlag, Berlin, pp. 257-280.
- KARASICK, M., D. LIEBER, AND L. R. NACKMAN (1991). "Efficient Delaunay Triangulation Using Rational Arithmetic", *ACM Transactions on Graphics*, 10(1), pp. 71-91.
- KASS, M. (1992). "CONDOR: Constraint-Based Dataflow", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 321-330.
- KASSON, J. M. AND W. PLOUFFE (1992). "An Analysis of Selected Computer Interchange Color Spaces", *ACM Transactions on Graphics*, 11(4), pp. 373-405.
- KAUFMAN, A. (1987). "Efficient Algorithms for 3D Scan-Conversion of Parametric Curves, Surfaces, and Volumes", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 171-179.
- KAWAGUCHI, Y. (1982). "A Morphological Study of the Form of Nature", in proceedings of SIGGRAPH '82, *Computer Graphics*, 16(3), pp. 223-232.
- KAY, T. L. AND J. T. KAJIYA (1986). "Ray Tracing Complex Scenes", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 269-278.
- KAY, D. C. AND J. R. LEVINE (1992). *Graphics File Formats*, Windcrest/McGraw-Hill, New York.
- KELLEY, A. D., M. C. MALIN, AND G. M. NIELSON (1988). "Terrain Simulation using a Model of Stream Erosion", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 263-268.
- KENT, J. R., W. E. CARLSON, AND R. E. PARENT (1992). "Shape Transformation for Polyhedral Objects", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 47-54.
- KIRK, D. AND J. ARVO (1991). "Unbiased Sampling Techniques for Image Synthesis", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 153-156.
- KIRK, D., ED. (1992). *Graphics Gems III*, Academic Press, San Diego, CA.
- KNUTH, D. E. (1987). "Digital Halftones by Dot Diffusion", *ACM Transactions on Graphics*, 6(4), pp. 245-273.
- KOCHANEK, D. H. U. AND R. H. BARTELS (1984). "Interpolating Splines with Local Tension, Continuity, and Bias Control", in proceedings of SIGGRAPH '84, *Computer Graphics*, 18(3), pp. 33-41.

- KOH, E.-K. AND D. HEARN (1992). "Fast Generation and Surface Structuring Methods for Terrain and Other Natural Phenomena", in proceedings of Eurographs '92 *Computer Graphics Forum*, 11(3), pp. C-169-180.
- KORIEN, J. U. AND N. I. BADLER (1982). "Techniques for Generating the Goal-Directed Motion of Articulated Structures", *IEEE Computer Graphics and Applications*, 2(9), pp. 71-81.
- KORIEN, J. U. AND N. I. BADLER (1983). "Temporal antialiasing in Computer-Generated Animation", in proceedings of SIGGRAPH '83, *Computer Graphics*, 17(3), pp. 377-388.
- LASSETER, J. (1987). "Principles of Traditional Animation Applied to 3D Computer Animation", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 35-44.
- LAUR, D. AND P. HANRAHAN (1991). "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 285-288.
- LAUREL, B. (1990). *The Art of Human-Computer Interface Design*, Addison-Wesley, Reading, MA.
- LEE, M. E., R. A. REDNER, AND S. P. USELTON (1985). "Statistically Optimized Sampling for Distributed Ray Tracing", in proceedings of SIGGRAPH '85, *Computer Graphics*, 19(3), pp. 61-68.
- LEVINATHAL, A. AND T. PORTER (1984). "CHAP — A SIMD Graphics Processor", in proceedings of SIGGRAPH '84, *Computer Graphics*, 18(3), pp. 77-82.
- LEVOY, M. (1988). "Display of Surfaces from Volume Data", *IEEE Computer Graphics and Applications*, 8(3), pp. 29-37.
- LEVOY, M. (1990). "A Hybrid Ray Tracer for Rendering Polygon and Volume Data", *IEEE Computer Graphics and Applications*, 10(2), pp. 33-40.
- LEWIS, J.-P. (1989). "Algorithms for Solid Noise Synthesis", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 263-270.
- LIANG, Y.-D. AND B. A. BARSKY (1983). "An Analysis and Algorithm for Polygon Clipping", *CACM*, 26(11), pp. 868-877.
- LIANG, Y.-D. AND B. A. BARSKY (1984). "A New Concept and Method for Line Clipping", *ACM Transactions on Graphics*, 3(1), pp. 1-22.
- LIEN, S.-L., M. SHANTZ, AND V. PRATT (1987). "Adaptive Forward Differencing for Rendering Curves and Surfaces", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 111-118.
- LINDLEY, C. A. (1992). *Practical Ray Tracing in C*, John Wiley & Sons, New York.
- LISCHINSKI, D., F. TAMPIERI, AND D. P. GREENBERG (1993). "Combining Hierarchical Radiosity and Discontinuity Meshing", in proceedings of SIGGRAPH '93, *Computer Graphics*, pp. 199-208.
- LITWINOWICZ, P. C. (1991). "Inkwell: A 2 1/2-D Animation System", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 113-122.
- LODDING, K. N. (1983). "Iconic Interfacing", *IEEE Computer Graphics and Applications*, 3(2), pp. 11-20.
- LOKE, T.-S., D. TAN, H.-S. SEAH, ET AL. (1992). "Rendering Fireworks Displays", *IEEE Computer Graphics and Applications*, 12(3), pp. 33-43.
- LOOMIS, J., H. POIZNER, U. BELLUGI, ET AL. (1983). "Computer Graphic Modeling of American Sign Language", in proceedings of SIGGRAPH '83, *Computer Graphics*, 17(3), pp. 105-114.
- LORENSEN, W. E. AND H. CLINE (1987). "Marching Cubes: A High-Resolution 3D Surface Construction Algorithm", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 163-169.
- MACKINLAY, J. D., S. K. CARD, AND G. G. ROBERTSON (1990). "Rapid Controlled Movement Through a Virtual 3D Workspace", *SIGGRAPH 90*, pp. 171-176.
- MACKINLAY, J. D., G. G. ROBERTSON, AND S. K. CARD (1991). "The Perspective Wall: Detail and Context Smoothly Integrated", *CHI '91*, pp. 173-179.
- MAGENAT-THALMANN, N. AND D. THALMANN (1985). *Computer Animation: Theory and Practice*, Springer-Verlag, Tokyo.
- MAGENAT-THALMANN, N. AND D. THALMANN (1987). *Image Synthesis*, Springer-Verlag, Tokyo.
- MAGENAT-THALMANN, N. AND D. THALMANN (1991). "Complex Models for Animating Synthetic Actors", *IEEE Computer Graphics and Applications*, 11(5), pp. 32-45.
- MANDELBROT, B. B. (1977). *Fractals: Form, Chance, and Dimension*, Freeman Press, San Francisco.
- MANDELBROT, B. B. (1982). *The Fractal Geometry of Nature*, Freeman Press, New York.
- MANTYLA, M. (1988). *An Introduction to Solid Modeling*, Computer Science Press, Rockville, MD.
- MAX, N. L. AND D. M. LERNER (1985). "A Two-and-a-Half-D Motion Blur Algorithm", in proceedings of SIGGRAPH '85, *Computer Graphics*, 19(3), pp. 85-94.
- MAX, N. L. (1986). "Atmospheric Illumination and Shadows", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 117-124.
- MAX, N. L. (1990). "Cone-Spheres", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 59-62.
- METAXAS, D. AND D. TERZOPoulos (1992). "Dynamic Deformation of Solid Primitives with Constraints", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 309-312.
- MEYER, G. W., H. E. RUSHMEIER, M. F. COHEN, ET AL. (1986). "An Experimental Evaluation of Computer Graphics Imagery", *ACM Transactions on Graphics*, 6(1), pp. 30-50.
- MEYER, G. W. AND D. P. GREENBERG (1988). "Color-Defective Vision and Computer Graphics Displays", *IEEE Computer Graphics and Applications*, 8(5), pp. 28-40.
- MEYERS, D., S. SKINNER, AND K. SLOAN (1992). "Surfaces from Contours", *ACM Transactions on Graphics*, 11(3), pp. 228-258.
- MILLER, G. S. P. (1988). "The Motion Dynamics of Snakes and Worms", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 169-178.

- MILLER, J. V., D. E. BREEN, W. E. LORENSON, ET AL. (1991). "Geometrically Deformed Models: A Method for Extracting Closed Geometric Models from Volume Data", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 217-226.
- MITCHELL, D. P. (1991). "Spectrally Optimal Sampling for Distribution Ray Tracing", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 157-165.
- MITCHELL, D. P. AND P. HANRAHAN (1992). "Illumination from Curved Reflectors", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 283-291.
- MIYATA, K. (1990). "A Method of Generating Stone Wall Patterns", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 387-394.
- MOLNAR, S., J. EYLES, AND J. POULTON (1992). "PixelFlow: High-Speed Rendering Using Image Composition", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 231-240.
- MOON, F. C. (1992). *Chaotic and Fractal Dynamics*, John Wiley & Sons, New York.
- MOORE, M. AND J. WILHELMS (1988). "Collision Detection and Response for Computer Animation", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 289-298.
- MORTENSON, M. E. (1985). *Geometric Modeling*, John Wiley & Sons, New York.
- MURAKI, S. (1991). "Volumetric Shape Description of Range Data Using the 'Bobby Model'", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 227-235.
- MUSGRAVE, F. K., C. E. KOLB, AND R. S. MACE (1989). "The Synthesis and Rendering of Eroded Fractal Terrains", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 41-50.
- MYERS, B. A. AND W. BUXTON (1986). "Creating High-Interactive and Graphical User Interfaces by Demonstration", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 249-258.
- NAYLOR, B., J. AMANATIDES, AND W. THIBAUT (1990). "Merging BSP Trees Yields Polyhedral Set Operations", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 115-124.
- NEWMAN, W. H. (1968). "A System for Interactive Graphical Programming", *SJCC*, Thompson Books, Washington, D. C., pp. 47-54.
- NEWMAN, W. H. AND R. F. SPROULL (1979). *Principles of Interactive Computer Graphics*, McGraw-Hill, New York.
- NGO, J. T. AND J. MARKS (1993). "Spacetime Constraints Revisited", in proceedings of SIGGRAPH '93, *Computer Graphics*, pp. 343-350.
- NICHOLL, T. M., D. T. LEE, AND R. A. NICHOLL (1987). "An Efficient New Algorithm for 2D Line Clipping: Its Development and Analysis", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 253-262.
- NIELSON, G. M., B. SHRIVER, AND L. ROSENBLUM, ED. (1990). *Visualization in Scientific Computing*, IEEE Computer Society Press, Los Alamitos, CA.
- NIELSON, G. M. (1993). "Scattered Data Modeling", *IEEE Computer Graphics and Applications*, 13(1), pp. 60-70.
- NISHIMURA, H. (1985). "Object Modeling by Distribution Function and a Method of Image Generation", *Journal Electronics Comm. Conf.* '85, 168(4), pp. 718-725.
- NISHITA, T. AND E. NAKAMAE (1986). "Continuous-Tone Representation of Three-Dimensional Objects Illuminated by Sky Light", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 125-132.
- NISHITA, T., T. SIRAI, K. TADAMURA, ET AL. (1993). "Display of the Earth Taking into Account Atmospheric Scattering", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 175-182.
- NORTON, A. (1982). "Generation and Display of Geometric Fractals in 3-D", in proceedings of SIGGRAPH '82, *Computer Graphics*, 16(3), pp. 61-67.
- NSF INVITATIONAL WORKSHOP (1992). "Research Directions in Virtual Environments", *Computer Graphics*, 26(3), pp. 153-177.
- OKABE, H., H. IMAOKA, T. TOMIHA, ET AL. (1992). "Three-Dimensional Apparel CAD System", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 105-110.
- OPENGL ARCHITECTURE REVIEW BOARD (1993). *OpenGL Programming Guide*, Addison-Wesley, Reading, MA.
- OPPENHEIMER, P. E. (1986). "Real-Time Design and Animation of Fractal Plants and Trees", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 55-64.
- OSF/MOTIF (1989). *OSF/Motif Style Guide*, Open Software Foundation, Prentice-Hall, Englewood Cliffs, NJ.
- PAINTER, J. AND K. SLOAN (1989). "Antialiased Ray Tracing by Adaptive Progressive Refinement", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 281-288.
- PANG, A. T. (1990). "Line-Drawing Algorithms for Parallel Machines", *IEEE Computer Graphics and Applications*, 10(5), pp. 54-59.
- PAVLIDIS, T. (1982). *Algorithms For Graphics and Image Processing*, Computer Science Press, Rockville, MD.
- PAVLIDIS, T. (1983). "Curve Fitting with Conic Splines", *ACM Transactions on Graphics*, 2(1), pp. 1-31.
- PEACHEY, D. R. (1986). "Modeling Waves and Surf", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 65-74.
- PEITGEN, H.-O. AND P. H. RICHTER (1986). *The Beauty of Fractals*, Springer-Verlag, Berlin.
- PEITGEN, H.-O. AND D. SAUPE, ED. (1988). *The Science of Fractal Images*, Springer-Verlag, Berlin.
- PENTLAND, A. AND J. WILLIAMS (1989). "Good Vibrations: Modal Dynamics for Graphics and Animation", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 215-222.
- PERLIN, K. AND E. M. HOFFERT (1989). "Hypertexture", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 253-262.
- PHILLIPS, R. L. (1977). "A Query Language for a Network Data Base with Graphical Entities", in proceedings of SIGGRAPH '77, *Computer Graphics*, 11(2), pp. 179-185.

