splines [BREW77], are useful for this situation. One member of this spline family is able to interpolate the points P_1 to P_{m-1} from the sequence of points P_0 to P_m. In addition, the tangent vector at point P_i is parallel to the line connecting points P_{i-1} and P_{i+1}, as shown in Fig. 11.32. However, these splines do not posess the convex-hull property. The natural (interpolating) splines also interpolate points, but without the local control afforded by the Catmull–Rom splines.

Designating M_{CR} as the Catmull-Rom basis matrix and using the same geometry matrix G_{Bs_i} of Eq. (11.32) as was used for B-splines, the representation is

$$Q^i(t) = T \cdot M_{\text{CR}} \cdot G_{Bs_i} = \frac{1}{2} \cdot T \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix}. \tag{11.47}$$

A method for rapidly displaying Catmull–Rom curves is given in [BARR88b].

Another spline is the *uniformly shaped β-spline* [BARS88; BART87], which has two additional parameters, β_1 and β_2, to provide further control over shape. It uses the same geometry matrix G_{Bs_i} as the B-spline, and has the convex hull property. The *β-spline basis matrix* M_β is

$$M_\beta = \frac{1}{\delta} \begin{bmatrix} -2\beta_1^3 & 2(\beta_2 + \beta_1^3 + \beta_1^2 + \beta_1) & -2(\beta_2 + \beta_1^2 + \beta_1 + 1) & 2 \\ 6\beta_1^3 & -3(\beta_2 + 2\beta_1^3 + 2\beta_1^2) & 3(\beta_2 + 2\beta_1^2) & 0 \\ -6\beta_1^3 & 6(\beta_1^3 - \beta_1) & 6\beta_1 & 0 \\ 2\beta_1^3 & \beta_2 + 4(\beta_1^2 + \beta_1) & 2 & 0 \end{bmatrix},$$

$$\delta = \beta_2 + 2\beta_1^3 + 4\beta_1^2 + 4\beta_1 + 2. \tag{11.48}$$

The first parameter, β_1, is called the *bias* parameter; the second parameter, β_2, is called the *tension* parameter. If $\beta_1 = 1$ and $\beta_2 = 0$, M_β reduces to M_{Bs} (Eq. (11.34)) for B-splines. As β_1 is increased beyond 1, the spline is "biased," or influenced more, by the tangent vector on the parameter-increasing side of the points, as shown in Fig. 11.33. For values of β_1 less than 1, the bias is in the other direction. As the tension parameter β_2 increases, the spline is pulled closer to the lines connecting the control points, as seen in Fig. 11.34.

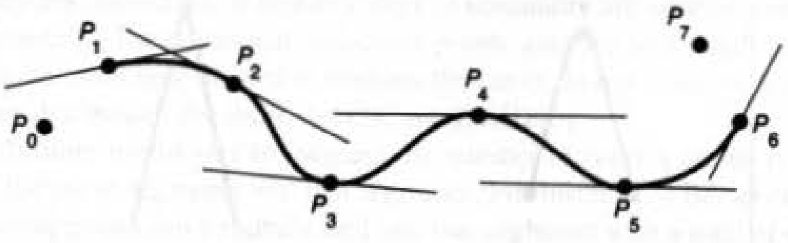


Fig. 11.32 A Catmull–Rom spline. The points are interpolated by the spline, which passes through each point in a direction parallel to the line between the adjacent points. The straight line segments indicate these directions.

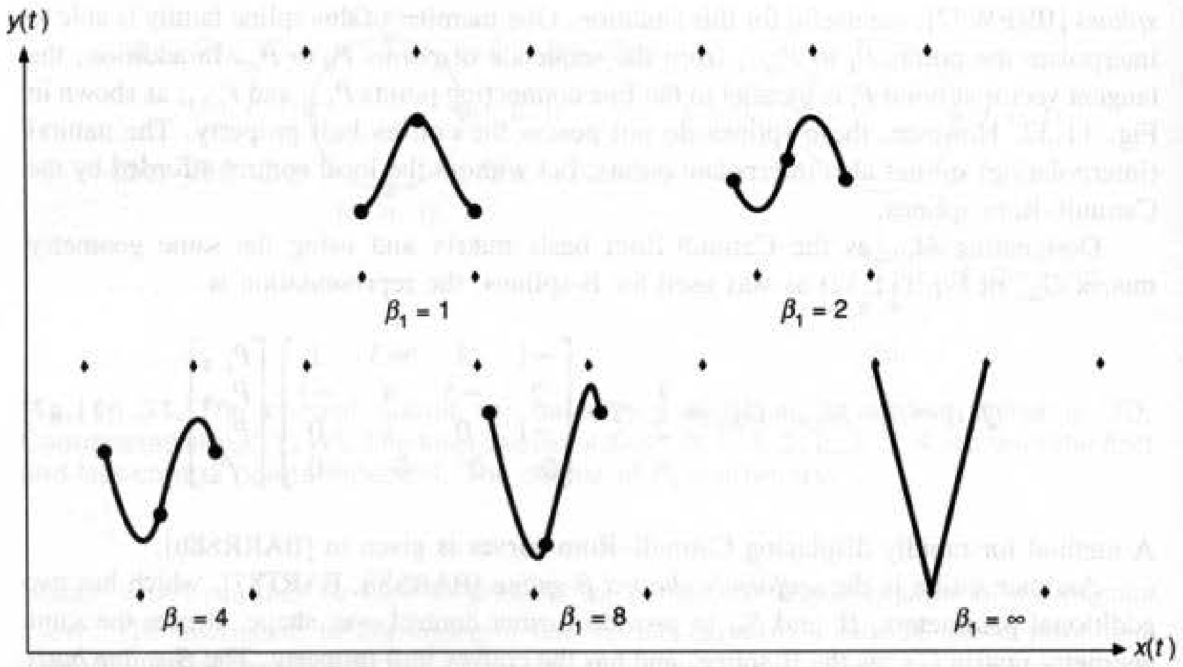


Fig. 11.33 Effect on a uniformly shaped β-spline of increasing the bias parameter β_1 from 1 to infinity.

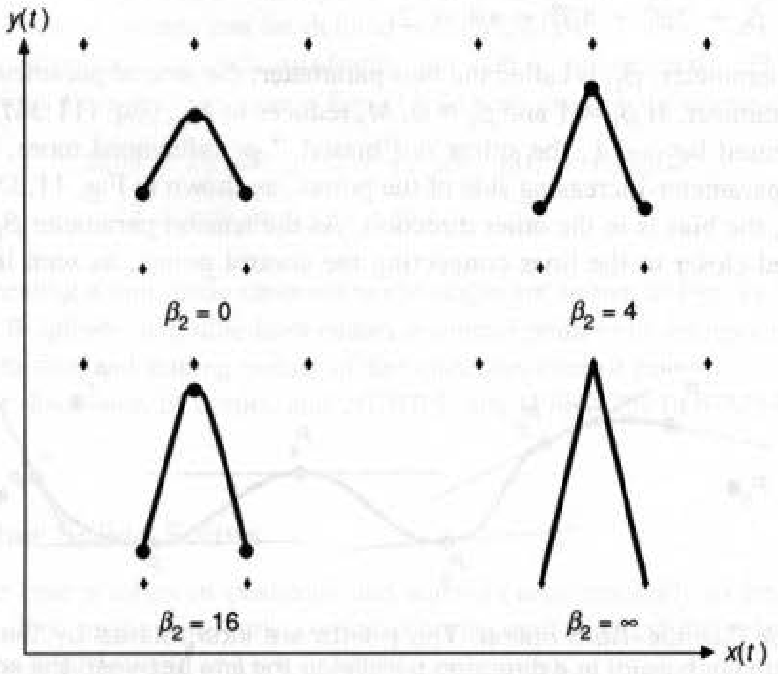


Fig. 11.34 Effect on a uniformly shaped β-spline of increasing the tension parameter β_2 from 0 to infinity.

These β-splines are called *uniformly shaped* because β_1 and β_2 apply uniformly, over all curve segments. This global effect of β_1 and β_2 violates the spirit of local control. *Continuously shaped β-splines* and *discretely shaped β splines* associate distinct values of β_1 and β_2 with each control point, providing a local, rather than a global, effect [BARS83; BART87].

Although β-splines provide more shape generality than do B-splines, they are G^2 but only C^0 continuous at the join points. This is not a drawback in geometric modeling, but can introduce a velocity discontinuity into an animation's motion path. In the special case $\beta_1 = 1$, the β-splines are C^1 continuous.

A variation on the Hermite form useful for controlling motion paths in animation was developed by Kochanek and Bartels [KOCH84]. The tangent vector R_i at point P_i is

$$R_i = \frac{(1 - a_i)(1 + b_i)(1 + c_i)}{2}(P_i - P_{i-1}) + \frac{(1 - a_i)(1 - b_i)(1 - c_i)}{2}(P_{i+1} - P_i), \tag{11.49}$$

and the tangent vector R_{i+1} at point P_{i+1} is

$$\begin{aligned} R_{i+1} &+ \frac{(1 - a_{i+1})(1 + b_{i+1})(1 - c_{i+1})}{2}(P_i - P_{i-1}) \\ &+ \frac{(1 - a_{i+1})(1 - b_{i+1})(1 + c_{i+1})}{2}(P_{i+1} - P_i). \end{aligned} \tag{11.50}$$

See Exercise 11.16 for how to find the basis matrix M_{KB}.

The parameters a_i, b_i, and c_i control different aspects of the path in the vicinity of point P_i: a_i controls how sharply the curve bends, b_i is comparable to β_1, the bias of β-splines, and c_i controls continuity at P_i. This last parameter is used to model the rapid change in direction of an object that bounces off another object.

11.2.7 Subdividing Curves

Suppose you have just created a connected series of curve segments to approximate a shape you are designing. You manipulate the control points, but you cannot quite get the shape you want. Probably, there are not enough control points to achieve the desired effect. There are two ways to increase the number of control points. One is the process of *degree elevation*: The degree of the splines is increased from 3 to 4 or more. This adjustment is sometimes necessary, especially if higher orders of continuity are needed, but is generally undesirable because of the additional inflection points allowed in a single curve and the additional computational time needed to evaluate the curve. In any case, the topic is beyond the scope of our discussion; for more details, see [FARI86].

The second, more useful way to increase the number of control points is to *subdivide* one or more of the curve segments into two segments. For instance, a Bézier curve segment with its four control points can be subdivided into two segments with a total of seven control points (the two new segments share a common point). The two new segments exactly match the one original segment until any of the control points are actually moved; then, one or both of the new segments no longer matches the original. For nonuniform B-splines, a more general process know as *refinement* can be used to add an arbitrary number of control points to a curve. Another reason for subdividing is to display a curve or surface. We elaborate on this in Section 11.2.9, where we discuss tests for stopping the subdivision.

Given a Bézier curve $Q(t)$ defined by points P_1, P_2, P_3, and P_4, we want to find a left curve defined by points L_1, L_2, L_3, and L_4, and a right curve defined by points R_1, R_2, R_3, and R_4, such that the left curve is coincident with Q on the interval $0 \le t < \frac{1}{2}$ and the right curve is coincident with Q on the interval $\frac{1}{2} \le t < 1$. The subdivision can be accomplished using a geometric construction technique developed by de Casteljau [DECA59] to evaluate a Bézier curve for any value of t. The point on the curve for a parameter value of t is found by drawing the construction line L_2H so that it divides P_1P_2 and P_2P_3 in the ratio of $t{:}(1 - t)$, HR_3 so that it similarly divides P_2P_3 and P_3P_4, and L_3R_2 to likewise divide L_2H and HR_3. The point L_4 (which is also R_1) divides L_3R_2 by the same ratio and gives the point $Q(t)$. Figure 11.35 shows the construction for $t = \frac{1}{2}$, the case of interest.

All the points are easy to compute with an add and shift to divide by 2:

$$L_2 = (P_1 + P_2)/2,\ H = (P_2 + P_3)/2,\ L_3 = (L_2 + H)/2,\ R_3 = (P_3 + P_4)/2,$$
$$R_2 = (H + R_3)/2,\ L_4 = R_1 = (L_3 + R_2)/2. \tag{11.51}$$

These results can be reorganized into matrix form, so as to give the left Bézier division matrix D_B^L and the right Bézier division matrix D_B^R. These matrices can be used to find the geometry matrices G_B^L of points for the left Bézier curve and G_B^R for the right Bézier curve:

$$G_B^L = D_B^L \cdot G_B = \frac{1}{8}\begin{bmatrix} 8 & 0 & 0 & 0 \\ 4 & 4 & 0 & 0 \\ 2 & 4 & 2 & 0 \\ 1 & 3 & 3 & 1 \end{bmatrix}\begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix},$$

$$G_B^R = D_B^R \cdot G_B = \frac{1}{8}\begin{bmatrix} 1 & 3 & 3 & 1 \\ 0 & 2 & 4 & 2 \\ 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 8 \end{bmatrix}\begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}. \tag{11.52}$$

Notice that each of the new control points L_i and R_i is a weighted sum of the points P_i, with the weights positive and summing to 1. Thus, each of the new control points is in the convex hull of the set of original control points. Therefore, the new control points are no farther from the curve $Q(t)$ than are the original control points, and in general are closer

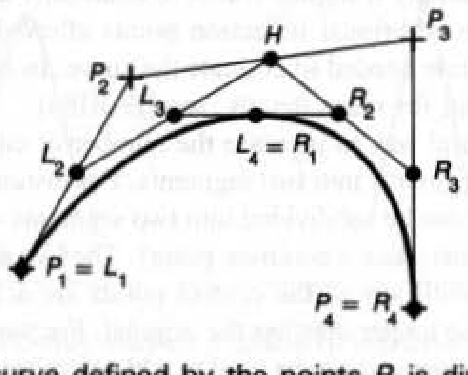


Fig. 11.35 The Bézier curve defined by the points P_i is divided at $t = \frac{1}{2}$ into a left curve defined by the points L_i and a right curve defined by the points R_i.

than the original points. This *variation-diminishing* property is true of all splines that have the convex-hull property. Also notice the symmetry between D^L_B and D^R_B, a consequence of the symmetry in the geometrical construction.

Dividing the Bézier curve at $t = \frac{1}{2}$ often gives the interactive user the control needed, but it might be better to allow the user to indicate a point on the curve at which a split is to occur. Given such a point, it is easy to find the approximate corresponding value of t. The splitting then proceeds as described previously, except that the construction lines are divided in the $t{:}(1 - t)$ ratio (see Exercise 11.22).

The corresponding matrixes D^L_{Bs} and D^R_{Bs} for B-spline subdivision are

$$G^L_{Bs_i} = D^L_{Bs} \cdot G_{Bs_i} = \frac{1}{8}\begin{bmatrix} 4 & 4 & 0 & 0 \\ 1 & 6 & 1 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 1 & 6 & 1 \end{bmatrix}\begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix},$$

$$G^R_{Bs_i} = D^R_{Bs} \cdot G_{Bs_i} = \frac{1}{8}\begin{bmatrix} 1 & 6 & 1 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 1 & 6 & 1 \\ 0 & 0 & 4 & 4 \end{bmatrix}\begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix}. \tag{11.53}$$

Careful examination of these two equations shows that the four control points of G_{Bs_i} are replaced by a total of five new control points, shared between $G^L_{Bs_i}$ and $G^R_{Bs_i}$. However, the spline segments on either side of the one being divided are still defined by some of the control points of G_{Bs_i}. Thus, changes to any of the five new control points or four old control points cause the B-spline to be no longer connected. This problem is avoided with nonuniform B-splines.

Subdividing nonuniform B-splines is not as is simple as specifying two splitting matrices, because there is no explicit basis matrix: The basis is defined recursively. The basic approach is to add knots to the knot sequence. Given a nonuniform B-spline defined by the control points $P_0, \ldots, P_n$ and a value of $t = t'$ at which to add a knot (and hence to add a control point), we want to find the new control points $Q_0, \ldots, Q_{n+1}$ that define the same curve. (The value t' might be determined via user interaction—see Exercise 11.21.) The value t' satisfies $t_j < t' \le t_{j+1}$. Böhm [BÖHM80] has shown that the new control points are given by

$$\begin{aligned} Q_0 &= P_0 \\ Q_i &= (1 - a_i)P_{i-1} + a_iP_i, \quad 1 \le i \le n \\ Q_{n+1} &= P_n \end{aligned} \tag{11.54}$$

where a_i is given by:

$$\begin{aligned} a_i &= 1, & & 1 \le i \le j - 3 \\ a_i &= \frac{t' - t_i}{t_{i+3} - t_i}, & & j - 2 \le i \le j \quad \text{(division by zero is zero)} \\ a_i &= 0, & & j + 1 \le i \le n. \end{aligned} \tag{11.55}$$

This algorithm is a special case of the more general Oslo algorithm [COHE80], which inserts any number of knots into a B-spline in a single set of computations. If more than a few knots are being inserted at once, the Oslo algorithm is more efficient than is the Böhm algorithm.

As an example of Böhm subdivision, consider the knot sequence (0, 0, 0, 0, 1, 1, 1, 1), the four x coordinates of the control point vector (5.0, 8.0, 9.0, 6.0), and a new knot at $t = 0.5$ (i.e., $n = 3$, $j = 3$). The a_i values defined by Eq. (11.55) are (0.5, 0.5, 0.5). Applying Eq. (11.54) we find that the x coordinates of the new Q_i control points are (5.0, 6.5, 8.5, 7.5, 6.0).

Notice that adding a knot causes two old control points to be replaced by three new control points. Furthermore, segments adjacent to the subdivided segment are defined only in terms of the new control points. Contrast this to the less attractive case of uniform–B-spline subdivision, in which four old control points are replaced by five new ones and adjacent segments are defined with the old control points, which are no longer used for the two new segments.

Hierarchical B-spline refinement is another way to gain finer control over the shape of a curve [FORS89]. Additional control points are added to local regions of a curve, and a hierarchical data structure is built relating the new control points to the original points. Use of a hierarchy allows further additional refinement. Storing the hierarchy rather than replacing the old control points with the larger number of new control points means that the original control points can continue to be used for coarse overall shaping of the curve, while at the same time the new control points can be used for control of details.

11.2.8 Conversion Between Representations

It is often necessary to convert from one representation to another. That is, given a curve represented by geometry vector G_1 and basis matrix M_1, we want to find the equivalent geometry matrix G_2 for basis matrix M_2 such that the two curves are identical: $T \cdot M_2 \cdot G_2 = T \cdot M_1 \cdot G_1$. This equality can be rewritten as $M_2 \cdot G_2 = M_1 \cdot G_1$. Solving for G_2, the unknown geometry matrix, we find

$$M_2^{-1} \cdot M_2 \cdot G_2 = M_2^{-1} \cdot M_1 \cdot G_1, \text{ or } G_2 = M_2^{-1} \cdot M_1 \cdot G_1 = M_{1,2} \cdot G_1. \quad (11.56)$$

That is, the matrix that converts a known geometry vector for representation 1 into the geometry vector for representation 2 is just $M_{1,2} = M_2^{-1} \cdot M_1$.

As an example, in converting from B-spline to Bézier form, the matrix $M_{Bs,B}$ is

$$M_{Bs,B} = M_B^{-1} \cdot M_{Bs} = \frac{1}{6}\begin{bmatrix} 1 & 4 & 1 & 0 \\ 0 & 4 & 2 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix}. \quad (11.57)$$

The inverse is

$$M_{B,Bs} = M_{Bs}^{-1} \cdot M_B = \begin{bmatrix} 6 & -7 & 2 & 0 \\ 0 & 2 & -1 & 0 \\ 0 & -1 & 2 & 0 \\ 0 & 2 & -7 & 6 \end{bmatrix}. \quad (11.58)$$

Nonuniform B-splines have no explicit basis matrix; recall that a nonuniform B-spline over four points with a knot sequence of (0, 0, 0, 0, 1, 1, 1, 1) is just a Bézier curve. Thus, a way to convert a nonuniform B-spline to any of the other forms is first to convert to Bézier form by adding multiple knots using either the Böhm or Oslo algorithms mentioned in Section 11.2.7, to make all knots have multiplicity 4. Then the Bézier form can be converted to any of the other forms that have basis matrices.

11.2.9 Drawing Curves

There are two basic ways to draw a parametric cubic. The first is by iterative evaluation of $x(t)$, $y(t)$, and $z(t)$ for incrementally spaced values of t, plotting lines between successive points. The second is by recursive subdivision that stops when the control points get sufficiently close to the curve itself.

In the introduction to Section 11.2, simple brute-force iterative evaluation display was described, where the polynomials were evaluated at successive values of t using Horner's rule. The cost was nine multiplies and 10 additions per 3D point. A much more efficient way repeatedly to evaluate a cubic polynomial is with *forward differences*. The forward difference $\Delta f(t)$ of a function $f(t)$ is

$$\Delta f(t) = f(t + \delta) - f(t),\ \delta > 0, \tag{11.59}$$

which can be rewritten as

$$f(t + \delta) = f(t) + \Delta f(t). \tag{11.60}$$

Rewriting Eq. (11.60) in iterative terms, we have

$$f_{n+1} = f_n + \Delta f_n, \tag{11.61}$$

where f is evaluated at equal intervals of size δ, so that $t_n = n\delta$ and $f_n = f(t_n)$.

For a third-degree polynomial,

$$f(t) = at^3 + bt^2 + ct + d = T \cdot C, \tag{11.62}$$

so the forward difference is

$$\begin{aligned}\Delta f(t) &= a(t + \delta)^3 + b(t + \delta)^2 + c(t + \delta) + d - (at^3 + bt^2 + ct + d) \quad (11.63)\\ &= 3at^2\delta + t(3a\delta^2 + 2b\delta) + a\delta^3 + b\delta^2 + c\delta.\end{aligned}$$

Thus, $\Delta f(t)$ is a second-degree polynomial. This is unfortunate, because evaluating Eq. (11.61) still involves evaluating $\Delta f(t)$, plus an addition. But forward differences can be applied to $\Delta f(t)$ to simplify its evaluation. From Eq. (11.61), we write

$$\Delta^2 f(t) = \Delta(\Delta f(t)) = \Delta f(t + \delta) - \Delta f(t). \tag{11.64}$$

Applying this to Eq. (11.63) gives

$$\Delta^2 f(t) = 6a\delta^2 t + 6a\delta^3 + 2b\delta^2. \tag{11.65}$$

This is now a first-degree equation in t. Rewriting (11.64) and using the index n, we obtain

$$\Delta^2 f_n = \Delta f_{n+1} - \Delta f_n. \tag{11.66}$$

Reorganizing and replacing n by $n - 1$ yields

$$\Delta f_n = \Delta f_{n-1} + \Delta^2 f_{n-1}. \tag{11.67}$$

Now, to evaluate Δf_n for use in Eq. (11.61), we evaluate $\Delta^2 f_{n-1}$ and add it to Δf_{n-1}. Because $\Delta^2 f_{n-1}$ is linear in t, this is less work than is evaluating Δf_n directly from the second-degree polynomial Eq. (11.63).

The process is repeated once more to avoid direct evaluation of Eq. (11.65) to find $\Delta^2 f(t)$:

$$\Delta^3 f(t) = \Delta(\Delta^2 f(t)) = \Delta^2 f(t + \delta) - \Delta^2 f(t) = 6a\delta^3. \tag{11.68}$$

The third forward difference is a constant, so further forward differences are not needed. Rewriting Eq. (11.68) with n, and with $\Delta^3 f_n$ as a constant yields

$$\Delta^2 f_{n+1} = \Delta^2 f_n + \Delta^3 f_n = \Delta^2 f_n + 6a\delta^3. \tag{11.69}$$

One further rewrite, replacing n with $n - 2$, completes the development:

$$\Delta^2 f_{n-1} = \Delta^2 f_{n-2} + 6a\delta^3. \tag{11.70}$$

This result can be used in Eq. (11.67) to calculate Δf_n, which is then used in Eq. (11.61) to find f_{n+1}.

To use the forward differences in an algorithm that iterates from $n = 0$ to $n\delta = 1$, we compute the initial conditions with Eqs. (11.62), (11.63), (11.65), and (11.68) for $t = 0$. They are

$$f_0 = d,\ \Delta f_0 = a\delta^3 + b\delta^2 + c\delta,\ \Delta^2 f_0 = 6a\delta^3 + 2b\delta^2,\ \Delta^3 f_0 = 6a\delta^3. \tag{11.71}$$

These initial condition calculations can be done by direct evaluation of the four equations. Note that, however, if we define the vector of initial differences as D, then

$$D = \begin{bmatrix} f_0 \\ \Delta f_0 \\ \Delta^2 f_0 \\ \Delta^3 f_0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ \delta^3 & \delta^2 & \delta & 0 \\ 6\delta^3 & 2\delta^2 & 0 & 0 \\ 6\delta^3 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}. \tag{11.72}$$

Rewriting, with the 4×4 matrix represented as $E(\delta)$, yields $D = E(\delta)C$. Because we are dealing with three functions, $x(t)$, $y(t)$, and $z(t)$, there are three corresponding sets of initial conditions, $D_x = E(\delta)C_x$, $D_y = E(\delta)C_y$, and $D_z = E(\delta)C_z$.

Based on this derivation, the algorithm for displaying a parametric cubic curve is given in Fig. 11.36. This procedure needs just nine additions and no multiplies per 3D point, and just a few adds and multiplies are used for initialization! This is considerably better than the 10 additions *and* nine multiplies needed for the simple brute-force iterative evaluation using Horner's rule. Notice, however, that error accumulation can be an issue with this algorithm, and sufficient fractional precision must be carried to avoid it. For instance, if $n = 64$ and integer arithmetic is used, then 16 bits of fractional precision are needed; if $n = 256$, 22 bits are needed [BART87].

Recursive subdivision is the second way to display a curve. Subdivision stops adaptively, when the curve segment in question is flat enough to be approximated by a line.

```
void DrawCurveFwdDif (
      int n,                 /* number of steps used to draw a curve */
      double x, double Δx, double Δ²x, double Δ³x,
      /* initial values for x(t) polynomial at t = 0, computed as D_x = E(δ)C_x. */
      double y, double Δy, double Δ²y, double Δ³y,
      /* initial values for y(t) polynomial at t = 0, computed as D_y = E(δ)C_y. */
      double z, double Δz, double Δ²z, double Δ³z
      /* initial values for z(t) polynomial at t = 0, computed as D_z = E(δ)C_z. */
      /* The step size δ used to calculate D_x, D_y, and D_z is 1/n . */
)
{
   int i;

   MoveAbs3 (x, y, z);        /* Go to start of curve */
   for (i = 0; i < n; i++) {
      x += Δx;   Δx += Δ²x;   Δ²x += Δ³x;
      y += Δy;   Δy += Δ²y;   Δ²y += Δ³y;
      z += Δz;   Δz += Δ²z;   Δ²z += Δ³z;
      LineAbs3 (x, y, z);     /* Draw a short line segment */
   }
}  /* DrawCurveFwdDif */
```

Fig. 11.36 Procedure to display a parametric curve using forward differences.

Details vary for each type of curve, because the subdivision process is slightly different for each curve, as is the flatness test. The general algorithm is given in Fig. 11.37.

The Bézier representation is particularly appropriate for recursive subdivision display. Subdivision is fast, requiring only six shifts and six adds in Eq. (11.51). The test for "straightness" of a Bézier curve segment is also simple, being based on the convex hull formed by the four control points; see Fig. 11.38. If the larger of distances d_2 and d_3 is less than some threshold ε, then the curve is approximated by a straight line drawn between the endpoints of the curve segment. For the Bézier form, the endpoints are P_1 and P_4, the first and last control points. Were some other form used, the flatness test would be more complicated and the endpoints might have to be calculated. Thus, conversion to Bézier form is appropriate for display by recursive subdivision.

More detail on recursive-subdivision display can be found in [BARS85; BARS87; LANE80]. If the subdivision is done with integer arithmetic, less fractional precision is

```
void DrawCurveRecSub (curve, ε)
{
   if (Straight (curve, ε))      /* Test control points to be within ε of a line */
      DrawLine (curve);
   else {
      SubdivideCurve (curve, leftCurve, rightCurve);
      DrawCurveRecSub (leftCurve, ε);
      DrawCurveRecSub (rightCurve, ε);
   }
}  /* DrawCurveRecSub */
```

Fig. 11.37 Procedure to display a curve by recursive subdivision. Straight is a procedure that returns **true** if the curve is sufficiently flat.

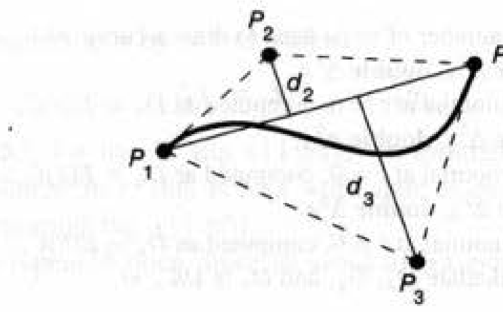


Fig. 11.38 Flatness test for a curve segment. If d_2 and d_3 are both less than some ε, the segment is declared to be flat and is approximated by the line segment P_1P_4.

needed than in the forward difference method. If at most eight levels of recursion are needed (a reasonable expectation), only 3 bits of fractional precision are necessary; if 16, 4 bits.

Recursive subdivision is attractive because it avoids unnecessary computation, whereas forward differencing uses a fixed subdivision. The forward-difference step size δ must be small enough that the portion of the curve with the smallest radius of curvature is approximated satisfactorily. For nearly straight portions of the curve, a much larger step size would be acceptable. [LANE80a] gives a method for calculating the step size δ to obtain a given maximum deviation from the true curve. On the other hand, recursive subdivision takes time to test for flatness. An alternative is to do recursive subdivision down to a fixed depth, avoiding the flatness test at the cost of some extra subdivision (see Exercise 11.30).

A hybrid approach, adaptive forward differencing [LIEN87; SHAN87; SHAN89], uses the best features of both the forward differencing and subdivision methods. The basic strategy is forward differencing, but an adaptive step size is used. Computationally efficient methods to double or halve the step size are used to keep it close to 1 pixel. This means that essentially no straight-line approximations are used between computed points.

11.2.10 Comparison of the Cubic Curves

The different types of parametric cubic curves can be compared by several different criteria, such as ease of interactive manipulation, degree of continuity at join points, generality, and speed of various computations using the representations. Of course, it is not necessary to choose a single representation, since it is possible to convert between all representations, as discussed in Section 11.2.8. For instance, nonuniform rational B-splines can be used as an internal representation, while the user might interactively manipulate Bézier control points or Hermite control points and tangent vectors. Some interactive graphics editors provide the user with Hermite curves while representing them internally in the Bézier form supported by PostScript [ADOB85a]. In general, the user of an interactive CAD system may be given several choices, such as Hermite, Bézier, uniform B-splines, and nonuniform B-splines. The nonuniform rational B-spline representation is likely to be used interally, because it is the most general.

Table 11.2 compares most of the curve forms mentioned in this section. Ease of interactive manipulation is not included explicitly in the table, because the latter is quite

TABLE 11.2 COMPARISON OF SEVEN DIFFERENT FORMS OF PARAMETRIC CUBIC CURVES

	Hermite	Bézier	Uniform B-spline	Uniformly shaped β-spline	Nonuniform B-spline	Catmull–Rom	Kochanek–Bartels
Convex hull defined by control points	N/A	Yes	Yes	Yes	Yes	No	No
Interpolates some control points	Yes	Yes	No	No	No	Yes	Yes
Interpolates all control points	Yes	No	No	No	No	Yes	Yes
Ease of subdivision	Good	Best	Avg	Avg	High	Avg	Avg
Continuities inherent in representation	C^0 G^0	C^0 G^0	C^2 G^2	C^0 G^2	C^2 G^2	C^1 G^1	C^1 G^1
Continuities easily achieved	C^1 G^1	C^1 G^1	C^2 G^{2*}	C^1 G^{2*}	C^2 G^{2*}	C^1 G^1	C^1 G^1
Number of parameters controlling a curve segment	4	4	4	6†	5	4	7

*Except for special case discussed in Section 11.2.
†Four of the parameters are local to each segment, two are global to the entire curve.

application specific. "Number of parameters controlling a curve segment" is the four geometrical constraints plus other parameters, such as knot spacing for nonuniform splines, β_1 and β_2 for β-splines, or a, b, or c for the Kochanek–Bartels case. "Continuity easily achieved" refers to constraints such as forcing control points to be collinear to allow G^1 continuity. Because C^n continuity is more restrictive than G^n, any form that can attain C^n can by definition also attain at least G^n.

When only geometric continuity is required, as is often the case for CAD, the choice is narrowed to the various types of splines, all of which can achieve both G^1 and G^2 continuity. Of the three types of splines in the table, uniform B-splines are the most limiting. The possibility of multiple knots afforded by nonuniform B-splines gives more shape control to the user, as does the use of the β_1 and β_2 shape parameters of the β-splines. Of course, a good user interface that allows the user to exploit this power easily is important.

To interpolate the digitized points of a camera path or shape contour, Catmull–Rom or Kochanek–Bartels splines are preferable. When a combination of interpolation and tangent vector control is desired, the Bézier or Hermite form is best.

It is customary to provide the user with the ability to drag control points or tangent vectors interactively, continually displaying the updated spline. Figure 11.23 shows such a sequence for B-splines. One of the disadvantages of B-splines in some applications is that the control points are not on the spline itself. It is possible, however, not to display the control points, allowing the user instead to interact with the knots (which must be marked so they can be selected). When the user selects a knot and moves it by some $(\Delta x, \Delta y)$, the control point weighted most heavily in determining the position of the join point is also moved by $(\Delta x, \Delta y)$. The join does not move the full $(\Delta x, \Delta y)$, because it is a weighted sum of several control points, only one of which was moved. Therefore, the cursor is repositioned on the join. This process is repeated in a loop until the user stops dragging.

11.3 PARAMETRIC BICUBIC SURFACES

Parametric bicubic surfaces are a generalization of parametric cubic curves. Recall the general form of the parametric cubic curve $Q(t) = T \cdot M \cdot G$, where G, the geometry vector, is a constant. First, for notational convenience, we replace t with s, giving $Q(s) = S \cdot M \cdot G$. If we now allow the points in G to vary in 3D along some path that is parameterized on t, we have

$$Q(s, t) = S \cdot M \cdot G(t) = S \cdot M \cdot \begin{bmatrix} G_1(t) \\ G_2(t) \\ G_3(t) \\ G_4(t) \end{bmatrix}. \tag{11.73}$$

Now, for a fixed t_1, $Q(s, t_1)$ is a curve because $G(t_1)$ is constant. Allowing t to take on some new value—say, t_2—where $t_2 - t_1$ is very small, $Q(s, t)$ is a slightly different curve. Repeating this for arbitrarily many other values of t_2 between 0 and 1, an entire family of curves is defined, each arbitrarily close to another curve. The set of all such curves defines a surface. If the $G_i(t)$ are themselves cubics, the surface is said to be a *parametric bicubic surface*.

Continuing with the case that the $G_i(t)$ are cubics, each can be represented as $G_i(t) = T \cdot M \cdot \mathbf{G}_i$, where $\mathbf{G}_i = [g_{i1} \quad g_{i2} \quad g_{i3} \quad g_{i4}]^T$ (the $\mathbf{G}$ and g are used to distinguish from the G used for the curve). Hence, g_{i1} is the first element of the geometry vector for curve $G_i(t)$, and so on.

Now let us transpose the equation $G_i(t) = T \cdot M \cdot \mathbf{G}_i$, using the identity $(A \cdot B \cdot C)^T = C^T \cdot B^T \cdot A^T$. The result is $G_i(t) = \mathbf{G}_i^T \cdot M^T \cdot T^T = [g_{i1} \quad g_{i2} \quad g_{i3} \quad g_{i4}] \cdot M^T \cdot T^T$. If we now substitute this result in Eq. (11.73) for each of the four points, we have

$$Q(s, t) = S \cdot M \cdot \begin{bmatrix} g_{11} & g_{12} & g_{13} & g_{14} \\ g_{21} & g_{22} & g_{23} & g_{24} \\ g_{31} & g_{32} & g_{33} & g_{34} \\ g_{41} & g_{42} & g_{43} & g_{44} \end{bmatrix} \cdot M^T \cdot T^T \tag{11.74}$$

or

$$Q(s, t) = S \cdot M \cdot \mathbf{G} \cdot M^T \cdot T^T, 0 \le s, t \le 1. \tag{11.75}$$

Written separately for each of x, y, and z, the form is

$$\begin{aligned} x(s, t) &= S \cdot M \cdot \mathbf{G}_x \cdot M^T \cdot T^T, \\ y(s, t) &= S \cdot M \cdot \mathbf{G}_y \cdot M^T \cdot T^T, \\ z(s, t) &= S \cdot M \cdot \mathbf{G}_z \cdot M^T \cdot T^T. \end{aligned} \tag{11.76}$$

Given this general form, we now move on to examine specific ways to specify surfaces using different geometry matrixes.

11.3.1 Hermite Surfaces

Hermite surfaces are completely defined by a 4×4 geometry matrix $\mathbf{G}_H$. Derivation of $\mathbf{G}_H$ follows the same approach used to find Eq. (11.75). We further elaborate the derivation here, applying it just to $x(s, t)$. First, we replace t by s in Eq. (11.13), to get $x(s) = S \cdot M_H \cdot G_{H_x}$. Rewriting this further so that the Hermite geometry vector G_{H_x} is not constant, but is rather a function of t, we obtain

$$x(s, t) = S \cdot M_H \cdot G_{H_x}(t) = S \cdot M_H \begin{bmatrix} P_1(t) \\ P_4(t) \\ R_1(t) \\ R_4(t) \end{bmatrix}_x . \tag{11.77}$$

The functions $P_{1_x}(t)$ and $P_{4_x}(t)$ define the x components of the starting and ending points for the curve in parameter s. Similarly, $R_{1_x}(t)$ and $R_{4_x}(t)$ are the tangent vectors at these points. For any specific value of t, there are two specific endpoints and tangent vectors. Figure 11.39 shows $P_1(t)$, $P_4(t)$, and the cubic in s that is defined when $t = 0.0, 0.2, 0.4, 0.6, 0.8$, and 1.0. The surface patch is essentially a cubic interpolation between $P_1(t) = Q(0, t)$ and $P_4(t) = Q(1, t)$ or, alternatively, between $Q(s, 0)$ and $Q(s, 1)$.

In the special case that the four interpolants $Q(0, t)$, $Q(1, t)$, $Q(s, 0)$, and $Q(s, 1)$ are straight lines, the result is a *ruled surface*. If the interpolants are also coplanar, then the surface is a four-sided planar polygon.

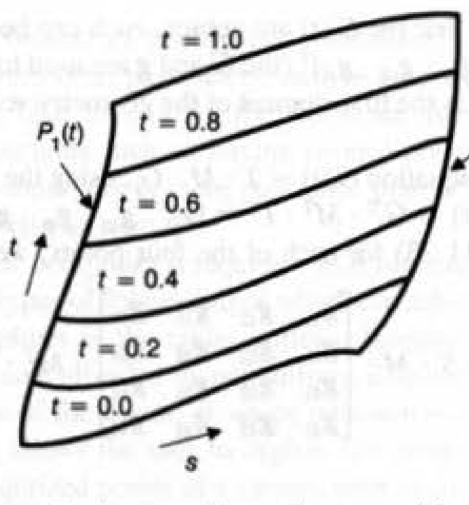


Fig. 11.39 Lines of constant parameter values on a bicubic surface: $P_1(t)$ is at $s = 0$, $P_4(t)$ is at $s = 1$.

Continuing with the derivation, let each of $P_1(t)$, $P_4(t)$, $R_1(t)$, and $R_4(t)$ be represented in Hermite form as

$$P_{1_x}(t) = T \cdot M_{\rm H} \begin{bmatrix} g_{11} \\ g_{12} \\ g_{13} \\ g_{14} \end{bmatrix}_x, \qquad P_{4_x}(t) = T \cdot M_{\rm H} \begin{bmatrix} g_{21} \\ g_{22} \\ g_{23} \\ g_{24} \end{bmatrix}_x, \tag{11.78}$$

$$R_{1_x}(t) = T \cdot M_{\rm H} \begin{bmatrix} g_{31} \\ g_{32} \\ g_{33} \\ g_{34} \end{bmatrix}_x, \qquad R_{4_x}(t) = T \cdot M_{\rm H} \begin{bmatrix} g_{41} \\ g_{42} \\ g_{43} \\ g_{44} \end{bmatrix}_x.$$

These four cubics can be rewritten together as a single equation:

$$[P_1(t) \quad P_4(t) \quad R_1(t) \quad R_4(t)]_x = T \cdot M_{\rm H} \cdot G_{{\rm H}_x}^{\rm T}, \tag{11.79}$$

where

$$G_{{\rm H}_x} = \begin{bmatrix} g_{11} & g_{12} & g_{13} & g_{14} \\ g_{21} & g_{22} & g_{23} & g_{24} \\ g_{31} & g_{32} & g_{33} & g_{34} \\ g_{41} & g_{42} & g_{43} & g_{44} \end{bmatrix}_x. \tag{11.80}$$

Transposing both sides of Eq. (11.79) results in

$$\begin{bmatrix} P_1(t) \\ P_4(t) \\ R_1(t) \\ R_4(t) \end{bmatrix}_x = \begin{bmatrix} g_{11} & g_{12} & g_{13} & g_{14} \\ g_{21} & g_{22} & g_{23} & g_{24} \\ g_{31} & g_{32} & g_{33} & g_{34} \\ g_{41} & g_{42} & g_{43} & g_{44} \end{bmatrix}_x M_{\rm H}^{\rm T} \cdot T^{\rm T} = G_{{\rm H}_x} \cdot M_{\rm H}^{\rm T} \cdot T^{\rm T}. \tag{11.81}$$

Substituting Eq. (11.81) into Eq. (11.77) yields

$$x(s, t) = S \cdot M_H \cdot G_{H_x} \cdot M_H^T \cdot T^T; \qquad (11.82)$$

similarly,

$$y(s, t) = S \cdot M_H \cdot G_{H_y} \cdot M_H^T \cdot T^T, \; z(s, t) = S \cdot M_H \cdot G_{H_z} \cdot M_H^T \cdot T^T. \qquad (11.83)$$

The three 4 × 4 matrixes G_{H_x}, G_{H_y}, and G_{H_z} play the same role for Hermite surfaces as did the single matrix G_H for curves. The meanings of the 16 elements of G_H can be understood by relating them back to Eqs. (11.77) and (11.78). The element g_{11_x} is $x(0, 0)$ because it is the starting point for $P_{1_x}(t)$, which is in turn the starting point for $x(s, 0)$. Similarly, g_{12_x} is $x(0, 1)$ because it is the ending point of $P_{1_x}(t)$, which is in turn the starting point for $x(s, 1)$. Furthermore, g_{13_x} is $\partial x/\partial t(0, 0)$ because it is the starting tangent vector for $P_{1_x}(t)$, and g_{33_x} is $\partial^2 x/\partial s \partial t(0, 0)$ because it is the starting tangent vector of $R_{1_x}(t)$, which in turn is the starting slope of $x(s, 0)$.

Using these interpretations, we can write G_{H_x} as

$$G_{H_x} = \begin{bmatrix} x(0, 0) & x(0, 1) & \frac{\partial}{\partial t}x(0, 0) & \frac{\partial}{\partial t}x(0, 1) \\ x(1, 0) & x(1, 1) & \frac{\partial}{\partial t}x(1, 0) & \frac{\partial}{\partial t}x(1, 1) \\ \frac{\partial}{\partial s}x(0, 0) & \frac{\partial}{\partial s}x(0, 1) & \frac{\partial^2}{\partial s \partial t}x(0, 0) & \frac{\partial^2}{\partial s \partial t}x(0, 1) \\ \frac{\partial}{\partial s}x(1, 0) & \frac{\partial}{\partial s}x(1, 1) & \frac{\partial^2}{\partial s \partial t}x(1, 0) & \frac{\partial^2}{\partial s \partial t}x(1, 1) \end{bmatrix}. \qquad (11.84)$$

The upper-left 2 × 2 portion of G_{H_x} contains the x coordinates of the four corners of the patch. The upper-right and lower-left 2 × 2 areas give the x coordinates of the tangent vectors along each parametric direction of the patch. The lower-right 2 × 2 portion has at its corners the partial derivatives with respect to both s and t. These partials are often called the *twists*, because the greater they are, the greater the corkscrewlike twist at the corners. Figure 11.40 shows a patch whose corners are labeled to indicate these parameters.

This Hermite form of bicubic patches is an alternative way to express a restricted form of the *Coons patch* [COON67]. These more general patches permit boundary curves and slopes to be any curves. (The Coons patch was developed by the late Steven A. Coons [HERZ80], an early pioneer in CAD and computer graphics after whom SIGGRAPH's prestigious *Steven A. Coons Award for Outstanding Contributions to Computer Graphics* is named.) When the four twist vectors in the Hermite form are all zero, the patches are also called Ferguson surfaces after another early developer of surface representations [FERG64; FAUX79].

Just as the Hermite cubic permits C^1 and G^1 continuity from one curve segment to the next, so too the Hermite bicubic permits C^1 and G^1 continuity from one patch to the next. First, to have C^0 continuity at an edge, the matching curves of the two patches must be identical, which means the control points for the two surfaces must be identical along the edge. The necessary conditions for C^1 continuity are that the control points along the edge and the tangent and twist vectors across the edge be equal. For G^1 continuity, the tangent

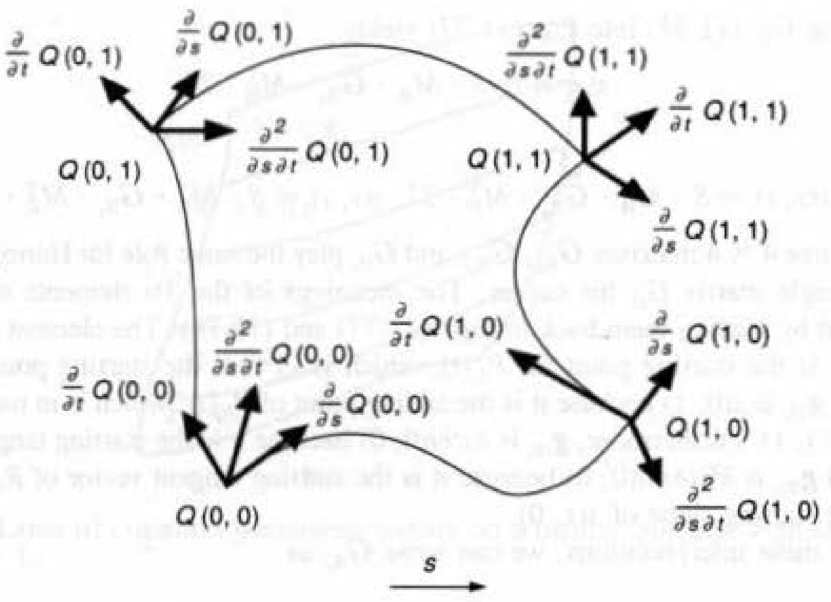

$\frac{\partial}{\partial t} Q(0,1)$
$\frac{\partial}{\partial s} Q(0,1)$
$\frac{\partial^2}{\partial s \partial t} Q(1,1)$
$\frac{\partial}{\partial t} Q(1,1)$
$\frac{\partial^2}{\partial s \partial t} Q(0,1)$
$Q(0,1)$
$Q(1,1)$
$\frac{\partial}{\partial s} Q(1,1)$
t
$\frac{\partial}{\partial t} Q(1,0)$
$\frac{\partial}{\partial s} Q(1,0)$
$\frac{\partial^2}{\partial s \partial t} Q(0,0)$
$\frac{\partial}{\partial t} Q(0,0)$
$\frac{\partial}{\partial s} Q(0,0)$
$Q(1,0)$
$Q(0,0)$
$\frac{\partial^2}{\partial s \partial t} Q(1,0)$
s

Fig. 11.40 Components of the geometry matrix for a Hermite surface. Each vector is a 3-tuple, the x component of which is given by Eq. (11.84).

vector requirement is relaxed so that the vectors must be in the same direction, but do not need to have the same magnitude. If the common edge for patch 1 is at $s = 1$ and that for patch 2 is at $s = 0$, as in Fig. 11.41, then the values in some rows of the geometry matrices for the two patches must reflect the G^1 conditions, as indicated here:

$$\begin{array}{cc} \text{Patch 1} & \text{Patch 2} \\ \begin{bmatrix} - & - & - & - \\ g_{21} & g_{22} & g_{23} & g_{24} \\ - & - & - & - \\ g_{41} & g_{42} & g_{43} & g_{44} \end{bmatrix} & \begin{bmatrix} g_{21} & g_{22} & g_{23} & g_{24} \\ - & - & - & - \\ kg_{41} & kg_{42} & kg_{43} & kg_{44} \\ - & - & - & - \end{bmatrix} \end{array} \quad k > 0. \tag{11.85}$$

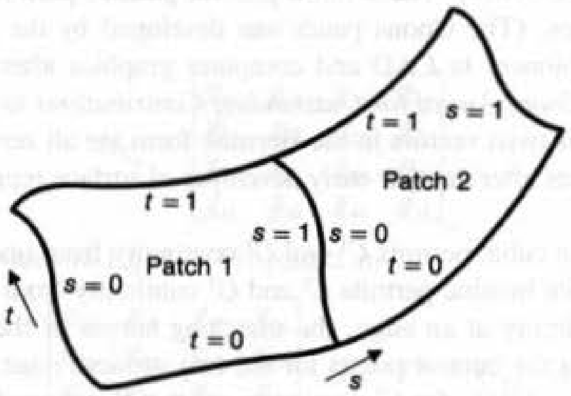


Fig. 11.41 Two joined surface patches.

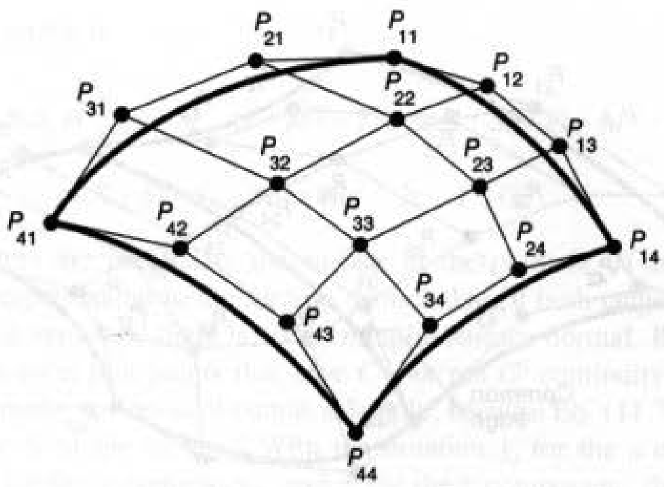


Fig. 11.42 Sixteen control points for a Bézier bicubic patch.

Entries marked with a dash can have any value. If four patches joined at a common corner and along the edges emanating from that corner are to have G^1 continuity, then the relationships are more complex; see Exercise 11.25.

11.3.2 Bézier Surfaces

The Bézier bicubic formulation can be derived in exactly the same way as the Hermite cubic. The results are

$$x(s, t) = S \cdot M_B \cdot G_{B_x} \cdot M_B^T \cdot T^T,$$
$$y(s, t) = S \cdot M_B \cdot G_{B_y} \cdot M_B^T \cdot T^T, \qquad (11.86)$$
$$z(s, t) = S \cdot M_B \cdot G_{B_z} \cdot M_B^T \cdot T^T.$$

The Bézier geometry matrix G consists of 16 control points, as shown in Fig. 11.42. Bézier surfaces are attractive in interactive design for the same reason as Bézier curves are: Some of the control points interpolate the surface, giving convenient precise control, whereas tangent vectors also can be controlled explicitly. When Bézier surfaces are used as an internal representation, their convex-hull property and easy subdivision are attractive.

C^0 and G^0 continuity across patch edges is created by making the four common control points equal. G^1 continuity occurs when the two sets of four control points on either side of the edge are collinear with the points on the edge. In Fig. 11.43, the following sets of control points are collinear and define four line segments whose lengths all have the same ratio k: (P_{13}, P_{14}, P_{15}), (P_{23}, P_{24}, P_{25}), (P_{33}, P_{34}, P_{35}), and (P_{43}, P_{44}, P_{45}).

An alternative way to maintain interpatch continuity, described in more detail in [FAUX79; BEZI70], requires that, at each corner of an edge across which continuity is desired, the corner control points and the control points immediately adjacent to the corner be coplanar.

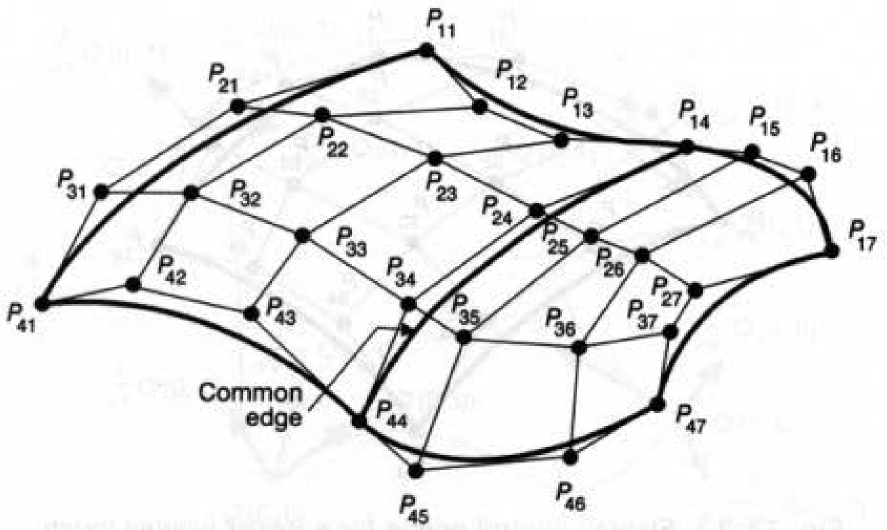


Fig. 11.43 Two Bézier patches joined along the edge P_{14}, P_{24}, P_{34}, and P_{44}.

11.3.3 B-Spline Surfaces

B-spline patches are represented as

$$\begin{aligned} x(s, t) &= S \cdot M_{Bs} \cdot G_{Bs_x} \cdot M_{Bs}{}^{T} \cdot T^{T}, \\ y(s, t) &= S \cdot M_{Bs} \cdot G_{Bs_y} \cdot M_{Bs}{}^{T} \cdot T^{T}, \\ z(s, t) &= S \cdot M_{Bs} \cdot G_{Bs_z} \cdot M_{Bs}{}^{T} \cdot T^{T}. \end{aligned} \tag{11.87}$$

C^2 continuity across boundaries is automatic with B-splines; no special arrangements of control points are needed except to avoid duplicate control points, which create discontinuities.

Bicubic nonuniform and rational B-spline surfaces and other rational surfaces are similarly analogous to their cubic counterparts. All the techniques for subdivision and display carry over directly to the bicubic case.

11.3.4 Normals to Surfaces

The normal to a bicubic surface, needed for shading (Chapter 16), for performing interference detection in robotics, for calculating offsets for numerically controlled machining, and for doing other calculations, is easy to find. From Eq. (11.75), the s tangent vector of the surface $Q(s, t)$ is

$$\begin{aligned} \frac{\partial}{\partial s} Q(s, t) &= \frac{\partial}{\partial s} (S \cdot M \cdot G \cdot M^{T} \cdot T^{T}) = \frac{\partial}{\partial s} (S) \cdot M \cdot G \cdot M^{T} \cdot T^{T} \\ &= [3s^2 \quad 2s \quad 1 \quad 0] \cdot M \cdot G \cdot M^{T} \cdot T^{T}, \end{aligned} \tag{11.88}$$

and the t tangent vector is

$$\frac{\partial}{\partial t}Q(s, t) = \frac{\partial}{\partial t}(S \cdot M \cdot G \cdot M^{\mathrm{T}} \cdot T^{\mathrm{T}}) = S \cdot M \cdot G \cdot M^{\mathrm{T}} \cdot \frac{\partial}{\partial t}(T^{\mathrm{T}})$$
$$= S \cdot M \cdot G \cdot M^{\mathrm{T}} \cdot [3t^2 \quad 2t \quad 1 \quad 0]^{\mathrm{T}}. \qquad (11.89)$$

Both tangent vectors are parallel to the surface at the point (s, t) and, therefore, their cross-product is perpendicular to the surface. Notice that, if both tangent vectors are zero, the cross-product is zero, and there is no meaningful surface normal. Recall that a tangent vector can go to zero at join points that have C^1 but not G^1 continuity.

Each of the tangent vectors is of course a 3-tuple, because Eq. (11.75) represents the x, y, and z components of the bicubic. With the notation x_s for the x component of the s tangent vector, y_s for the y component, and z_s for the z component, the normal is

$$\frac{\partial}{\partial s}Q(s, t) \times \frac{\partial}{\partial t}Q(s, t) = [y_s z_t - y_t z_s \quad z_s x_t - z_t x_s \quad x_s y_t - x_t y_s]. \qquad (11.90)$$

The surface normal is a biquintic (two-variable, fifth-degree) polynomial and hence is fairly expensive to compute. [SCHW82] gives a bicubic approximation that is satisfactory as long as the patch itself is relatively smooth.

11.3.5 Displaying Bicubic Surfaces

Like curves, surfaces can be displayed either by iterative evaluation of the bicubic polynomials or by subdivision, which is essentially an adaptive evaluation of the bicubic polynomials. We consider first iterative evaluation, and then subdivision.

Iterative evaluation is best suited for displaying bicubic patches in the style of Fig. 11.44. Each of the curves of constant s and constant t on the surface is itself a cubic, so display of each of the curves is straightforward, as in Fig. 11.45.

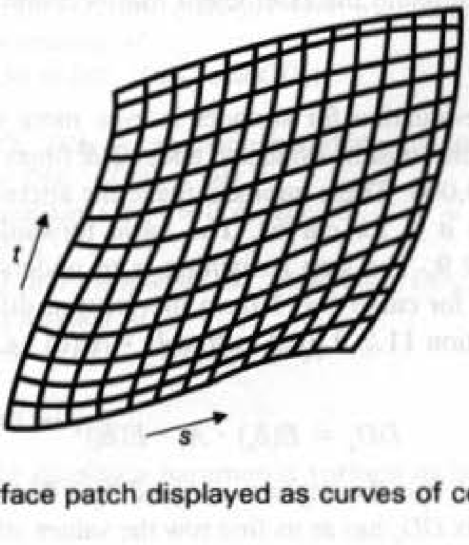


Fig. 11.44 A single surface patch displayed as curves of constant s and constant t.

```
typedef double Coeffs[4][4][3];
void DrawSurface (
      Coeffs coefficients,        /* Coefficients for Q(s,t) */
      int n_s,                    /* No. of curves of constant s to draw, typically 5–10 */
      int n_t,                    /* No. of curves of constant t to draw, typically 5–10 */
      int n)                      /* No. of steps used to draw each curve, typically 20–100 */
{
   double δ = 1.0 / n;            /* Step size to use in drawing each curve */
   double δ_s = 1.0 / (n_s−1);    /* Step size in s to increment to next curve of constant t */
   double δ_t = 1.0 / (n_t−1);    /* Step size in t to increment to next curve of constant s */
   int i, j; double s, t;
   /* Draw n_s curves of constant s, for s = 0, δ_s, 2δ_s, ... 1 */
   for (i = 0, s = 0.0; i < n_s; i++, s += δ_s) {
      /* Draw a curve of constant s, varying t from 0 to 1. */
      /* X, Y, and Z are functions to evaluate the bicubics. */
      MoveAbs3 (X (s, 0.0), Y (s, 0.0), Z (s, 0.0));
      for (j = 1, t = δ; j < n; j++, t += δ) {
         /* n−1 steps are used as t varies from δ to 1 for each curve. */
         LineAbs3 (X (s, t), Y (s, t), Z (s, t));
      }
   }
   /* Draw n_t curves of constant t, for t = 0, δ_t, 2δ_t, ... 1 */
   for (i = 0, t = 0.0; i < n_t; i++, t += δ_t) {
      /* Draw a curve of constant t, varying s from 0 to 1. */
      MoveAbs3 (X (0.0, t), Y (0.0, t), Z (0.0, t));
      for (j = 1, s = δ; j < n; j++, s += δ) {
         /* n−1 steps are used as s varies from δ to 1 for each curve. */
         LineAbs3 (X (s, t), Y (s, t), Z (s, t));
      }
   }
} /* DrawSurface */
```

Fig. 11.45 Procedure to display bicubic patch as a grid. Procedures X(*s*, *t*), Y(*s*, *t*), and Z(*s*, *t*) evaluate the surface using the coefficient matrix *coefficients*.

Brute-force iterative evaluation for surfaces is even more expensive than for curves, because the surface equations must be evaluated about $2/\delta^2$ times. For $\delta = 0.1$, this value is 200; for $\delta = 0.01$, it is 20,000. These numbers make the alternative, forward differencing even more attractive than it is for curves. The basic forward-differencing method was developed in Section 11.2.9. The step remaining is to understand how to calculate the initial forward differences for curve $i + 1$ from the forward differences for curve i.

The derivation in Section 11.2.9 used to find $D = E(\delta) \cdot C$ for curves can be used to find

$$DD_x = E(\delta_s) \cdot A_x \cdot E(\delta_t)^{\mathrm{T}} \tag{11.91}$$

where δ_s is the step size in s, δ_t is the step size in t, and A_x is the 4×4 matrix of coefficients for $x(s, t)$. The 4×4 matrix DD_x has as its first row the values $x(0, 0)$, $\Delta_t x(0, 0)$, $\Delta_t^2 x(0, 0)$, and $\Delta_t^3 x(0, 0)$ (the notation Δ_t means forward difference on t, as opposed to s). Thus, the first row can be used to calculate $x(0, t)$ in increments of δ_t.

After $x(0, t)$ has been computed, how can $x(\delta_s, t)$ be computed, to draw another curve of constant s? (This is the step of calculating the initial forward differences for curve $i + 1$ from the forward differences for curve i.) The other rows of DD_x are the first, second, and third forward differences on s of the first row's forward differences. Therefore, applying the following equations to the rows of DD_x

$$\begin{aligned} \text{row } 1 &:= \text{row } 1 + \text{row } 2 \\ \text{row } 2 &:= \text{row } 2 + \text{row } 3 \\ \text{row } 3 &:= \text{row } 3 + \text{row } 4 \end{aligned} \tag{11.92}$$

```
typedef double Coeffs[4][4][3];

void DrawSurfaceFwdDif (
        Coeffs A,   /* Coefficients for $Q(s, t)$ */
        int $n_s$,      /* Number of curves of constant $s$ to be drawn, typically 5 to 10. */
        int $n_t$,      /* Number of curves of constant $t$ to be drawn, typically 5 to 10. */
        int n)      /* Number of steps to use in drawing each curve, typically 20 to 100. */
{
    /* Initialize */
    double $\delta_s = 1.0 / (n_s - 1.0)$; double $\delta_t = 1.0 / (n_t - 1.0)$;
    /* "*" indicates matrix multiplication */
    $DD_x = E(\delta_s) * A_x * E(\delta_t)^T$;
    $DD_y = E(\delta_s) * A_y * E(\delta_t)^T$;
    $DD_z = E(\delta_s) * A_z * E(\delta_t)^T$;

    /* Draw $n_s$ curves of constant s, for $s = 0, \delta_s, 2\delta_s, \ldots 1$ */
    for ($i = 0; i < n_s; i$++) {
        /* Procedure from Section 11.2.9 to draw a curve */
        DrawCurveFwdDif (n, First row of $DD_x$, First row of $DD_y$, First row of $DD_z$);
        /* Prepare for next iteration */
        Apply equation 11.92 to $DD_x$, $DD_y$, and $DD_z$;
    }
    /* Transpose $DD_x$, $DD_y$, $DD_z$ so can continue working with rows */
    $DD_x = {DD_x}^T$; $DD_y = {DD_y}^T$; $DD_z = {DD_z}^T$;
    /* Draw $n_t$ curves of constant t, for $t = 0, \delta_t, 2\delta_t, 3\delta_t, \ldots 1$ */
    for ($i = 0; i < n_t; i$++) {
        DrawCurveFwdDif (n, First row of $DD_x$, First row of $DD_y$, First row of $DD_z$);
        /* Prepare for next iteration */
        Apply equation 11.92 to $DD_x$, $DD_y$, and $DD_z$;
    }
}   /* DrawSurfaceFwdDif */
```

Fig. 11.46 Procedure to display a parametric surface as lines of constant s and t.

```
void DrawSurfaceRecSub (surface, $\epsilon$)
{
    /* Test whether control points are within $\epsilon$ of a plane */
    if (Flat (surface, $\epsilon$))
        DrawQuadrilateral (surface);      /* If within $\epsilon$, draw the surface as a quadrilateral */

    else {                                /* If not flat enough, subdivide into four patches */
        SubdivideSurface (surface, &surfaceLL, &surfaceLR, &surfaceRL, &surfaceRR);
        DrawSurfaceRecSub (surfaceLL, $\epsilon$);
        DrawSurfaceRecSub (surfaceLR, $\epsilon$);
        DrawSurfaceRecSub (surfaceRL, $\epsilon$);
        DrawSurfaceRecSub (surfaceRR, $\epsilon$);
    }
} /* DrawSurfaceRecSub */
```

Fig. 11.47 Procedure to display a parametric surface as a shaded surface. DrawQuadrilateral renders individual, nearly flat quadrilaterals. Flat returns **true** if the surface is within ε of being flat.

yields in the first row of DD_z the terms $x(\delta_s, 0)$, $\Delta_t x(\delta_s, 0)$, $\Delta_t^2 x(\delta_s, 0)$, and $\Delta_t^3 x(\delta_s, 0)$. These quantities are now used to compute $x(\delta_s, t)$, using forward differences, as before. Then the steps of Eq. (11.92) are again repeated, and the first row of D is used to compute $x(2\delta_s, t)$, and so on.