- PHONG, B. T. (1975). "Illumination for Computer-Generated Images", *CACM*, 18(6), pp. 311-317.
- PINEDA, J. (1988). "A Parallel Algorithm for Polygon Rasterization", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 17-20.
- PITTEWAY, M. L. V. AND D. J. WATKINSON (1980). "Bresenham's Algorithm with Gray Scale", *CACM*, 23(11), pp. 625-626.
- PLATT, J. C. AND A. H. BARR (1988). "Constraint Methods for Flexible Models", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 279-288.
- PORTER, T. AND T. DUFF (1984). "Compositing Digital Images", in proceedings of SIGGRAPH '84, *Computer Graphics*, 18(3), pp. 253-259.
- POTMESIL, M. AND I. CHAKRAVARTY (1982). "Synthetic Image Generation with a Lens and Aperture Camera Model" *ACM Transactions on Graphics*, 1(2), pp. 85-108.
- POTMESIL, M. AND I. CHAKRAVARTY (1983). "Modeling Motion Blur in Computer-Generated Images", in proceedings of SIGGRAPH '83, *Computer Graphics*, 17(3), pp. 389-399.
- POTMESIL, M. AND E. M. HOFFERT (1987). "FRAMES: Software Tools for Modeling, Rendering and Animation of 3D Scenes", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 85-93.
- POTMESIL, M. AND E. M. HOFFERT (1989). "The Pixel Machine: A Parallel Image Computer", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 69-78.
- PRATT, W. K. (1). *Digital Image Processing*, John Wiley & Sons, New York.
- PREPARATA, F. P. AND M. I. SHAMOS (1985). *Computational Geometry*, Springer-Verlag, New York.
- PRESS, W. H., S. A. TEUKOLSKY, W. T. VETTERLING, ET AL. (1992). *Numerical Recipes in C*, Cambridge University Press, Cambridge, England.
- PRUSINKIEWICZ, P., M. S. HAMMEL, AND E. MJOLNESS (1993). "Animation of Plant Development", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 351-360.
- PRUYN, P. W. AND D. P. GREENBERG (1993). "Exploring 3D Computer Graphics in Cockpit Avionics", *IEEE Computer Graphics and Applications*, 13(3), pp. 28-35.
- QUEK, L.-H. AND D. HEARN (1988). "Efficient Space-Subdivision Methods in Ray-Tracing Algorithms", University of Illinois, Department of Computer Science Report UIUCDCS-R-88-1468.
- RAIBERT, M. H. AND J. K. HODGINS (1991). "Animation of Dynamic Legged Locomotion", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 349-358.
- REEVES, W. T. (1983). "Particle Systems: A Technique for Modeling a Class of Fuzzy Objects", *ACM Transactions on Graphics*, 2(2), pp. 91-108.
- REEVES, W. T. (1983). "Particle Systems—A Technique for Modeling a Class of Fuzzy Objects", in proceedings of SIGGRAPH '83, *Computer Graphics*, 17(3), pp. 359-376.
- REEVES, W. T. AND R. BLAU (1985). "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems", in proceedings of SIGGRAPH '85, *Computer Graphics*, 19(3), pp. 313-321.
- REEVES, W. T., D. H. SALESIN, AND R. L. COOK (1987). "Rendering Antialiased Shadows with Depth Maps", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 283-291.
- REQUICHA, A. A. G. AND J. R. ROSSIGNAC (1992). "Solid Modeling and Beyond", *IEEE Computer Graphics and Applications*, 12(5), pp. 31-44.
- REYNOLDS, C. W. (1982). "Computer Animation with Scripts and Actors", in proceedings of SIGGRAPH '82, *Computer Graphics*, 16(3), pp. 289-296.
- REYNOLDS, C. W. (1987). "Flocks, Herds, and Schools: A Distributed Behavioral Model", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 25-34.
- RIESENFIELD, R. F. (1981). "Homogeneous Coordinates and Projective Planes in Computer Graphics", *IEEE Computer Graphics and Applications*, 1(1), pp. 50-55.
- ROBERTSON, P. K. (1988). "Visualizing Color Gamuts: A User Interface for the Effective Use of Perceptual Color Spaces in Data Displays", *IEEE Computer Graphics and Applications*, 8(5), pp. 50-64.
- ROBERTSON, G. G., J. D. MACKINLAY AND S. K. CARD (1991). "Cone Trees: Animated 3D Visualizations of Hierarchical Information", *CHI '91*, pp. 189-194.
- ROGERS, D. F. AND R. A. EARNshaw, ED. (1987). *Techniques for Computer Graphics*, Springer-Verlag, New York.
- ROGERS, D. F. AND J. A. ADAMS (1990). *Mathematical Elements for Computer Graphics*, McGraw-Hill, New York.
- ROSENTHAL, D. S. H., ET AL. (1982). "The Detailed Semantics of Graphics Input Devices", in proceedings of SIGGRAPH '82, *Computer Graphics*, 16(3), pp. 33-38.
- RUBINE, D. (1991). "Specifying Gestures by Example", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 329-337.
- RUSHMEIER, H. AND K. TORRANCE (1987). "The Zonal Method for Calculating Light Intensities in the Presence of a Participating Medium", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 293-302.
- RUSHMEIER, H. E. AND K. E. TORRANCE (1990). "Extending the Radiosity Method to Include Specularly Reflecting and Translucent Materials", *ACM Transactions on Graphics*, 9(1), pp. 1-27.
- SABELLA, P. (1988). "A Rendering Algorithm for Visualizing 3D Scalar Fields", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 51-58.
- SABIN, M. A. (1985). "Contouring: The State of the Art", in *Fundamental Algorithms for Computer Graphics*, R. A. Earnshaw, ed, Springer-Verlag, Berlin, pp. 411-482.
- SALESIN, D. AND R. BARZEL (1993). "Adjustable Tools: An Object-Oriented Interaction Metaphor", *ACM Transactions on Graphics*, 12(1), pp. 103-107.
- SAMET, H. AND R. E. WEBBER (1985). "Sorting a Collection of Polygons using Quadrees", *ACM Transactions on Graphics*, 4(3), pp. 182-222.

- SAMET, H. AND M. TAMMINEN (1985). "Bintrees, CSG Trees, and Time", in proceedings of SIGGRAPH '85, *Computer Graphics*, 19(3), pp. 121-130.
- SAMET, H. AND R. E. WEBBER (1988). "Hierarchical Data Structures and Algorithms for Computer Graphics: Part 1", *IEEE Computer Graphics and Applications*, 8(4), pp. 59-75.
- SAMET, H. AND R. E. WEBBER (1988). "Hierarchical Data Structures and Algorithms for Computer Graphics: Part 2", *IEEE Computer Graphics and Applications*, 8(3), pp. 48-68.
- SCHIEFLER, R. W. AND J. GETTYS (1986). "The X Window System", *ACM Transactions on Graphics*, 5(2), pp. 79-109.
- SCHOENEMAN, C., J. DORSEY, B. SMITS, ET AL. (1993). "Global Illumination", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 143-146.
- SCHRODER, P. AND P. HANRAHAN (1993). "On the Form Factor Between Two Polygons", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 163-164.
- SCHWARTZ, M. W., W. B. COWAN, AND J. C. BEATTY (1987). "An Experimental Comparison of RGB, YIQ, LAB, HSV, and Opponent Color Models", *ACM Transactions on Graphics*, 6(2), pp. 123-158.
- SEDERBERG, T. W. AND E. GREENWOOD (1992). "A Physically Based Approach to 2-D Shape Bending", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 25-34.
- SEDERBERG, T. W., P. GAO, G. WANG, ET AL. (1993). "2D Shape Blending: An Intrinsic Solution to the Vertex Path Problem", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 15-18.
- SEGAL, M. (1990). "Using Tolerances to Guarantee Valid Polyhedral Modeling Results", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 105-114.
- SEGAL, M., C. KOROBKIN, R. VAN WIDENFELT, ET AL. (1992). "Fast Shadows and Lighting Effects Using Texture Mapping", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 249-252.
- SEQUIN, C. H. AND E. K. SMYRL (1989). "Parameterized Ray-Tracing", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 307-314.
- SHERR, S. (1993). *Electronic Displays*, John Wiley & Sons, New York.
- SHILLING, A. AND W. STRASSER (1993). "EXACT: Algorithm and Hardware Architecture for an Improved A-Buffer", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 85-92.
- SHIRLEY, P. (1990). "A Ray Tracing Method for Illumination Calculation in Diffuse-Specular Scenes", *Graphics Interface '90*, pp. 205-212.
- SHNEIDERMAN, B. (1986). *Designing the User Interface*, Addison-Wesley, Reading, MA.
- SHOEMAKE, K. (1985). "Animating Rotation with Quaternion Curves", in proceedings of SIGGRAPH '85, *Computer Graphics*, 19(3), pp. 245-254.
- SIBERT, J. L., W. D. HURLEY, AND T. W. BLESER (1986). "An Object-Oriented User Interface Management System", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 259-268.
- SILLION, F. X. AND C. PUECH (1989). "A General Two-Pass Method Integrating Specular and Diffuse Reflection", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 335-344.
- SILLION, F. X., J. R. ARVO, S. H. WESTIN, ET AL. (1991). "A Global Illumination Solution for General Reflectance Distributions", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 187-196.
- SIMS, K. (1990). "Particle Animation and Rendering Using Data Parallel Computation", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 405-413.
- SIMS, K. (1991). "Artificial Evolution for Computer Graphics", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 319-328.
- SINGH, B., J. C. BEATTY, K. S. BOOTH, ET AL. (1983). "A Graphics Editor for Benesh Movement Notation", in proceedings of SIGGRAPH '83, *Computer Graphics*, 17(3), pp. 51-62.
- SMITH, A. R. (1978). "Color Gamut Transform Pairs", *Computer Graphics*, 12(3), pp. 12-19.
- SMITH, A. R. (1979). "Tint Fill", *Computer Graphics*, 13(2), pp. 276-283.
- SMITH, A. R. (1984). "Plants, Fractals, and Formal Languages", in proceedings of SIGGRAPH '84, *Computer Graphics*, 18(3), pp. 1-10.
- SMITH, R. B. (1987). "Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic", *IEEE Computer Graphics and Applications*, 7(9), pp. 42-50.
- SMITH, A. R. (1987). "Planar 2-Pass Texture Mapping and Warping", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 263-272.
- SMITS, B. E., J. R. ARVO, AND D. H. SALESIN (1992). "An Importance-Driven Radiosity Algorithm", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 273-282.
- SNYDER, J. M. AND J. T. KAJIYA (1992). "Generative Modeling: A Symbolic System for Geometric Modeling", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 369-378.
- SNYDER, J. M., A. R. WOODBURY, K. FLEISCHER, ET AL. (1993). "Interval Method for Multi-Point Collisions between Time-Dependent Curved Surfaces", in proceedings of SIGGRAPH '93, *Computer Graphics*, pp. 321-334.
- SPOULL, R. F. AND I. E. SUTHERLAND (1968). "A Clipping Divider", *AFIPS Fall Joint Computer Conference*.
- STAM, J. AND E. FIUME (1993). "Turbulent Wind Fields for Gaseous Phenomena", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 369-376.
- STETTNER, A. AND D. P. GREENBERG (1989). "Computer Graphics Visualization for Acoustic Simulation", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 195-206.

- STRASSMANN, S. (1986). "Hairy Brushes", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 225-232.
- STRAUSS, P. S. AND R. CAREY (1992). "An Object-Oriented 3D Graphics Toolkit", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 341-349.
- SUNG, H. C. K., G. ROGERS, AND W. J. KUBITZ (1990). "A Critical Evaluation of PEX", *IEEE Computer Graphics and Applications*, 10(6), pp. 65-75.
- SUTHERLAND, I. E. (1963). "Sketchpad: A Man-Machine Graphical Communication System", AFIPS Spring Joint Computer Conference, 23 pp. 329-346.
- SUTHERLAND, I. E., R. F. SPROULL, AND R. SCHUMACKER (1974). "A Characterization of Ten Hidden Surface Algorithms", *ACM Computing Surveys*, 6(1), pp. 1-55.
- SUTHERLAND, I. E. AND G. W. HODGMAN (1974). "Reentrant Polygon Clipping", *CACM*, 17(1), pp. 32-42.
- SWEZEY, R. W. AND E. G. DAVIS (1983). "A Case Study of Human Factors Guidelines in Computer Graphics", *IEEE Computer Graphics and Applications*, 3(8), pp. 21-30.
- TAKALA, T. AND J. HAHN (1992). "Sound Rendering", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 211-220.
- TANNAS, J., LAWRENCE E., ED. (1985). *Flat-Panel Displays and CRTs*, Van Nostrand Reinhold Company, New York.
- TELLER, S. AND P. HANRAHAN (1993). "Global Visibility Algorithms for Illumination Computations", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 239-246.
- TERZOPOULOS, D., J. PLATT, A. H. BARR, ET AL. (1987). "Elastically Deformable Models", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 205-214.
- THALMANN, D., ED. (1990). *Scientific Visualization and Graphics Simulation*, John Wiley & Sons, Chichester, England.
- THIBAUT, W. C. AND B. F. NAYLOR (1987). "Set Operations on Polyhedra using Binary Space Partitioning Trees", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 153-162.
- TORBERG, J. G. (1987). "A Parallel Processor Architecture for Graphics Arithmetic Operations", in proceedings of SIGGRAPH '87, *Computer Graphics*, 21(4), pp. 197-204.
- TORRANCE, K. E. AND E. M. SPARROW (1967). "Theory for Off-Specular Reflection from Roughened Surfaces", *J. Optical Society of America*, 57(9), pp. 1105-1114.
- TRAVIS, D. (1991). *Effective Color Displays*, Academic Press, London.
- TUFTE, E. R. (1983). *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, CN.
- TUFTE, E. R. (1990). *Envisioning Information*, Graphics Press, Cheshire, CN.
- TURKOWSKI, K. (1982). "Antialiasing Through the Use of Coordinate Transformations", *ACM Transactions on Graphics*, 1(3), pp. 215-234.
- UPSON, C. AND M. KEELER (1988). "VBUFFER: Visible Volume Rendering", in proceedings of SIGGRAPH '88, *Computer Graphics*, 22(4), pp. 59-64.
- UPSON, C., T. FAULHABER JR., D. KAMINS, ET AL. (1989). "The Application Visualization System: A Computational Environment for Scientific Visualization", *IEEE Computer Graphics and Applications*, 9(4), pp. 30-42.
- UPSTILL, S. (1990). *The RenderMan Companion*, Addison-Wesley, Reading, MA.
- VAN DE PANNE, M. AND E. FILME (1993). "Sensor-Actuator Networks", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 335-342.
- VAN WIJK, J. J. (1991). "Spot Noise-Texture Synthesis for Data Visualization", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 309-318.
- VEENSTRA, J. AND N. AHUJA (1988). "Line Drawings of Octree-Represented Objects", *ACM Transactions on Graphics*, 7(1), pp. 61-75.
- VELHO, L. AND J. D. M. GOMES (1991). "Digital Halftoning with Space-Filling Curves", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 81-90.
- VON HERZEN, B., A. H. BARR, AND H. R. ZATZ (1990). "Geometric Collisions for Time-Dependent Parametric Surfaces", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 39-48.
- WALLACE, V. L. (1976). "The Semantics of Graphic Input Devices", in proceedings of SIGGRAPH '76, *Computer Graphics*, 10(1), pp. 61-65.
- WALLACE, J. R., K. A. ELMQUIST, AND E. A. HAINES (1989). "A Ray-Tracing Algorithm for Progressive Radiosity", in proceedings of SIGGRAPH '89, *Computer Graphics*, 23(3), pp. 315-324.
- WANGER, L. R., J. A. FERWERDA, AND D. P. GREENBERG (1992). "Perceiving Spatial Relationships in Computer-Generated Images", *IEEE Computer Graphics and Applications*, 12(3), pp. 44-58.
- WARE, C. (1988). "Color Sequences for Univariate Maps: Theory, Experiments, and Principles", *IEEE Computer Graphics and Applications*, 8(5), pp. 41-49.
- WARN, D. R. (1983). "Lighting Controls for Synthetic Images", in proceedings of SIGGRAPH '83, *Computer Graphics*, 17(3), pp. 13-21.
- WARNOCK, J. AND D. K. WYATT (1982). "A Device-Independent Graphics Imaging Model for Use with Raster Devices", in proceedings of SIGGRAPH '82, *Computer Graphics*, 16(3), pp. 313-319.
- WATT, A. (1989). *Fundamentals of Three-Dimensional Computer Graphics*, Addison-Wesley, Wokingham, England.
- WATT, M. (1990). "Light-Water Interaction Using Backward Beam Tracing", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 377-386.
- WATT, A. AND M. WATT (1992). *Advanced Animation and Rendering Techniques*, Addison-Wesley, Wokingham, England.
- WEGHORST, H., G. HOOPER, AND D. P. GREENBERG (1984). "Improved Computational Methods for Ray Tracing", *ACM Transactions on Graphics*, 3(1), pp. 52-69.
- WEIL, J. (1986). "The Synthesis of Cloth Objects", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 49-54.

- WEILER, K. AND P. ATHERTON (1977). "Hidden-Surface Removal Using Polygon Area Sorting", in proceedings of SIGGRAPH '77, *Computer Graphics*, 11(2), pp. 214-222.
- WEILER, K. (1980). "Polygon Comparison Using a Graph Representation", in proceedings of SIGGRAPH '80, *Computer Graphics*, 14(3), pp. 10-18.
- WESTIN, S. H., J. R. ARVO, AND K. E. TORRANCE (1992). "Predicting Reflectance Functions from Complex Surfaces", in proceedings of SIGGRAPH '92, *Computer Graphics*, 26(2), pp. 255-264.
- WESTOVER, L. (1990). "Footprint Evaluation for Volume Rendering", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 367-376.
- WHITTED, T. (1980). "An Improved Illumination Model for Shaded Display", *CACM*, 23(6), pp. 343-349.
- WHITTED, T. AND D. M. WEIMER (1982). "A Software Testbed for the Development of 3D Raster Graphics Systems", *ACM Transactions on Graphics*, 1(1), pp. 43-58.
- WHITTED, T. (1983). "Antialiased Line Drawing Using Brush Extrusion", in proceedings of SIGGRAPH '83, *Computer Graphics*, 17(3), pp. 151-156.
- WILHELMS, J. (1987). "Toward Automatic Motion Control", *IEEE Computer Graphics and Applications*, 7(4), pp. 11-22.
- WILHELMS, J. AND A. V. GELDER (1991). "A Coherent Projection Approach for Direct Volume Rendering", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 275-284.
- WILHELMS, J. AND A. VAN GELDER (1992). "Octrees for Faster Isosurface Generation", *ACM Transactions on Graphics*, 11(3), pp. 201-227.
- WILLIAMS, L. (1990). "Performance-Driven Facial Animation", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 235-242.
- WILLIAMS, P. L. (1992). "Visibility Ordering Meshed Polyhedra", *ACM Transactions on Graphics*, 11(2), pp. 103-126.
- WITKIN, A. AND W. WELCH (1990). "Fast Animation and Control of Nonrigid Structures", in proceedings of SIGGRAPH '90, *Computer Graphics*, 24(4), pp. 243-252.
- WITKIN, A. AND M. KASS (1991). "Reaction-Diffusion Textures", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 299-308.
- WOLFRAM, S. (1991). *Mathematica*, Addison-Wesley, Reading, MA.
- WOO, A., P. POULIN, AND A. FOURNIER (1990). "A Survey of Shadow Algorithms", *IEEE Computer Graphics and Applications*, 10(6), pp. 13-32.
- WRIGHT, W. E. (1990). "Parallelization of Bresenham's Line and Circle Algorithms", *IEEE Computer Graphics and Applications*, 10(5), pp. 60-67.
- WU, X. (1991). "An Efficient Antialiasing Technique", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 143-152.
- WYSZECKI, G. AND W. S. STILES (1982). *Color Science*, John Wiley & Sons, New York.
- WYVILL, G., B. WYVILL, AND C. MCPHEETERS (1987). "Solid Texturing of Soft Objects", *IEEE Computer Graphics and Applications*, 7(12), pp. 20-26.
- YAEGER, L., C. UPSON, AND R. MYERS (1986). "Combining Physical and Visual Simulation: Creation of the Planet Jupiter for the Film '2010'", in proceedings of SIGGRAPH '86, *Computer Graphics*, 20(4), pp. 85-94.
- YAGEL, R., D. COHEN, AND A. KAUFMAN (1992). "Discrete Ray Tracing", *IEEE Computer Graphics and Applications*, 12(5), pp. 19-28.
- YAMAGUCHI, K., T. L. KUNII, AND FUJIMURA (1984). "Octree-Related Data Structures and Algorithms", *IEEE Computer Graphics and Applications*, 4(1), pp. 53-59.
- YOUNG, D. A. (1990). *The X Window System - Programming and Applications with Xt, OSF/Motif Edition*, Prentice-Hall, Englewood Cliffs, NJ.
- ZELEZNICK, R. C., D. B. CONNER, M. M. WLOKA, ET AL. (1991). "An Object-Oriented Framework for the Integration of Interactive Animation Techniques", in proceedings of SIGGRAPH '91, *Computer Graphics*, 25(4), pp. 105-112.
- ZELTZER, D. (1982). "Motor Control Techniques for Figure Animation", *IEEE Computer Graphics and Applications*, 2(9), pp. 53-60.
- ZHANG, Y. AND R. E. WEBBER (1993). "Space Diffusion: An Improved Parallel Halftoning Technique Using Space-Filling Curves", in proceedings of SIGGRAPH '93, *Computer Graphics Proceedings*, pp. 305-312.

Subject Index

A

Absolute coordinates, 96
A-buffer algorithm, 475–76
Acoustic digitizer, 66–67
Active edge list, 122, 477
Active-matrix LCD, 47
Adaptive sampling, 538–40
Adaptive spatial subdivision:
 BSP tree, 362
 ray tracing, 536–38
Additive color model, 569, 572
Affine transformation, 208
Aliasing, 171
Alignment (text), 166
Ambient light, 497 (*see also* Illumination models)
Ambient reflection coefficient, 499
American National Standards Institute (ANSI), 78
Angle:
 direction (vector), 606
 incidence, 499
 phase, 595
 refraction, 509
 rotation, 186
 specular reflection, 501
Angstrom, 566
Animation, 584
 accelerations, 591–94
 action specifications, 587
 applications, 5–7, 17–18, 19–24
 cels, 588
 color-table, 586–87
 direct motion specification, 594–95
 double buffering, 55
 dynamics, 595–96
 frame-by-frame, 585
 functions, 586
 goal-directed, 595
 in-betweens, 585
 inverse dynamics, 596
 inverse kinematics, 596
 key frame, 585
 key-frame system, 587
 kinematics, 588, 595–96
 Kochanek-Bartels splines, 325–27
 languages, 587
 morphing, 18, 588–91
 motion specification, 594–96
 object definitions, 585
 parametrized system, 587
 physically based modeling, 393–95, 588, 596
 raster methods, 586–87
 real-time, 55, 585, 586
 scene description, 587
 scripting system, 588
 storyboard, 585
ANSI (American National Standards Institute), 78

Antialiasing:

 area boundaries, 176–78
 area sampling, 172, 174, 539
 filtering, 174–75
 lines, 172–76
 Nyquist sampling interval, 171
 Pitteway-Watkinson, 177–78
 pixel phasing, 172, 175
 pixel-weighting masks, 174, 555
 prefiltering, 172
 postfiltering, 172
 in ray tracing, 538–43
 stochastic sampling, 540–43
 supersampling, 172–74, 538–40
 surface boundaries, 538–43
 in texture mapping, 554–56
Application icon, 273
Applications (*see* Graphics applications)
Approximation spline, 316
Area clipping, 237–44
Area filling: (*see also* Fill: area)
 antialiasing, 176–78
 boundary-fill algorithm, 127–30
 bundled attributes, 169
 curved boundaries, 126–30
 flood-fill algorithm, 130
 functions, 131
 hatch, 158, 161
 nonzero winding number rule, 125–26
 odd-even rule, 125
 scan-line algorithm, 117–27
 soft fill, 162–63
 tint fill, 162
 unbundled attributes, 168
Area sampling, 172, 174, 539
Aspect ratio, 40
Aspect source flag, 168
Area-subdivision visibility algorithm, 482–85
Artificial reality (*see* Virtual reality)
Attenuation function, 506
Attribute, 77
 area-fill, 158–63, 169
 bundled, 168–69
 brush, 149–52
 character, 163–68, 169–70
 color, 154–57
 curve, 152–54
 grayscale, 157
 individual, 168
 inquiry functions, 170
 intensity level, 155 (*see also* Color: Intensity levels)
 line color, 149–50, 168–69
 line type, 144–46, 168–69
 line width, 146–49, 168–69
 marker, 167–68, 170
 parameter, 144
 pen, 149–52

 structure, 253–54
 system list, 144
 table, 306
 text, 163–68, 169–70
 unbundled, 168

Axis:

 reflection, 201
 rotation, 186, 413–20
 shear, 203
Axis vector (rotation), 414–15
Axis vectors (basis), 609
Axonometric projection, 440

B

Back-face detection, 471–72
Back plane (clipping), 447
Background (ambient) light, 497
Bar chart, 11–12, 137–38
Barn doors (light control), 504
Baseline (character), 164
Base vector, 609 (*see also* Basis)
Basis:
 coordinate vectors, 609
 normal, 609
 orthogonal, 609
 orthonormal, 609
Basis functions, 319 (*see also* Blending functions)
Basis matrix (spline), 320
Beam-penetration CRT, 42–43 (*see also* Cathode-ray tube)
Bernstein polynomials, 327
Beta parameter, 345
Beta-spline, 345–47
Bevel join, 149
Bézier:
 blending functions, 327–28
 B-spline conversions, 350
 closed curve, 330
 cubic curve, 331–33
 curves, 327–33
 design techniques, 330–31
 matrix, 333
 properties, 329–30
 surfaces, 333–34
Bias parameter (spline), 325, 346
Binary space-partitioning tree, 362 (*see also* BSP tree)
Binding (language), 78
Bisection root finding, 622
BitBit (bit-block transfer), 210
Bit map, 40 (*see also* Frame buffer)
Bitmap font, 132–33
Blending functions, 319
 Bézier, 327–28
 B-spline, 335