Drawing the curves of constant t is done similarly. To calculate $x(s, 0)$, we apply Eq. (11.92) to DD_z but substitute *column* for *row*. Alternatively, DD_z^{T} can be used in place of DD_z, so that rows are still used to calculate $x(s, 0)$, $x(s, \delta_t)$, $x(s, 2\delta_t)$, and so on.

The algorithm for displaying a surface as lines of constant s and t using forward differences is shown in Fig. 11.46.

Surface display using recursive subdivision is also a simple extension from the procedure DrawCurveRecSub of Section 11.2.9, and is presented in Fig. 11.47. Nearly planar quadrilaterals are created, then are displayed by procedure DrawQuadrilateral as shaded flat quadrilaterals using the methods described in Chapters 15 and 16. The process is again simplest if the Bézier form is used.

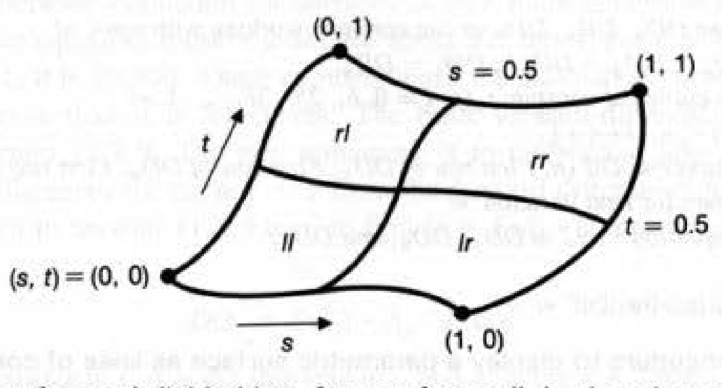


Fig. 11.48 A surface subdivided into four surfaces, *ll*, *lr*, *rl*, and *rr*, each of which is 'flatter' than the original surface.

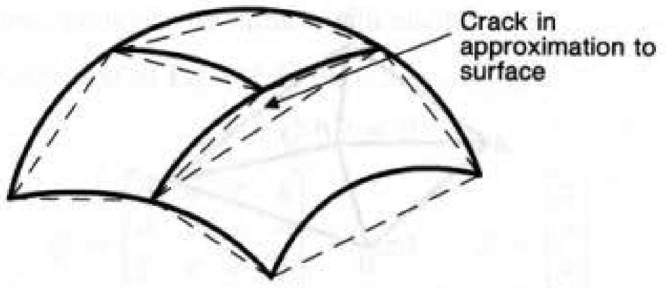


Fig. 11.49 A subdivided surface whose three approximating quadrilaterals introduce a crack into the approximation.

Flatness is tested by computing the plane passing through three of the four corner control points and finding the distance from each of the other 13 control points to the plane; the maximum distance must be less than ε. This is just a generalization of the flatness test used for curves. [LANE79] discusses a different way to perform a more efficient flatness test. Of course, the recursion can again be done to a constant depth, eliminating the flatness test at the price of needless subdivisions.

Surface subdivision is done by splitting the surface along one parameter, say *s*, and then splitting each of the two resulting surfaces along *t*. The curve-splitting methods discussed in Section 11.2.7 are applied to each set of four control points running in the direction of the parameter along which the split is being made. Figure 11.48 shows the idea, with the resulting four surfaces labeled *ll*, *lr*, *rl*, and *rr* as an extension of the notation used in Fig. 11.35. Alternatively, the patch can be split along only one parameter, if the patch is locally flat along the other parameter. [LANE80a] gives a thorough theoretical account of the subdivision process and also describes how it can be used to find curve–curve and surface–surface intersections.

One critical problem with adaptive subdivision is the possibility of cracks between approximating quadrilaterals, as seen in Fig. 11.49. The cracks occur because of the different levels of subdivision applied to adjoining patches. They can be avoided by subdividing to a fixed depth or by making the flatness threshold ε very small. Both solutions, however, cause needless subdivisions. Another way to avoid cracks is to modify adjoining approximating quadrilaterals, as suggested in Fig. 11.50. This basic strategy is

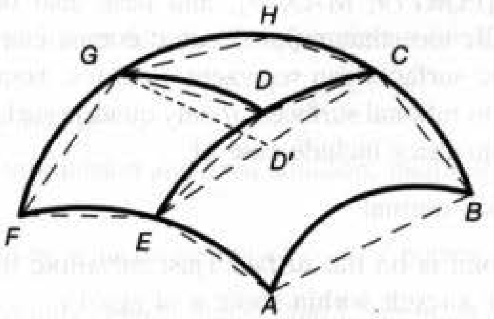


Fig. 11.50 Crack elimination in recursive subdivision. A naive algorithm displays quadrilaterals *ABCE, EFGD,* and *GDCH*. A more sophisticated algorithm displays quadrilaterals *ABCE, EFGD', and GD'CH*. Vertex *D'* is used instead of *D*.

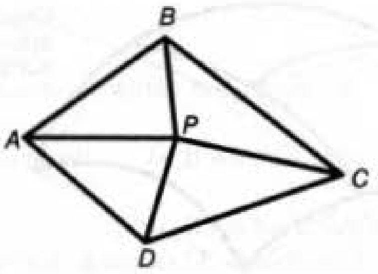


Fig. 11.51 A quadrilateral *ABCD* to be displayed is first subdivided into four triangles. The new point *P* is the average of points *A, B, C,* and *D*.

used by Clark [CLAR79] and by Barsky, DeRose, and Dippé [BARS87].

Procedure DrawQuadrilateral is given a nearly flat quadrilateral to display. The best way to display the quadrilateral as a shaded surface is to subdivide it further into four triangles, as shown in Fig. 11.51. This strategy avoids visual artifacts.

It is sometimes desirable to display only a portion of a bicubic surface. For instance, there may be a hole in a patch caused by a pipe going through the surface. Displaying only a portion can be done with *trimming curves*, which are just spline curves defined in (s, t) parameter space instead of in (x, y, z) space on the bicubic surface. When displaying a surface that is constrained by a trim curve using DrawSurfaceFwdDif (see Fig. 11.46), the (s, t) values of the trim curve are used to start and stop the iteration.

Other useful ways to display bicubic surfaces are presented in [FORR79].

11.4 QUADRIC SURFACES

The implicit surface equation of the form

$$f(x, y, z) = ax^2 + by^2 + cz^2 + 2dxy + 2eyz + 2fxz + 2gx + 2hy + 2jz + k = 0 \tag{11.93}$$

defines the family of quadric surfaces. For example, if $a = b = c = -k = 1$ and the remaining coefficients are zero, a unit sphere is defined at the origin. If a through f are zero, a plane is defined. Quadric surfaces are particularly useful in specialized applications such as molecular modeling [PORT79; MAX79], and have also been integrated into solid-modeling systems. Recall, too, that rational cubic curves can represent conic sections; similarly, rational bicubic surfaces can represent quadrics. Hence, the implicit quadratic equation is an alternative to rational surfaces, *if* only quadric surfaces are being represented. Other reasons for using quadrics include ease of

- Computing the surface normal
- Testing whether a point is on the surface (just substitute the point into Eq. (11.93), evaluate, and test for a result within some ε of zero)
- Computing z given x and y (important in hidden-surface algorithms—see Chapter 15) and

- Calculating intersections of one surface with another.

An alternative representation of Eq. (11.93) is:

$$P^{\mathrm{T}} \cdot Q \cdot P = 0, \tag{11.94}$$

$$\text{with} \quad Q = \begin{bmatrix} a & d & f & g \\ d & b & e & h \\ f & e & c & j \\ g & h & j & k \end{bmatrix} \quad \text{and} \quad P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{11.95}$$

The surface represented by Q can be easily translated and scaled. Given a 4×4 transformation matrix M of the form developed in Chapter 5, the transformed quadric surface Q' is given by

$$Q' = (M^{-1})^{\mathrm{T}} \cdot Q \cdot M^{-1}. \tag{11.96}$$

The normal to the implicit surface defined by $f(x, y, z) = 0$ is the vector $[df/dx \quad df/dy \quad df/dz]$. This surface normal is much easier to evaluate than is the surface normal to a bicubic surface discussed in Section 11.3.4.

11.5 SUMMARY

This chapter has only touched on some of the important ideas concerning curve and surface representation, but it has given sufficient information so that you can implement interactive systems using these representations. Theoretical treatments of the material can be found in texts such as [BART87; DEBO78; FARI88; FAUX79; MORT85].

Polygon meshes, which are piecewise linear, are well suited for representing flat-faced objects but are seldom satisfactory for curve-faced objects. Piecewise continuous parametric cubic curves and bicubic surfaces are widely used in computer graphics and CAD to represent curve-faced objects because they

- Permit multiple values for a single value of x or y
- Represent infinite slopes
- Provide local control, such that changing a control point affects only a local piece of the curve
- Can be made either to interpolate or to approximate control points, depending on application requirements
- Are computationally efficient
- Permit refinement by subdivision and knot addition, facilitating display and interactive manipulation
- Are easily transformed by transformation of control points.

Although we have discussed only cubics, higher- and lower-order surfaces can also be used. The texts mentioned previously generally develop parametric curves and surfaces for the general case of order n.

EXERCISES

11.1 Develop the equations, similar to Eq. (11.2), for the coefficients A and B of the plane equation. Assume that the polygon vertices are enumerated counterclockwise as viewed toward the plane from the positive side of the plane. The surface normal—given by A, B, and C—points toward the positive side of the plane (which accounts for the need to negate the area computed for B, as discussed in Section 11.1.3).

11.2 Write a program to calculate the plane equation coefficients, given n vertices of a polygon that is approximately planar. The vertices are enumerated in a counterclockwise direction, as defined in Exercise 11.1. Test the program with $n = 3$ for several known planes; then test it for larger n.

11.3 Find the geometry matrix and basis matrix for the parametric representation of a straight line given in Eq. (11.11).

11.4 Implement the procedure DrawCurve given in Fig 11.18. Display a number of curves, varying the coefficients cx, cy, and cz. Try to make the curve correspond to some of the curve segments shown in figures in this chapter. Why it this difficult to do?

11.5 Show that, for a 2D curve $[x(t) \quad y(t)]$, G^1 continuity means that the geometric slope dy/dx is equal at the join points between segments.

11.6 Let $\gamma(t) = (t, t^2)$ for $0 \le t \le 1$, and let $\eta(t) = (2t + 1, t^3 + 4t + 1)$ for $0 \le t \le 1$. Notice that $\gamma(1) = (1, 1) = \eta(0)$, so γ and η join with C^0 continuity.

a. Plot $\eta(t)$ and $\gamma(t)$ for $0 \le t \le 1$.

b. Do $\eta(t)$ and $\gamma(t)$ meet with C^1 continuity at the join point? (You will need to compute the vectors $\frac{d\gamma}{dt}(1)$ and $\frac{d\eta}{dt}(0)$ to check this.)

c. Do $\eta(t)$ and $\gamma(t)$ meet with G^1 continuity at the join point? (You will need to check ratios from part (b) to determine this).

11.7 Consider the paths

$$\gamma(t) = (t^2 - 2t + 1,\ t^3 - 2t^2 + t) \quad \text{and} \quad \eta(t) = (t^2 + 1,\ t^3),$$

both defined on the interval $0 \le t \le 1$. The curves join, since $\gamma(1) = (1, 0) = \eta(0)$. Show that they meet with C^1 continuity, but not with G^1 continuity. Plot both curves as functions of t to demonstrate exactly why this happens.

11.8 Show that the two curves $\gamma(t) = (t^2 - 2t, t)$ and $\eta(t) = (t^2 + 1, t + 1)$ are both C^1 and G^1 continuous where they join at $\gamma(1) = \eta(0)$.

11.9 Analyze the effect on a B-spline of having in sequence four collinear control points.

11.10 Write a program to accept an arbitrary geometry matrix, basis matrix, and list of control points, and to draw the corresponding curve.

11.11 Find the conditions under which two joined Hermite curves have C^1 continuity.

11.12 Suppose the equations relating the Hermite geometry to the Bézier geometry were of the form $R_1 = \beta(P_2 - P_1)$, $R_4 = \beta(P_4 - P_3)$. Consider the four equally spaced Bézier control points $P_1 = (0, 0)$, $P_2 = (1, 0)$, $P_3 = (2, 0)$, $P_4 = (3, 0)$. Show that, for the parametric curve $Q(t)$ to have constant velocity from P_1 to P_4, the coefficient β must be equal to 3.

11.13 Write an interactive program that allows the user to create and refine piecewise continuous cubic curves. Represent the curves internally as B-splines. Allow the user to specify how the curve is to be interactively manipulated—as Hermite, Bézier, or B-splines.

11.14 Show that duplicate interior control points on a B-spline do not affect the C^2 continuity at the join point. Do this by writing out the equations for the two curve segments formed by the control points $P_{i-1}, P_i, P_{i+1}, P_{i+2} = P_{i+1}, P_{i+3}$. Evaluate the second derivative of the first segment at $t = 1$, and that of the second segment at $t = 0$. They should be the same.

11.15 Find the blending functions for the Catmull–Rom splines of Eq. (11.47). Do they sum to 1, and are they everyone nonzero? If not, the spline is not contained in the convex hull of the points.

11.16 Using Eqs. (11.49), (11.50), and (11.19), find the basis matrix M_{KB} for the Kochanek-Bartels spline, using the geometry matrix G_{Bs_i} of Eq. (11.32).

11.17 Write an interactive program that allows the user to create, to manipulate interactively, and to refine piecewise continuous β-spline curves. Experiment with the effect of varying β_1 and β_2.

11.18 Write an interactive program that allows the user to create, to manipulate interactively, and refine piecewise continuous Kochanek–Bartels curves. Experiment with the effect of varying a, b, and c.

11.19 Implement both the forward-difference and recursive-subdivision curve-display procedures. Compare execution times for displaying various curves such that the curves are equally smooth to the eye.

11.20 Why is Eq. (11.36) for uniform B-splines written as $Q_i(t - t_i)$, whereas Eq. (11.43) for nonuniform B-splines is written as $Q_i(t)$?

11.21 Given a 2D nonuniform B-spline and an (x, y) value on the curve, write a program to find the corresponding value of t. Be sure to consider the possibility that, for a given value of x (or y), there may be multiple values of y (or x).

11.22 Given a value t at which a Bézier curve is to be split, use the de Casteljau construction shown in Fig. 11.35 to find the division matrices $D_B^L(t)$ and $D_B^R(t)$.

11.23 Apply the methodology used to derive Eq. (11.82) for Hermite surfaces to derive Eq. (11.86) for Bézier surfaces.

11.24 Write programs to display parametric cubic curves and surfaces using forward differences and recursive subdivision. Vary the step size δ and error measure ε to determine the effects of these parameters on the appearance of the curve.

11.25 Given four Hermite patches joined at a common corner and along the edges emanating from that corner, show the four geometry matrices and the relations that must hold between elements of the matrices.

11.26 Let $t_0 = 0, t_1 = 1, t_2 = 3, t_3 = 4, t_4 = 5$. Using these values, compute $B_{0,4}$ and each of the functions used in its definition. Then plot these functions on the interval $-3 \le t \le 8$.

11.27 Expand the recurrence relation of Eq. (11.44) into an explicit expression for $B_{i,4}(t)$. Use Fig. 11.26 as a guide.

11.28 Write a program that displays a nonuniform, non-rational B-spline, given as input a knot sequence and control points. Provide a user-controllable option to calculate the $B_{i,4}(t)$ in two ways: (a) using the recurrence relations of Eq. (11.44), and (b) using the explicit expression found in Exercise 11.27. Measure how much time is taken by each method. Is the faster method necessarily the better one?

11.29 Expand the program from Exercise 11.28 to allow interactive input and modification of the B-splines.

11.30 Write procedures to subdivide a curve recursively in two ways: adaptively, with a flatness test, and fixed, with a uniform level of subdivision. Draw a curve first with the adaptive subdivision, noting the deepest level of subdivision needed. Now draw the same curve with fixed subdivision as deep as that needed for the adaptive case. Compare the execution time of the two procedures for a variety of curves.

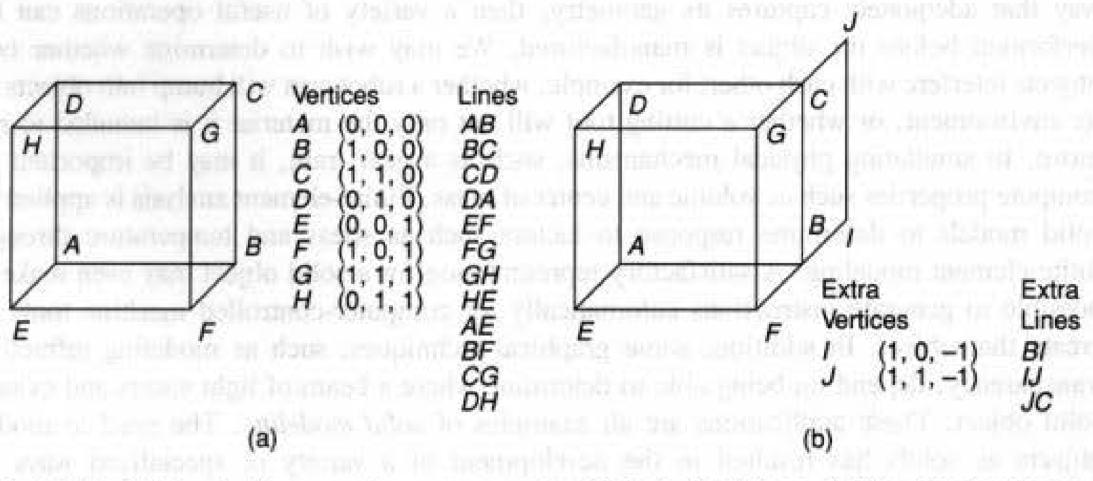
12 Solid Modeling

The representations discussed in Chapter 11 allow us to describe curves and surfaces in 2D and 3D. Just as a set of 2D lines and curves does not need to describe the boundary of a closed area, a collection of 3D planes and surfaces does not necessarily bound a closed volume. In many applications, however, it is important to distinguish between the inside, outside, and surface of a 3D object and to be able to compute properties of the object that depend on this distinction. In CAD/CAM, for example, if a solid object can be modeled in a way that adequately captures its geometry, then a variety of useful operations can be performed before the object is manufactured. We may wish to determine whether two objects interfere with each other; for example, whether a robot arm will bump into objects in its environment, or whether a cutting tool will cut only the material it is intended to remove. In simulating physical mechanisms, such as a gear train, it may be important to compute properties such as volume and center of mass. Finite-element analysis is applied to solid models to determine response to factors such as stress and temperature through finite-element modeling. A satisfactory representation for a solid object may even make it possible to generate instructions automatically for computer-controlled machine tools to create that object. In addition, some graphical techniques, such as modeling refractive transparency, depend on being able to determine where a beam of light enters and exits a solid object. These applications are all examples of *solid modeling*. The need to model objects as solids has resulted in the development of a variety of specialized ways to represent them. This chapter provides a brief introduction to these representations.

12.1 REPRESENTING SOLIDS

A representation's ability to encode things that *look* like solids does not by itself mean that the representation is adequate for representing solids. Consider how we have represented objects so far, as collections of straight lines, curves, polygons, and surfaces. Do the lines of Fig. 12.1(a) define a solid cube? If each set of four lines on each side of the object is assumed to bound a square face, then the figure is a cube. However, there is nothing in the representation given that requires the lines to be interpreted this way. For example, the same set of lines would be used to draw the figure if any or all of the faces were missing. What if we decide that each planar loop of connected lines in the drawing by definition determines a polygonal face? Then, Fig. 12.1(b) would consist of all of the faces of Fig. 12.1(a), plus an extra "dangling" face, producing an object that does not bound a volume. As we shall see in Section 12.5, some extra constraints are needed if we want to ensure that a representation of this sort models only solids.

Requicha [REQU80] provides a list of the properties desirable in a solid representation scheme. The *domain* of representation should be large enough to allow a useful set of physical objects to be represented. The representation should ideally be *unambiguous*: There should be no question as to what is being represented, and a given representation should correspond to one and only one solid, unlike the one in Fig. 12.1(a). An unambiguous representation is also said to be *complete*. A representation is *unique* if it can be used to encode any given solid in only one way. If a representation can ensure uniqueness, then operations such as testing two objects for equality are easy. An *accurate* representation allows an object to be represented without approximation. Much as a graphics system that can draw only straight lines forces us to create approximations of smooth curves, some solid modeling representations represent many objects as approximations. Ideally, a representation scheme should make it *impossible to create an invalid representation* (i.e., one that does not correspond to a solid), such as Fig. 12.1(b). On the other hand, it should be *easy to create a valid representation*, typically with the aid of an interactive solid modeling system. We would like objects to maintain *closure* under



Vertices		Lines
A	(0, 0, 0)	*AB*
B	(1, 0, 0)	*BC*
C	(1, 1, 0)	*CD*
D	(0, 1, 0)	*DA*
E	(0, 0, 1)	*EF*
F	(1, 0, 1)	*FG*
G	(1, 1, 1)	*GH*
H	(0, 1, 1)	*HE*
		AE
		BF
		CG
		DH

Extra Vertices		Extra Lines
I	(1, 0, −1)	*BI*
J	(1, 1, −1)	*IJ*
		JC

Fig. 12.1 (a) A wireframe cube composed of 12 straight lines. (b) A wireframe cube with an extra face.

rotation, translation, and other operations. This means that performing these operations on valid solids should yield only valid solids. A representation should be *compact* to save space, which in turn may save communication time in a distributed system. Finally, a representation should allow the use of *efficient* algorithms for computing desired physical properties, and, most important for us, for creating pictures.

Designing a representation with all these properties is difficult indeed, and compromises are often necessary. As we discuss the major representations in use today, our emphasis will be on providing enough detail to be able to understand how these representations can be interfaced to graphics software. More detail, with an emphasis on the solid modeling aspects, can be found in [REQU80; MORT85; MANT88].

12.2 REGULARIZED BOOLEAN SET OPERATIONS

No matter how we represent objects, we would like to be able to combine them in order to make new ones. One of the most intuitive and popular methods for combining objects is by *Boolean set operations,* such as union, difference, and intersection, as shown in Fig. 12.2. These are the 3D equivalents of the familiar 2D Boolean operations. Applying an ordinary Boolean set operation to two solid objects, however, does not necessarily yield a solid object. For example, the ordinary Boolean intersections of the cubes in Fig. 12.3(a) through (e) are a solid, a plane, a line, a point, and the null object, respectively.

Rather than using the ordinary Boolean set operators, we will instead use the *regularized Boolean set operators* [REQU77], denoted $\cup^*$, $\cap^*$, and $-^*$, and defined such that operations on solids always yield solids. For example, the regularized Boolean intersection of the objects shown in Fig. 12.3 is the same as their ordinary Boolean intersection in cases (a) and (e), but is empty in (b) through (d).

To explore the difference between ordinary and regularized operators, we can consider any object to be defined by a set of points, partitioned into interior points and boundary points, as shown in Fig. 12.4(a). *Boundary* points are those points whose distance from the object and the object's complement is zero. Boundary points need not be part of the object. A *closed* set contains all its boundary points, whereas an *open* set contains none. The union of a set with the set of its boundary points is known as the set's *closure,* as shown in Fig. 12.4(b), which is itself a closed set. The *boundary* of a closed set is the set of its boundary points, whereas the *interior,* shown in Fig. 12.4(c), consists of all of the set's other points, and thus is the complement of the boundary with respect to the object. The *regularization* of

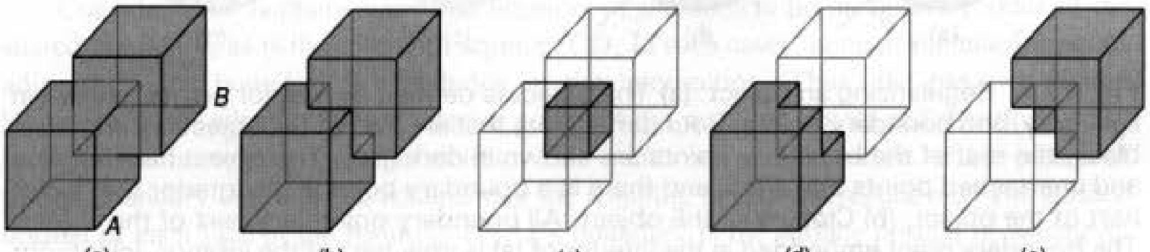


Fig. 12.2 Boolean operations. (a) Objects *A* and *B*, (b) $A \cup B$, (c) $A \cap B$, (d) $A - B$, and (e) $B - A$.

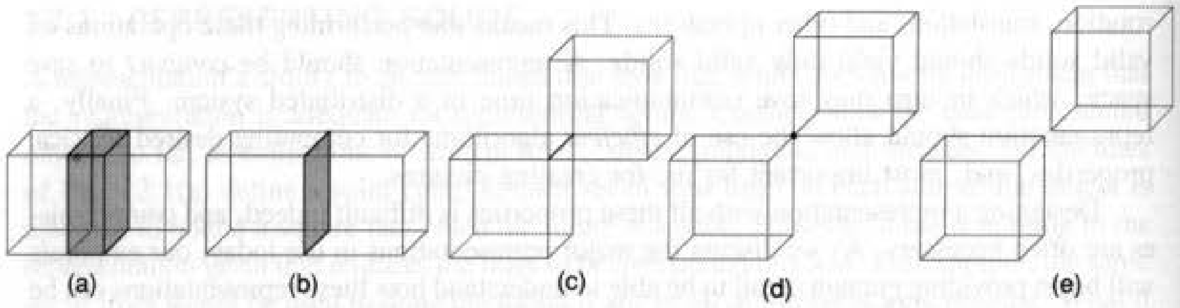


Fig. 12.3 The ordinary Boolean intersection of two cubes may produce (a) a solid, (b) a plane, (c) a line, (d) a point, or (e) the null set.

a set is defined as the closure of the set's interior points. Figure 12.4(d) shows the closure of the object in Fig. 12.4(c) and, therefore, the regularization of the object in Fig. 12.4(a). A set that is equal to its own regularization is known as a *regular set*. Note that a regular set can contain no boundary point that is not adjacent to some interior point; thus, it can have no "dangling" boundary points, lines, or surfaces. We can define each regularized Boolean set operator in terms of the corresponding ordinary Boolean set operator as

$$A \; op^* \; B = \text{closure}(\text{interior}(A \; op \; B)), \tag{12.1}$$

where *op* is one of $\cup$, $\cap$, or $-$. The regularized Boolean set operators produce only regular sets when applied to regular sets.

We now compare the ordinary and regularized Boolean set operations as performed on regular sets. Consider the two objects of Fig. 12.5(a), positioned as shown in Fig. 12.5(b). The ordinary Boolean intersection of two objects contains the intersection of the interior and boundary of each object with the interior and boundary of the other, as shown in Fig. 12.5(c). In contrast, the regularized Boolean intersection of two objects, shown in Fig. 12.5(d), contains the intersection of their interiors and the intersection of the interior of each with the boundary of the other, but only a subset of the intersection of their boundaries. The criterion used to define this subset determines how regularized Boolean

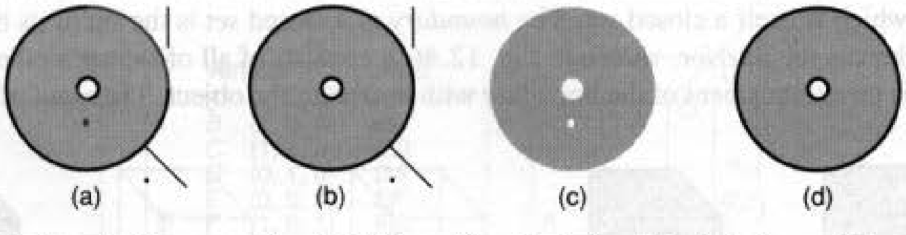


Fig. 12.4 Regularizing an object. (a) The object is defined by interior points, shown in light gray, and boundary points. Boundary points that are part of the object are shown in black; the rest of the boundary points are shown in dark gray. The object has dangling and unattached points and lines, and there is a boundary point in the interior that is not part of the object. (b) Closure of the object. All boundary points are part of the object. The boundary point embedded in the interior of (a) is now part of the interior. (c) Interior of the object. Dangling and unattached points and lines have been eliminated. (d) Regularization of the object is the closure of its interior.

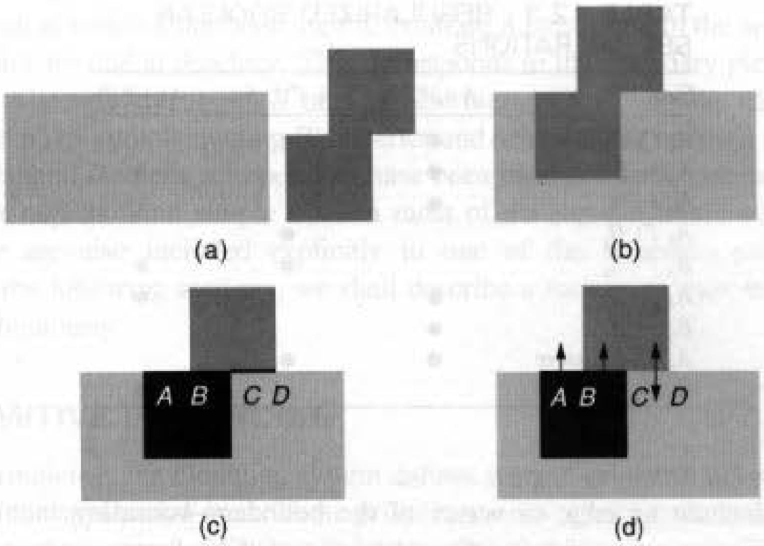


Fig. 12.5 Boolean intersection. (a) Two objects, shown in cross-section. (b) Positions of object prior to intersection. (c) Ordinary Boolean intersection results in a dangling face, shown as line *CD* in cross-section. (d) Regularized Boolean intersection includes a piece of shared boundary in the resulting boundary if both objects lie on the same side of it (*AB*), and excludes it if the objects lie on opposite sides (*CD*). Boundary–interior intersections are always included (*BC*).

intersection differs from ordinary Boolean intersection, in which all parts of the intersection of the boundaries are included.

Intuitively, a piece of the boundary–boundary intersection is included in the regularized Boolean intersection if and only if the interiors of both objects lie on the same side of this piece of shared boundary. Since the interior points of both objects that are directly adjacent to that piece of boundary are in the intersection, the boundary piece must also be included to maintain closure. Consider the case of a piece of shared boundary that lies in coplanar faces of two polyhedra. Determining whether the interiors lie on the same side of a shared boundary is simple if both objects are defined such that their surface normals point outward (or inward). The interiors are on the same side if the normals point in the same direction. Thus, segment *AB* in Fig. 12.5(d) is included. Remember that those parts of one object's boundary that intersect with the other object's interior, such as segment *BC*, are always included.

Consider what happens when the interiors of the objects lie on opposite sides of the shared boundary, as is the case with segment *CD*. In such cases, none of the interior points adjacent to the boundary are included in the intersection. Thus, the piece of shared boundary is not adjacent to any interior points of the resulting object and therefore is not included in the regularized intersection. This additional restriction on which pieces of shared boundary are included ensures that the resulting object is a regular set. The surface normal of each face of the resulting object's boundary is the surface normal of whichever surface(s) contributed that part of the boundary. (As we shall see in Chapter 16, surface normals are important in shading objects.) Having determined which faces lie in the

TABLE 12.1 REGULARIZED BOOLEAN SET OPERATIONS

Set	$A \cup^* B$	$A \cap^* B$	$A -^* B$
$A_i \cap B_i$	●	●	
$A_i - B$	●		●
$B_i - A$	●		
$A_b \cap B_i$		●	
$B_b \cap A_i$		●	●
$A_b - B$	●		●
$B_b - A$	●		
$A_b \cap B_b$ *same*	●	●	
$A_b \cap B_b$ *diff*			●

boundary, we include an edge or vertex of the boundary–boundary intersection in the boundary of the intersection if it is adjacent to one of these faces.