- Blending functions (*cont.*)
 cardinal, 325
 Hermite, 323
 Block transfer, 210
 Blobby object, 314
 Body:
 character, 164
 nonrigid, 393
 rigid, 185, 196
 Boolean operations:
 area-fill, 161
 raster transformations, 210
 Boundary conditions (*spline*), 317, 318–19
 Boundary-fill algorithms:
 8-connected region, 127
 4-connected region, 127–30
 Boundary representation, 305
 Bounding:
 box, 161
 rectangle, 94, 161
 volume, 535
 Box covering, 366
 Box filter, 174–75
 Box dimension, 366
 B-rep (boundary representation), 305
 Bresenham's algorithm:
 circle, 98
 line, 88–92
 Brightness (light), 566
 Brownian motion, 372
 Brush and pen attributes, 149–52
 BSP:
 ray tracing, 536
 tree, 362
 visibility algorithm, 481–82
 B-spline:
 Bézier conversions, 350
 blending functions, 335
 Cox-deBoor recursion formulas, 335
 cubic, 339–41
 curves, 334–44
 knot vector, 335
 local control, 335, 336
 matrix, 341
 nonuniform, 336, 343–44
 nonuniform rational (NURB), 347
 open, 336, 341–44
 periodic, 337–41
 properties, 335–36
 quadratic, 338–39, 342–44
 rational, 347
 surfaces, 344–45
 tension parameter, 341
 uniform, 336–44
 Buffer, 40 (*see also* Frame buffer)
 Bump function, 558 (*see also* Frame mapping)
 Bump mapping, 558–59
 Bundled attributes, 168–69
 Bundle table, 168
 Business visualization, 25, 395 (*see also* Data visualization)
 Butt line cap, 147
 Button box, 61, 279
- C
- Cabinet projection, 443
 CAD, 4–11
 Calligraphic (vector) display, 41
 Camera viewing, 433–36
 Camera lens effects, 541
 Capline (character), 164
 Cardinal spline, 323–25
- Cardioid, 139–40
 Cartesian coordinates, 600–601, 602
 Cathode-ray tube, 36–40 (*see also* Video monitors)
 aspect ratio, 40
 beam intensity, 38
 beam penetration, 42–43
 color, 42–45
 components, 37–38
 electron gun, 37
 delta-delta shadow mask, 43
 electrostatic beam deflection, 38–39
 focusing, 38
 high-definition, 40
 inline shadow-mask, 44
 magnetic beam deflection, 37, 38
 persistence, 39
 phosphor, 37–39
 refresh rate, 40–41
 resolution, 39–40
 RGB, 45
 shadow-mask, 43–44
 Catmull-Rom spline, 325
 Cavalier projection, 443
 Cell array, 131
 Cell encoding, 56
 Cels, 588
 Center of projection, 438
 Central structure store (CSS), 251
 CGI (Computer Graphics Interface), 79
 CGM (Computer Graphics Metafile), 79
 Character:
 attributes, 163–68
 baseline, 164
 body, 164
 bottom line, 164
 capline, 164
 color, 164
 descender, 164
 fonts, 132, 163
 functions, 163–168
 generation, 132–34
 grid, 55–56, 132–33
 height, 164
 italic, 163
 kern, 164
 outline fonts, 55–56, 132–133
 text precision, 166–167
 topline, 164
 typeface, 132–33, 163
 up vector, 165
 width, 164–65
 Characteristic polygon, 316
 Chart:
 bar, 11–12, 137–38
 pie, 11–12, 138–40
 time, 11, 13
 line, 11, 136–37
 Choice input device, 275, 279
 Chromaticity, 567
 diagram, 569–71
 values, 569
 CIE (International Commission on Illumination), 568
 Circle equation
 Cartesian, 97
 nonparametric, 97, 619
 parametric, 97, 619
 polar, 97
 Circle-generating algorithms, 97–102
 Bresenham, 98
 midpoint, 98–102
 midpoint function, 96–99
 midpoint decision parameters, 99
- Circle symmetry, 97–98
 Clipping:
 areas, 237–44
 Cohen-Sutherland line algorithm, 226–30, 232
 curves, 244
 Cyrus-Beck line algorithm, 230
 exterior, 245, 246
 hardware implementation, 463–64
 in homogeneous coordinates, 461–63
 Liang-Barsky line algorithm, 230–32
 Liang-Barsky polygon algorithm, 243
 Nichol-Lee-Nichol line algorithm, 233–35
 nonrectangular window, 235
 in normalized coordinates, 224, 458–61
 parallel methods, 239
 parametric, 230–32
 planes, 447–50, 456–63
 points, 225
 polygons, 237–43
 region codes, 227, 460
 straight line segments, 225–37, 456, 460–61
 Sutherland-Hodgman polygon algorithm, 238–42
 text, 244, 245
 three-dimensional, 456–63
 two-dimensional, 224–45
 view volumes, 447–50, 456–63
 Weiler-Atherton polygon algorithm, 242–43
 window, 224
 in world coordinates, 224
 CMY color model, 574–75
 Codes (ray tracing), 541
 Coefficient:
 ambient-reflection, 499
 diffuse-reflection, 498
 matrix, 620
 specular-reflection, 501–2
 transparency, 510
 Cohen-Sutherland line-clipping algorithm, 226–30, 232
 Coherence, 119–24, 471
 Color:
 chromaticity, 567
 chromaticity diagram, 569–71
 chromaticity values, 569
 coding, 25, 396
 complementary, 568, 570
 cube, 572–73 (*see also* Color models)
 dominant frequency, 566
 dominant wavelength, 566, 569–70
 fill, 154–63
 gamut, 568, 570–71
 hue, 566, 575, 579
 illuminant C, 570
 in illumination models, 507–8
 intuitive concepts, 571–72
 lightness (HLS parameter), 579
 line, 149–52, 166–69
 lookup table, 155–56
 marker, 168, 170
 matching functions, 568
 model, 565, 568
 monitor, 42–45 (*see also* Video monitor)
 nonspectral, 571
 perception, 566–67
 primaries, 568
 pure, 567, 569
 purity, 567
 purple line, 570
 RGB, 155–57
 saturation, 567, 575, 579
 selection considerations, 580–81
 shades, 571, 577

- spectrum (electromagnetic), 565
 - standard CIE primaries, 568–69
 - table, 155–56
 - text, 164, 169
 - tints, 571, 577
 - tones, 571, 577
 - tristimulus vision theory, 572
 - value (HSV parameter), 575
 - Color model, 565, 568
 - additive, 569, 572
 - CMY, 574–75
 - HLS, 579–80
 - HSB (*see* HSV model)
 - HSV, 575–77
 - HSV-RGB conversion, 578–79
 - RGB, 572–73
 - RGB-CMY conversion, 575
 - XYZ, 569
 - YIQ, 574
 - Color-table animation, 586–87
 - Column vector, 611
 - Command icon, 273
 - Commission Internationale de l'Éclairage (CIE), 568
 - Complementary colors, 568, 570
 - Complex number:
 - absolute value, 616
 - conjugate, 616
 - Euler's formula, 617
 - imaginary part, 615
 - length (modulus), 616
 - modulus, 616
 - ordered-pair representation, 615
 - polar representation, 616–17
 - pure imaginary, 615
 - real part, 615
 - roots, 617
 - Complex plane, 615
 - Composite monitor, 44–45
 - Composition (matrix), 191
 - Computed tomography (CT), 32
 - Computer-aided design (CAD), 4–11
 - Computer-aided surgery, 33
 - Computer art, 13–18
 - Computer Graphics Interface (CGI), 79
 - Computer Graphics Metafile (CGM), 79
 - Concatenation (matrix), 191, 612–13
 - Concave polygon splitting, 235–37
 - Cone filter, 174, 175
 - Cone receptors, 572
 - Cone tracing, 540 (*see also* Ray tracing)
 - Conic curves, 110–12, 348–49
 - Conjugate (complex), 616
 - Constant-intensity shading, 522–23
 - Constraints, 288–89
 - Constructive solid geometry (CSG), 356
 - mass calculations, 359
 - octree methods, 361–62
 - ray-casting methods, 357–59
 - volume calculations, 358–59
 - Continuity conditions (spline):
 - geometric, 318–19
 - parametric, 317–18
 - Continuity parameter, 325
 - Continuous-tone images, 515, 516 (*see also* Halftone)
 - Contour (intensity border), 515, 518
 - Contour plots:
 - applications, 11, 12, 25
 - surface lines, 489–90
 - three-dimensional (isosurfaces), 398
 - two-dimensional (isolines), 396–97
 - Contraction (tensor), 402
 - Control graph, 316
 - Control icon, 273
 - Control operations, 78
 - Control point (spline), 316
 - Control polygon, 316
 - Control surface (terrain), 376–77
 - Convex hull, 316
 - Coordinate-axis rotations, 409–13
 - Coordinate-axis vectors (basis), 609
 - Coordinate extents, 94
 - Coordinate point, 602, 605, 612
 - Coordinates:
 - absolute, 96
 - current position, 96
 - homogeneous, 189
 - relative, 96
 - screen, 114
 - Coordinate system:
 - Cartesian, 600–601, 602
 - curvilinear, 602
 - cylindrical, 603–4
 - device, 76
 - left-handed, 435, 602
 - local, 76, 265
 - master, 76, 265
 - modeling, 76, 265, 426–29
 - normalized device, 76
 - normalized projection, 458
 - orthogonal, 603
 - polar, 601–2, 604
 - right-handed, 602
 - screen, 54, 76, 114
 - spherical, 604
 - three-dimensional, 602–4
 - transformation of, 205–7, 219–20, 426–29
 - two-dimensional, 600–602
 - unit, 435–38
 - viewing, 218, 219–20, 432–36
 - world, 76
 - Copy function, 210
 - Cox-deBoor recursion formulas, 335
 - Cramer's rule, 621
 - Cross hatch fill, 158, 159
 - Cross product (vector), 608–9
 - CRT, 36–40 (*see also* Cathode-ray tube)
 - CSG, 356 (*see also* Constructive solid geometry)
 - CT (Computed Tomography) scan, 32
 - Cubic spline, 112, 319
 - beta, 346–47
 - Bézier, 331–33
 - B-spline, 339–41
 - interpolation, 320–27
 - Current event record, 286
 - Current position, 96
 - Curve
 - attributes, 152–54
 - beta spline, 345–46
 - Bézier spline, 327
 - B-spline, 334–35
 - cardinal spline, 323–24
 - cardioid, 139–40
 - Catmull-Rom spline, 325
 - circle, 97, 111
 - conic section, 110–12, 348–49
 - ellipse, 102–3
 - fractal, 362–68 (*see also* Fractal curves)
 - generalized function, 113
 - Hermite spline, 322
 - hyperbola, 111, 112
 - Koch (fractal), 367
 - Kochanek-Bartels spline, 325
 - Limaçon, 139–40
 - natural spline, 321
 - Overhauser spline, 325
 - parabola, 112
 - parallel algorithms, 112–13
 - parametric representations, 112, 619
 - piecewise construction, 315–16
 - polynomial, 112
 - spiral, 139–40
 - spline, 112, 315–20 (*see also* Spline curve)
 - superquadric, 312–13
 - symmetry considerations, 97–98, 103, 112
 - Curved surface,
 - ellipsoid, 311
 - parametric representations, 619–20
 - quadric, 310–12
 - rendering (*see* Surface rendering)
 - sphere, 311
 - spline, 316 (*see also* Spline surface)
 - superquadric, 312–13
 - torus, 311–12
 - visibility, 487–90, (*see also* Visible-surface detection)
 - Curvilinear coordinates, 602
 - Cutaway views, 300, 302
 - Cylindrical coordinates, 603–4
 - Cyrus-Beck line-clipping algorithm, 230
- ## D
- Damping constant, 595
 - Dashed line, 144–46
 - Data glove, 64, 65, 292–93 (*see also* Virtual reality)
 - Data tablet, 64 (*see also* Digitizer)
 - Data visualization:
 - applications, 25–31
 - contour plots, 396–97
 - field lines, 400
 - glyphs, 403
 - isolines, 396–97
 - isosurfaces, 398
 - multivariate fields, 402–3
 - pseudo-color methods, 396
 - scalar fields, 395–99
 - streamlines, 400
 - tensor fields, 401–2
 - vector fields, 400–401
 - volume rendering, 399
 - DDA line algorithm, 87–88
 - Deflection coils, 37, 38 (*see also* Cathode-ray tube)
 - Delta-delta shadow-mask CRT, 43
 - Density function (blobby object), 314
 - Depth-buffer algorithm, 472–75
 - Depth cueing, 299–300
 - Depth-sorting algorithm, 478–81
 - Descender (character), 164
 - Detectability filter, 284–85
 - Determinant, 613–14
 - Device codes, 281–82
 - Device coordinates, 76
 - Differential scaling, 188
 - Diffuse reflection, 497–500
 - Diffuse refraction, 509
 - Digitizer, 64
 - accuracy, 65, 66
 - acoustic, 66–67
 - applications, 13–15
 - electromagnetic, 65–66
 - locator device, 277
 - resolution, 65, 66
 - sonic, 66
 - stroke device, 277
 - three-dimensional, 67
 - valuator device, 278

- Dimension
 - Euclidean, 363
 - fractal, 363, 364-66
 - fractional, 363
 - Directed line segment (vector), 605
 - Direction angles, 606
 - Direction cosines, 606
 - Direct-view storage tube (DVST), 45
 - Display
 - coprocessor, 55
 - controller, 53
 - devices, 36-52 (*see also* Video monitors; Display processors)
 - file, 42, 56
 - list, 42-84
 - processor, 55-56
 - processing unit, 56
 - program, 42, 56
 - Distance
 - point to line, 279-80
 - ray-tracing path, 531-34
 - Distributed light source, 496
 - Distributed ray tracing, 540-43
 - Distribution ray tracing, 541
 - Dithering, 519
 - dot diffusion method, 522
 - error diffusion method, 520-22
 - matrix, 520
 - noise, 519-20
 - ordered-dither method, 520
 - random, 520-21
 - Dominant frequency, 566
 - Dominant wavelength, 566, 569-70
 - Dot product, 607-8
 - Dot-matrix printer, 72
 - Dot-diffusion algorithm, 522
 - Double buffering, 55
 - Dragging, 291
 - Drawing methods, 291-92
 - DVST, 45
 - Dynamics, 595-96 (*see also* Animation)
- I**
- Edge list (table), 121-22, 306-7, 476-77
 - Edge vector, 126
 - 8-connected region, 127
 - Elastic material (nonrigid object), 393
 - Electromagnetic spectrum, 565
 - Electron beam (*see also* Cathode-ray tube)
 - convergence, 38
 - electrostatic deflection, 38, 39
 - focusing, 38
 - intensity, 38
 - magnetic deflection, 37, 38
 - spot size, 39-40
 - Electron gun, 37 (*see also* Cathode-ray tube)
 - Electrostatic printer, 73
 - Electrothermal printer, 73
 - Element (structure), 255
 - Element pointer, 255
 - Elevation view, 440
 - Ellipse
 - Cartesian equation, 102-3
 - focus point, 102
 - midpoint algorithm, 103-10
 - parametric representation, 103
 - properties, 103
 - symmetry, 103
 - Ellipsoid, 311-12
 - Emissive displays (emitter), 45
 - Energy cloth-modeling function, 394
 - Energy distribution (light source), 567
 - Energy propagation (radiosity), 544
 - Environment array, 552
 - Environment mapping, 552
 - Error-diffusion algorithm, 520-22
 - Euler's formula, 617 (*see also* Complex numbers)
 - Even-odd polygon-filling rule, 125
 - Event, 285
 - input mode, 281, 285-87
 - queue, 285
 - Explicit representation, 618
 - Exploded view, 300, 301
 - Exterior clipping, 245, 246
- F**
- False-position root finding, 622
 - Far plane (clipping), 447
 - Fast Phong shading, 526-27
 - Feedback, 275-76
 - Field lines, 400
 - Fill
 - algorithms (*see* Area filling)
 - area, 77, 117
 - attributes, 158-63 (*see also* Area filling)
 - color, 158
 - hatch, 159, 161
 - patterns, 159-62
 - soft, 162-63
 - styles, 158
 - tint, 162
 - Filter
 - box, 174, 175
 - cone, 174, 175
 - function, 174
 - Gaussian, 174-75
 - structure, 253-54, 284-85
 - workstation pick detectability, 284-85
 - Fixed position (scaling), 188, 193, 421
 - Flaps (light control), 504
 - Flat-plane display, 45
 - emissive, 45
 - gas-discharge, 45
 - light-emitting diode (LED), 46-47
 - liquid-crystal (LCD), 47-48
 - nonemissive, 45
 - passive-matrix, 47
 - plasma, 45-46
 - thin-film electroluminescent, 46
 - Flat shading, 522
 - Flight simulators, 21-24
 - Flood-fill algorithm, 130
 - Flood gun, 45
 - Focus point (ellipse), 102
 - Font, 132 (*see also* Typeface)
 - bitmap, 132-33
 - cache, 133
 - outline, 132, 133
 - proportionally spaced, 164
 - Force constant, 393
 - Form factors (radiosity), 546
 - Forward differences, 351-53
 - 4-connected region, 127-29
 - Fractal
 - affine constructions, 372-78
 - box-covering methods, 366
 - Brownian motion, 372-75
 - characteristics, 362-63
 - classification, 364
 - dimension, 363, 364-67
 - generation procedures, 363-64
 - generator, 367
 - geometric constructions, 367-71
 - geometry, 362
 - initiator, 367
 - invariant set, 364
 - random midpoint-displacement methods, 373-78
 - self-affine, 364
 - self-inverse, 364
 - self-inversion methods, 385-87
 - self-similar, 364
 - self-similarity, 362
 - self-squaring, 364
 - self-squaring methods, 378-85
 - similarity dimension, 365
 - subdivision methods, 373-78
 - topological covering methods, 365-66
 - Fractal curve
 - Brownian motion, 372
 - dimension, 366
 - fractional Brownian motion, 372-75
 - geometric constructions, 367-68
 - invariant, 379-87
 - inversion construction methods, 385-87
 - Julia set, 379
 - Koch, 367
 - Mandelbrot set boundary, 381-84
 - midpoint displacement, 373-75
 - Peano, 366
 - self-affine, 372-75
 - self-inverse, 385-87
 - self-similar, 367-71
 - self-squaring, 379-84
 - snowflake, 367-68
 - Fractal solid, 366
 - Fractal surface
 - Brownian, 372-78
 - dimension, 366
 - four-dimensional, 384-85
 - geometric constructions, 369-71
 - midpoint displacement, 373-78
 - self-similar, 369-71
 - self-squaring, 384-85
 - surface rendering, 376
 - terrain, 372-78
 - Fractional Brownian motion, 372-78
 - Fractional dimension, 366
 - Frame (animation), 585
 - Frame buffer, 40, 84
 - bit-block transfers, 210
 - copy function, 210
 - loading intensity values, 94-95
 - lookup table, 155-56, 513
 - raster transformations, 210-11
 - read function, 210
 - resolution, 40
 - write function, 210
 - Frame mapping, 559-60
 - Fresnel reflection laws, 501
 - Frequency spectrum (electromagnetic), 565
 - Front plane (clipping), 447
 - Full-color system, 45
 - Frustum, 447
 - Functions, 77-78 (*see also* Function Index)
- G**
- Gamma correction, 513-15
 - Gamut (color), 568, 570-71
 - Gas-discharge displays, 45
 - Gaussian bump, 314
 - Gaussian density function, 314-15
 - Gaussian elimination, 621

- Gaussian filter, 174–75
 Gauss-Seidel method, 621
 Generator (fractal), 367
 Geometric continuity (spline), 318–19
 Geometric models, 261
 Geometric-object properties, 114–17
 Geometric production rules, 387–89
 Geometric table, 306–7
 Geometric transformations, 77, 184, 408
 GKS (Graphical Kernel System), 78
 GL (Graphics Library), 76, 251, 264, 327, 432, 434, 435, 439, 458
 Global lighting effects, 497, 527, 544
 Glyph, 403
 Goal-directed motion, 595
 Gouraud shading model, 523–25
 Graftal, 389
 Graphical user interface:
 applications, 34
 backup and error handling, 274–75
 components, 272–76
 feedback, 275–76
 help facilities, 274
 icons, 34, 273
 interactive techniques, 288–93
 menus, 34, 273
 model, 272
 user dialogue, 272–73
 user's model, 272
 windows, 34, 273
 Graphics applications,
 advertising, 8, 17–18
 agriculture, 27, 28
 animations, 5–7, 17–18, 19–24
 architecture, 10, 11
 art, 13–18
 astronomy, 25
 business, 11–13, 17–18, 25, 31
 CAD, 4–11
 cartography, 11
 education, 21–24
 engineering, 4–9
 entertainment, 18–21
 facility planning, 9, 10
 flight simulators, 21–24
 geology, 32
 graphs and charts, 11–13
 image processing, 32–33
 manufacturing, 8–9
 mathematics, 14–17, 25–27
 medicine, 32–33
 modeling and simulations, 4–8, 21–25, 25–31
 physical sciences, 25–31, 32
 publishing, 17
 scientific visualization, 25–31
 simulations, 5–10, 21–31
 simulators, 21–25
 training, 21–24
 user interfaces, 34
 virtual reality, 5–8, 466–67
 Graphics controller, 55, 56
 Graphics functions, 77–78 (*see also* Function Index)
 Graphics monitors, 36–52 (*see also* Video monitors)
 Graphics software packages:
 basic functions, 77–78
 GKS, 78
 GL, 76, 251, 264, 327, 432, 434, 435, 439, 458
 PHIGS, 78
 PHIGS+, 78
 standards, 78–79
 three-dimensional, 302–3
 Graph plotting, 136–39 (*see also* Charts)
 Graphics tablet, 13–15, 64–67 (*see also* Digitizer)
- Gravitational acceleration, 111
 Gravity field, 290
 Grayscale, 157
 Grids:
 character, 55–56, 132–33
 in interactive constructions, 289–90
- ## H
- Halftone, 516
 approximations, 516–19
 color methods, 519
 dithering, 519–22
 patterns, 516
 Halfway vector, 503
 Hard-copy devices, 72–75
 Hatch fill, 159, 161
 Hausdorff-Besicovitch dimension, 366
 Head-mounted display, 6–7 (*see also* Virtual reality)
 Hemicycle (radiosity), 548–49
 Hermite spline, 322–23
 Hexcone (HSV), 576
 Hidden-line elimination, 490
 Hidden-surface elimination, 470 (*see also* Visible-surface detection)
 Hierarchical modeling, 266–68
 High-definition video monitor, 40
 Highlighting:
 as depth-cueing technique, 299–300
 primitives, 287
 specular reflections, 497, 500–504
 structures, 253–54, 287
 HLS color model, 579–80
 Homogeneous coordinates, 189
 Hooke's law, 393
 Horizontal retrace, 41
 Horner's polynomial factoring method, 351
 HSB color model (*see* HSV model)
 HSV color model, 575–77
 Hue, 566, 575, 579
 Hyperbola, 111, 112
- ## I
- Icon, 34, 273
 Ideal reflector, 498
 Illuminant C, 570
 Illumination model, 495
 ambient light, 497
 attenuation function, 506
 basic components, 497–511
 color considerations, 507–8
 combined diffuse-specular, 504
 diffuse reflection, 497–500
 flaps, 504
 ideal reflector, 498
 intensity attenuation, 505–6
 light sources, 496–97
 multiple light sources, 504
 opacity factor, 510
 Phong, 501–4
 refraction, 508–10
 shadows, 511
 Snell's law, 509
 specular reflection, 500–504
 spotlights, 504
 transmission vector, 510
 transparency, 508–11
 Warn, 504
 Image-order scanning, 554
 Image processing, 32–33
 Image scanners, 67, 68
 Image-space methods (visibility detection), 470
 Imaginary number, 615
 Impact printer, 72
 Implicit representation, 618
 In-between, 585
 Index of refraction, 509
 Initiator (fractal), 367
 Ink-jet printer, 72–73
 Inner product (vector), 607
 In-line shadow-mask CRT, 43
 Input devices:
 button box, 61, 279
 choice, 276, 279
 data glove, 64, 65, 292–93
 dials, 61, 62
 digitizer, 64–67, 277–80
 graphics tablet, 64
 initializing, 287–88
 joystick, 63–64, 277–80
 keyboard, 61, 277–80
 light pen, 70, 71
 locator, 276, 277
 logical classification, 276
 mouse, 61–62, 277–80
 pick, 276, 279–80
 scanner, 67, 68
 spaceball, 63
 string, 276, 277
 stroke, 276, 277
 switches, 61, 62
 three-dimensional sonic digitizers, 67
 touch panel, 68–70
 trackball, 63
 valuator, 276, 277–78
 voice systems, 70–71
 Input functions, 78, 281–87
 Input modes:
 concurrent use, 287
 event, 281, 285–87
 request, 281, 282–85
 sample, 281, 285
 Input priority, 283
 Inquiry functions, 170
 Inside-outside test:
 polygon odd-even rule, 125
 polygon nonzero winding number rule, 125–26
 spatial plane surface, 308
 Inside polygon face, 308
 Instance, 261 (*see also* Modeling)
 Integral equation solving:
 rectangle approximations, 622
 Simpson's rule, 623
 trapezoid rule, 623
 Monte Carlo methods, 623–24
 Intensity:
 attenuation, 505–6
 depth cueing, 299–300
 interpolation shading (Gouraud), 523
 modeling, 495–97 (*see also* Illumination models)
 radiosity model, 544–51
 Intensity level:
 adjusting (*see* Antialiasing)
 assigning, 512–13
 color lookup tables, 155–56
 contours (borders), 515, 518
 frame-buffer storage, 240
 gamma correction, 513–15
 ratio, 512
 RGB, 507–8
 video lookup table, 155, 513

Interactive picture construction techniques, 288-92
 Interlacing scan lines, 41
 International Commission on Illumination (CIE), 568
 Interpolation spline, 316
 Inverse geometric transformations, 190, 409, 413, 421-22
 Inverse dynamics, 596
 Inverse kinematics, 596
 Inverse matrix, 614
 Inverse quaternion, 618
 Inverse scanning, 554
 ISO (International Standards Organization), 78
 Isolines, 396-97
 Isometric joystick, 64
 isometric projection, 440-41
 Isosurfaces, 398

J

Jaggies, 85 (see also Antialiasing; Antialiasing)
 jittering, 541
 Joystick:
 as locator device, 277
 movable, 63-64
 as pick device, 279
 pressure sensitive (isometric), 63, 64
 as stroke device, 277
 asvaluator device, 278
 Julia set, 379

K

Kern, 164
 Keyboard, 61
 as choice device, 279
 as locator device, 277
 as pick device, 280
 as string device, 277
 asvaluator device, 278
 Key frame, 585
 Key-frame system, 587
 Kinematics, 588, 595-96 (see also Animation)
 Knot vector, 335
 Kochanek-Bartels spline, 325-27
 Koch curve, 367