The results of each regularized operator may be defined in terms of the ordinary operators applied to the boundaries and interiors of the objects. Table 12.1 shows how the regularized operators are defined for any objects A and B; Fig. 12.6 shows the results of performing the operations. A_b and A_i are A's boundary and interior, respectively. $A_b \cap B_b$ *same* is that part of the boundary shared by A and B for which A_i and B_i lie on the same side. This is the case for some point b on the shared boundary if at least one point i adjacent to it is a member of both A_i and B_i. $A_b \cap B_b$ *diff* is that part of the boundary shared by A and B for which A_i and B_i lie on opposite sides. This is true for b if it is adjacent to no such point i. Each regularized operator is defined by the union of the sets associated with those rows that have a ● in the operator's column.

Note that, in all cases, each piece of the resulting object's boundary is on the boundary of one or both of the original objects. When computing $A \cup^* B$ or $A \cap^* B$, the surface normal of a face of the result is inherited from the surface normal of the corresponding face of one or both original objects. In the case of $A -^* B$, however, the surface normal of each

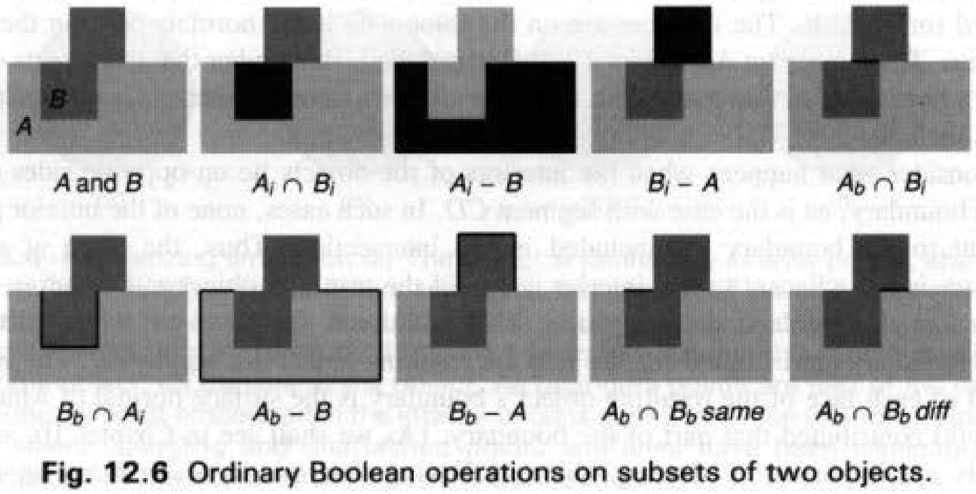


Fig. 12.6 Ordinary Boolean operations on subsets of two objects.

face of the result at which B has been used to excavate A must point in the *opposite* direction from B's surface normal at that face. This corresponds to the boundary pieces $A_b \cap B_b$ *diff* and $B_b \cap A_i$. Alternatively, $A -^* B$ may be rewritten as $A \cap^* \overline{B}$. We can obtain $\overline{B}$ (the complement of B) by complementing B's interior and reversing the normals of its boundary.

The regularized Boolean set operators have been used as a user–interface technique to build complex objects from simple ones in most of the representation schemes we shall discuss. They are also included explicitly in one of the schemes, constructive solid geometry. In the following sections, we shall describe a variety of ways to represent solid objects unambiguously.

12.3 PRIMITIVE INSTANCING

In *primitive instancing*, the modeling system defines a set of primitive 3D solid shapes that are relevant to the application area. These primitives are typically parameterized not just in terms of the transformations of Chapter 7, but on other properties as well. For example, one primitive object may be a regular pyramid with a user-defined number of faces meeting at the apex. Primitive instances are similar to parameterized objects, such as the menus of Chapter 2, except that the objects are solids. A parameterized primitive may be thought of as defining a family of parts whose members vary in a few parameters, an important CAD concept known as *group technology*. Primitive instancing is often used for relatively complex objects, such as gears or bolts, that are tedious to define in terms of Boolean combinations of simpler objects, yet are readily characterized by a few high-level parameters. For example, a gear may be parameterized by its diameter or number of teeth, as shown in Fig. 12.7.

Although we can build up a hierarchy of primitive instances, each leaf-node instance is still a separately defined object. In primitive instancing, no provisions are made for combining objects to form a new higher-level object, using, for example, the regularized Boolean set operations. Thus, the only way to create a new kind of object is to write the code that defines it. Similarly, the routines that draw the objects or determine their mass properties must be written individually for each primitive.

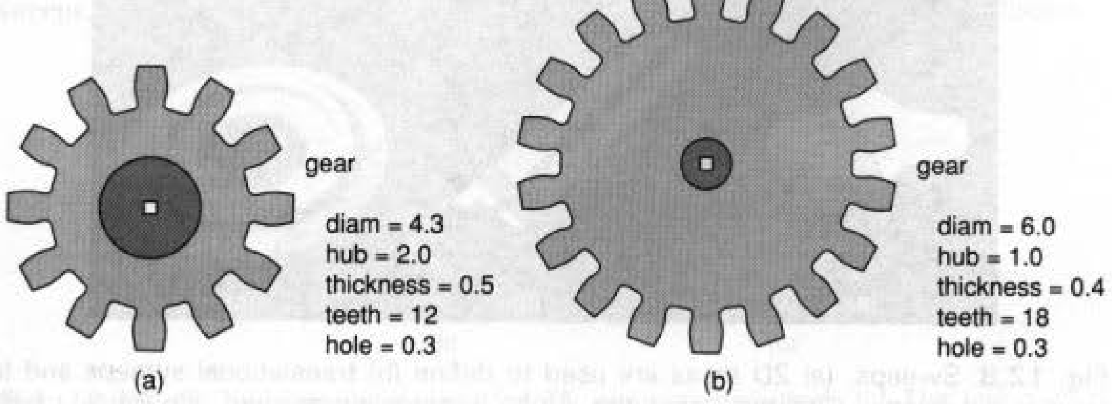


Fig. 12.7 Two gears defined by primitive instancing.

12.4 SWEEP REPRESENTATIONS

Sweeping an object along a trajectory through space defines a new object, called a *sweep*. The simplest kind of sweep is defined by a 2D area swept along a linear path normal to the plane of the area to create a volume. This is known as a *translational sweep* or *extrusion* and is a natural way to represent objects made by extruding metal or plastic through a die with the desired cross-section. In these simple cases, each sweep's volume is simply the swept object's area times the length of the sweep. Simple extensions involve scaling the cross-section as it is swept to produce a tapered object or sweeping the cross-section along a linear path that is not normal to it. *Rotational sweeps* are defined by rotating an area about an axis. Figure 12.8 shows two objects and simple translational and rotational sweeps generated using them.

The object being swept does not need to be 2D. Sweeps of solids are useful in modeling the region swept out by a machine-tool cutting head or robot following a path, as shown in Fig. 12.9. Sweeps whose generating area or volume changes in size, shape, or orientation as they are swept and that follow an arbitrary curved trajectory are called *general sweeps*. General sweeps of 2D cross-sections are known as *generalized cylinders* in computer vision [BINF71] and are usually modeled as parameterized 2D cross-sections swept at right angles along an arbitrary curve. General sweeps are particularly difficult to model efficiently. For example, the trajectory and object shape may make the swept object intersect itself, making volume calculations complicated. As well, general sweeps do not always generate solids. For example, sweeping a 2D area in its own plane generates another 2D area.

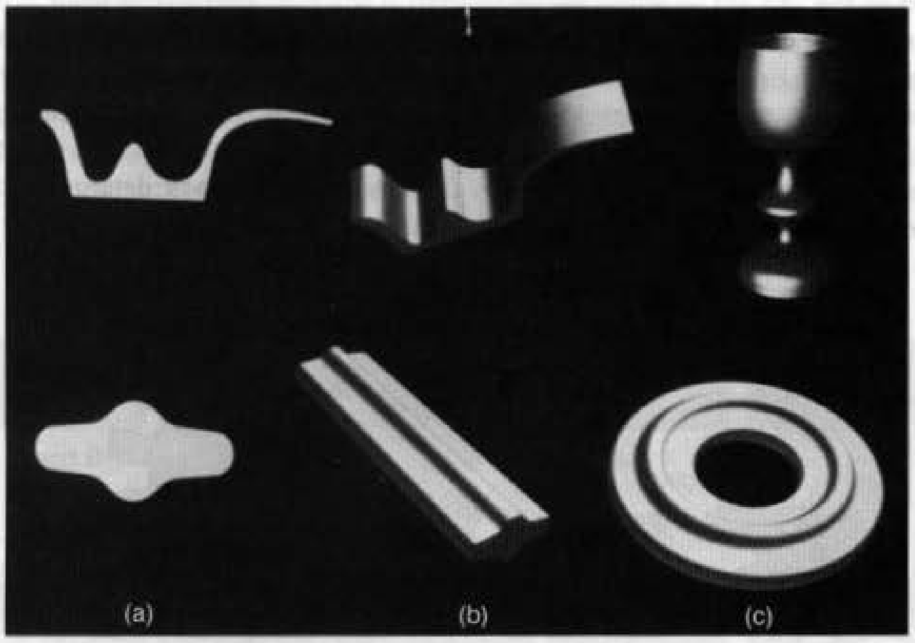
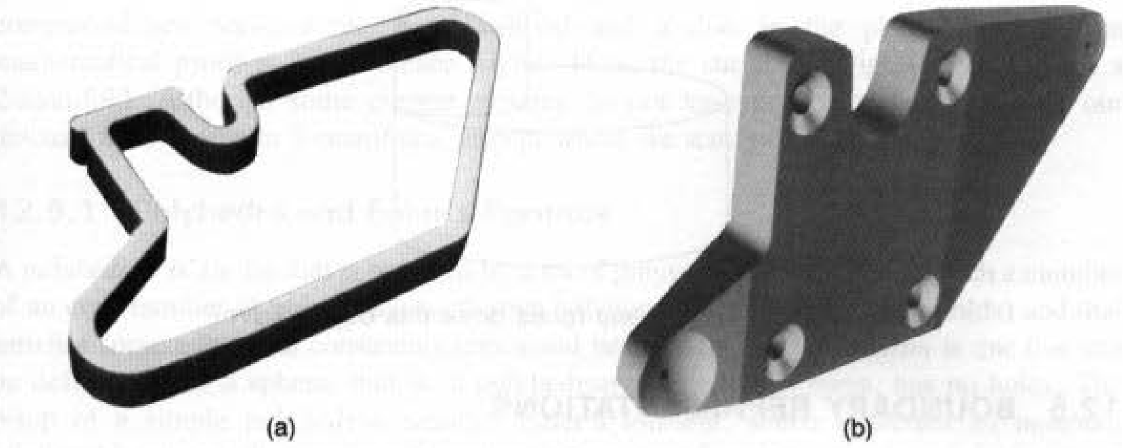


Fig. 12.8 Sweeps. (a) 2D areas are used to define (b) translational sweeps and (c) rotational sweeps. (Created using the Alpha_1 modeling system. Courtesy of the University of Utah.)



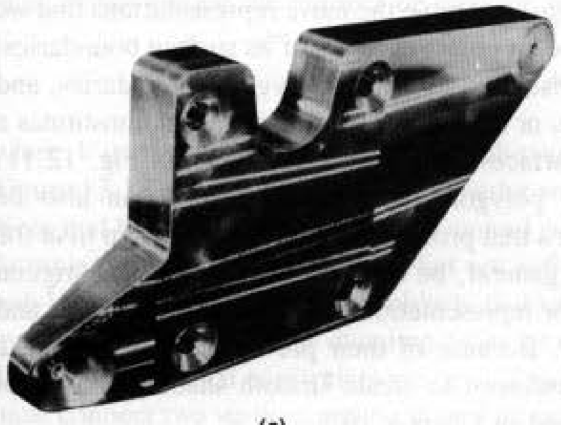


Fig. 12.9 (a) Path of a cutting tool, modeled as a solid sweep, is used to define (b) model of an aircraft part. (c) Photograph of actual part that is milled from automatically generated instructions. (Created using the Alpha_1 modeling system. Courtesy of the University of Utah.)

In general, it is difficult to apply regularized Boolean set operations to sweeps without first converting to some other representation. Even simple sweeps are not closed under regularized Boolean set operations. For example, the union of two simple sweeps is in general not a simple sweep, as shown in Fig. 12.10. Despite problems of closure and calculation, however, sweeps are a natural and intuitive way to construct a variety of objects. For this reason, many solid modeling systems allow users to construct objects as sweeps, but store the objects in one of the other representations that we shall discuss.

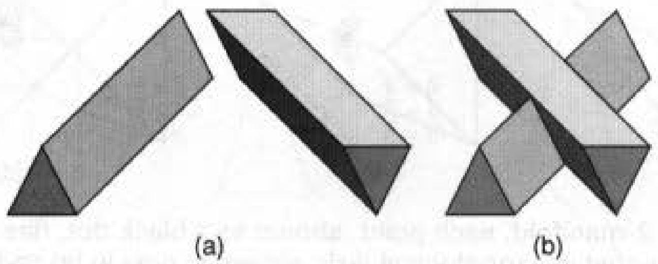


Fig. 12.10 (a) Two simple sweeps of 2D objects (triangles). (b) The union of the sweeps shown in (a) is not itself a simple sweep of a 2D object.

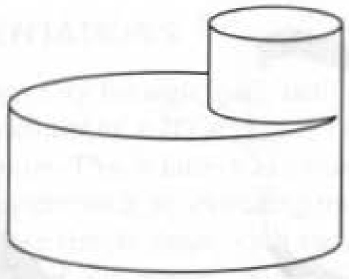


Fig. 12.11 How many faces does this object have?

12.5 BOUNDARY REPRESENTATIONS

Boundary representations (also known as *b-reps*) resemble the naive representations that we discussed in Section 12.1 in that they describe an object in terms of its surface boundaries: vertices, edges, and faces. Some b-reps are restricted to planar, polygonal boundaries, and may even require faces to be convex polygons or triangles. Determining what constitutes a face can be particularly difficult if curved surfaces are allowed, as shown in Fig. 12.11. Curved faces are often approximated with polygons. Alternatively, they can also be represented as surface patches if the algorithms that process the representation can treat the resulting intersection curves, which will, in general, be of higher order than the original surfaces. B-reps grew out of the simple vector representations used in earlier chapters and are used in many current modeling systems. Because of their prevalence in graphics, a number of efficient techniques have been developed to create smooth shaded pictures of polygonal objects; many of these are discussed in Chapter 16.

Many b-rep systems support only solids whose boundaries are *2-manifolds*. By definition, every point on a 2-manifold has some arbitrarily small neighborhood of points around it that can be considered topologically the same as a disk in the plane. This means that there is a continuous one-to-one correspondence between the neighborhood of points and the disk, as shown in Fig. 12.12(a) and (b). For example, if more than two faces share an edge, as in Fig. 12.12(c), any neighborhood of a point on that edge contains points from each of those faces. It is intuitively obvious that there is no continuous one-to-one

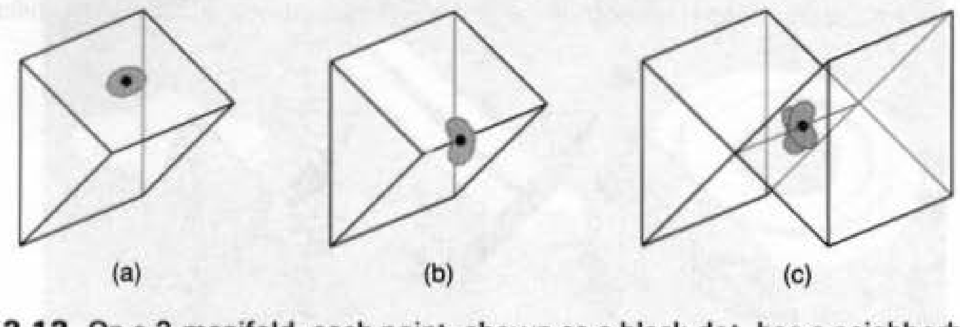


Fig. 12.12 On a 2-manifold, each point, shown as a black dot, has a neighborhood of surrounding points that is a topological disk, shown in gray in (a) and (b). (c) If an object is not a 2-manifold, then it has points that do not have a neighborhood that is a topological disk.

correspondence between this neighborhood and a disk in the plane, although the mathematical proof is by no means trivial. Thus, the surface in Fig. 12.12(c) is not a 2-manifold. Although some current systems do not have this restriction, we limit our discussion of b-reps to 2-manifolds, except where we state otherwise.

12.5.1 Polyhedra and Euler's Formula

A *polyhedron* is a solid that is bounded by a set of polygons whose edges are each a member of an even number of polygons (exactly two polygons in the case of 2-manifolds) and that satisfies some additional constraints (discussed later). A *simple polyhedron* is one that can be deformed into a sphere; that is, a polyhedron that, unlike a torus, has no holes. The b-rep of a simple polyhedron satisfies Euler's formula, which expresses an invariant relationship among the number of vertices, edges, and faces of a simple polyhedron:

$$V - E + F = 2, \tag{12.2}$$

where V is the number of vertices, E is the number of edges, and F is the number of faces. Figure 12.13 shows some simple polyhedra and their numbers of vertices, edges, and faces. Note that the formula still applies if curved edges and nonplanar faces are allowed. Euler's formula by itself states necessary but not sufficient conditions for an object to be a simple polyhedron. One can construct objects that satisfy the formula but do not bound a volume, by attaching one or more dangling faces or edges to an otherwise valid solid, as in Fig. 12.1(b). Additional constraints are needed to guarantee that the object is a solid: each edge must connect two vertices and be shared by exactly two faces, at least three edges must meet at each vertex, and faces must not interpenetrate.

A generalization of Euler's formula applies to 2-manifolds that have faces with holes:

$$V - E + F - H = 2(C - G), \tag{12.3}$$

where H is the number of holes in the faces, G is the number of holes that pass through the object, and C is the number of separate components (parts) of the object, as shown in Fig. 12.14. If an object has a single component, its G is known as its *genus*; if it has multiple components, then its G is the sum of the genera of its components. As before, additional constraints are also needed to guarantee that the objects are solids.

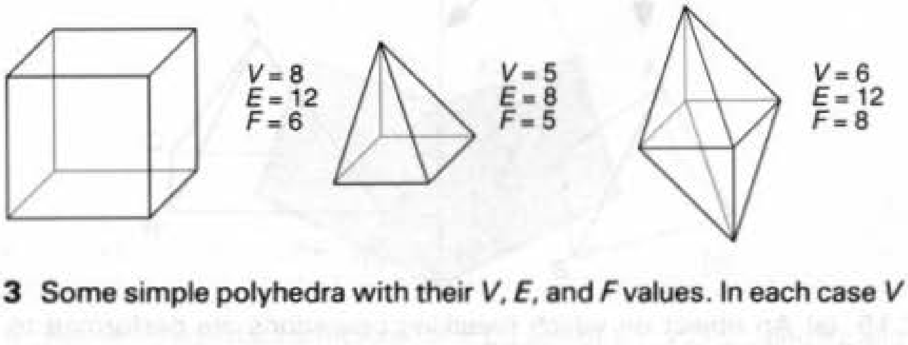


Fig. 12.13 Some simple polyhedra with their V, E, and F values. In each case $V - E + F = 2$.

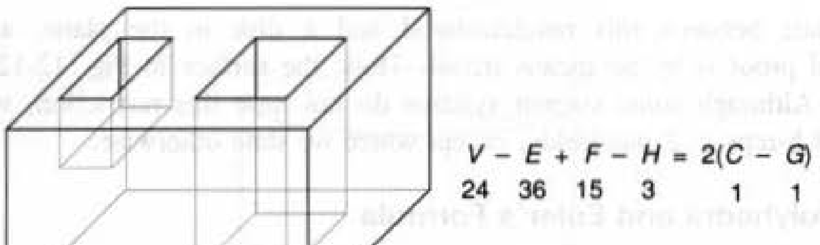


Fig. 12.14 A polyhedron classified according to Eq. (12.3), with two holes in its top face and one hole in its bottom face.

Baumgart introduced the notion of a set of *Euler operators* that operate on objects satisfying Euler's formula to transform the objects into new objects that obey the formula as well, by adding and removing vertices, edges, and faces [BAUM74]. Braid, Hillyard, and Stroud [BRAI78] show how a small number of Euler operators can be composed to transform objects, provided that intermediate objects are not required to be valid solids, whereas Mäntylä [MANT88] proves that all valid b-reps can be constructed by a finite sequence of Euler operators. Other operators that do not affect the number of vertices, edges, or faces may be defined that *tweak* an object by moving the existing vertices, edges, or faces, as shown in Fig. 12.15.

Perhaps the simplest possible b-rep is a list of polygonal faces, each represented by a list of vertex coordinates. To represent the direction in which each polygon faces, we list a polygon's vertices in clockwise order, as seen from the exterior of the solid. To avoid replicating coordinates shared by faces, we may instead represent each vertex of a face by an index into a list of coordinates. In this representation, edges are represented implicitly by

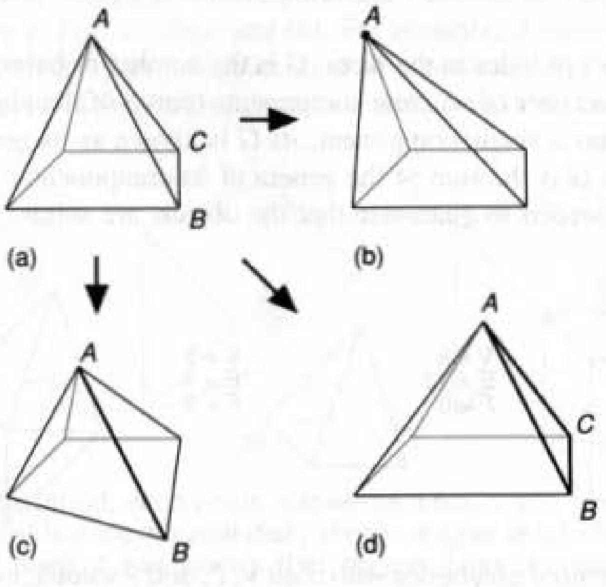


Fig. 12.15 (a) An object on which tweaking operations are performed to move (b) vertex *A*, (c) edge *AB*, and (d) face *ABC*.

the pairs of adjacent vertices in the polygon vertex lists. Edges may instead be represented explicitly as pairs of vertices, with each face now defined as a list of indices into the list of edges. These representations were discussed in more detail in Section 11.1.1.

12.5.2 The Winged-Edge Representation

Simple representations make certain computations quite expensive. For example, discovering the two faces shared by an edge (e.g., to help prove that a representation encodes a valid solid) requires searching the edge lists of all the faces. More complex b-reps have been designed to decrease the cost of these computations. One of the most popular is the *winged-edge* data structure developed by Baumgart [BAUM72; BAUM75]. As shown in Fig. 12.16, each edge in the winged-edge data structure is represented by pointers to its two vertices, to the two faces sharing the edge, and to four of the additional edges emanating from its vertices. Each vertex has a backward pointer to one of the edges emanating from it, whereas each face points to one of its edges. Note that we traverse the vertices of an edge in opposite directions when following the vertices of each of its two faces in clockwise order. Labeling the edge's vertices *n* and *p*, we refer to the face to its right when traversing the edge from *n* to *p* as its *p* face, and the face to its right when traversing the edge from *p* to *n* as its *n* face. For edge *E*1 in Fig. 12.16, if *n* is *V*1 and *p* is *V*2, then *F*1 is *E*1's *p* face, and *F*2 is its *n* face. The four edges to which each edge points can be classified as follows. The two edges that share the edge's *n* vertex are the next (clockwise) edge of the *n* face, and the previous (counterclockwise) edge of the *p* face, *E*3 and *E*2, respectively. The two edges that share the edge's *p* vertex are the next (clockwise) edge of the *p* face, and the previous (counterclockwise) edge of the *n* face, *E*4 and *E*5, respectively. These four edges are the "wings" from which the winged-edge data structure gets its name.

Note that the data structure described here handles only faces that have no holes. This limitation can be removed by representing each face as a set of edge loops—a clockwise outer loop and zero or more counterclockwise inner loops for its holes—as described in

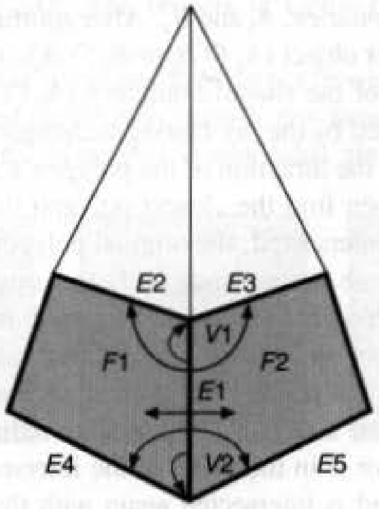


Fig. 12.16 Winged-edge data structure for *E*1. Each of *V*1, *V*2, *F*1, and *F*2 also have a backward pointer to one of their edges (not shown).

Section 19.1. Alternatively, a special auxiliary edge can be used to join each hole's boundary to the outer boundary. Each auxiliary edge is traversed twice, once in each direction, when a circuit of its face's edges is completed. Since an auxiliary edge has the same face on both of its sides, it can be easily identified because its two face pointers point to the same face.

A b-rep allows us to query which faces, edges, or vertices are adjacent to each face, edge, or vertex. These queries correspond to nine kinds of *adjacency relationships*. The winged-edge data structure makes it possible to determine in constant time which vertices or faces are associated with an edge. It takes longer to compute other adjacency relationships. One attractive property of the winged edge is that the data structures for the edges, faces, and vertices are each of a small, constant size. Only the number of instances of each data structure varies among objects. Weiler [WEIL85] and Woo [WOO85] discuss the space–time efficiency of the winged edge and a variety of alternative b-rep data structures.

12.5.3 Boolean Set Operations

B-reps may be combined, using the regularized Boolean set operators, to create new b-reps [REQU85]. Sarraga [SARR83] and Miller [MILL87] discuss algorithms that determine the intersections between quadric surfaces. Algorithms for combining polyhedral objects are presented in [TURN84; REQU85; PUTN86; LAID86], and Thibault and Naylor [THIB87] describe a method based on the binary space-partitioning tree representation of solids discussed in Section 12.6.4.

One approach [LAID86] is to inspect the polygons of both objects, splitting them if necessary to ensure that the intersection of a vertex, edge, or face of one object with any vertex, edge, or face of another, is a vertex, edge, or face of both. The polygons of each object are then classified relative to the other object to determine whether they lie inside, outside, or on its boundary. Referring back to Table 12.1, we note that since this is a b-rep, we are concerned with only the last six rows, each of which represents some part of one or both of the original object boundaries, A_b and B_b. After splitting, each polygon of one object is either wholly inside the other object ($A_b \cap B_i$ or $B_b \cap A_i$), wholly outside the other object (A_b − B or B_b − A), or part of the shared boundary ($A_b \cap B_b$ *same* or $A_b \cap B_b$ *diff*).

A polygon may be classified by the ray-casting technique discussed in Section 15.10.1. Here, we construct a vector in the direction of the polygon's surface normal from a point in the polygon's interior, and then find the closest polygon that intersects the vector in the other object. If no polygon is intersected, the original polygon is outside the other object. If the closest intersecting polygon is coplanar with the original polygon, then this is a boundary–boundary intersection, and comparing polygon normals indicates what kind of intersection it is ($A_b \cap B_b$ *same* or $A_b \cap B_b$ *diff*). Otherwise, the dot product of the two polygons' normals is inspected. A positive dot product indicates that the original polygon is inside the other object, whereas a negative dot product indicates that it is outside. A zero dot product occurs if the vector is in the plane of the intersected polygon; in this case, the vector is perturbed slightly and is intersected again with the other object's polygons.

Vertex-adjacency information can be used to avoid the overhead of classifying each polygon in this way. If a polygon is adjacent to (i.e., shares vertices with) a classified polygon and does not meet the surface of the other object, then it is assigned the same classification. All vertices on the common boundary between objects can be marked during the initial polygon-splitting phase. Whether or not a polygon meets the other object's surface can be determined by checking whether or not it has boundary vertices.

Each polygon's classification determines whether it is retained or discarded in the operation creating the composite object, as described in Section 12.2. For example, in forming the union, any polygon belonging to one object that is inside the other object is discarded. Any polygon from either object that is not inside the other is retained, except in the case of coplanar polygons. Coplanar polygons are discarded if they have opposite surface normals, and only one of a pair is retained if the directions of the surface normals are the same. Deciding which polygon to retain is important if the objects are made of different materials. Although $A \cup^* B$ has the same geometric meaning as $B \cup^* A$, the two may have visibly different results in this case, so the operation may be defined to favor one of its operands in the case of coplanar polygons.

12.5.4 Nonpolyhedral b-Reps

Unfortunately, polyhedral representations can only approximate objects that are not themselves polyhedral, and can require large amounts of data to approximate objects with curved surfaces acceptably. Consider the problem of representing a cylindrical object in a cylindrical hole with polyhedral b-reps, as shown in Fig. 12.17. If the boundaries of the actual objects touch, then even if the boundaries of the two polyhedral approximations are initially coincident as well, the approximations will intersect if one is slowly rotated, no matter how many polygons are used in the approximation.

One promising approach to exact b-reps allows sculpted surfaces defined by curves. The Alpha_1 [COHE83] and Geomod [TILL83] modeling systems model such free-form surfaces as tensor products of NURBS (see Section 11.2.5). Since each individual surface may not itself be closed, Thomas [THOM84] has developed an algorithm for Alpha_1 that performs regularized Boolean set operations on objects whose boundaries are only partially specified, as shown in Fig. 12.18. The objects in Color Plate I.31 were modeled with Alpha_1.

Because b-reps tile an object's surface, they do not provide a unique representation of a solid. In addition, as mentioned previously, many b-rep–based systems handle only objects whose surfaces are 2-manifolds. Note that 2-manifolds are not closed under regularized

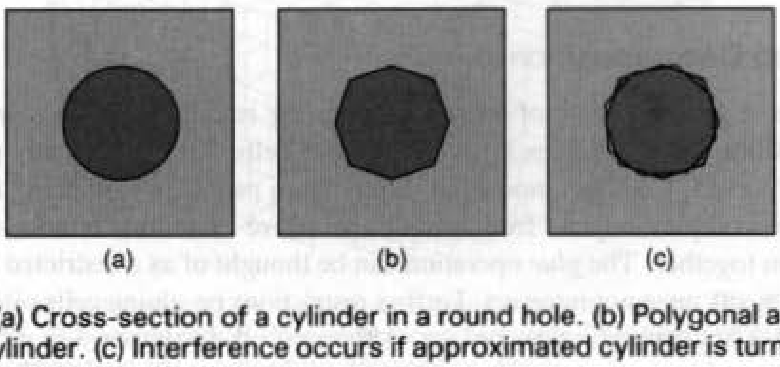


Fig. 12.17 (a) Cross-section of a cylinder in a round hole. (b) Polygonal approximation of hole and cylinder. (c) Interference occurs if approximated cylinder is turned relative to hole.

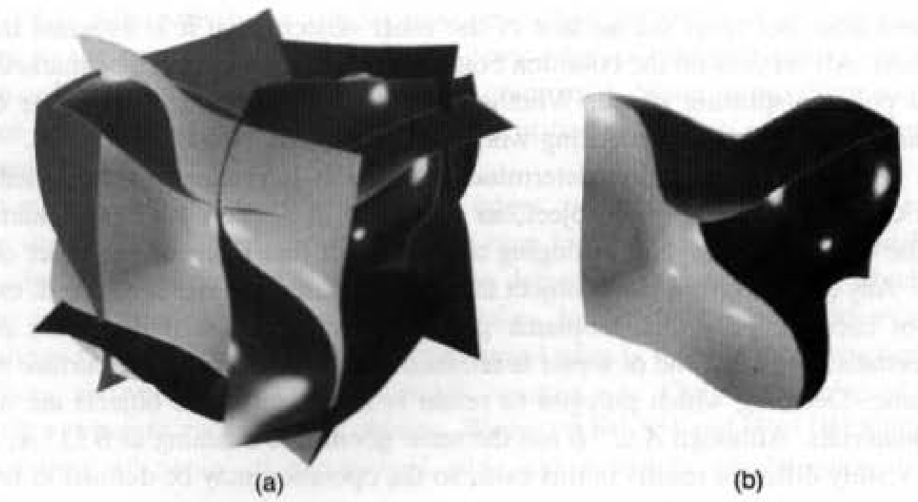


Fig. 12.18 Boolean set operations on partially bounded objects. (a) Six partially bounded sets. (b) Intersection of sets defines a wavy cube. (Courtesy of Spencer W. Thomas, University of Utah.)

Boolean set operations. The regularized union of two b-rep cubes positioned such that they share exactly one common edge, for example, should produce four faces sharing that common edge. This configuration is not allowed in some systems, however, such as those based on the winged-edge representation. Weiler [WEIL88] describes a nonmanifold, boundary-based modeling system that can handle wireframe objects, in addition to surfaces and solids, as illustrated in Color Plate I.32.

12.6 SPATIAL-PARTITIONING REPRESENTATIONS

In *spatial-partitioning* representations, a solid is decomposed into a collection of adjoining, nonintersecting solids that are more primitive than, although not necessarily of the same type as, the original solid. Primitives may vary in type, size, position, parameterization, and orientation, much like the different-shaped blocks in a child's block set. How far we decompose objects depends on how primitive the solids must be in order to perform readily the operations of interest.

12.6.1 Cell Decomposition

One of the most general forms of spatial partitioning is called *cell decomposition*. Each cell-decomposition system defines a set of primitive cells that are typically parameterized and are often curved. Cell decomposition differs from primitive instancing in that we can compose more complex objects from simple, primitive ones in a bottom-up fashion by "gluing" them together. The *glue* operation can be thought of as a restricted form of union in which the objects must not intersect. Further restrictions on gluing cells often require that two cells share a single point, edge, or face. Although cell-decomposition representation of an object is unambiguous, it is not necessarily unique, as shown in Fig. 12.19. Cell

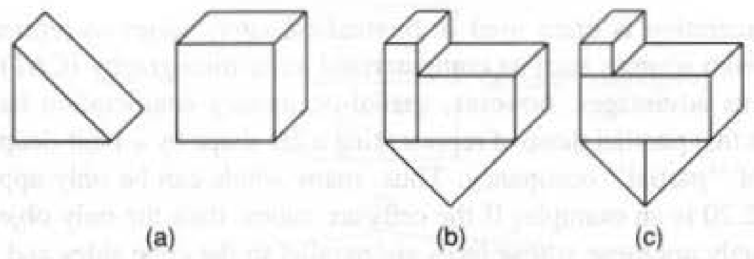


Fig. 12.19 The cells shown in (a) may be transformed to construct the same object shown in (b) and (c) in different ways. Even a single cell type is enough to cause ambiguity.

decompositions are also difficult to validate, since each pair of cells must potentially be tested for intersection. Nevertheless, cell decomposition is an important representation for use in finite element analysis.

12.6.2 Spatial-Occupancy Enumeration

Spatial-occupancy enumeration is a special case of cell decomposition in which the solid is decomposed into identical cells arranged in a fixed, regular grid. These cells are often called *voxels* (volume elements), in analogy to pixels. Figure 12.20 shows an object represented by spatial-occupancy enumeration. The most common cell type is the cube, and the representation of space as a regular array of cubes is called a *cuberille*. When representing an object using spatial-occupancy enumeration, we control only the presence or absence of a single cell at each position in the grid. To represent an object, we need only to decide which cells are occupied and which are not. The object can thus be encoded by a unique and unambiguous list of occupied cells. It is easy to find out whether a cell is inside or outside of the solid, and determining whether two objects are adjacent is simple as well. Spatial-

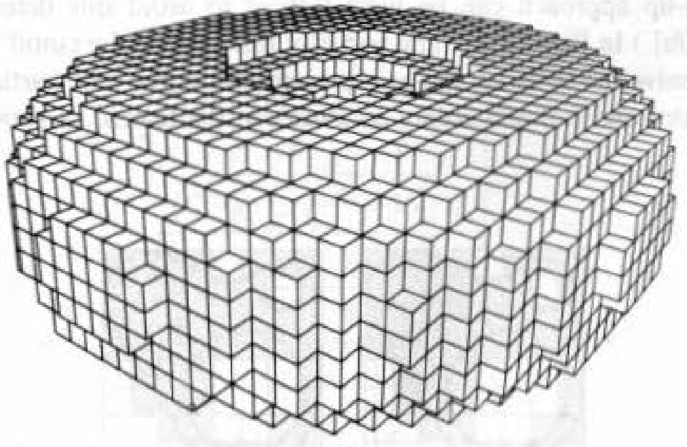


Fig. 12.20 Torus represented by spatial-occupancy enumeration. (By AHJ Christensen, SIGGRAPH '80 Conference Proceedings, *Computer Graphics* (14)3, July 1980. Courtesy of Association for Computing Machinery, Inc.)

occupancy enumeration is often used in biomedical applications to represent volumetric data obtained from sources such as computerized axial tomography (CAT) scans.

For all of its advantages, however, spatial-occupancy enumeration has a number of obvious failings that parallel those of representing a 2D shape by a 1-bit-deep bitmap. There is no concept of "partial" occupancy. Thus, many solids can be only approximated; the torus of Fig. 12.20 is an example. If the cells are cubes, then the only objects that can be represented exactly are those whose faces are parallel to the cube sides and whose vertices fall exactly on the grid. Like pixels in a bitmap, cells may in principle be made as small as desired to increase the accuracy of the representation. Space becomes an important issue, however, since up to n^3 occupied cells are needed to represent an object at a resolution of n voxels in each of three dimensions.

12.6.3 Octrees

Octrees are a hierarchical variant of spatial-occupancy enumeration, designed to address that approach's demanding storage requirements. Octrees are in turn derived from *quadtrees*, a 2D representation format used to encode images (see Section 17.7). As detailed in Samet's comprehensive survey [SAME84], both representations appear to have been discovered independently by a number of researchers, quadtrees in the late 1960s to early 1970s [e.g., WARN69; KLIN71] and octrees in the late 1970s to early 1980s [e.g., HUNT78; REDD78; JACK80; MEAG80; MEAG82a].

The fundamental idea behind both the quadtree and octree is the divide-and-conquer power of binary subdivision. A quadtree is derived by successively subdividing a 2D plane in both dimensions to form quadrants, as shown in Fig. 12.21. When a quadtree is used to represent an area in the plane, each quadrant may be full, partially full, or empty (also called black, gray, and white, respectively), depending on how much of the quadrant intersects the area. A partially full quadrant is recursively subdivided into subquadrants. Subdivision continues until all quadrants are homogeneous (either full or empty) or until a predetermined cutoff depth is reached. Whenever four sibling quadrants are uniformly full or empty, they are deleted and their partially full parent is replaced with a full or empty node. (A bottom-up approach can be used instead to avoid this deletion and merging process [SAME90b].) In Fig. 12.21, any partially full node at the cutoff depth is classified as full. The successive subdivisions can be represented as a tree with partially full quadrants at the internal nodes and full and empty quadrants at the leaves, as shown in Fig. 12.22.

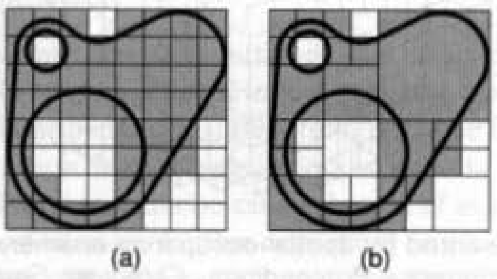


Fig. 12.21 An object represented using (a) spatial-occupancy enumeration (b) a quadtree.

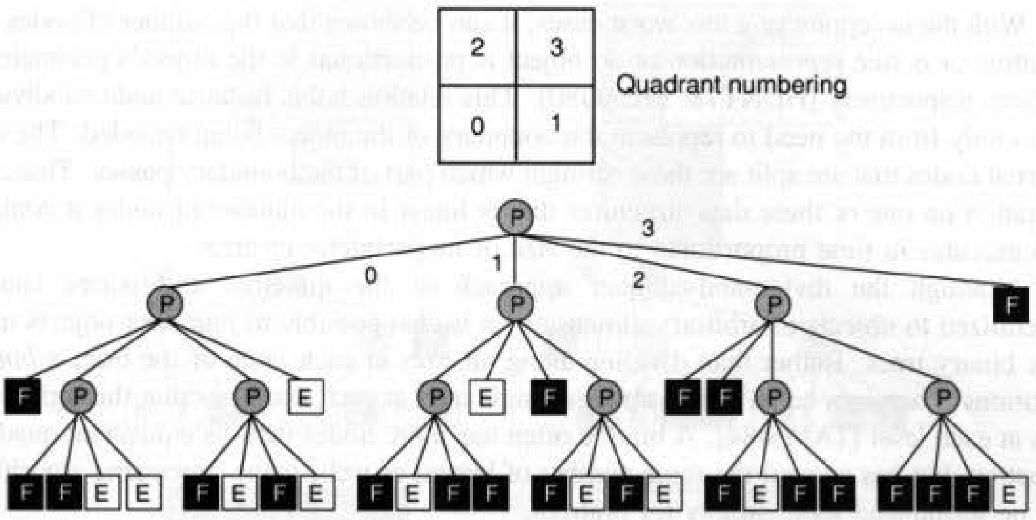


Fig. 12.22 Quadtree data structure for the object in Fig. 12.21. F = full, P = partially full, E = empty.

This idea can be compared to the Warnock area-subdivision algorithm discussed in Section 15.7.1. If the criteria for classifying a node as homogeneous are relaxed, allowing nodes that are above or below some threshold to be classified as full or empty, then the representation becomes more compact, but less accurate. The octree is similar to the quadtree, except that its three dimensions are recursively subdivided into octants, as shown in Fig. 12.23.

Quadrants are often referred to by the numbers 0 to 3, and octants by numbers from 0 to 7. Since no standard numbering scheme has been devised, mnemonic names are also used. Quadrants are named according to their compass direction relative to the center of their parent: NW, NE, SW, and SE. Octants are named similarly, distinguishing between left (L) and right (R), up (U) and down (D), and front (F) and back (B): LUF, LUB, LDF, LDB, RUF, RUB, RDF, and RDB.

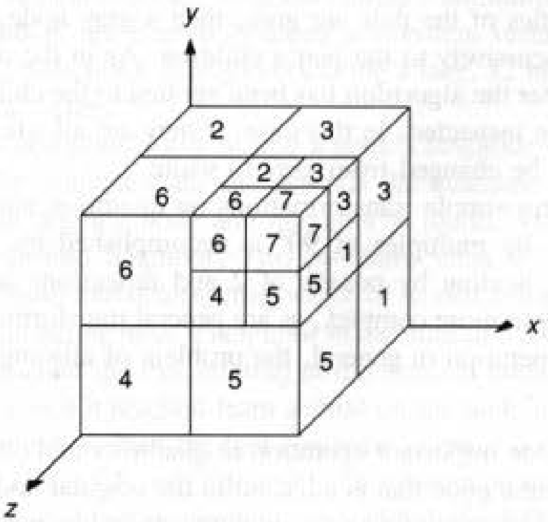


Fig. 12.23 Octree enumeration. Octant 0 is not visible.

With the exception of a few worst cases, it can be shown that the number of nodes in a quadtree or octree representation of an object is proportional to the object's perimeter or surface, respectively [HUNT78; MEAG80]. This relation holds because node subdivision arises only from the need to represent the boundary of the object being encoded. The only internal nodes that are split are those through which part of the boundary passes. Thus, any operation on one of these data structures that is linear in the number of nodes it contains also executes in time proportional to the size of its perimeter or area.

Although the divide-and-conquer approach of the quadtree and octree can be generalized to objects of arbitrary dimension, it is also possible to represent objects using only binary trees. Rather than dividing along all axes at each node of the tree, a *bintree* partitions space into equal halves about a single axis at each node, cycling through a new axis at each level [TAMM84]. A bintree often has more nodes than its equivalent quadtree or octree, but has at most the same number of leaves; as well, many processing algorithms can be formulated more simply for bintrees.

Boolean set operations and transformations. Much work has been done on developing efficient algorithms for storing and processing quadtrees and octrees [SAME84; SAME90a; SAME90b]. For example, Boolean set operations are straightforward for both quadtrees and octrees [HUNT79]. To compute the union or intersection U of two trees, S and T, we traverse both trees top-down in parallel. Figure 12.24 shows the operations for quadtrees; the generalization to octrees is straightforward. Each matching pair of nodes is examined. Consider the case of union. If either of the nodes in the pair is black, then a corresponding black node is added to U. If one of the pair's nodes is white, then the corresponding node is created in U with the value of the other node in the pair. If both nodes of the pair are gray, then a gray node is added to U, and the algorithm is applied recursively to the pair's children. In this last case, the children of the new node in U must be inspected after the algorithm has been applied to them. If they are all black, they are deleted and their parent in U is changed from gray to black. The algorithm for performing intersection is similar, except the roles of black and white are interchanged. If either of the nodes in a pair is white, then a corresponding white node is added to U. If one of the pair's nodes is black, then the corresponding node is created in U with the value of the other node in the pair. If both nodes of the pair are gray, then a gray node is added to U, and the algorithm is applied recursively to the pair's children. As in the union algorithm, if both nodes are gray, then after the algorithm has been applied to the children of the new node in U, the children must be inspected. In this case, if they are all white, they are deleted and their parent in U must be changed from gray to white.

It is easy to perform simple transformations on quadtrees and octrees. For example, rotation about an axis by multiples of 90° is accomplished by recursively rotating the children at each level. Scaling by powers of 2 and reflections are also straightforward. Translations are somewhat more complex, as are general transformations. In addition, as in spatial-occupancy enumeration in general, the problem of aliasing under general transformations is severe.

Neighbor finding. One important operation in quadtrees and octrees is finding a node's *neighbor*; that is, finding a node that is adjacent to the original node (sharing a face, edge, or vertex) and of equal or greater size. A quadtree node has neighbors in eight possible

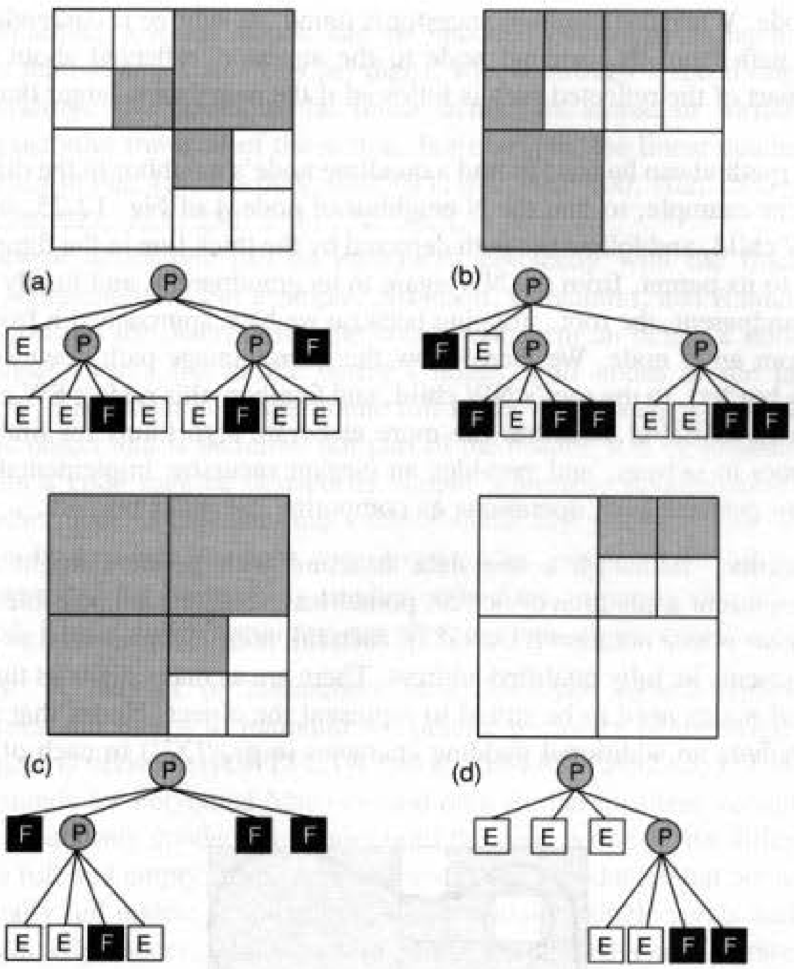


Fig. 12.24 Performing Boolean set operations on quadtrees. (a) Object *S* and its quadtree. (b) Object *T* and its quadtree. (c) $S \cup T$. (d) $S \cap T$.

directions. Its N, S, E, W neighbors are neighbors along a common edge, whereas its NW, NE, SW, and SE neighbors are neighbors along a common vertex. An octree node has neighbors in 26 possible directions: 6 neighbors along a face, 12 neighbors along an edge, and 8 neighbors along a vertex.

Samet [SAME89a] describes a way to find a node's neighbor in a specified direction. The method starts at the original node and ascends the quadtree or octree until the first common ancestor of the original node and neighbor is found. The tree is then traversed downward to find the desired neighbor. Two problems must be solved efficiently here: finding the common ancestor and determining which of its descendants is the neighbor. The simplest case is finding an octree node's neighbor in the direction *d* of one of its faces: L, R, U, D, F, or B. As we ascend the tree starting at the original node, the common ancestor will be the first node that is not reached from a child on the node's *d* side. For example, if the search is for an L neighbor, then the first common ancestor is the first node that is not reached from an LUF, LUB, LDF, or LDB child. This is true because a node that has been reached from one of these children cannot have any child that is left of (is an L neighbor of) the original node. When the common ancestor is found, its subtree is descended in a mirror image of the path from the original node to the ancestor, reflected about the common border. Only part of the reflected path is followed if the neighbor is larger than the original node.

A similar method can be used to find a quadtree node's neighbor in the direction of one of its edges. For example, to find the N neighbor of node *A* of Fig. 12.25, we begin at *A*, which is a NW child, and follow the path depicted by the thick line in the figure. We ascend from the NW to its parent, from the NW again to its grandparent, and finally from the SW to its great grandparent, the root, stopping because we have approached it from an S node, rather than from an N node. We then follow the mirror-image path downward (reflected about the N–S border), to the root's NW child, and finally to this node's SW child, which is a leaf. Samet [SAME89a] describes the more elaborate algorithms for finding edge and vertex neighbors in octrees, and provides an elegant recursive implementation that uses table lookup to perform such operations as computing the reflected path.

Linear notations. Although a tree data structure with pointers might at first seem necessary to represent a quadtree or octree, pointerless notations are possible. In the *linear quadtree* or *linear octree* notation [GARG82], each full node is represented as a sequence of digits that represents its fully qualified address. There are as many digits as there are levels. Only black leaf nodes need to be stored to represent the object. Nodes that are not at the lowest level include an additional padding character (e.g., "X") in each of their trailing

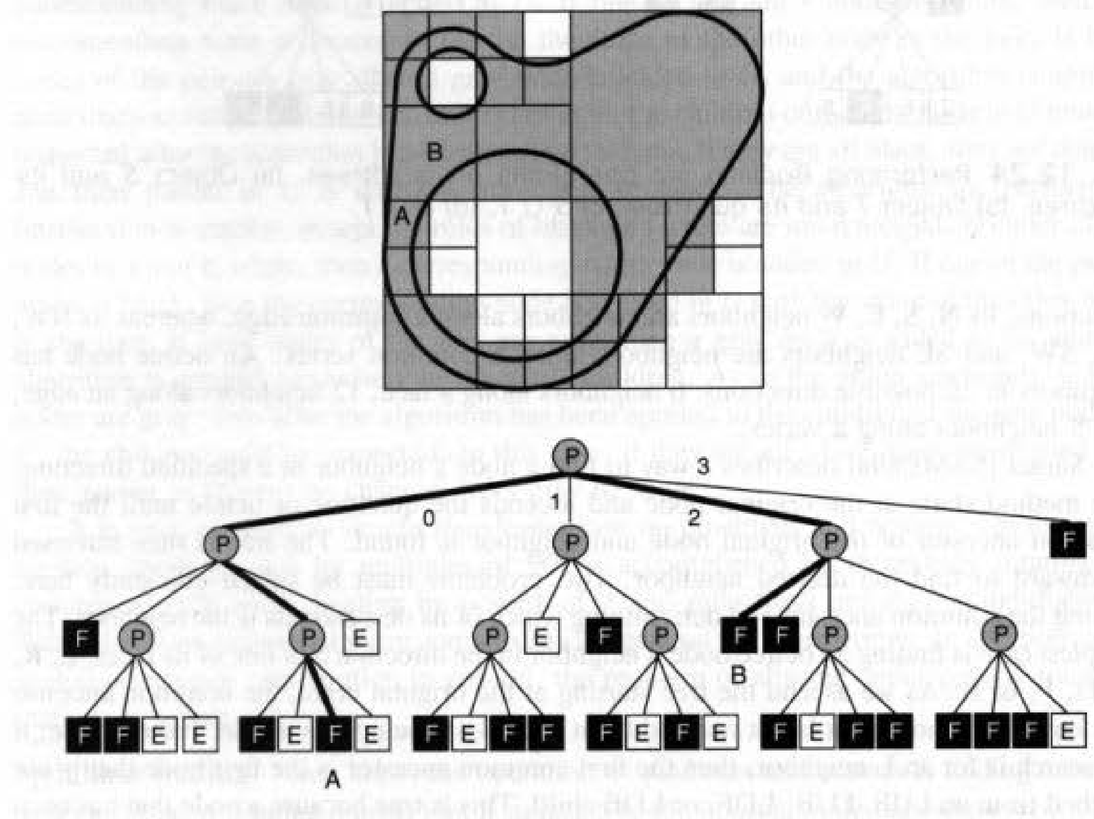


Fig. 12.25 Finding the neighbor of a quadtree node.

digits. For example, a linear octree can be encoded compactly using base-9 numbers (conveniently represented with 4 bits per digit), with 0 through 7 specifying octants and 8 indicating padding. The nodes in the linear octree are stored in sorted order, which represents a postorder traversal of the octree. For example, the linear quadtree representation of the object in Fig. 12.21 is 00X, 010, 011, 020, 022, 100, 102, 103, 12X, 130, 132, 20X, 21X, 220, 222, 223, 230, 231, 232, 3XX.

A number of operations can be performed efficiently with the linear-quadtree or linear-octree representation. For example, Atkinson, Gargantini, and Ramanath [ATKI84] present an algorithm for determining the voxels that form an octree's border by making successive passes over a list of the octree's nodes. Full nodes of the largest size are considered first. Each such node that abuts full nodes of the same size on all six sides is internal to the object and is therefore not part of the border; it is eliminated from the list. (Each neighbor's code may be derived by simple arithmetic manipulation of the node's code.) Any other node of this size may contain voxels that are part of the border; each of these nodes is broken into its eight constituent nodes, which replace it in the list. The algorithm is repeated for successively smaller node sizes, stopping after voxel-sized nodes are considered. Those nodes remaining are all the voxels on the object's border.

PM octrees. A number of researchers have developed hybrid representations that combine octrees and b-reps to maintain the precise geometry of the original b-rep from which the object is derived [HUNT81; QUIN82; AYAL85; CARL85; FUJI85]. These *PM octrees* (*PM* stands for Polygonal Map) expand on a similar quadtree variant [SAME90a]. The octree is recursively divided into nodes until the node is one of five different leaf types. In addition to full and empty, three new leaf nodes are introduced that are actually special kinds of partially full nodes: vertex nodes, which contain a single vertex and its connected faces and edges; edge nodes, which contain part of a single edge and its faces; and surface nodes, which are cut by a piece of a single face. Restricting the new leaf types to a set of simple geometries, each of which divides the node into exactly two parts, simplifies the algorithms that manipulate the representation, such as Boolean set operations [CARL87; NAVA89].

Section 18.11.4 discusses a number of architectures based on voxel and octree models. Section 15.8 discusses visible-surface algorithms for octrees.

12.6.4 Binary Space-Partitioning Trees

Octrees recursively divide space by planes that are always mutually perpendicular and that bisect all three dimensions at each level of the tree. In contrast, *binary space-partitioning* (BSP) *trees* recursively divide space into pairs of subspaces, each separated by a plane of arbitrary orientation and position. The binary-tree data structure created was originally used in determining visible surfaces in graphics, as described in Section 15.5.2. Thibault and Naylor [THIB87] later introduced the use of BSP trees to represent arbitrary polyhedra. Each internal node of the BSP tree is associated with a plane and has two child pointers, one for each side of the plane. Assuming that normals point out of an object, the left child is behind or inside the plane, whereas the right child is in front of or outside the plane. If the half-space on a side of the plane is subdivided further, then its child is the root of a subtree; if the half-space is homogeneous, then its child is a leaf, representing a region either entirely inside or entirely outside the polyhedron. These homogeneous regions are called "in" cells and "out" cells. To account for the limited numerical precision with which operations are performed, each node also has a "thickness" associated with its plane. Any point lying within this tolerance of the plane is considered to be "on" the plane.

The subdivision concept behind BSP trees, like that underlying octrees and quadtrees, is dimension-independent. Thus, Fig. 12.26(a) shows a concave polygon in 2D, bordered by black lines. "In" cells are shaded light gray, and the lines defining the half-spaces are shown in dark gray, with normals pointing to the outside. The corresponding BSP tree is shown in Fig. 12.26(b). In 2D, the "in" and "out" regions form a convex polygonal tessellation of the plane; in 3D, the "in" and "out" regions form a convex polyhedral tessellation of 3-space. Thus, a BSP tree can represent an arbitrary concave solid with holes as a union of convex "in" regions. Unlike octrees, but like b-reps, an arbitrary BSP tree does not necessarily represent a bounded solid. For example, the 3D BSP tree consisting of a single internal node, with "in" and "out" nodes as children, defines an object that is a half-space bounded by only one plane.

Consider the task of determining whether a point lies inside, outside, or on a solid, a problem known as *point classification* [TILO80]. A BSP tree may be used to classify a point by filtering that point down the tree, beginning at the root. At each node, the point is substituted into the node's plane equation and is passed recursively to the left child if it lies behind (inside) the plane, or to the right child if it lies in front of (outside) the plane. If the node is a leaf, then the point is given the leaf's value, either "out" or "in." If the point lies on a node's plane, then it is passed to both children, and the classifications are compared. If they are the same, then the point receives that value; if they are different, then the point lies on the boundary between "out" and "in" regions and is classified as "on." This approach can be extended to classify lines and polygons. Unlike a point, however, a line or polygon may lie partially on both sides of a plane. Therefore, at each node whose plane intersects the line or polygon, the line or polygon must be divided (clipped) into those parts that are in front of, in back of, or on the plane, and the parts classified separately.

Thibault and Naylor describe algorithms for building a BSP tree from a b-rep, for performing Boolean set operations to combine a BSP tree with a b-rep, and for determining those polygonal pieces that lie on a BSP tree's boundary [THIB87]. These algorithms operate on BSP trees whose nodes are each associated with a list of polygons embedded in

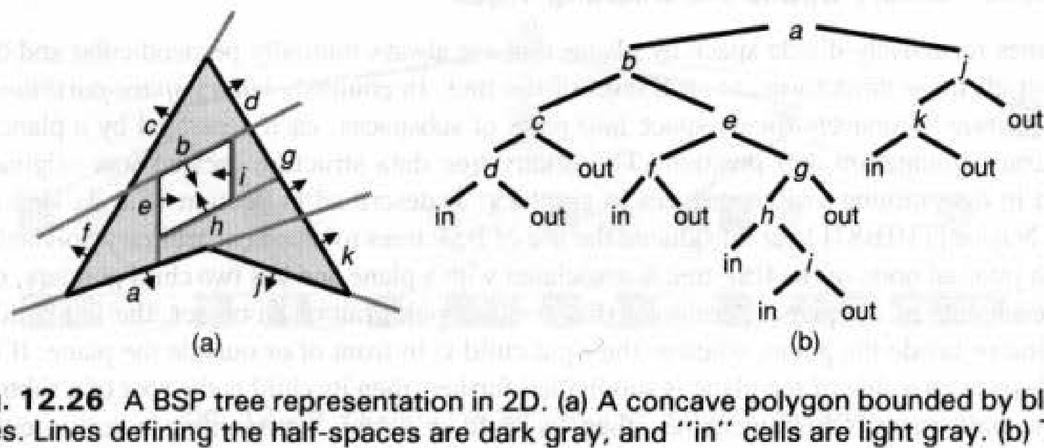


Fig. 12.26 A BSP tree representation in 2D. (a) A concave polygon bounded by black lines. Lines defining the half-spaces are dark gray, and "in" cells are light gray. (b) The BSP tree.

the node's plane. Polygons are inserted into the tree using a variant of the BSP tree building algorithm presented in Section 15.5.2.

Although BSP trees provide an elegant and simple representation, polygons are subdivided as the tree is constructed and as Boolean set operations are performed, making the notation potentially less compact than other representations. By taking advantage of the BSP tree's inherent dimension-independence, however, we can develop a closed Boolean algebra for 3D BSP trees that recursively relies on representing polygons as 2D trees, edges as 1D trees, and points as 0D trees [NAYL90].

12.7 CONSTRUCTIVE SOLID GEOMETRY

In *constructive solid geometry* (CSG), simple primitives are combined by means of regularized Boolean set operators that are included directly in the representation. An object is stored as a tree with operators at the internal nodes and simple primitives at the leaves (Fig. 12.27). Some nodes represent Boolean operators, whereas others perform translation, rotation, and scaling, much like the hierarchies of Chapter 7. Since Boolean operations are not, in general, commutative, the edges of the tree are ordered.

To determine physical properties or to make pictures, we must be able to combine the properties of the leaves to obtain the properties of the root. The general processing strategy is a depth-first tree walk, as in Chapter 7, to combine nodes from the leaves on up the tree. The complexity of this task depends on the representation in which the leaf objects are stored and on whether a full representation of the composite object at the tree's root must actually be produced. For example, the regularized Boolean set operation algorithms for b-reps, discussed in Section 12.5, combine the b-reps of two nodes to create a third b-rep and are difficult to implement. The much simpler CSG algorithm discussed in Section 15.10.3, on the other hand, produces a picture by processing the representations of the leaves without explicitly combining them. Other algorithms for creating pictures of CSG representations include [ATHE83; OKIN84; JANS85]; architectures that support CSG are discussed in Sections 18.9.2, 18.10.2 and 18.11.4.

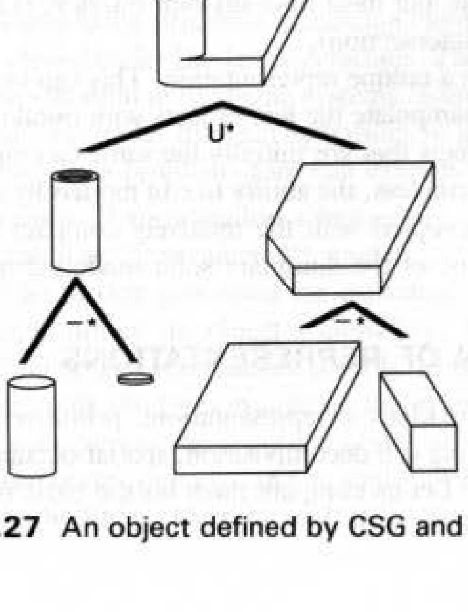


Fig. 12.27 An object defined by CSG and its tree.

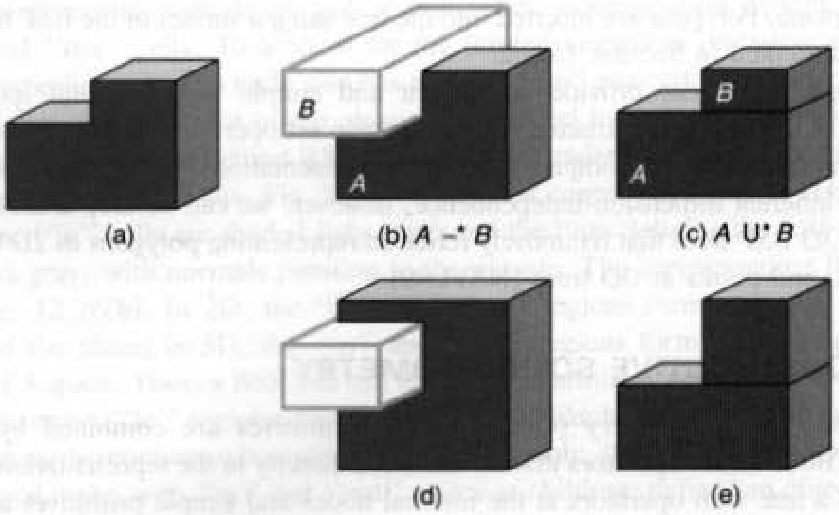


Fig. 12.28 The object shown in (a) may be defined by different CSG operations, as shown in (b) and (c). Tweaking the top face of (b) and (c) upward yields different objects, shown in (d) and (e).

In some implementations, the primitives are simple solids, such as cubes or spheres, ensuring that all regularized combinations are valid solids as well. In other systems, primitives include half-spaces, which themselves are not bounded solids. For example, a cube can be defined as the intersection of six half-spaces, or a finite cylinder as an infinite cylinder that is capped off at the top and bottom by planar half-spaces. Using half-spaces introduces a validity problem, since not all combinations produce solids. Half-spaces are useful, however, for operations such as slicing an object by a plane, which might otherwise be performed by using the face of another solid object. Without half-spaces, extra overhead is introduced, since the regularized Boolean set operations must be performed with the full object doing the slicing, even if only a single slicing face is of interest.