L

Lambertian reflector, 498
 Lambert's cosine law, 498
 Language binding, 78
 Laser printer, 72
 LCD (liquid-crystal display), 47-48
 Least-squares data fitting, 625
 LED (light-emitting diode), 46-47
 Left-handed coordinates, 435, 602
 Legible typeface, 132
 Length
 complex number, 616
 vector, 605
 L-grammar, 389
 Liang-Barsky clipping
 polygons, 243
 two-dimensional lines, 230-32
 Light:
 ambient, 497
 angle of incidence, 499
 chromaticity, 567

chromaticity diagram, 569-71
 diffuse reflection, 497-500
 diffuse refraction, 509
 frequency band, 565
 hue, 566
 ideal reflector, 498
 index of refraction, 509
 illuminant C, 570
 illumination model, 495 (see also Illumination models)
 intensity-level assignment, 512-13
 Lambert's cosine law, 498
 Phong specular model, 501-4
 properties, 565-68
 purity, 567
 reflection coefficients, 497-502
 refraction angle, 509
 saturation, 567, 575, 579
 spectrum, 565
 specular reflection, 500-504
 specular refraction, 509
 speed, 566
 transparency coefficient, 510
 wavelength, 566
 white, 567, 570

Light buffer (ray tracing), 537
 Light-emitting diode (LED), 46-47
 Lighting model, 495 (see also Illumination model)
 Lightness (HLS parameter), 579
 Light pen, 70, 71
 Light source:
 brightness, 566
 distributed, 496
 dominant frequency, 566
 dominant wavelength, 566
 energy distribution, 567
 frequency distribution, 565
 luminance, 566
 multiple, 504
 point, 496

Limaçon, 139, 140

Line:
 bundled attributes, 166-69
 chart, 11, 136-37
 clipping, 225-37 (see also Line clipping)
 color, 149-50
 contour, 11, 12, 25, 396-97
 dashed, 144-46
 function, 95-96
 parametric representation, 230, 444
 pen and brush options, 149, 154
 sampling, 87, 88-89
 slope-intercept equation, 86
 type, 144-46
 width, 146-49

Linear congruential generator, 624

Linear equation solving:

 Cramer's rule, 621
 Gaussian elimination, 621
 Gauss-Seidel, 621

Line caps, 147

Line clipping:

 Cohen-Sutherland, 226-30, 232
 Cyrus-Beck, 230
 Liang-Barsky, 230-32
 Nichol-Lee-Nichol, 233-35
 nonrectangular clip window, 235
 parallel methods, 239
 parametric, 230-32
 three-dimensional, 456

Line-drawing algorithms, 86-95
 antialiasing, 172-76
 Bresenham, 88-92

DDA, 87-88
 frame-buffer loading, 94-95
 parallel, 92-94
 Liquid-crystal display (LCD), 47-48
 Local coordinates, 76, 265
 Local control (spline), 332, 335, 336
 Local transformation matrix, 266
 Locator input device, 276, 277
 Logical input device, 276
 Look-at point, 434
 Lookup table, 155-56, 513
 Luminance, 544, 566

M

Mach band, 525
 Mandelbrot set, 381-84
 Marching cubes algorithm (see Isosurfaces)
 Marker, 133-34
 Marker attributes 167-68, 170
 Mask, 146, 517 (see also Pixel mask)
 Mass calculations (CSG), 359
 Master coordinates, 76, 265
 Matrix, 611
 addition, 612
 basis (spline), 320
 Bézier, 333
 B-spline, 341
 cardinal, 325
 coefficient, 620
 column, 611
 concatenation, 191, 612-13
 determinant, 613-14
 dither, 520
 Hermite, 323
 identity, 614
 inverse, 614
 multiplication, 612-13
 nonsingular, 614
 reflection, 201-3, 422
 row, 611
 rotation, 186, 190, 193, 410-12, 418-20
 scalar multiplication, 612
 scaling, 187, 193, 192, 421
 shear, 203-4, 423
 singular, 614
 spline characterization, 320
 square, 611
 translation, 185, 190, 408
 transpose, 613
 Medical applications, 32-33
 Menu, 34, 273
 Mesh (polygon), 306, 309-10
 Metaball model, 315
 Metafile, 79
 Metric tensor, 610-11
 Midpoint circle algorithm, 98-102
 Midpoint-displacement fractal generation, 373-78
 Midpoint ellipse algorithm, 103-10
 Miter join, 148-49
 Mode (input device), 281
 Model, 261
 Modeling, 261 (see also Graphics applications;
 Object representations; Illumination models)
 basic concepts, 260-64
 coordinates, 76, 265, 426-29
 display procedures, 261, 266
 geometric, 261
 hierarchical, 262-63
 instance, 261
 local coordinates, 265
 master coordinates, 265

modules, 262
 packages, 263–64
 physically based, 393–95, 588, 596
 representations, 261–62
 structure hierarchies, 266–68
 symbol, 261
 symbol hierarchies, 262–63
 transformations, 77, 265–68, 426–29

Modules, 262

Modulus (complex), 616

Monte Carlo methods, 623–24

Monitor, 36–52 (*see also* Video monitor)

Monitor response curve, 513

Morphing, 18, 588–91

Motion blur, 541, 542–43

Motion specification, 594–96

Mouse, 61–63
 as choice device, 279
 as locator device, 277
 as pick device, 279
 as stroke device, 277

Multivariate data visualization, 402–3

N

National Television System Committee (NTSC), 514, 573, 574

Natural spline, 321

Near plane (clipping), 447

Newton-Raphson root-finding, 621–22

Newton's second law of motion, 596

Nicholl-Lee-Nicholl line-clipping, 233–35

Noise (dither), 519–20

Nonemissive displays, 45

Nonemitter, 45

Nonlinear-equation solving:
 bisection, 622
 false-position, 622
 Newton-Raphson, 621–22

Nonparametric representations, 618–19

Nonrigid object, 393

Nonsingular matrix, 614

Nonspectral color, 571

Nonuniform B-splines, 336, 343–344

Nonuniform (differential) scaling, 188

Nonuniform rational B-spline (NURB), 347

Nonzero winding number rule, 125–26

Normal basis, 609

Normalized device coordinates, 76

Normalized projection coordinates, 458

Normalized view volumes, 458 (*see also* Clipping)

Normal vector:
 average (polygon mesh), 523
 curved surface, 558
 interpolation (Phong shading), 525
 plane surface, 308–9
 view-plane, 434–36

NTSC (National Television System Committee), 514, 573, 574

Numerical methods:
 bisection method, 622
 Cramer's rule, 621
 false-position method, 622
 Gaussian elimination, 621
 Gauss-Seidel method, 621
 integral evaluations, 622–24
 least-squares data fitting, 625
 linear equations, 620–21
 Monte Carlo methods, 623–24
 Newton-Raphson method, 621–22
 nonlinear equations, 621–22
 root finding, 621–22

Simpson's rule, 623
 trapezoid rule, 623

NURB (Nonuniform rational B-spline), 347

Nyquist sampling interval, 171

O

Object:
 nonrigid (flexible), 393
 as picture component, 77, 251
 rigid, 185, 196–97

Object geometry, 114–17

Object representation
 beta splines, 345–47
 Bézier splines, 327–34
 boundary (B-rep), 305
 B-splines, 334–45
 BSP trees, 362
 blobby surfaces, 314–15
 CSG methods, 356–59
 cubic spline interpolation, 320–27
 data visualization, 395–403
 explicit, 618
 fractal curves and surfaces, 362–87
 implicit, 618
 nonparametric, 618–19
 octrees, 359–62
 parametric, 619–20
 particle systems, 390–92
 physically based modeling, 393–95
 polygon, 305–10
 quadric surfaces, 310–12
 rational splines, 347–49
 shape grammars, 387–89
 space-partitioning methods, 305
 superquadrics, 312–14
 sweep constructions, 355–56

Object-space methods (visibility detection), 470

Oblique projection, 439, 441–43, 447–50, 452–53

Octree, 359
 CSG operations, 361–62
 generation, 360–61
 visibility detection, 362, 485–87
 volume element, 360
 voxel, 360

Odd-even polygon-filling rule, 125

One-point perspective projection, 446

Opacity factor, 510

Order (spline curve continuity), 317–19

Ordered dither, 520

Orthogonal basis, 609

Orthogonal coordinates, 603

Orthographic projections, 439, 441, 447–48

Orthonormal basis, 609

Outline font, 132, 133

Output primitives, 77
 cell array, 131, 132
 circle, 97–102
 character, 131–34
 conic section, 110–12
 ellipse, 102–10
 fill area, 117–30
 marker, 133–34
 point, 84–86
 polynomial, 112
 spline, 112
 straight line segment, 85, 86–94
 text, 131–33

Outside polygon face, 308

Overhauser spline, 325

P

Paintbrush programs, 13–16, 291–92

Painter's algorithm (depth sorting), 478

Panning, 219

Parabola, 112

Parallel algorithms:
 area-filling, 120–21
 curve-drawing, 112–13
 line-drawing, 92–94

Parallel projection, 298–99, 438

axonometric, 440
 cabinet, 443
 cavalier, 443
 elevation view, 440
 isometric, 440–41
 oblique, 439, 441–43, 447–50, 452–53
 orthographic, 439, 441, 447–48
 plan view, 440
 principal axes, 440
 shear transformation, 442, 453
 view volume, 447–50

Parametric continuity (spline), 317–18

Parametric representations, 619–20
 circle, 97, 619
 curve, 111–12, 619
 ellipse, 103
 ellipsoid, 311–12
 sphere, 311, 620
 spline, 112, 315–16
 straight line, 230, 444
 surface, 619–20
 torus, 311–12

Parametrized system, 587

Parity (odd-even) rule, 125

Particle systems, 390–92

Path (text), 166

Passive-matrix LCD, 47

Pattern fill, 159–61
 index, 159
 reference point, 159–60
 representation, 159
 size, 159
 tiling, 160

Pattern mapping, 554

Pattern recognition, 277

Peano curve, 366

Pel, 40

Pen and brush attributes, 149, 150, 154

Penumbra shadow, 542

Perfect reflector, 498

Persistence, 39

Perspective projection, 299, 438
 frustum, 447
 one-point, 446
 principal vanishing point, 446
 reference point, 438
 shear transformation, 454–56
 three-point, 446
 two-point, 446
 vanishing point, 446
 view volume, 447–49

PET (Positron emission tomography) 32–33

Phase angle, 595

PHIGS, 78 (*see also* Function Index)
 attributes, 145, 146, 149, 156, 158–59, 164–70
 input, 281–87, 302
 modeling, 267–68, 427
 output primitives, 95–96, 113, 131, 133, 302
 structures, 251–60
 three-dimensional transformations, 425–26
 three-dimensional viewing, 464–66

- PHIGS (*cont.*)
 two-dimensional transformations, 208-9
 two-dimensional viewing, 222-23
 workstation, 79
- PHIGS+, 78
- Phong specular-reflection model, 501-4
- Phong shading, 525-27
- Phosphor, 37-39
- Photorealism, 495
- Physically based modeling, 393-95, 588, 596
- Pick
 distance, 279-80, 288
 filter, 284-85
 identifier, 284
 input device, 276, 279-80
 window, 280
- Pickability (structure), 254
- Picking, 284
- Picture element (pixel), 40
- Piecewise approximation (spline), 315-16
- Pie chart, 11-12, 138-40
- Piteway-Watkins antialiasing, 177-78
- Pivot point, 186
- PixBit, 210
- Pixel, 40
 addressing, 114-15
 grid, 114
 mask, 146, 149-51, 152, 517
 patterns (halftone), 516
 phasing, 172
 ray, 528-29
 weighting mask, 174, 555
- Pixel-order scanning, 554-55
- Pixmap, 40
- Plane
 clipping, 456-61
 coefficients, 308
 complex, 615
 equations, 308-9
 far (clipping), 447
 inside-outside faces, 308
 near (clipping), 447
 normal vector, 308-9
- Plan view, 440
- Plasma-panel display, 45-46
- Plotters (*see also* Printers)
 beltbed, 74
 color, 73, 74
 drum, 74
 flatbed, 74, 75
 ink-jet, 72-73, 74
 laser, 72, 73, 74
 pen, 74, 75
 rollfeed, 74, 75
- Point
 clipping, 225
 control (spline), 316
 coordinate, 602, 605, 612
 plotting, 84-86
 sampling, 87
 as unit of character size, 164
- Point light source, 496
- Polar coordinates, 601-2, 604
- Polar form (complex number), 616-17
- Polygon
 active edge list, 122, 477
 characteristic, 316
 control, 316
 edge vector, 126
 fill, 117-27 (*see also* Area filling)
 inside face, 308
 inside-outside tests, 125-26 (*see also* Plane)
 mesh, 306, 309-10
 normal vector, 308-9
 outside face, 308
 plane equation, 307-9
 rendering (shading), 522-27
 ray intersection, 533-34
 sorted edge table, 121
 splitting, 235-37
 surface, 305-6
 surface detail, 553-54
 tables, 121-22, 306-7, 476-77
- Polygon clipping
 parallel methods, 239
 parametric methods, 243
 Sutherland-Hodgeman, 238-42
 three-dimensional, 456-57
 Weiler-Atherton, 242-43
- Polyline, 95-96
- Polyline connections, 148-49
- Polynomial curve, 110
- Position emission tomography (PET), 32-33
- Positioning methods, 288
- Postfiltering, 172 (*see also* Antialiasing)
- Posting (structures), 252
- Precision (text), 166-67
- Prefiltering, 172 (*see also* Antialiasing)
- Presentation graphics, 11-13
- Pressure-sensitive joystick, 63, 64
- Primary colors, 568, 569
- Primitives, 77 (*see also* Output primitives)
- Principal axes, 440
- Principal vanishing point, 446
- Printers
 dot-matrix, 72
 electrothermal, 73
 impact, 72
 laser, 72, 73, 74
 nonimpact, 72
 electrostatic, 73, 74
 ink-jet, 72-73, 74
- Priority
 structure, 252
 view-transformation input, 283
- Procedural object representation, 362-92
- Procedural texture mapping, 556-57
- Production rules, 387-89
- Progressive refinement (radiosity), 549-50
- Projecting square line cap, 147
- Projection
 axonometric, 440
 cabinet, 443
 cavalier, 443
 center of, 438
 frustum, 447
 isometric, 440-41
 oblique, 439, 441-43, 447-50, 452-53
 orthographic, 439, 441, 447-48
 parallel, 298, 439-43, 452-54
 perspective, 299, 439, 443-47, 454-56
 plane, 433
 reference point, 438
 vector, 450, 452-53
 view volume, 447
 window, 447
- Pseudo-color methods, 396
- Pure color, 567, 569
- Purity (light), 567
- Purple line, 570
- Quadrilateral mesh, 309-10
- Quadtree, 359
- Quaternion, 617
 addition, 618
 in fractal constructions, 384-85
 inverse, 618
 magnitude, 618
 multiplication, 618
 ordered-pair representation, 419, 618
 rotations, 419-20
 scalar multiplication, 618
 scalar part, 419, 617
 vector part, 419, 618
- ## R
- Radiant energy (Radiance), 544
- Radiosity model, 544-51
 energy transport equation, 546
 form factors, 546
 hemicycle, 548-49
 luminance, 544
 progressive refinement, 545-50
 reflectivity factor, 546
 surface enclosure, 546
- Random dither (noise), 520-21
- Random midpoint-displacement methods, 373-78
- Random-scan monitor, 41-42
 color, 42
 refresh display file, 42
- Random-scan system
 display file, 42, 56
 graphics controller, 56
 processing unit, 56
- Random walk, 372
- Raster animation, 586-87
- Raster ops, 210
- Raster-scan monitor, 40-41
 bilevel, 40
 bitmap, 40
 color, 42-45
 frame buffer, 40
 horizontal retrace, 41
 interlacing, 41
 pixel, 40
 pixmap, 40
 refresh buffer, 40
 vertical retrace, 41
- Raster-scan system
 cell encoding, 55
 display processor, 55
 run-length encoding, 55
 scan conversion, 55
 video controller, 53-55
- Raster transformations, 210-11
- Rational spline, 347-49
- Ray casting
 constructive solid geometry, 357-59
 visible-surface detection, 487-88
- Ray tracing, 527
 adaptive sampling, 538-40
 adaptive subdivision, 536-38
 antialiasing, 538-43
 area sampling, 539
 basic algorithm, 528-31
 bundles, 538
 camera-lens effects, 541
 cell traversal, 536-37
 codes, 541
 cone tracing, 540
 distributed, 540-43
 eye ray (*see* pixel ray)