We can think of the cell-decomposition and spatial-occupancy enumeration techniques as special cases of CSG in which the only operator is the implicit glue operator: the union of two objects that may touch, but must have disjoint interiors (i.e., the objects must have a null regularized Boolean intersection).

CSG does not provide a unique representation. This can be particularly confusing in a system that lets the user manipulate the leaf objects with tweaking operators. Applying the same operation to two objects that are initially the same can yield two different results, as shown in Fig. 12.28. Nevertheless, the ability to edit models by deleting, adding, replacing, and modifying subtrees, coupled with the relatively compact form in which models are stored, have made CSG one of the dominant solid modeling representations.

12.8 COMPARISON OF REPRESENTATIONS

We have discussed five main kinds of representations: primitive instancing, sweeps, b-reps, spatial partitioning (including cell decomposition, spatial-occupancy enumeration, octrees, and BSP trees), and CSG. Let us compare them on the basis of the criteria introduced in Section 12.1.

- *Accuracy*. Spatial-partitioning and polygonal b-rep methods produce only approximations for many objects. In some applications, such as finding a path for a robot, this is not a drawback, as long as the approximation is computed to an adequate (often relatively coarse) resolution. The resolution needed to produce visually pleasing graphics or to calculate object interactions with sufficient accuracy, however, may be too high to be practical. The smooth shading techniques discussed in Chapter 16 do not fix the visual artifacts caused by the all-too-obvious polygonal edges. Therefore, systems that support high-quality graphics often use CSG with nonpolyhedral primitives and b-reps that allow curved surfaces. Primitive instancing also can produce high-quality pictures, but does not allow two simpler objects to be combined with Boolean set operators.
- *Domain*. The domain of objects that can be represented by both primitive instancing and sweeps is limited. In comparison, spatial-partitioning approaches can represent any solid, although often only as an approximation. By providing other kinds of faces and edges in addition to polygons bounded by straight lines, b-reps can be used to represent a very wide class of objects. Many b-rep systems, however, are restricted to simple surface types and topologies. For example, they may be able to encode only combinations of quadrics that are 2-manifolds.
- *Uniqueness*. Only octree and spatial-occupancy–enumeration approaches guarantee the uniqueness of a representation: There is only one way to represent an object with a specified size and position. In the case of octrees, some processing must be done to ensure that the representation is fully reduced (i.e., that no gray node has all black children or all white children). Primitive instancing does not guarantee uniqueness in general: for example, a sphere may be represented by both a spherical and an elliptical primitive. If the set of primitives is chosen carefully, however, uniqueness can be ensured.
- *Validity*. Among all the representations, b-reps stand out as being the most difficult to validate. Not only may vertex, edge, and face data structures be inconsistent, but also faces or edges may intersect. In contrast, any BSP tree represents a valid spatial set, but not necessarily a bounded solid. Only simple local syntactic checking needs to be done to validate a CSG tree (which is always bounded, if its primitives are bounded) or an octree, and no checking is needed for spatial-occupancy enumeration.
- *Closure*. Primitives created using primitive instancing cannot be combined at all, and simple sweeps are not closed under Boolean operations. Therefore, neither is typically used as an internal representation in modeling systems. Although particular b-reps may suffer from closure problems under Boolean operations (e.g., the inability to represent other than 2-manifolds), these problem cases can often be avoided.
- *Compactness and efficiency*. Representation schemes are often classified by whether they produce "evaluated" or "unevaluated" models. *Unevaluated* models contain information that must be further processed (or evaluated) in order to perform basic operations, such as determining an object's boundary. With regard to the use of Boolean operations, CSG creates unevaluated models, in that each time computations are performed, we must walk the tree, evaluating the expressions. Consequently, the advantages of CSG are its compactness and the ability to record Boolean operations and changes of transformations quickly, and to undo all of these quickly since they involve only tree-node building. Octrees and BSP trees can also be considered

unevaluated models, as can a sequence of Euler operators that creates a b-rep. B-reps and spatial-occupancy enumeration, on the other hand, are often considered *evaluated* models insofar as any Boolean operations used to create an object have already been performed. Note that the use of these terms is relative; if the operation to be performed is determining whether a point is inside an object, for example, more work may be done evaluating a b-rep than evaluating the equivalent CSG tree.

As discussed in Chapter 15, a number of efficient algorithms exist for generating pictures of objects encoded using b-reps and CSG. Although spatial-occupancy enumeration and octrees can provide only coarse approximations for most objects, the algorithms used to manipulate them are in general simpler than the equivalents for other representations. They have thus been used in hardware-based solid modeling systems intended for applications in which the increased speed with which Boolean set operations can be performed on them outweighs the coarseness of the resulting images.

Some systems use multiple representations because some operations are more efficient with one representation than with another. For example, GMSOLID [BOYS82] uses CSG for compactness and a b-rep for quick retrieval of useful data not explicitly specified in CSG, such as connectivity. Although GMSOLID's CSG representation always reflects the current state of the object being modeled, its b-rep is updated only when the operations that require it are executed. Updating may be done by a background process, so that the user can perform other operations while waiting for the result. In addition to systems that maintain two completely separate representations, deriving one from the other when needed, there are also hybrid systems that go down to some level of detail in one scheme, then switch to another, but never duplicate information. The PM octrees discussed in Section 12.6.3 that combine octrees with b-reps provide examples. Some of the issues raised by the use of multiple representations and hybrid representations are addressed in [MILL89].

It is relatively easy to convert all the representations we have discussed to spatial-occupancy–enumeration or octree representations, but only as approximations. Such conversions are not invertible, because they lose information. In addition, it is easy to convert all representations exactly to b-reps and PM octrees. An algorithm for performing Boolean operations on b-reps, such as the one described in Section 12.5, can be used to convert CSG to b-rep by successive application to each level of the CSG tree, beginning with the polyhedral descriptions of the leaf primitives. Rossignac and Voelcker [ROSS89] have implemented an efficient CSG–to–b-rep conversion algorithm that identifies what they call the *active zone* of a CSG node—that part of the node that, if changed, will affect the final solid; only the parts of a solid within a node's active zone need be considered when Boolean operations are performed. On the other hand, conversion from b-rep into CSG is difficult, especially if an encoding into a minimal number of CSG operations is desired. Vossler [VOSS85b] describes a method for converting sweeps to CSG by automatically recognizing patterns of simpler sweeps that can be combined with Boolean operations to form the more complex sweep being converted.

As pointed out in Section 12.1, wireframe representations containing only vertex and edge information, with no reference to faces, are inherently ambiguous. Markowsky and Wesley, however, have developed an algorithm for deriving all polyhedra that could be represented by a given wireframe [MARK80] and a companion algorithm that generates all polyhedra that could produce a given 2D projection [WESL81].

12.9 USER INTERFACES FOR SOLID MODELING

Developing the user interface for a solid modeling system provides an excellent opportunity to put into practice the interface design techniques discussed in Chapter 9. A variety of techniques lend themselves well to graphical interfaces, including the direct application of regularized Boolean set operators, tweaking, and Euler operators. In CSG systems the user may be allowed to edit the object by modifying or replacing one of the leaf solids or subtrees. Blending and chamfering operations may be defined to smooth the transition from one surface to another. The user interfaces of successful systems are largely independent of the internal representation chosen. Primitive instancing is an exception, however, since it encourages user to think of objects in terms of special-purpose parameters.

In Chapter 11, we noted that there are many equivalent ways to describe the same curve. For example, the user interface to a curve-drawing system can let the user enter curves by controlling Hermite tangent vectors or by specifying Bezier control points, while storing curves only as Bezier control points. Similarly, a solid modeling system may let the user create objects in terms of several different representations, while storing them in yet another. As with curve representations, each different input representation may have some expressive advantage that makes it a natural choice for creating the object. For example, a b-rep system may allow an object to be defined as a translational or rotational sweep. The user interface may also provide different ways to define the same object within a single representation. For example, two of the many ways to define a sphere are to specify its center and a point on its surface, or to specify the two endpoints of a diameter. The first may be more useful for centering a sphere at a point, whereas the second may be better for positioning the sphere between two supports.

The precision with which objects must be specified often dictates that some means be provided to determine measurements accurately; for example, through a locator device or through numeric entry. Because the position of one object often depends on those of others, interfaces often provide the ability to constrain one object by another. A related technique is to give the user the ability to define grid lines to constrain object positions, as discussed in Section 8.2.1.

Some of the most fundamental problems of designing a solid modeling interface are those caused by the need to manipulate and display 3D objects with what are typically 2D interaction devices and displays. These general issues were discussed in more detail in Chapters 8 and 9. Many systems address some of these problems by providing multiple display windows that allow the user to view the object simultaneously from different positions.

12.10 SUMMARY

As we have seen, solid modeling is important in both CAD/CAM and graphics. Although useful algorithms and systems exist that handle the objects described so far, many difficult problems remain unsolved. One of the most important is the issue of robustness. Solid modeling systems are typically plagued by numerical instabilities. Commonly used algorithms require more precision to hold intermediate floating-point results than is available in hardware. For example, Boolean set operation algorithms may fail when presented with two objects, one of which is a very slightly transformed copy of the first.

Representations are needed for nonrigid, flexible, jointed objects. Work on transformations that bend and twist objects is described in Chapter 20. Many objects cannot be specified with total accuracy; rather, their shapes are defined by parameters constrained to lie within a range of values. These are known as "toleranced" objects, and correspond to real objects turned out by machines such as lathes and stampers [REQU84]. New representations are being developed to encode toleranced objects [GOSS88].

Common to all designed objects is the concept of "features," such as holes and chamfers, that are designed for specific purposes. One current area of research is exploring the possibility of recognizing features automatically and inferring the designer's intent for what each feature should accomplish [PRAT84]. This will allow the design to be checked to ensure that the features perform as intended. For example, if certain features are designed to give a part strength under pressure, then their ability to perform this function could be validated automatically. Future operations on the object could also be checked to ensure that the features' functionality was not compromised.

EXERCISES

12.1 Define the results of performing $\cup^*$ and $-^*$ for two polyhedral objects in the same way as the result of performing $\cap^*$ was defined in Section 12.2. Explain how the resulting object is constrained to be a regular set, and specify how the normal is determined for each of the object's faces.

12.2 Consider the task of determining whether or not a legal solid is the null object (which has no volume). How difficult is it to perform this test in each of the representations discussed?

12.3 Consider a system whose objects are represented as sweeps and can be operated on using the regularized Boolean set operators. What restrictions must be placed on the objects to ensure closure?

12.4 Implement the algorithms for performing Boolean set operations on quadtrees or on octrees.

12.5 Explain why an implementation of Boolean set operations on quadtrees or octrees does not need to address the distinction between the ordinary and regularized operations described in Section 12.2.

12.6 Although the geometric implications of applying the regularized Boolean set operators are unambiguous, it is less clear how object properties should be treated. For example, what properties should be assigned to the intersection of two objects made of different materials? In modeling actual objects, this question is of little importance, but in the artificial world of graphics, it is possible to intersect any two materials. What solutions do you think would be useful?

12.7 Explain how a quadtree or octree could be used to speed up 2D or 3D picking in a graphics package.

12.8 Describe how to perform point classification in primitive instancing, b-rep, spatial occupancy enumeration, and CSG.

13 Achromatic and Colored Light

The growth of raster graphics has made color and gray scale an integral part of contemporary computer graphics. Color is an immensely complex subject, one that draws on concepts and results from physics, physiology, psychology, art, and graphic design. Many researchers' careers have been fruitfully devoted to developing theories, measurement techniques, and standards for color. In this chapter, we introduce some of the areas of color that are most relevant to computer graphics.

The color of an object depends not only on the object itself, but also on the light source illuminating it, on the color of the surrounding area, and on the human visual system. Furthermore, some objects reflect light (wall, desk, paper), whereas others also transmit light (cellophane, glass). When a surface that reflects only pure blue light is illuminated with pure red light, it appears black. Similarly, a pure green light viewed through glass that transmits only pure red will also appear black. We postpone some of these issues by starting our discussion with achromatic sensations—that is, those described as black, gray, and white.

13.1 ACHROMATIC LIGHT

Achromatic light is what we see on a black-and-white television set or display monitor. An observer of achromatic light normally experiences none of the sensations we associate with red, blue, yellow, and so on. Quantity of light is the only attribute of achromatic light. Quantity of light can be discussed in the physics sense of energy, in which case the terms *intensity* and *luminance* are used, or in the psychological sense of perceived intensity, in which case the term *brightness* is used. As we shall discuss shortly, these two concepts are related but are not the same. It is useful to associate a scalar with different intensity levels, defining 0 as black and 1 as white; intensity levels between 0 and 1 represent different grays.

A black-and-white television can produce many different intensities at a single pixel position. Line printers, pen plotters, and electrostatic plotters produce only two levels: the white (or light gray) of the paper and the black (or dark gray) of the ink or toner deposited on the paper. Certain techniques, discussed in later sections, allow such inherently *bilevel* devices to produce additional intensity levels.

13.1.1 Selecting Intensities—Gamma Correction

Suppose we want to display 256 different intensities. Which 256 intensity levels should we use? We surely do not want 128 in the range of 0 to 0.1 and 128 more in the range of 0.9 to 1.0, since the transition from 0.1 to 0.9 would certainly appear discontinuous. We might initially distribute the levels evenly over the range 0 to 1, but this choice ignores an important characteristic of the eye: that it is sensitive to ratios of intensity levels rather than to absolute values of intensity. That is, we perceive the intensities 0.10 and 0.11 as differing just as much as the intensities 0.50 and 0.55. (This nonlinearity is easy to observe: Cycle through the settings on a three-way 50–100–150-watt lightbulb; you will see that the step from 50 to 100 seems much greater than the step from 100 to 150.) On a brightness (that is, perceived intensity) scale, the differences between intensities of 0.10 and 0.11 and between intensities of 0.50 and 0.55 are equal. Therefore, the intensity levels should be spaced logarithmically rather than linearly, to achieve equal steps in brightness.

To find 256 intensities starting with the lowest attainable intensity I_0 and going to a maximum intensity of 1.0, with each intensity r times higher than the preceding intensity, we use the following relations:

$$I_0 = I_0,\ I_1 = rI_0,\ I_2 = rI_1 = r^2I_0,\ I_3 = rI_2 = r^3I_0,\ \ldots,\ I_{255} = r^{255}I_0 = 1. \quad (13.1)$$

Therefore,

$$r = (1/I_0)^{1/255},\ I_j = r^jI_0 = (1/I_0)^{j/255}\ I_0 = I_0^{(255-j)/255} \qquad \text{for } 0 \le j \le 255, \quad (13.2)$$

and in general for $n + 1$ intensities,

$$r = (1/I_0)^{1/n},\ I_j = I_0^{(n-j)/n} \qquad \text{for } 0 \le j \le n. \quad (13.3)$$

With just four intensities ($n = 3$) and an I_0 of $\frac{1}{8}$ (an unrealistically large value chosen for illustration only), Eq. (13.3) tells us that $r = 2$, yielding intensity values of $\frac{1}{8}$, $\frac{1}{4}$, $\frac{1}{2}$, and 1.

The minimum attainable intensity I_0 for a CRT is anywhere from about $\frac{1}{200}$ up to $\frac{1}{40}$ of the maximum intensity of 1.0. Therefore, typical values of I_0 are between 0.005 and 0.025. The minimum is not 0, because of light reflection from the phosphor within the CRT. The ratio between the maximum and minimum intensities is called the *dynamic range*. The exact value for a specific CRT can be found by displaying a square of white on a field of black and measuring the two intensities with a photometer. This measurement is taken in a completely darkened room, so that reflected ambient light does not affect the intensities. With an I_0 of 0.02, corresponding to a dynamic range of 50, Eq. (13.2) yields $r = 1.0154595 \ldots$, and the first few and last two intensities of the 256 intensities from Eq. (13.1) are 0.0200, 0.0203, 0.0206, 0.0209, 0.0213, 0.0216, . . ., 0.9848, 1.0000.

Displaying the intensities defined by Eq. (13.1) on a CRT is a tricky process, and recording them on film is even more difficult, because of the nonlinearities in the CRT and film. For instance, the intensity of light output by a phosphor is related to the number of electrons N in the beam by

$$I = kN^{\gamma} \tag{13.4}$$

for constants k and γ. The value of γ is in the range 2.2 to 2.5 for most CRTs. The number of electrons N is proportional to the control-grid voltage, which is in turn proportional to the value V specified for the pixel. Therefore, for some other constant K,

$$I = KV^{\gamma}, \text{ or } V = (I/K)^{1/\gamma}. \tag{13.5}$$

Now, given a desired intensity I, we first determine the nearest I_j by searching through a table of the available intensities as calculated from Eq. (13.1) or from its equivalent:

$$j = ROUND(log_r(I/I_0)). \tag{13.6}$$

After j is found, we calculate

$$I_j = r^j I_0. \tag{13.7}$$

The next step is to determine the pixel value V_j needed to create the intensity I_j, by using Eq. (13.5):

$$V_j = ROUND\ ((I_j/K)^{1/\gamma}). \tag{13.8}$$

If the raster display has no look-up table, then V_j is placed in the appropriate pixels. If there is a look-up table, then j is placed in the pixel and V_j is placed in entry j of the table.

The values of K, γ, and I_0 depend on the CRT in use, so in practice the look-up table is loaded by a method based on actual measurement of intensities [CATM79; COWA83; HALL89]. Use of the look-up table in this general manner is called *gamma correction*, so named for the exponent in Eq. (13.4). If the display has hardware gamma correction, then I_j rather than V_j is placed in either the refresh buffer or look-up table.

Without the use of ratio-based intensity values and gamma correction, quantization errors (from approximating a true intensity value with a displayable intensity value) will be more conspicuous near black than near white. For instance, with 4 bits and hence 16 intensities, a quantization round-off error of as much as $\frac{1}{32} = 0.031$ is possible. This is 50 percent of intensity value $\frac{1}{16}$, and only 3 percent of intensity value 1.0. Using the ratio-based intensities and gamma correction, the maximum quantization error as a percentage of brightness (perceived intensity) is constant.

A natural question is, "How many intensities are enough?" By "enough," we mean the number needed to reproduce a continuous-tone black-and-white image such that the reproduction appears to be continuous. This appearance is achieved when the ratio r is 1.01 or less (below this ratio, the eye cannot distinguish between intensities I_j and I_{j+1}) [WYSZ82, p. 569]. Thus, the appropriate value for n, the number of intensity levels, is found by equating r to 1.01 in Eq. (13.3):

$$r = (1/I_0)^{1/n} \quad \text{or} \quad 1.01 = (1/I_0)^{1/n}. \tag{13.9}$$

TABLE 13.1 DYNAMIC RANGE ($1/I_0$) AND NUMBER OF REQUIRED INTENSITIES $n = \log_{1.01}(1/I_0)$ FOR SEVERAL DISPLAY MEDIA

Display media	Typical dynamic range	Number of intensities, n
CRT	50–200	400–530
Photographic prints	100	465
Photographic slides	1000	700
Coated paper printed in B/W*	100	465
Coated paper printed in color	50	400
Newsprint printed in B/W	10	234

*B/W = black and white.

Solving for n gives

$$n = \log_{1.01}(1/I_0), \tag{13.10}$$

where $1/I_0$ is the dynamic range of the device.

The dynamic range $1/I_0$ for several display media, and the corresponding n, which is the number of intensity levels needed to maintain $r = 1.01$ and at the same time to use the full dynamic range, are shown in Table 13.1. These are theoretical values, assuming perfect reproduction processes. In practice, slight blurring due to ink bleeding and small amounts of random noise in the reproduction decreases n considerably for print media. For instance, Fig. 13.1 shows a continuous-tone photograph; and the succeeding five Figs. 13.2 to 13.6 reproduce the same photograph at 4, 8, 16, 32, and 64 intensity levels. With four and eight levels, the transitions or contours between one intensity level and the next are quite conspicuous, because the ratio r between successive intensities is considerably greater that the ideal 1.01. Contouring is barely detectable with 32 levels, and for these particular

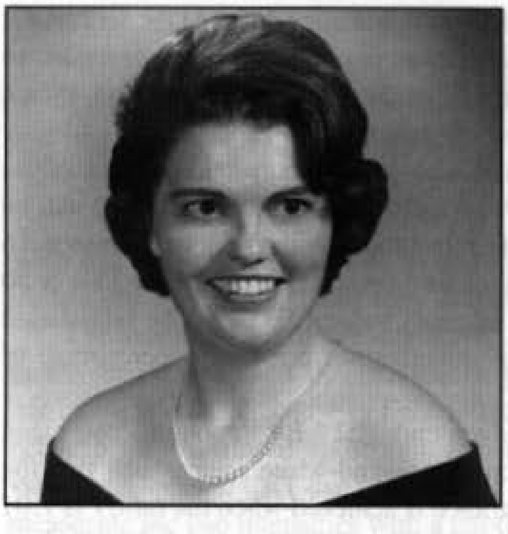


Fig. 13.1 A continuous-tone photograph.

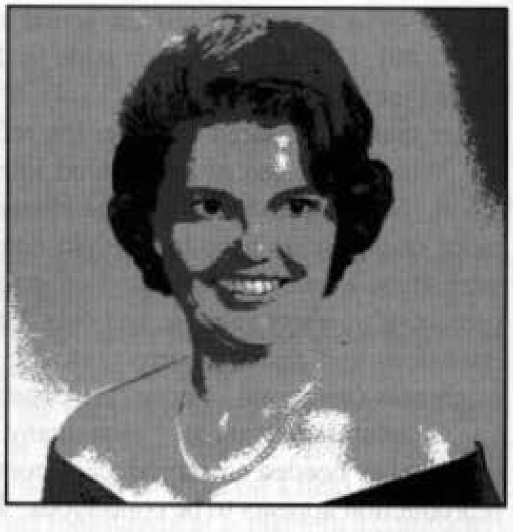


Fig. 13.2 A continuous-tone photograph reproduced with four intensity levels. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

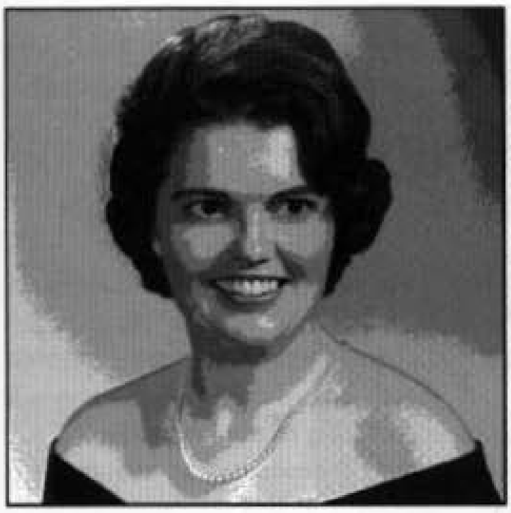


Fig. 13.3 A continuous-tone photograph reproduced with eight intensity levels. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

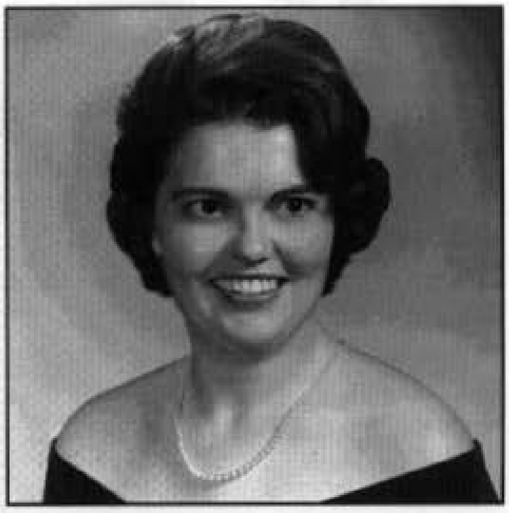


Fig. 13.4 A continuous-tone photograph reproduced with 16 intensity levels. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

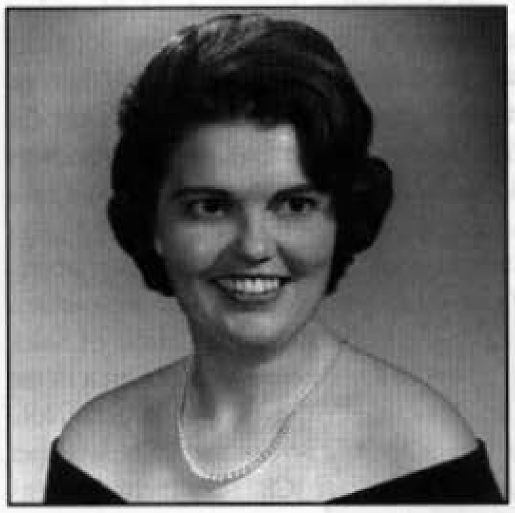


Fig. 13.5 A continuous-tone photograph reproduced with 32 intensity levels. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

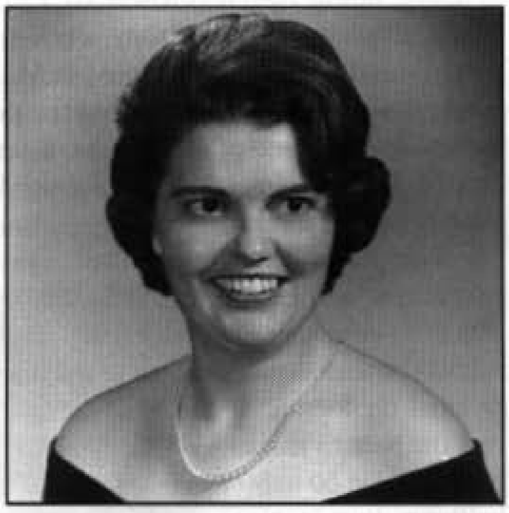


Fig. 13.6 A continuous-tone photograph reproduced with 64 intensity levels. Differences from the picture in Fig. 13.5 are quite subtle. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

images disappears with 64. This suggests that 64 intensity levels is the absolute minimum needed for contour-free printing of continuous-tone black-and-white images on paper such as is used in this book. For a well-adjusted CRT in a perfectly black room, however, the higher dynamic range means that many more levels are demanded.

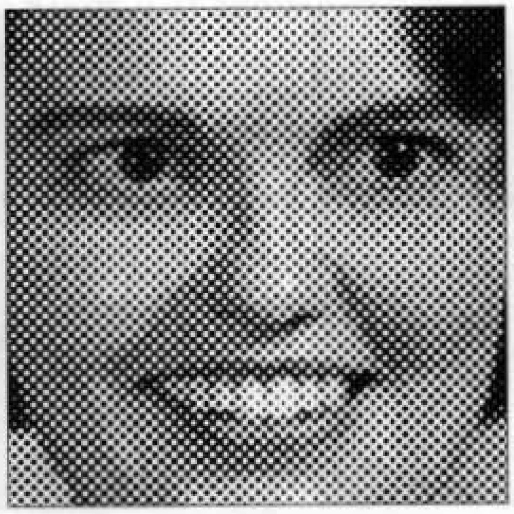


Fig. 13.7 Enlarged halftone pattern. Dot sizes vary inversely with intensity of original photograph. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

13.1.2 Halftone Approximation

Many displays and hardcopy devices are bilevel—they produce just two intensity levels—and even 2- or 3-bit-per-pixel raster displays produce fewer intensity levels than we might desire. How can we expand the range of available intensities? The answer lies in the *spatial integration* that our eyes perform. If we view a very small area from a sufficiently large viewing distance, our eyes average fine detail within the small area and record only the overall intensity of the area.

This phenomenon is exploited in printing black-and-white photographs in newspapers, magazines, and books, in a technique called *halftoning* (also called *clustered-dot ordered dither*[1] in computer graphics). Each small resolution unit is imprinted with a circle of black ink whose area is proportional to the blackness $1 - I$ (where I = intensity) of the area in the original photograph. Figure 13.7 shows part of a halftone pattern, greatly enlarged. Note that the pattern makes a 45° angle with the horizontal, called the *screen angle*. Newspaper halftones use 60 to 80 variable-sized and variable-shaped areas [ULIC87] per inch, whereas halftones in magazines and books use 110 to 200 per inch.

Graphics output devices can approximate the variable-area circles of halftone reproduction. For example, a 2×2 pixel area of a bilevel display can be used to produce five different intensity levels at the cost of halving the spatial resolution along each axis. The patterns shown in Fig. 13.8 can be used to fill the 2×2 areas with the number of "on" pixels that is proportional to the desired intensity. Figure 13.9 shows a face digitized as a 351×351 image array and displayed with 2×2 patterns.

An $n \times n$ group of bilevel pixels can provide $n^2 + 1$ intensity levels. In general, there is a tradeoff between spatial resolution and intensity resolution. The use of a 3×3 pattern

[1]The "ordered dither" contrasts with "random dither," an infrequently used technique.

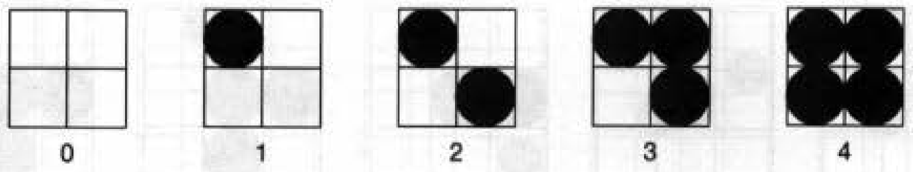


Fig. 13.8 Five intensity levels approximated with four 2 × 2 dither patterns.

cuts spatial resolution by one-third on each axis, but provides 10 intensity levels. Of course, the tradeoff choices are limited by our visual acuity (about 1 minute of arc in normal lighting), the distance from which the image is viewed, and the dots-per-inch resolution of the graphics device.

One possible set of patterns for the 3 × 3 case is shown in Fig. 13.10. Note that these patterns can be represented by the *dither matrix*

$$\begin{bmatrix} 6 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 2 & 7 \end{bmatrix}. \tag{13.11}$$

To display an intensity I, we turn on all pixels whose values are less than I.

The $n \times n$ pixel patterns used to approximate the halftones must be designed not to introduce visual artifacts in an area of identical intensity values. For instance, if the pattern in Fig. 13.11 were used, rather than the one in Fig. 13.10, horizontal lines would appear in any large area of the image of intensity 3. Second, the patterns must form a *growth sequence* so that any pixel intensified for intensity level j is also intensified for all levels $k > j$. This minimizes the differences in the patterns for successive intensity levels, thereby minimizing the contouring effects. Third, the patterns must grow outward from the center, to create the effect of increasing dot size. Fourth, for hardcopy devices such as laser printers and film recorders that are poor at reproducing isolated "on" pixels, all pixels that are "on" for a

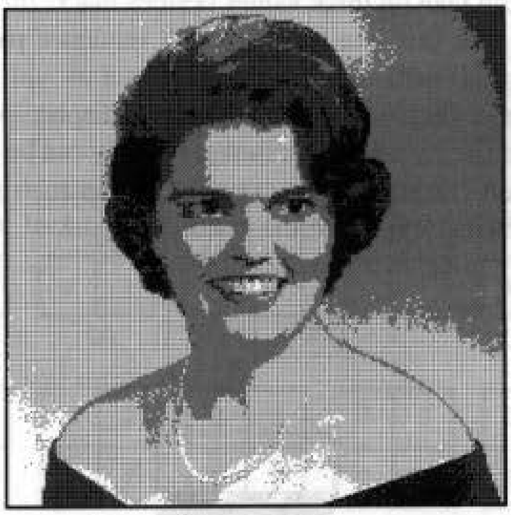


Fig. 13.9 A continuous-tone photograph, digitized to a resolution of 351 × 351 and displayed using the 2 × 2 patterns of Fig. 13.8. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

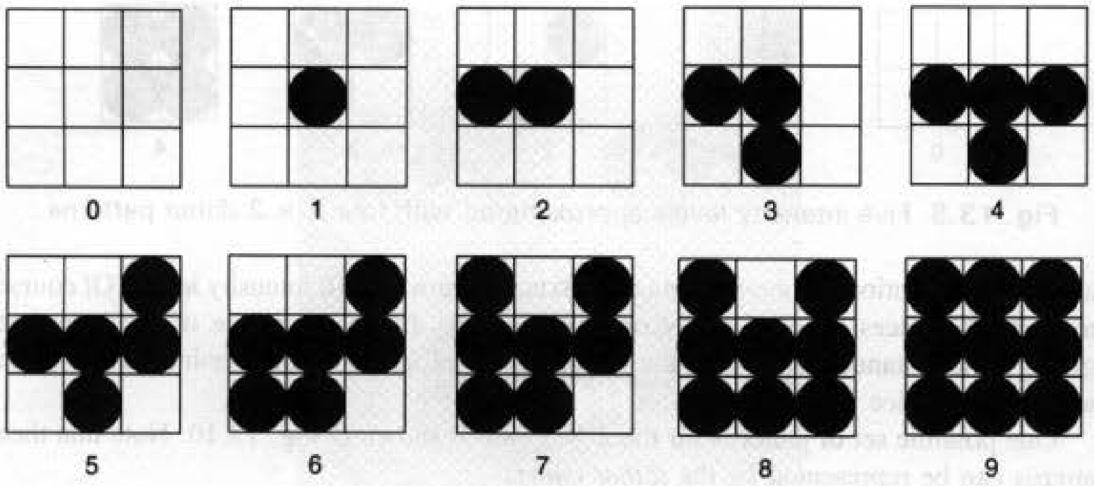


Fig. 13.10 Ten intensity levels approximated with 3 × 3 dither patterns.

particular intensity must be adjacent to other "on" pixels; a pattern such as that in Fig. 13.12 is not acceptable. This is the meaning of the term *clustered-dot* in "clustered-dot ordered dither." Holladay [HOLL80] has developed a widely used method for defining dither matrices for clustered-dot ordered dither. For high-quality reproduction of images, n must be 8 to 10, theoretically allowing 65 to 101 intensity levels. To achieve an effect equivalent to the 150-circle-per-inch printing screen, we thus require a resolution of from $150 \times 8 = 1200$ up to $150 \times 10 = 1500$ pixels per inch. High-quality film recorders can attain this resolution, but cannot actually show all the intensity levels because patterns made up of single black or white pixels may disappear.

Halftone approximation is not limited to bilevel displays. Consider a display with 2 bits per pixel and hence four intensity levels. The halftone technique can be used to increase the number of intensity levels. If we use a 2 × 2 pattern, we have a total of 4 pixels at our disposal, each of which can take on three values besides black; this allows us to display $4 \times 3 + 1 = 13$ intensities. One possible set of growth sequence patterns in this situation is shown in Fig. 13.13.

Unlike a laser printer, a CRT display is quite able to display individual dots. Hence, the clustering requirement on the patterns discussed previously can be relaxed and *dispersed-dot ordered dither* can be used. There are many possible dither matrices: Bayer [BAYE73] has developed dither matrices that minimize the texture introduced into the displayed images. For the 2 × 2 case, the dither matrix, called $D^{(2)}$, is

$$D^{(2)} = \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}. \tag{13.12}$$

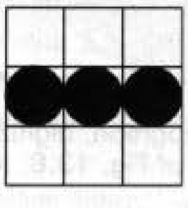


Fig. 13.11 A 3 × 3 dither pattern inappropriate for halftoning.

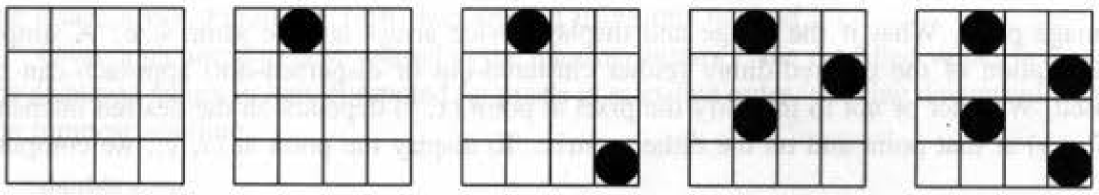


Fig. 13.12 Part of a 4 × 4 ordered dither dot pattern in which several of the patterns have single, nonadjacent dots. Such disconnected patterns are unacceptable for halftoning on laser printers and for printing presses.

This represents the set of patterns of Figure 13.8.

Larger dither matrices can be found using a recurrence relation [JUDI74] to compute $D^{(2n)}$ from $D^{(n)}$. With $U^{(n)}$ defined as an $n \times n$ matrix of 1s, that is,

$$U^{(n)} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 1 & 1 & \cdots & 1 \\ 1 & 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 1 & 1 & \cdots & 1 \end{bmatrix}, \tag{13.13}$$

the recurrence relation is

$$D^{(n)} = \begin{bmatrix} 4D^{(n/2)} + D^{(2)}_{00}U^{(n/2)} & 4D^{(n/2)} + D^{(2)}_{01}U^{(n/2)} \\ 4D^{(n/2)} + D^{(2)}_{10}U^{(n/2)} & 4D^{(n/2)} + D^{(2)}_{11}U^{(n/2)} \end{bmatrix}. \tag{13.14}$$

Applying this relation to $D^{(2)}$ produces

$$D^{(4)} = \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix}. \tag{13.15}$$

The techniques presented thus far have assumed that the image array being shown is smaller than the display device's pixel array, so that multiple display pixels can be used for one

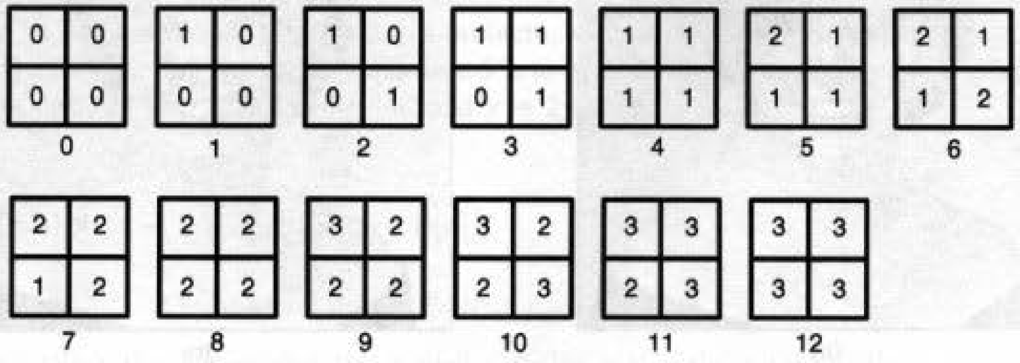


Fig. 13.13 Dither patterns for intensity levels 0 to 13 approximated using 2 × 2 patterns of pixels, with 2 bits (four intensity levels) per pixel. The intensities of the individual pixels sum to the intensity level for each pattern.

image pixel. What if the image and display device arrays are the same size? A simple adaptation of the ordered-dither (either clustered-dot or dispersed-dot) approach can be used. Whether or not to intensify the pixel at point (x, y) depends on the desired intensity $S(x, y)$ at that point and on the dither matrix. To display the point at (x, y), we compute

$$i = x \text{ modulo } n, \quad j = y \text{ modulo } n. \tag{13.16}$$

Then, if

$$S(x, y) > D_{ij}^{(n)}, \tag{13.17}$$

the point at (x, y) is intensified; otherwise, it is not. That is, 1 pixel in the image array controls 1 pixel in the display array. Notice that large areas of fixed image intensity are displayed exactly as when the image-array size is less than the display-array size, so the effect of equal image and display arrays is apparent only in areas where intensity varies.

Figure 13.14(a) is a face digitized at 512 × 512 and shown on a 512 × 512 bilevel display using $D^{(8)}$. Compare this bilevel result to the multilevel pictures shown earlier in this section. Further pictures displayed by means of ordered dither appear in [JARV76a; JARV76b; ULIC87], where more ways to display continuous-tone images on bilevel displays are described.

Error diffusion, another way to handle the case when the image and display array sizes are equal, was developed by Floyd and Steinberg [FLOY75]; its visual results are often satisfactory. The error (i.e., the difference between the exact pixel value and the approximated value actually displayed) is added to the values of the four image-array pixels to the right of and below the pixel in question: $\frac{7}{16}$ of the error to the pixel to the right, $\frac{3}{16}$ to the pixel below and to the left, $\frac{5}{16}$ to the pixel immediately below, and $\frac{1}{16}$ to the pixel below

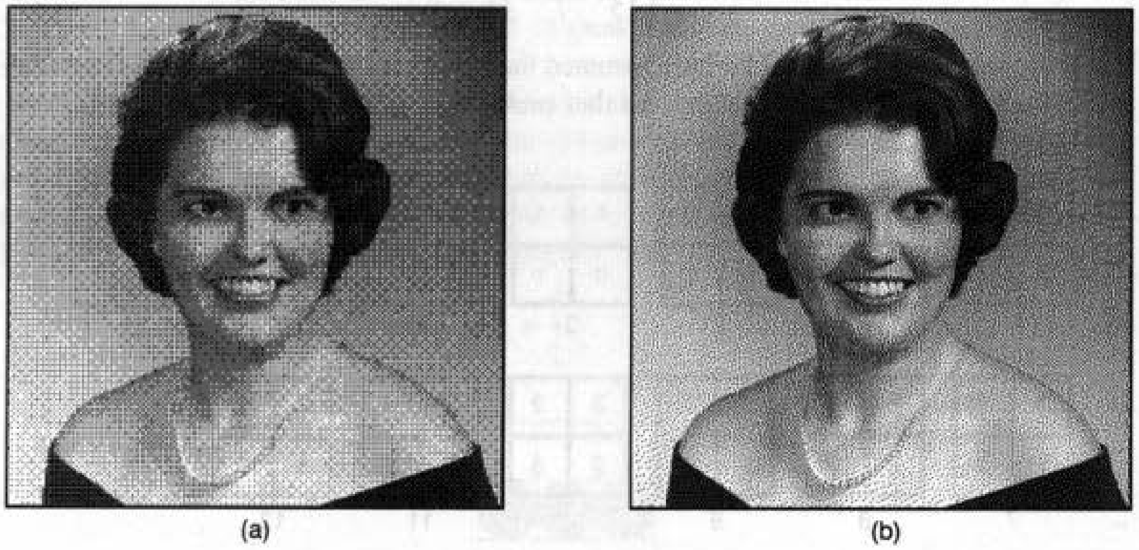


Fig. 13.14 A continuous-tone photograph reproduced with (a) $D^{(8)}$ ordered dither, and (b) Floyd-Steinberg error diffusion. (Courtesy of Alan Paeth, University of Waterloo Computer Graphics Lab.)

and to the right. This has the effect of spreading, or diffusing, the error over several pixels in the image array. Figure 13.14(b) was created using this method.

Given a picture S to be displayed in the intensity matrix I, the modified values in S and the displayed values in I are computed for pixels in scan-line order, working downward from the topmost scanline.

```
double error;
K = Approximate (S[x][y]);     /* Approximate S to nearest displayable intensity */
I[x][y] = K;                   /* Draw the pixel at (x, y). */
error = S[x][y] - K;           /* Error term */

/* Step 1: spread 7/16 of error into the pixel to the right, at (x + 1, y). */
S[x + 1][y] += 7 * error/16;

/* Step 2: spread 3/16 of error into pixel below and to the left. */
S[x - 1][y - 1] += 3 * error/16;

/* Step 3: spread 5/16 of error into pixel below. */
S[x][y - 1] += 5 * error/16;

/* Step 4: spread 1/16 of error below and to the right. */
S[x + 1][y - 1] += error/16;
```

To avoid introducing visual artifacts into the displayed image, we must ensure that the four errors sum exactly to *error*; no roundoff errors can be allowed. This can be done by calculating the step 4 error term as *error* minus the error terms from the first three steps. The function *Approximate* returns the displayable intensity value closest to the actual pixel value. For a bilevel display, the value of S is simply rounded to 0 or 1.

Even better results can be obtained by alternately scanning left to right and right to left; on a right-to-left scan, the left–right directions for errors in steps 1, 2, and 4 are reversed. For a detailed discussion of this and other error-diffusion methods, see [ULIC87]. Other approaches are discussed in [KNUT87].

Suppose the size of the image array is less than the size of the display array, the number of intensities in the image and display are equal, and we wish to display the image at the size of the display array. A simple case of this is an 8-bit-per-pixel, 512×512 image and an 8-bit-per-pixel, 1024×1024 display. If we simply replicate image pixels horizontally and vertically in the display array, the replicated pixels on the display will form squares that are quite obvious to the eye. To avoid this problem, we can interpolate intensities to calculate the pixel values. For instance, if an image S is to be displayed at double resolution, then the intensities I to display (with $x = 2x'$ and $y = 2y'$) are

$$I[x][y] = S[x'][y'];$$

$$I[x+1][y] = \tfrac{1}{2}(S[x'][y'] + S[x'+1][y']);$$

$$I[x][y+1] = \tfrac{1}{2}(S[x'][y'] + S[x'][y'+1]);$$

$$I[x+1][y+1] = \tfrac{1}{4}(S[x'][y'] + S[x'+1][y'] + S[x'][y'+1] + S[x'+1][y'+1]);$$

See Section 17.4 for a discussion of two-pass scaling transformations of images, and Section 3.17 for a description of the filtering and image-reconstruction techniques that are applicable to this problem.

13.2 CHROMATIC COLOR

The visual sensations caused by colored light are much richer than those caused by achromatic light. Discussions of color perception usually involve three quantities, known as hue, saturation, and lightness. *Hue* distinguishes among colors such as red, green, purple, and yellow. *Saturation* refers to how far color is from a gray of equal intensity. Red is highly saturated; pink is relatively unsaturated; royal blue is highly saturated; sky blue is relatively unsaturated. Pastel colors are relatively unsaturated; unsaturated colors include more white light than do the vivid, saturated colors. *Lightness* embodies the achromatic notion of perceived intensity of a reflecting object. *Brightness*, a fourth term, is used instead of lightness to refer to the perceived intensity of a self-luminous (i.e., emitting rather than reflecting light) object, such as a light bulb, the sun, or a CRT.

It is necessary to specify and measure colors if we are to use them precisely in computer graphics. For reflected light, we can do this by visually comparing a sample of unknown color against a set of "standard" samples. The unknown and sample colors must be viewed under a standard light source, since the perceived color of a surface depends both on the surface and on the light under which the surface is viewed. The widely used Munsell color-order system includes sets of published standard colors [MUNS76] organized in a 3D space of hue, value (what we have defined as lightness), and chroma (saturation). Each color is named, and is ordered so as to have an equal perceived "distance" in color space (as judged by many observers) from its neighbors. [KELL76] gives an extensive discussion of standard samples, charts depicting the Munsell space, and tables of color names.

In the printing industry and graphic design profession, colors are typically specified by matching to printed color samples. Color Plate I.33 is taken from a widely used color-matching system.

Artists often specify color as different tints, shades, and tones of strongly saturated, or pure, pigments. A *tint* results from adding white pigment to a pure pigment, thereby decreasing saturation. A *shade* comes from adding a black pigment to a pure pigment, thereby decreasing lightness. A *tone* is the consequence of adding both black and white pigments to a pure pigment. All these steps produce different colors of the same hue, with varying saturation and lightness. Mixing just black and white pigments creates grays. Figure 13.15 shows the relationship of tints, shades, and tones. The percentage of pigments that must be mixed to match a color can be used as a color specification. The Ostwald [OSTW31] color-order system is similar to the artist's model.

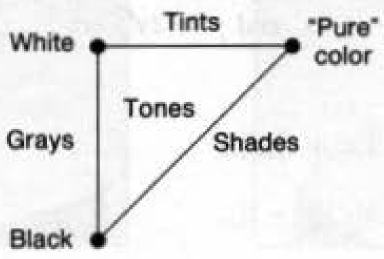


Fig. 13.15 Tints, tones, and shades.

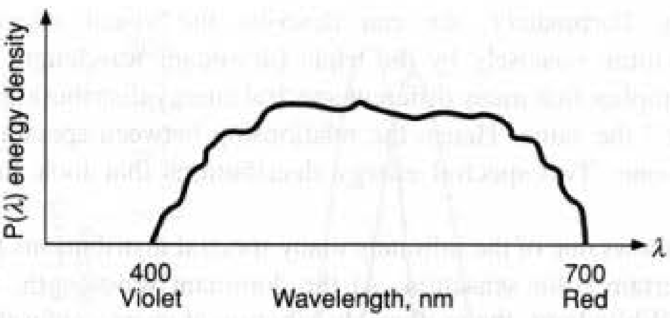


Fig. 13.16 Typical spectral energy distribution $P(\lambda)$ of a light.

13.2.1 Psychophysics

The Munsell and artists' pigment-mixing methods are subjective: They depend on human observers' judgments, the lighting, the size of the sample, the surrounding color, and the overall lightness of the environment. An objective, quantitative way of specifying colors is needed, and for this we turn to the branch of physics known as *colorimetry*. Important terms in colorimetry are dominant wavelength, excitation purity, and luminance.

Dominant wavelength is the wavelength of the color we "see" when viewing the light, and corresponds to the perceptual notion of hue[2]; *excitation purity* corresponds to the saturation of the color; *luminance* is the amount or intensity of light. The excitation purity of a colored light is the proportion of pure light of the dominant wavelength and of white light needed to define the color. A completely pure color is 100 percent saturated and thus contains no white light, whereas mixtures of a pure color and white light have saturations somewhere between 0 and 100 percent. White light and hence grays are 0 percent saturated, containing no color of any dominant wavelength. The correspondences between these perceptual and colorimetry terms are as follows:

Perceptual term	**Colorimetry**
Hue	Dominant wavelength
Saturation	Excitation purity
Lightness (reflecting objects)	Luminance
Brightness (self-luminous objects)	Luminance

Fundamentally, light is electromagnetic energy in the 400- to 700-nm wavelength part of the spectrum, which is perceived as the colors from violet through indigo, blue, green, yellow, and orange to red. Color Plate I.34 shows the colors of the spectrum. The amount of energy present at each wavelength is represented by a spectral energy distribution $P(\lambda)$, such as shown in Fig. 13.16 and Color Plate I.34. The distribution represents an infinity of numbers, one for each wavelength in the visible spectrum (in practice, the distribution is represented by a large number of sample points on the spectrum, as measured by a

[2]Some colors, such as purple, have no true dominant wavelength, as we shall see later.

spectroradiometer). Fortunately, we can describe the visual effect of any spectral distribution much more concisely by the triple (dominant wavelength, excitation purity, luminance). This implies that many different spectral energy distributions produce the same color: They "look" the same. Hence the relationship between spectral distributions and colors is many-to-one. Two spectral energy distributions that look the same are called *metamers*.

Figure 13.17 shows one of the infinitely many spectral distributions $P(\lambda)$, or metamers, that produces a certain color sensation. At the dominant wavelength, there is a spike of energy of level e_2. White light, the uniform distribution of energy at level e_1, is also present. The excitation purity depends on the relation between e_1 and e_2: when $e_1 = e_2$, excitation purity is 0 percent; when $e_1 = 0$, excitation purity is 100 percent. Brightness, which is proportional to the integral of the product of the curve and the luminous efficiency function (defined later), depends on both e_1 and e_2. In general, spectral distributions may be more complex than the one shown, and it is not possible to determine the dominant wavelength merely by looking at the spectral distributions. In particular, the dominant wavelength may *not* be the one whose component in the spectral distribution is largest.

How does this discussion relate to the red, green, and blue phosphor dots on a color CRT? And how does it relate to the *tristimulus theory* of color perception, which is based on the hypothesis that the retina has three kinds of color sensors (called cones), with peak sensitivity to red, green, or blue lights? Experiments based on this hypothesis produce the spectral-response functions of Fig. 13.18. The peak blue response is around 440 nm; that for green is about 545 nm; that for red is about 580 nm. (The terms "red" and "green" are somewhat misleading here, as the 545 nm and 580 nm peaks are actually in the yellow range.) The curves suggest that the eye's response to blue light is much less strong than is its response to red or green.

Figure 13.19 shows the *luminous-efficiency function*, the eye's response to light of constant luminance, as the dominant wavelength is varied: our peak sensitivity is to yellow-green light of wavelength around 550 nm. There is experimental evidence that this curve is just the sum of the three curves shown in Fig. 13.18.

The tristimulus theory is intuitively attractive because it corresponds loosely to the notion that colors can be specified by positively weighted sums of red, green, and blue (the so-called primary colors). This notion is almost true: The three color-matching functions in

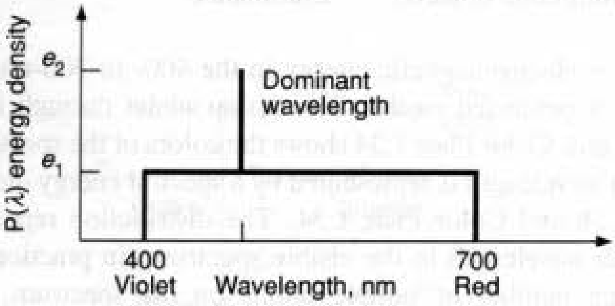


Fig. 13.17 Spectral energy distribution, $P(\lambda)$, illustrating dominant wavelength, excitation purity, and luminance.

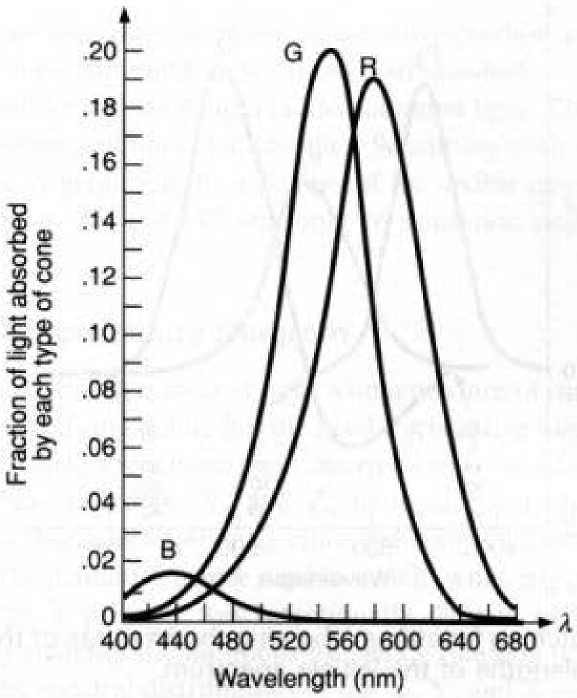


Fig. 13.18 Spectral-response functions of each of the three types of cones on the human retina.

Fig. 13.20 show the amounts of red, green, and blue light needed by an average observer to match a color of constant luminance, for all values of dominant wavelength in the visible spectrum.

A negative value in Fig. 13.20 means that we cannot match the color by adding together the primaries. However, if one of the primaries is added to the color sample, the sample can then be matched by a mixture of the other two primaries. Hence, negative values in Fig. 13.20 indicate that the primary was added to the color being matched. The need for negative values does not mean that the notion of mixing red, green, and blue to obtain other

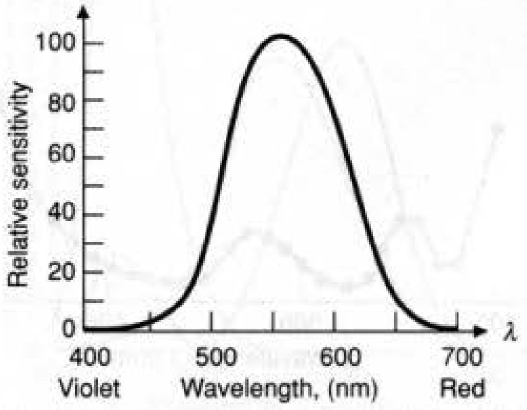


Fig. 13.19 Luminous-efficiency function for the human eye.

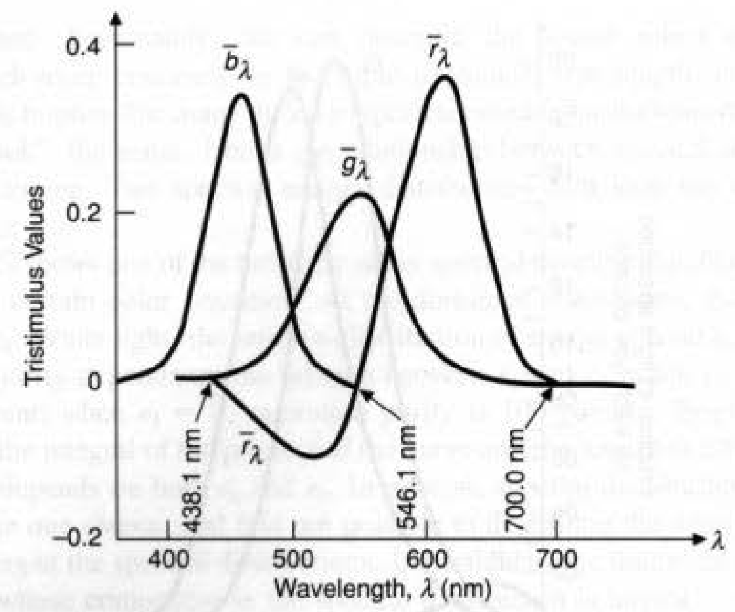


Fig. 13.20 Color-matching functions, showing the amounts of three primaries needed to match all the wavelengths of the visible spectrum.

colors is invalid; on the contrary, a huge range of colors can be matched by positive amounts of red, green, and blue. Otherwise, the color CRT would not work! It does mean, however, that certain colors cannot be produced by RGB mixes, and hence cannot be shown on an ordinary CRT.

The human eye can distinguish hundreds of thousands of different colors in color space, when different colors are judged side by side by different viewers who state whether the colors are the same or different. As shown in Fig. 13.21, when colors differ only in hue, the wavelength between just noticably different colors varies from more than 10 nm at the extremes of the spectrum to less than 2 nm around 480 nm (blue) and 580 nm (yellow).

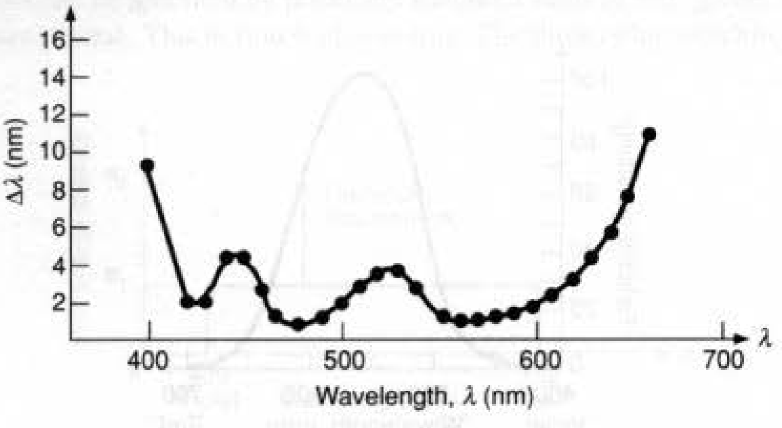


Fig. 13.21 Just-noticable color differences as a function of wavelength λ. The ordinate indicates the minimum detectable difference in wavelength between adjacent color samples. (Source: Data are from [BEDF58].)

Except at the spectrum extremes, however, most distinguished hues are within 4 nm. Altogether about 128 fully saturated hues can be distinguished.

The eye is less sensitive to hue changes in less saturated light. This is not surprising: As saturation tends to 0 percent, all hues tend to white. Sensitivity to changes in saturation for a fixed hue and lightness is greater at the extremes of the visible spectrum, where about 23 distinguishable steps exist. Around 575 nm, only 16 saturation steps can be distinguished [JONE26].

13.2.2 The CIE Chromaticity Diagram

Matching and therefore defining a colored light with a mixture of three fixed primaries is a desirable approach to specifying color, but the need for negative weights suggested by Fig. 13.20 is awkward. In 1931, the *Commission Internationale de l'Éclairage (CIE)* defined three standard primaries, called **X**, **Y**, and **Z**, to replace red, green, and blue in this matching process. The three corresponding color-matching functions, $\bar{x}_\lambda$, $\bar{y}_\lambda$, and $\bar{z}_\lambda$, are shown in Fig. 13.22. The primaries can be used to match, with only positive weights, all the colors we can see. The **Y** primary was intentionally defined to have a color-matching function $\bar{y}_\lambda$ that exactly matches the luminous-efficiency function of Fig. 13.19. Note that $\bar{x}_\lambda$, $\bar{y}_\lambda$, and $\bar{z}_\lambda$ are not the spectral distributions of the **X**, **Y**, and **Z** colors, just as the curves in Fig. 13.20 are not the spectral distributions of red, green, and blue. They are merely auxilliary functions used to compute how much of **X**, **Y**, and **Z** should be mixed together to generate a metamer of any visible color.

The color-matching functions are defined tabularly at 1-nm intervals, and are found in texts such as [WYSZ82; BILL81]. The distributions were defined for color samples that

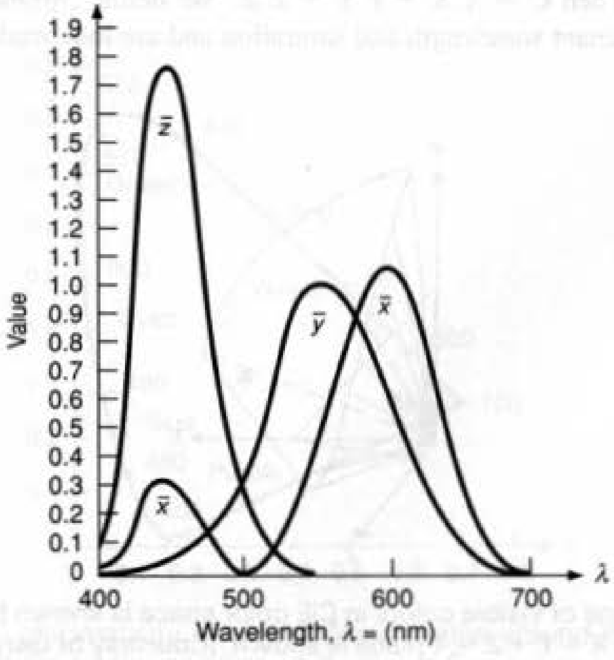


Fig. 13.22 The color-matching functions $\bar{x}_\lambda$, $\bar{y}_\lambda$, and $\bar{z}_\lambda$, for the 1931 CIE **X**, **Y**, **Z** primaries.

subtend a 2° field of view on the retina. The original 1931 tabulation is normally used in work relevant to computer graphics. A later 1964 tabulation for a 10° field of view is not generally useful, because it emphasizes larger areas of constant color than are normally found in graphics.

The three CIE color-matching functions are linear combinations of the color-matching functions from Fig. 13.20. This means that the definition of a color with red, green, and blue lights can be converted via a linear transformation into its definition with the CIE primaries, and vice versa.

The amounts of **X**, **Y**, and **Z** primaries needed to match a color with a spectral energy distribution $P(\lambda)$, are:

$$X = k \int P(\lambda)\, \bar{x}_\lambda\, d\lambda, \quad Y = k \int P(\lambda)\, \bar{y}_\lambda\, d\lambda, \quad Z = k \int P(\lambda)\, \bar{z}_\lambda\, d\lambda. \qquad (13.18)$$

For self-luminous objects like a CRT, k is 680 lumens/watt. For reflecting objects, k is usually selected such that bright white has a Y value of 100; then other Y values will be in the range of 0 to 100. Thus,

$$k = \frac{100}{\int P_w(\lambda)\bar{y}_\lambda\, d\lambda} \qquad (13.19)$$

where $P_w(\lambda)$ is the spectral energy distribution for whatever light source is chosen as the standard for white. In practice, these integrations are performed by summation, as none of the energy distributions are expressed analytically.

Figure 13.23 shows the cone-shaped volume of XYZ space that contains visible colors. The volume extends out from the origin into the postive octant, and is capped at the smooth curved line terminating the cone.

Let (X, Y, Z) be the weights applied to the CIE primaries to match a color **C**, as found using Eq. (13.18). Then $\mathbf{C} = X\,\mathbf{X} + Y\,\mathbf{Y} + Z\,\mathbf{Z}$. We define *chromaticity* values (which depend only on dominant wavelength and saturation and are independent of the amount of

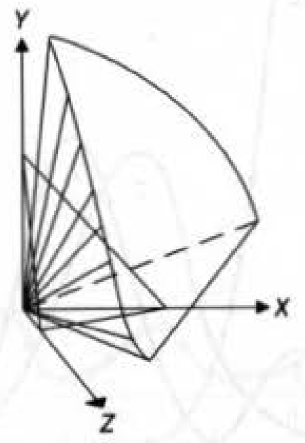


Fig. 13.23 The cone of visible colors in CIE color space is shown by the lines radiating from the origin. The $X + Y + Z = 1$ plane is shown. (Courtesy of Gary Meyer, Program of Computer Graphics, Cornell University, 1978.)

luminous energy) by normalizing against $X + Y + Z$, which can be thought of as the total amount of light energy:

$$x = \frac{X}{(X + Y + Z)},\ y = \frac{Y}{(X + Y + Z)},\ z = \frac{Z}{(X + Y + Z)}. \tag{13.20}$$

Notice that $x + y + z = 1$. That is, x, y, and z are on the $(X + Y + Z = 1)$ plane of Fig. 13.23. Color plate II.1 shows the $X + Y + Z = 1$ plane as part of CIE space, and also shows an orthographic view of the plane along with the projection of the plane onto the (X, Y) plane. This latter projection is just the CIE chromaticity diagram.