Q

- Quadric curves, 310
- Quadric surfaces, 310-12

- equation, 531
 intersection calculations, 531–35
 jittering, 541
 light-buffer method, 537
 motion blur, 541, 542–43
 pixel (primary) ray, 528–29
 polygon intersection, 533–34
 in radiosity model, 550
 reflection ray, 529, 530–31
 refraction ray, 529, 530
 secondary ray, 529
 shadow ray, 529–30
 space subdivision, 535–38
 sphere intersection, 532–33
 stochastic sampling, 540
 supersampling, 538–40
 tree, 529
 uniform subdivision, 536
 Read function, 210
 Readable typeface, 132
 Real-time animation, 55, 585, 586
 Reference point (viewing), 218, 219, 438
 Reflection:
 angle of incidence, 499
 axis, 201
 coefficients, 497–502
 diffuse, 497–500
 Fresnel laws, 501
 halfway vector, 503
 Lambertian, 498
 mapping, 552
 plane, 422
 ray, 529
 specular, 500–504, 530
 vector, 501–3, 530
 Reflection transformation, 201–3, 423
 Reflectivity, 498
 Reflectivity factor (radiosity), 446
 Refraction:
 angle, 509
 diffuse, 509
 index, 509
 ray, 529, 530
 Snell's law, 509
 specular, 509
 transmission vector, 510, 530–31
 transparency coefficient, 510
 vector, 510, 530–31
 Refresh buffer, 40 (*see also* Frame buffer)
 Refresh CRT, 37–45 (*see also* Cathode-ray tube)
 Refresh display file, 42
 Refresh rate (CRT), 40–41
 Region codes (clipping),
 three-dimensional, 460
 two-dimensional, 227
 Relative coordinates, 96
 Rendering (*see* Surface rendering)
 Request input mode, 281, 282–85
 Resolution:
 display device, 39–40
 half-tone approximations, 518
 Retrace (electron beam), 41
 REYES, 475
 RGB chromaticity coordinates, 573
 RGB color model, 572–73
 RGB monitor, 45 (*see also* Video monitor)
 Right-hand coordinate system, 602
 Right-hand rule, 608
 Rigid-body transformation, 185, 196–97
 Rigid motion, 196
 Roots:
 nonlinear equations, 621–22
 complex numbers, 617
 Rotation:
 angle, 186
 axis, 186, 413–20
 axis vector, 414–15
 composition, 191
 inverse, 190, 413
 matrix representation, 190, 192–93, 410–12,
 418–19, 420
 pivot point, 186
 quaternion, 419–20
 raster methods, 211
 three-dimensional, 409–20
 two-dimensional, 186–87, 190, 191, 192–93
 x axis, 411–12
 y axis, 412
 z axis, 409–11
 Rotational polygon-splitting method, 237
 Round join, 148–149
 Round line cap, 147
 Row vector, 611
 Rubber-band methods, 290, 291
 Run-length encoding, 56

S

 Sample input mode, 281, 285
 Sampling:
 adaptive, 538–40
 area, 172, 174
 line, 87, 88–89
 Nyquist interval, 171
 point, 87
 supersampling, 172–74, 538–40
 weighted, 174
 Sans serif typeface, 132
 Saturation (light), 567
 Scalar data-field visualization, 395–99
 Scalar input methods, 277–78
 Scalar product of two vectors, 607–8
 Scaling:
 in arbitrary directions, 193–94
 composition, 192
 curved objects, 188
 differential, 188
 factors, 187, 421
 fixed point, 188, 421
 inverse, 190, 421–22
 matrix representation, 190, 421
 nonuniform (differential), 188, 421
 parameters (factors), 187, 421
 raster methods, 211
 three-dimensional, 420–22
 two-dimensional, 187–88, 190, 192, 193–94
 uniform, 187–88, 421
 Scan conversion, 55
 areas, 117–30
 characters, 132–33
 circles, 98–102
 curved-boundary areas, 126–30
 curved lines, 110–13
 ellipses, 103–10
 patterned fill, 159–63
 points, 84, 85–86
 polygons, 117–27
 straight lines, 86–94 (*see also* Line-drawing
 algorithms)
 structure-list traversal, 252
 Scan line, 40
 Scan-line interlacing, 41
 Scan-line algorithms:
 area filling, 117–27, 158–63
 visible-surface detection, 476–78
 Scanner, 67, 68
 Scanning:
 image-order, 554
 inverse, 554
 pixel-order, 554–55
 texture, 554
 Scientific visualization, 25, 395 (*see also* Data
 visualization)
 Screen coordinates, 54, 76, 114 (*see also* Coordinate
 system, device)
 Scripting system (animation), 588
 Secondary ray, 529
 Segment, 77, 251
 Self-affine fractals, 364, 372–78
 Self-inverse fractals, 364, 385–87
 Self-similar fractals, 364, 367–71
 Self-squaring fractals, 364, 378–85
 Serif typeface, 132
 Shades (color), 571, 577
 Shading algorithm: (*see* Surface rendering)
 Shading model, 495 (*see also* Illumination model)
 Shadow mask, 43
 Shadow ray, 529–30
 Shadows:
 modeling 511, 529–30, 542
 penumbra, 542
 umbra, 542
 Shape grammars, 387–90
 Shear:
 axis, 203
 matrix, 423
 in projection mapping, 442, 453, 454–56
 three-dimensional, 423
 two-dimensional, 203–5
 x-direction, 203
 y-direction, 204
 z-direction, 423
 Shift vector, 184 (*see also* Translation)
 Similarity dimension, 365
 Simpson's rule, 623
 Simulations, 5–10, 21–31 (*see also* Graphics
 applications)
 Simulators, 21–25
 Simultaneous linear equation solving, 620–21
 Singular matrix, 614
 Sketching, 13–16, 291–92
 Snell's law, 509
 Snowflake (fractal), 367–68
 Soft fill, 162–63
 Software standards, 78–79
 Solid angle, 544–45, 604
 Solid modeling: (*see also* Surface; Curved
 surface)
 applications, 4, 5, 8, 9
 constructive solid geometry, 356–59
 sweep constructions, 355–56
 Solid texture, 556
 Sonic digitizer, 66
 Sorted edge table, 121
 Spaceball, 63
 SpaceGraph system, 49
 Space-partitioning methods (ray tracing):
 adaptive, 536–38
 light buffer, 537
 ray bundles, 538
 uniform, 536
 Space-partitioning representations, 305
 Specular reflection, 497, 500–504, 530
 angle, 501
 coefficient 501–2
 Fresnel laws, 501
 halfway vector, 503
 parameter 501

- Specular reflection (*cont.*)
 Phong model, 501-4
 vector, 501-3, 530
- Specular refraction, 509
- Speed of light, 566
- Sphere, 311, 620
- Spherical coordinates, 604
- Spiral, 139-40
- Spline curve, 112, 315-16
 approximation, 316
 basis functions, 319
 basis matrix, 320
 beta-spline, 345-47
 Bézier, 327-33
 bias parameter, 325, 346
 blending functions, 319
 B-spline, 334-44
 cardinal, 323-25
 Catmull-Rom, 325
 characteristic polygon, 316
 continuity conditions, 317-19
 continuity parameter, 325
 control graph, 316
 control points, 316
 conversions, 349-50
 convex hull, 316
 cubic interpolation, 320-27
 displaying, 351-55
 Hermite, 322-23
 interpolation, 316
 knot vector, 335
 Kochanek-Bartels, 325-27
 local control, 332, 335, 336
 matrix representation, 320
 natural, 321
 NURB, 347
 Overhauser, 325
 rational, 347-49
 tension parameter, 324, 325, 341, 346
- Spline generation:
 Horner's method, 351
 forward-difference method, 351-53
 subdivision methods, 353-55
- Spline surface, 316
 Bézier, 333-34
 B-spline, 344-45
- Splitting concave polygons:
 rotational method, 237
 vector method, 236
- Spotlights, 504
- Spring constant, 393
- Spring network (nonrigid body), 393
- Square matrix, 611
- Stairstep effect, 85
- Steradian, 544-45, 604
- Stereoscopic:
 glasses, 51
 headsets, 52
 views, 6, 7, 50-52, 292, 293, 300-301
 virtual-reality applications, 5-7, 50-52
- Stochastic Sampling, 540
- Storyboard, 585
- Streamlines, 400
- String input device, 276, 277
- String precision (text), 166, 167
- Stroke input device, 276, 277
- Stroke precision (text) 166-67
- Stroke-writing display, 41 (*see also* Video monitors, random-scan)
- Structure, 77, 251
 attributes, 253-54
 basic functions, 251-54
 central structure store (CSS), 251
 concepts, 251-52
 copying, 260
 creation, 251-52
 deletion, 253, 260
 displaying (posting), 252
 editing, 254-60
 element, 255
 element pointer, 255
 filters, 253, 284-85
 hierarchy, 266-68
 highlighting filter, 253-54
 lists, 252
 metafile, 79
 pickability, 254
 posting, 252
 priority, 252
 relabeling, 253
 traversal, 252
 unposting, 252-53
 visibility, 253
 workstation filters, 254, 284-85
- Subdivision methods:
 adaptive ray tracing, 536-38
 BSP tree, 362
 fractal generation, 373-78
 octree, 359-62
 spline generation, 353-55
 uniform ray tracing, 536
- Subtractive color model (CMY), 574-75
- Superquadric, 312-14
- Supersampling, 172-74, 538-40
- Surface:
 blobby, 314-15
 curved, 310 (*see also* Curved surfaces)
 fractal, 366, 369-85
 parametric representation, 619-20
 plane, 305-9
 quadric, 310-12
 spline, 316 (*see also* Spline surface)
 superquadric, 312-14
 weighting, 174
- Surface detail, 553-60
 bump mapping, 558-59
 environment mapping, 552
 frame mapping, 559-60
 image-order scanning, 554
 inverse scanning, 554
 pattern mapping, 554
 pixel-order scanning, 554
 polygon mesh, 553-54
 procedural texturing, 556-57
 solid texture mapping, 556
 texture mapping, 554-56
 texture scanning, 554
- Surface enclosure (radiosity), 546
- Surface normal vector, 308-9, 523, 558
- Surface rendering, 297-98, 495
 antialiasing, 538-43
 bump mapping, 558-59
 constant-intensity shading, 522-23
 environment mapping, 552
 fast Phong shading, 526-27
 flat shading, 522
 frame mapping, 559-60
 Gouraud shading, 523-25
 intensity interpolation, 523
 Mach bands, 525
 normal-vector interpolation, 525
 Phong shading, 525-27
 polygon methods, 522-27
 polygon surface detail, 553-54
 procedural texturing, 556-57
 radiosity, 544-50
 ray-tracing, 527-43
 texture mapping, 554-56
- Surface shading (*see* Surface rendering)
- Sutherland-Hodgeman polygon-clipping, 238-42
- Sweep representations, 355-56
- Symbol, 261
 hierarchies, 262-63
 instance, 261
 in modeling, 261-64
- Symmetry:
 circle, 97-98
 in curve-drawing algorithms, 97-98, 103, 112
 ellipse, 103
- T
- Table (polygon)
 attribute, 306
 edge, 121-22, 306-7, 476-77
 geometric, 306-7
 sorted edge table, 121
 vertex, 306-7
- Tablet, 64-67 (*see also* Digitizer)
- Task planning, 13
- Tension parameter (spline), 324, 325, 341, 346
- Tensor, 610
 contraction, 402
 data-field visualization, 401-2
 metric, 610-11
- Terrain (fractal), 372-78
- Tessellated surface, 306
- Text: (*see also* Character)
 alignment, 166
 attributes, 163-67, 169-70
 clipping, 244, 245
 generation, 132-33
 path, 166
 precision, 166-67
- Texture, 553 (*see also* Surface rendering)
 mapping, 554-56
 procedural methods, 556-57
 scanning, 554
 solid, 556
 space, 554, 556-57
- Thin-film electroluminescent display, 46
- Three-point perspective projection, 446
- Tiling, 160, 306
- Time chart, 11, 13
- Tint (color), 571, 577
- Tint fill, 162
- Tone (color), 571, 577
- Topline (character), 164
- Topological covering, 365-66
- Touch panel, 68-70
- Trackball, 63
- Transformation:
 affine, 208
 basic geometric, 184-200, 408-22
 commutative, 194-95
 composite, 191-200, 423-25
 computational efficiency, 195-97
 coordinate system, 205-7, 426-29
 functions, 208-9, 425-26
 geometric, 77, 184
 instance, 265-68
 local, 265-68
 matrix representations, 188-90
 modeling, 77, 265-68, 426-29
 noncommutative, 194-95
 parallel projection, 298-99, 438
 perspective projection, 299, 438
 raster methods 210-11

- reflection, 201–3, 422
 rotation, 186–87, 190–93, 409–20
 scaling, 187–88, 190, 192–94, 420–22
 shear, 203–5, 423
 three-dimensional geometric, 408–22
 three-dimensional viewing, 432–56
 translation, 184–85, 190, 191, 408–9
 two-dimensional geometric, 184–205
 two-dimensional viewing, 217–22
 viewing, 77, 217–22, 432–56
 window-to-viewport, 217, 220–22
 workstation, 221–22, 466
 world-to-viewing coordinate, 218–20, 437–38
- Translation:**
 composition, 191
 curved object, 185
 distances, 184, 408
 inverse, 190, 409
 matrix representation, 190, 408
 raster methods, 210
 three-dimensional, 408–9
 two-dimensional, 184–85, 190, 191
 vector, 184, 408
- Transmission vector (refraction), 510, 530–31
- Transparency** (*see also* Refraction; Ray tracing)
 coefficient, 510
 modeling, 508–11
 opacity factor, 510
 vector, 510, 530–31
- Transpose (matrix), 613
- Trapezoid rule, 623
- Traversal state list, 252
- Triangle strip, 309
- Tristimulus vision theory, 572
- True-color system, 45
- Twist angle, 434
- Two-point perspective projection, 446
- Typeface**, 131–33 (*see also* Font)
 legible, 132
 readable, 132
 sans serif, 132
 serif, 132
- U**
- Umbra shadow, 542
- Unbundled attributes, 168
- Uniform B-splines, 336–44
- Uniform scaling, 187–88, 421
- Uniform spatial subdivision:
 octree, 359–62
 ray tracing, 536
- Unit cube (clipping), 458
- Up vector (character), 165
- User dialogue, 272–73
- User help facilities, 274
- User interface, 34, 272–76, 288–93 (*see also*
 Graphical user interface)
- User model, 272
- uv* coordinate system, 435–38
- uv* plane, 435
- V**
- Valuator input device, 276, 277–78
- Value (HSV parameter), 575
- Vanishing point, 446
- Varifocal mirror, 49
- Vector, 605, 611–12
 addition, 607
- basis, 609
- column, 611
- components, 605
- cross product, 608–9
- data-field visualization, 400–401
- direction angles, 606
- direction cosines, 606
- dot (inner) product, 607–8
- knot, 335
- magnitude (length), 605
- polygon edge, 126
- product, 608–9
- projection, 450, 452–53
- in quaternion representation, 419, 618
- reflection, 501–3, 530
- rotation, 414–15
- row, 611
- scalar multiplication, 607
- scalar (dot) product, 607–8
- space, 609
- specular reflection, 500–504, 530
- surface normal, 308–9, 523, 558
- transmission (refraction), 510, 530–31
- translation, 184, 408
- Vector method (polygon splitting), 236
- Vector monitor, 41
- Vertex table, 306–7
- Vertical retrace, 41
- Video controller, 53–55
- Video lookup table, 155, 513
- Video monitor (*see also* Cathode-ray tube)
 calligraphic, 41
 color CRT, 42–45
 composite, 44–45
 direct-view storage tube (DVST), 45
 emissive, 45
 flat-panel, 45
 full-color, 45
 gas-discharge, 45
 LCD (liquid crystal device), 47–48
 LED (light-emitting diode), 46–47
 nonemissive, 45
 plasma panel, 45–46
 random-scan, 41–42
 raster-scan, 40–41
 refresh CRT, 37–45
 resolution, 39–40
 RGB, 45
 stereoscopic, 50–52
 thin-film electroluminescent, 46
 three-dimensional, 49
 true-color, 45
 vector, 41
- View:**
 look-at point, 434
 reference point, 218, 219, 438
 up vector, 219, 434
 twist angle, 434
- Viewing:**
 stereoscopic, 6, 7, 50–52, 292, 293, 300–301
 three-dimensional, 297
 two-dimensional, 217–45
- Viewing coordinates:**
 left-handed, 435
 three-dimensional, 433–34
 two-dimensional, 218, 219–20
- Viewing transformation:**
 back (far) clipping plane, 447
 clipping, 224–45, 456–63
 front (near) clipping plane, 447
 frustum, 447
 functions, 222–23, 464–66
 hardware implementation, 463–64
- input priority, 283
 normalized projection coordinates, 458
 normalized view volume, 458–61
 pipeline, 217–19, 432–33
 three-dimensional, 432–33
 two-dimensional, 217–22
 viewport, 217, 458–60
 view volume, 447
 window, 217, 447
 workstation mapping, 221–22, 466
- Viewing table**, 223, 465
- View plane**, 433–34
 normal vector, 434
 position, 434–35
 window, 447
- Viewport:**
 clipping, 224, 460–61
 function, 222–23
 priority, 283
 three-dimensional (*see* View volume)
 two-dimensional, 217
 workstation, 222
- View reference point**, 218, 219, 434
- View-up vector**, 219, 434
- View volume**, 447
 unit cube, 458
 normalized, 458
 perspective, 447–49
 parallel, 447–50
- View window**, 447
- Virtual reality:**
 applications, 5–8, 466–67
 display devices, 51–52
 input devices, 64
 environments, 292–93
- Visible structure**, 253
- Visible-line detection**, 490 (*see also* Depth culling)
- Visible-surface detection**, 470
 A-buffer method, 475–76
 algorithm classification, 470–71
 area-subdivision method, 482–85
 back-face detection, 471–72
 BSP-tree method, 481–82
 comparison of algorithms, 491–92
 curved surfaces, 487–90
 depth-buffer (z-buffer) method, 472–75
 depth-sorting method, 478–81
 function, 490–91
 image-space methods, 470
 object-space methods, 470
 octree methods, 485–87
 painter's algorithm (depth sorting), 478
 ray-casting method, 487–88
 scan-line method, 476–78
 surface contour plots, 489–90
 wireframe methods, 490
- Vision** (tristimulus theory), 572
- Visualization:**
 applications, 25–31
 methods, 395–403 (*see also* Data visualization)
 voice systems, 70–71
- Volume calculations** (CSG), 358–59
- Volume element**, 360
- Volume rendering**, 399
- Voxel**, 360
- W**
- Warn lighting model, 504–5
- Wavelength (light), 566
- Weighted sampling, 174, 555
- Weighting surface, 174

Weiler-Atherton polygon-clipping algorithm, 242-43
 White light, 567, 570
 Winding number, 125
 Window:
 functions, 222-23, 465
 manager, 34, 273
 nonrectangular, 217
 pick, 280
 projection, 447
 rotated, 218, 219-20
 three-dimensional viewing, 432-56
 two-dimensional viewing, 217
 user-interface, 34, 273
 view-plane, 433-34
 workstation, 221-22, 465
 Windowing transformation, 217
 panning, 219
 zooming, 218-19
 Window-to-viewport mapping, 217, 220-22
 Wireframe, 4, 5, 298
 Wireframe visibility algorithms, 490

Workstation
 in graphics applications, 57-60
 identifier, 79
 PHIGS, 79
 pick filter, 284-85
 structure filters, 254, 284-85
 transformation, 221-22, 466
 window, 221-22, 465
 viewport, 222, 465
 World coordinates, 76
 World-to-Viewing coordinate transformation, 218, 219-20, 437-38
 Write function, 210

X

x-axis rotation, 411-12
 x-direction shear, 203
 X Window System, 272
 XYZ color model, 569

Y

y-axis rotation, 412
 y-direction shear, 204
 YIQ color model, 574

Z

z-axis rotation, 409-11
 z-buffer algorithm, 472 (*see also* Depth-buffer algorithm)
 z-direction shear, 423
 Z mouse, 62-63
 zooming, 218-19

Function Index

A

awaitEvent, 285

B

buildTransformationMatrix, 209
buildTransformationMatrix3, 426

C

cellArray, 131
changeStructureIdentifier, 253
closeStructure, 251
composeMatrix, 209
composeMatrix3, 426
composeTransformationMatrix, 209
composeTransformationMatrix3, 426
copyAllElementsFromStructure, 260

D

deleteAllStructures, 253
deleteElement, 257
deleteElementRange, 258
deleteElementsBetweenLabels, 260
deleteStructure, 253
deleteStructureNetwork, 268

E

emptyStructure, 258
evaluateViewMappingMatrix, 222
evaluateViewMappingMatrix3, 465
evaluateViewOrientationMatrix, 222
evaluateViewOrientationMatrix3, 464
executeStructure, 267

F

fillArea, 131
fillArea3, 302
fillAreaSet, 131
fillCircle, 131
fillCircleArc, 131
fillEllipse, 131
fillEllipseArc, 131
fillRectangle, 131

G

generalizedDrawingPrimitive, 113
getChoice, 286
getLocator, 286
getLocator3, 302
getPick, 286
getPixel, 86
getString, 286
getStroke, 286
getValuator, 286

I

initializeChoice, 287
initializeLocator, 287
initializePick, 287
initializeString, 287
initializeStroke, 287
initializeValuator, 287
inquire, 170

L

label, 258

O

offsetElementPointer, 256
openStructure, 251

P

polyline, 96
polyline3, 302
polymarker, 133
postStructure, 252

R

requestChoice, 284
requestLocator, 282-83
requestPick, 284
requestString, 283
requestStroke, 282-83
requestValuator, 284
rotate, 208
rotateX, 425
rotateY, 425
rotateZ, 425

S

sampleChoice, 285
sampleLocator, 285
samplePick, 285
sampleString, 285
sampleStroke, 285
sampleValuator, 285
scale, 208
scale3, 425
setCharacterExpansionFactor, 165
setCharacterHeight, 164
setCharacterSpacing, 165
setCharacterUpVector, 165
setChoiceMode, 281
setColourRepresentation, 156
setEditMode, 256
setElementPointer, 255
setElementPointerAtLabel, 259
setHighlightingFilter, 254
setHLHSIdentifier, 491
setIndividualASF, 168
setInteriorColourIndex, 158
setInteriorIndex, 169
setInteriorRepresentation, 169
setInteriorStyle, 158
setInteriorStyleIndex, 159
setInvisibilityFilter, 253
setLinetype, 145
setLinewidthScaleFactor, 146
setLocalTransformation, 267
setLocalTransformation3, 426
setLocatorMode, 281-82
setMarkerSizeScaleFactor, 167
setMarkerType, 167
setPatternReferencePoint, 159
setPatternRepresentation, 159
setPatternSize, 159
setPickFilter, 284
setPickIdentifier, 284
setPickMode, 281-82
setPixel, 85, 161
setPolylineColourIndex, 149
setPolylineIndex, 169
setPolylineRepresentation, 168
setPolymarkerColourIndex, 168
setPolymarkerIndex, 170
setPolymarkerRepresentation, 170
setStringMode, 281
setStrokeMode, 281
setTextAlignment, 166
setTextColourIndex, 164
setTextFont, 164
setTextIndex, 170
setTextMode, 281-82
setTextPath, 166
setTextPrecision, 166

Function Index

setTextRepresentation, 169
setValuatorMode, 281
setViewIndex, 223, 466
setViewRepresentation, 223
setViewRepresentation3, 465
setViewTransformationInputPriority, 283
setWorkstationViewport, 223
setWorkstationViewport3, 466
setWorkstationWindow, 223

setWorkstationWindow3, 466

T

text, 133
text3, 302
transformPoint, 209
transformPoint3, 426

translate, 208
translate3, 302, 425–26

U

unpostAllStructures, 253
unpostStructure, 252