If we specify x and y, then z is determined by $z = 1 - x - y$. We cannot recover X, Y, and Z from x and y, however. To recover them, we need one more piece of information, typically Y, which carries luminance information. Given (x, y, Y), the transformation to the corresponding (X, Y, Z) is

$$X = \frac{x}{y}Y,\quad Y = Y, Z = \frac{1 - x - y}{y}Y. \tag{13.21}$$

Chromaticity values depend only on dominant wavelength and saturation and are independent of the amount of luminous energy. By plotting x and y for all visible colors, we obtain the CIE chromaticity diagram shown in Fig. 13.24, which is the projection onto the (X, Y) plane of the $(X + Y + Z = 1)$ plane of Fig. 13.23. The interior and boundary of the horseshoe-shaped region represent all visible chromaticity values. (All perceivable colors with the same chromaticity but different luminances map into the same point within this region.) The 100 percent spectrally pure colors of the spectrum are on the curved part of the boundary. A standard white light, meant to approximate sunlight, is formally defined by a

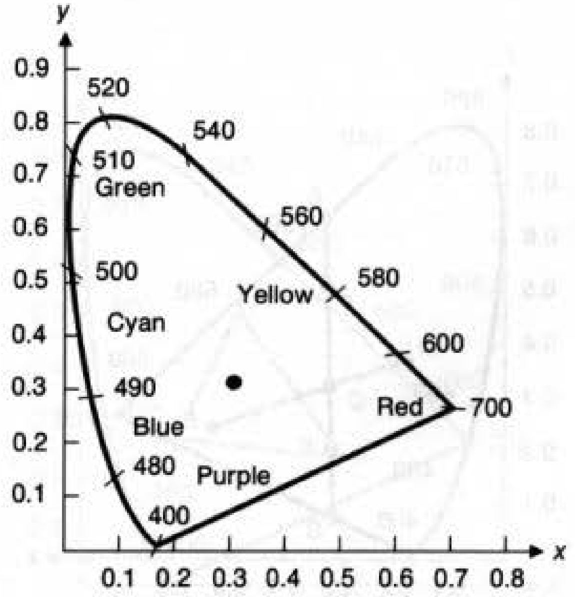


Fig. 13.24 The CIE chromaticity diagram. Wavelengths around the periphery are in nanometers. The dot marks the position of illuminant C.

light source *illuminant C*, marked by the center dot. It is near but not at the point where $x = y = z = \frac{1}{3}$. Illuminant C was defined by specifying a spectral power distribution that is close to daylight at a correlated color temperature of 6774° Kelvin.

The CIE chromaticity diagram is useful in many ways. For one, it allows us to measure the dominant wavelength and excitation purity of any color by matching the color with a mixture of the three CIE primaries. (Instruments called *colorimeters* measure tristimulus X, Y, and Z values, from which chromaticity coordinates are computed using Eq. (13.20). *Spectroradiometers* measure both the spectral energy distribution and the tristimulus values.) Now suppose the matched color is at point A in Fig. 13.25. When two colors are added together, the new color lies somewhere on the straight line in the chromaticity diagram connecting the two colors being added. Therefore, color A can be thought of as a mixture of "standard" white light (illuminant C) and the pure spectral light at point B. Thus, B defines the dominant wavelength. The ratio of length AC to length BC, expressed as a percentage, is the excitation purity of A. The closer A is to C, the more white light A includes and the less pure it is.

The chromaticity diagram factors out luminance, so color sensations that are luminance-related are excluded. For instance; brown, which is an orange-red chromaticity at very low luminance relative to its surrounding area, does not appear. It is thus important to remember that the chromaticity diagram is not a full color palette. There is an infinity of planes in (X, Y, Z) space, each of which projects onto the chromaticity diagram and each of which loses luminance information in the process. The colors found on each such plane are all different.

Complementary colors are those that can be mixed to produce white light (such as D and E in Fig. 13.25). Some colors (such as F in Fig. 13.25) cannot be defined by a

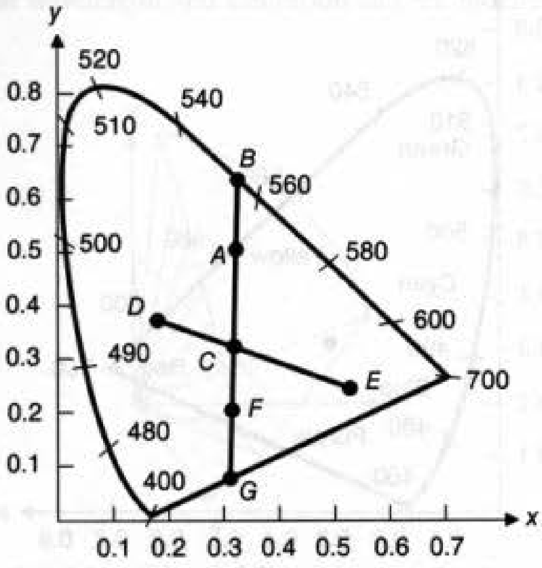


Fig. 13.25 Colors on the chromaticity diagram. The dominant wavelength of color A is that of color B. Colors D and E are complementary colors. The dominant wavelength of color F is defined as the complement of the dominant wavelength of color A.

dominant wavelength and are thus called *nonspectral*. In this case, the dominant wavelength is said to be the complement of the wavelength at which the line through F and C intersects the horseshoe part of the curve at point B, and is designated by a "c" (here about 555 nm c). The excitation purity is still defined by the ratio of lengths (here CF to CG). The colors that must be expressed by using a complementary dominant wavelength are the purples and magentas; they occur in the lower part of the CIE diagram. Complementary colors still can be made to fit the dominant wavelength model of Fig. 13.17, in the sense that if we take a flat spectral distribution and delete some of the light at frequency B, the resulting color will be perceived as F.

Another use of the CIE chromaticity diagram is to define *color gamuts*, or color ranges, that show the effect of adding colors together. Any two colors, say I and J in Fig. 13.26, can be added to produce any color along their connecting line by varying the relative amounts of the two colors being added. A third color K (see Fig. 13.26) can be used with various mixtures of I and J to produce the gamut of all colors in triangle IJK, again by varying relative amounts. The shape of the diagram shows why visible red, green, and blue cannot be additively mixed to match all colors: No triangle whose vertices are within the visible area can completely cover the visible area.

The chromaticity diagram is also used to compare the gamuts available on various color display and hardcopy devices. Color Plate II.2 shows the gamuts for a color television monitor, film, and print. The chromaticity coordinates for the phosphors in two typical monitors are these:

	Short-persistence phosphors			**Long-persistence phosphors**		
	Red	Green	Blue	Red	Green	Blue
x	0.61	0.29	0.15	0.62	0.21	0.15
y	0.35	0.59	0.063	0.33	0.685	0.063

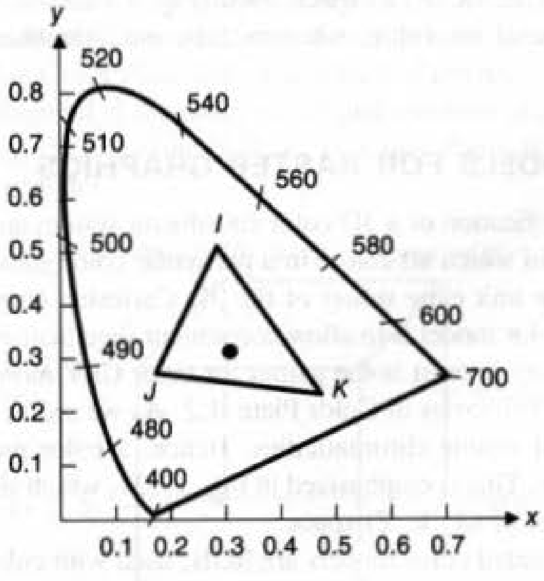


Fig. 13.26 Mixing colors. All colors on the line IJ can be created by mixing colors I and J; all colors in the triangle IJK can be created by mixing colors I, J, and K.

The smallness of the print gamut with respect to the color-monitor gamut suggests that, if images originally seen on a monitor must be faithfully reproduced by printing, a reduced gamut of colors should be used with the monitor. Otherwise, accurate reproduction will not be possible. If, however, the goal is to make a pleasing rather than an exact reproduction, small differences in color gamuts are less important. A discussion of color-gamut compression can be found in [HALL89].

There is a problem with the CIE system. Consider the distance from color $C_1 = (X_1, Y_1, Z_1)$ to color $C_1 + \Delta C$, and the distance from color $C_2 = (X_2, Y_2, Z_2)$ to color $C_2 + \Delta C$, where $\Delta C = (\Delta X, \Delta Y, \Delta Z)$. Both distances are equal to ΔC, yet in general they will not be perceived as being equal. This is because of the variation, throughout the spectrum, of the just noticable differences discussed in Section 13.2.1. A *perceptually uniform* color space is needed, in which two colors that are equally distant are *perceived* as equally distant by viewers.

The 1976 CIE LUV uniform color space was developed in response to this need. With (X_n, Y_n, Z_n) as the coordinates of the color that is to be defined as white, the space is defined by

$$
\begin{aligned}
L^* &= 116\,(Y/Y_n)^{1/3} - 16,\ Y/Y_n > 0.01\\
u^* &= 13\,L^*\,(u' - u'_n),\\
v^* &= 13\,L^*\,(v' - v'_n),\\
u' &= \frac{4X}{X + 15Y + 3Z},\quad v' = \frac{9Y}{X + 15Y + 3Z},\\
u'_n &= \frac{4X_n}{X_n + 15Y_n + 3Z_n},\quad v'_n = \frac{9Y_n}{X_n + 15Y_n + 3Z_n}.
\end{aligned}
\tag{13.22}
$$

The shape of the 3D volume of visible colors defined by these equations is of course different from that for CIE (X, Y, Z) space itself (Fig. 13.23).

With this background on color, we now turn our attention to color in computer graphics.

13.3 COLOR MODELS FOR RASTER GRAPHICS

A color model is a specification of a 3D color coordinate system and a visible subset in the coordinate system within which all colors in a particular color gamut lie. For instance, the RGB color model is the unit cube subset of the 3D Cartesian coordinate system.

The purpose of a color model is to allow convenient specification of colors within some color gamut. Our primary interest is the gamut for color CRT monitors, as defined by the RGB (red, green, blue) primaries in Color Plate II.2. As we see in this color plate, a color gamut is a subset of all visible chromaticities. Hence, a color model cannot be used to specify all visible colors. This is emphasized in Fig. 13.27, which shows that the gamut of a CRT monitor is a subset of (X, Y, Z) space.

Three hardware-oriented color models are RGB, used with color CRT monitors, YIQ, the broadcast TV color system, and CMY (cyan, magenta, yellow) for some color-printing devices. Unfortunately, none of these models are particularly easy to use, because they do not relate directly to intuitive color notions of hue, saturation, and brightness. Therefore,

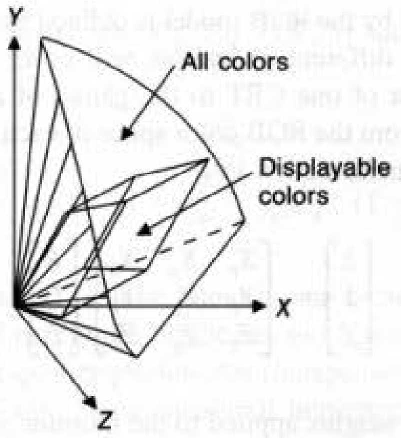


Fig. 13.27 The color gamut for a typical color monitor within the CIE color space. The range of colors that can be displayed on the monitor is clearly smaller than that of all colors visible in CIE space. (Courtesy of Gary Meyer, Program of Computer Graphics, Cornell University, 1978.)

another class of models has been developed with ease of use as a goal. Several such models are described in [GSPC79; JOBL78; MEYE80; SMIT78]. We discuss three, the HSV (sometimes called HSB), HLS, and HVC models.

With each model is given a means of converting to some other specification. For RGB, this conversion is to CIE's (X, Y, Z) space. This conversion is important, because CIE is the worldwide standard. For all of the other models, the conversion is to RGB; hence, we can convert from, say, HSV to RGB to the CIE standard.

13.3.1 The RGB Color Model

The red, green, and blue (RGB) color model used in color CRT monitors and color raster graphics employs a Cartesian coordinate system. The RGB primaries are *additive* primaries; that is, the individual contributions of each primary are added together to yield the result, as suggested in Color Plate II.3. The subset of interest is the unit cube shown in Fig. 13.28. The main diagonal of the cube, with equal amounts of each primary, represents the gray levels: black is (0, 0, 0); white is (1, 1, 1). Color Plates II.4 and II.5 show several views of the RGB color model.

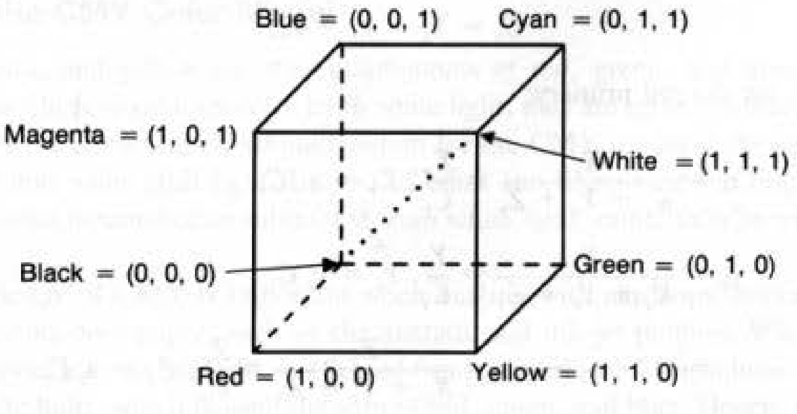


Fig. 13.28 The RGB cube. Grays are on the dotted main diagonal.

The color gamut covered by the RGB model is defined by the chromaticities of a CRT's phosphors. Two CRTs with different phosphors will cover different gamuts. To convert colors specified in the gamut of one CRT to the gamut of another CRT, we can use the transformations M_1 and M_2 from the RGB color space of each monitor to the (X, Y, Z) color space. The form of each transformation is:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X_r & X_g & X_b \\ Y_r & Y_g & Y_b \\ Z_r & Z_g & Z_b \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}, \tag{13.23}$$

where X_r, X_g, and X_b are the weights applied to the monitor's RGB colors to find X, and so on.

Defining M as the 3×3 matrix of color-matching coefficients, we write Eq. (13.23) as

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = M \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{13.24}$$

With M_1 and M_2 the matricies that convert from each of the two monitor's gamuts to CIE, $M_2^{-1}M_1$ converts from the RGB of monitor 1 to the RGB of monitor 2. This matrix product is all that is needed if the color in question lies in the gamuts of both monitors. What if a color C_1 is in the gamut of monitor 1 but is not in the gamut of monitor 2? The corresponding color $C_2 = M_2^{-1}M_1C_1$ will be outside the unit cube and hence will not be displayable. A simple but not very satisfactory solution is to clamp the color values—that is, to replace values of R, G, or B that are less than 0 with 0, and values that are greater than 1 with 1. More satisfactory but also more complex approaches are described in [HALL89].

The chromaticity coordinates for the RGB phosphors are usually available from CRT manufacturers' specifications. If not, a colorimeter can also be used to measure the chromaticity coordinates directly, or a spectroradiometer can be used to measure $P(\lambda)$, which can then be converted to chromaticity coordinates using Eqs. (13.18), (13.19), and (13.20). Denoting the coordinates by (x_r, y_r) for red, (x_g, y_g) for green, and (x_b, y_b) for blue, and defining C_r as

$$C_r = X_r + Y_r + Z_r, \tag{13.25}$$

we can write, for the red primary;

$$x_r = \frac{X_r}{X_r + Y_r + Z_r} = \frac{X_r}{C_r},\ X_r = x_r\,C_r,$$

$$y_r = \frac{Y_r}{X_r + Y_r + Z_r} = \frac{Y_r}{C_r},\ Y_r = y_r\,C_r,$$

$$z_r = (1 - x_r - y_r) = \frac{Z_r}{X_r + Y_r + Z_r} = \frac{Z_r}{C_r},\ Z_r = z_r\,C_r. \tag{13.26}$$

With similar definitions for C_g and C_b, Eq. (13.23) can be rewritten as:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} x_r C_r & x_g C_g & x_b C_b \\ y_r C_r & y_g C_g & y_b C_b \\ (1 - x_r - y_r) C_r & (1 - x_g - y_g) C_g & (1 - x_b - y_b) C_b \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \quad (13.27)$$

The unknowns C_r, C_g, and C_b can be found in one of two ways [MEYE83]. First, the luminances Y_r, Y_g, and Y_b of maximum-brightness red, green, and blue may be known or can be measured with a high-quality photometer (inexpensive meters can be off by factors of 2 to 10 on the blue reading). These measured luminances can be combined with the known y_r, y_g, and y_b to yield

$$C_r = Y_r / y_r,\ C_g = Y_g / y_g,\ C_b = Y_b / y_b. \quad (13.28)$$

These values are then substituted into Eq. (13.27), and the conversion matrix M is thus expressed in terms of the known quantities (x_r, y_r), (x_g, y_g), (x_b, y_b), Y_r, Y_g, and Y_b.

We can also remove the unknown variables from Eq. (13.27) if we know or can measure the X_w, Y_w, and Z_w for the white color produced when $R = G = B = 1$. For this case, Eq. (13.27) can be rewritten as

$$\begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix} = \begin{bmatrix} x_r & x_g & x_b \\ y_r & y_g & y_b \\ (1 - x_r - y_r) & (1 - x_g - y_g) & (1 - x_b - y_b) \end{bmatrix} \begin{bmatrix} C_r \\ C_g \\ C_b \end{bmatrix}, \quad (13.29)$$

solved for C_r, C_g, and C_b (the only unknowns), and the resulting values substituted into Eq. (13.27). Alternatively, it may be that the white color is given by x_w, y_w, and Y_w; in this case, before solving Eq. (13.29), we first find

$$X_w = x_w \frac{Y_w}{y_w},\ Z_w = z_w \frac{Y_w}{y_w}. \quad (13.30)$$

13.3.2 The CMY Color Model

Cyan, magenta, and yellow are the complements of red, green, and blue, respectively. When used as filters to subtract color from white light, they are called *subtractive primaries*. The subset of the Cartesian coordinate system for the CMY model is the same as that for RGB except that white (full light) instead of black (no light) is at the origin. Colors are specified by what is removed or subtracted from white light, rather than by what is added to blackness.

A knowledge of CMY is important when dealing with hardcopy devices that deposit colored pigments onto paper, such as electrostatic and ink-jet plotters. When a surface is coated with cyan ink, no red light is reflected from the surface. Cyan subtracts red from the reflected white light, which is itself the sum of red, green, and blue. Hence, in terms of the additive primaries, cyan is white minus red, that is, blue plus green. Similarly, magenta absorbs green, so it is red plus blue; yellow absorbs blue, so it is red plus green. A surface coated with cyan and yellow absorbs red and blue, leaving only green to be reflected from illuminating white light. A cyan, yellow, and magenta surface absorbs red, green, and blue, and therefore is black. These relations, diagrammed in Fig. 13.29, can be seen in Color Plate II.6 and are represented by the following equation:

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{13.31}$$

The unit column vector is the RGB representation for white and the CMY representation for black.

The conversion from RGB to CMY is then

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix}. \tag{13.32}$$

These straightforward transformations can be used for converting the eight colors that can be achieved with binary combinations of red, green, and blue into the eight colors achievable with binary combinations of cyan, magenta, and yellow. This conversion is relevant for use on ink-jet and xerographic color printers.

Another color model, CMYK, uses black (abbreviated as K) as a fourth color. CMYK is used in the four-color printing process of printing presses and some hard-copy devices.

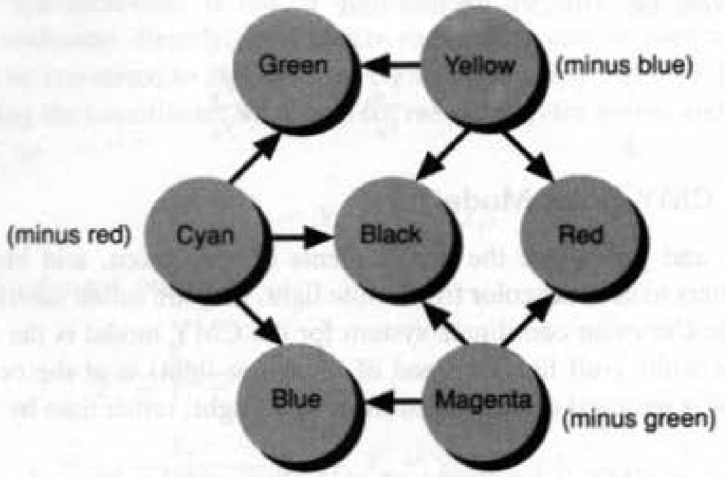


Fig. 13.29 Subtractive primaries (cyan, magenta, yellow) and their mixtures. For instance, cyan and yellow combine to green.

Given a CMY specification, black is used in place of equal amounts of C, M, and Y, according to the relations:

$$K = \min(C, M, Y);$$
$$C = C - K;$$
$$M = M - K;$$
$$Y = Y - K; \tag{13.34}$$

This is discussed further in Section 13.4 and in [STON88].

13.3.3 The YIQ Color Model

The YIQ model is used in U.S. commercial color television broadcasting and is therefore closely related to color raster graphics. YIQ is a recoding of RGB for transmission efficiency and for downward compatibility with black-and-white television. The recoded signal is transmitted using the National Television System Committee (NTSC) [PRIT77] system.

The *Y* component of YIQ is not yellow but luminance, and is defined to be the same as the CIE Y primary. Only the *Y* component of a color TV signal is shown on black-and-white televisions: the chromaticity is encoded in *I* and *Q*. The YIQ model uses a 3D Cartesian coordinate system, with the visible subset being a convex polyhedron that maps into the RGB cube.

The RGB-to-YIQ mapping is defined as follows:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{13.33}$$

The quantities in the first row reflect the relative importance of green and red and the relative unimportance of blue in brightness. The inverse of the RGB-to-YIQ matrix is used for the YIQ-to-RGB conversion.

Equation (13.33) assumes that the RGB color specification is based on the standard NTSC RGB phosphor, whose CIE coordinates are

	Red	**Green**	**Blue**
x	0.67	0.21	0.14
y	0.33	0.71	0.08

and for which the white point, illuminant C, is $x_w = 0.31$, $y_w = 0.316$, and $Y_w = 100.0$.

Specifying colors with the YIQ model solves a potential problem with material being prepared for broadcast television: Two different colors shown side by side on a color monitor will appear to be different, but, when converted to YIQ and viewed on a monochrome monitor, they may appear to be the same. This problem can be avoided by specifying the two colors with different Y values in the YIQ color model space (i.e., by adjusting only the Y value to disambiguate them).

The YIQ model exploits two useful properties of our visual system. First, the system is more sensitive to changes in luminance than to changes in hue or saturation; that is, our ability to discriminate spatially color information is weaker than our ability to discriminate spatially monochrome information. This suggests that more bits of bandwidth should be used to represent Y than are used to represent I and Q, so as to provide higher resolution in Y. Second, objects that cover a very small part of our field of view produce a limited color sensation, which can be specified adequately with one rather than two color dimensions. This suggests that either I or Q can have a lower bandwidth than the other. The NTSC encoding of YIQ into a broadcast signal uses these properties to maximize the amount of information transmitted in a fixed bandwidth: 4 MHz is assigned to Y, 1.5 to I, and 0.6 to Q. Further discussion of YIQ can be found in [SMIT78; PRIT77].

13.3.4 The HSV Color Model

The RGB, CMY, and YIQ models are hardware-oriented. By contrast, Smith's HSV (hue, saturation, value) model [SMIT78] (also called the HSB model, with B for brightness) is user-oriented, being based on the intuitive appeal of the artist's tint, shade, and tone. The coordinate system is cylindrical, and the subset of the space within which the model is defined is a hexcone, or six-sided pyramid, as in Fig. 13.30. The top of the hexcone corresponds to $V = 1$, which contains the relatively bright colors. The colors of the $V = 1$ plane are *not* all of the same perceived brightness, however. Color Plates II.7 and II.8 show the color model.

Hue, or H, is measured by the angle around the vertical axis, with red at 0°, green at 120°, and so on (see Fig. 13.30). Complementary colors in the HSV hexcone are 180°

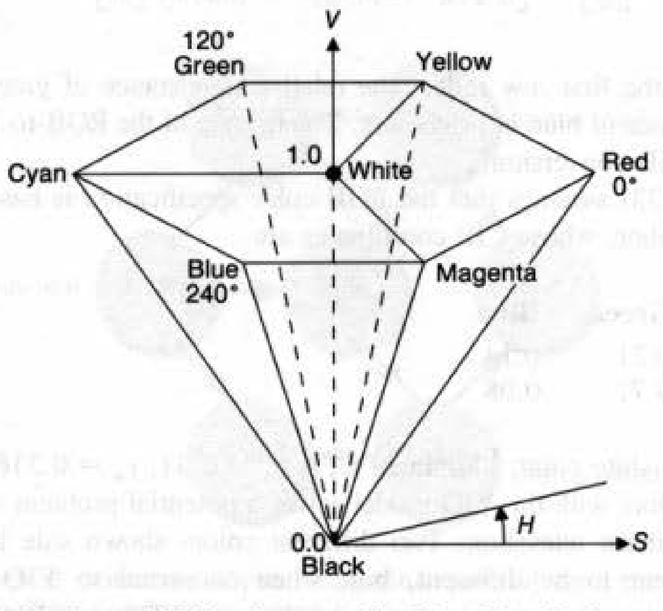


Fig. 13.30 Single-hexcone HSV color model. The $V = 1$ plane contains the RGB model's $R = 1$, $G = 1$, and $B = 1$ planes in the regions shown.

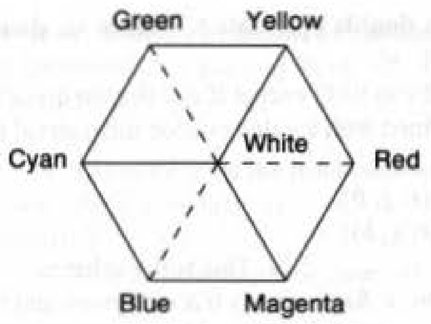


Fig. 13.31 RGB color cube viewed along the principal diagonal. Visible edges of the cube are solid; invisible edges are dashed.

opposite one another. The value of S is a ratio ranging from 0 on the center line (V axis) to 1 on the triangular sides of the hexcone. Saturation is measured relative to the color gamut represented by the model, which is, of course, a subset of the entire CIE chromaticity diagram. Therefore, saturation of 100 percent in the model is less than 100 percent excitation purity.

The hexcone is one unit high in V, with the apex at the origin. The point at the apex is black and has a V coordinate of 0. At this point, the values of H and S are irrelevant. The point $S = 0$, $V = 1$ is white. Intermediate values of V for $S = 0$ (on the center line) are the grays. When $S = 0$, the value of H is irrelevant (called by convention UNDEFINED). When S is not zero, H is relevant. For example, pure red is at $H = 0$, $S = 1$, $V = 1$. Indeed, any color with $V = 1$, $S = 1$ is akin to an artist's pure pigment used as the starting point in mixing colors. Adding white pigment corresponds to decreasing S (without changing V). Shades are created by keeping $S = 1$ and decreasing V. Tones are created by decreasing both S and V. Of course, changing H corresponds to selecting the pure pigment with which to start. Thus, H, S, and V correspond to concepts from the artists' color system, and are not exactly the same as the similar terms introduced in Section 13.2.

The top of the HSV hexcone corresponds to the projection seen by looking along the principal diagonal of the RGB color cube from white toward black, as shown in Fig. 13.31. The RGB cube has subcubes, as illustrated in Fig. 13.32. Each subcube, when viewed along

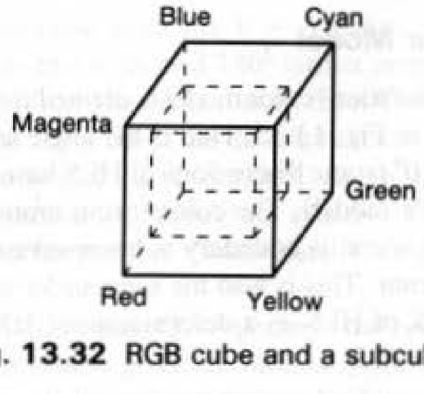


Fig. 13.32 RGB cube and a subcube.

```
void RGB_To_HSV (double r, double g, double b, double *h, double *s, double *v)
/* Given: r, g, b, each in [0,1]. */
/* Desired: h in [0,360), s and v in [0,1] except if s = 0, then h = UNDEFINED, */
/* which is some constant defined with a value outside the interval [0,360]. */
{
    double max = Maximum (r, g, b);
    double min = Minimum (r, g, b);
    *v = max;                          /* This is the value v. */
    /* Next calculate saturation, s. Saturation is 0 if red, green and blue are all 0 */
    *s = (max != 0.0) ? ((max - min) / max) : 0.0;
    if (*s == 0.0)
        *h = UNDEFINED;
    else {                             /* Chromatic case: Saturation is not 0, */
        double delta = max - min;      /* so determine hue. */
        if (r == max)
            *h = (g - b) /delta;       /* Resulting color is between yellow and magenta */
        else if (g == max)
            *h = 2.0 + (b - r) / delta; /* Resulting color is between cyan and yellow */
        else if (b == max)
            *h = 4.0 + (r - g) / delta; /* Resulting color is between magenta and cyan */
        *h *= 60.0;                    /* Convert hue to degrees */
        if (*h < 0.0)
            *h += 360.0;               /* Make sure hue is nonnegative */
    } /* Chromatic case */
} /* RGB_To_HSV */
```

Fig. 13.33 Algorithm for converting from RGB to HSV color space.

its main diagonal, is like the hexagon in Fig. 13.31, except smaller. Each plane of constant V in HSV space corresponds to such a view of a subcube in RGB space. The main diagonal of RGB space becomes the V axis of HSV space. Thus, we can see intuitively the correspondence between RGB and HSV. The algorithms of Figs. 13.33 and 13.34 define the correspondence precisely by providing conversions from one model to the other.

13.3.5 The HLS Color Model

The HLS (hue, lightness, saturation) color model is defined in the double-hexcone subset of a cylindrical space, as seen in Fig. 13.35. Hue is the angle around the vertical axis of the double hexcone, with red at 0° (some discussions of HLS have blue at 0°; we place red at 0° for consistency with the HSV model). The colors occur around the perimeter in the same order as in the CIE diagram when its boundary is traversed counterclockwise: red, yellow, green, cyan, blue, and magenta. This is also the same order as in the HSV single-hexcone model. In fact, we can think of HLS as a deformation of HSV, in which white is pulled

```
void HSV_To_RGB (double *r, double *g, double *b, double h, double s, double v)
/* Given: h in [0,360] or UNDEFINED, s and v in [0,1]. */
/* Desired: r, g, b, each in [0,1]. */
{
    if (s == 0.0) {          /* The color is on the black-and-white center line. */
        /* Achromatic color: There is no hue. */
        if (h == UNDEFINED) {
            *r = v;          /* This is the achromatic case. */
            *g = v;
            *b = v;
        } else
            Error();         /* By our convention, error if s = 0 and h has a value. */
    } else {
        double f,p,q,t;      /* Chromatic color: s != 0, so there is a hue. */
        int i;

        if (h == 360.0)      /* 360 degrees is equivalent to 0 degrees. */
            h = 0.0;
        h /= 60.0;           /* h is now in [0,6). */
        i = floor (h);       /* Floor returns the largest integer <= h */
        f = h - i;           /* f is the fractional part of h. */
        p = v * (1.0 - s);
        q = v * (1.0 - (s * f));
        t = v * (1.0 - (s * (1.0 - f)));
        switch(i) {
            case 0: *r = v;  *g = t;  *b = p; break;
            case 1: *r = q;  *g = v;  *b = p; break;
            case 2: *r = p;  *g = v;  *b = t; break;
            case 3: *r = p;  *g = q;  *b = v; break;
            case 4: *r = t;  *g = p;  *b = v; break;
            case 5: *r = v;  *g = p;  *b = q; break;
        }
    } /* Chromatic case */
} /* HSV_To_RGB */
```

Fig. 13.34 Algorithm for converting from HSV to RGB color space.

upward to form the upper hexcone from the $V = 1$ plane. As with the single-hexcone model, the complement of any hue is located 180° farther around the double hexcone, and saturation is measured radially from the vertical axis, from 0 on the axis to 1 on the surface. Lightness is 0 for black (at the lower tip of the double hexcone) to 1 for white (at the upper tip). Color Plate II.9 shows a view of the HLS model. Again, the terms hue, lightness, and saturation in this model are similar to, but are not exactly identical to, the same terms as they were introduced in an earlier section. Color Plate II.10 shows a different view of the space.

The procedures of Figs. 13.36 and 13.37 perform the conversions between HLS and RGB. They are modified from those given by Metrick [GSPC79] to leave H UNDEFINED when $S = 0$, and to have $H = 0$ for red rather than for blue.

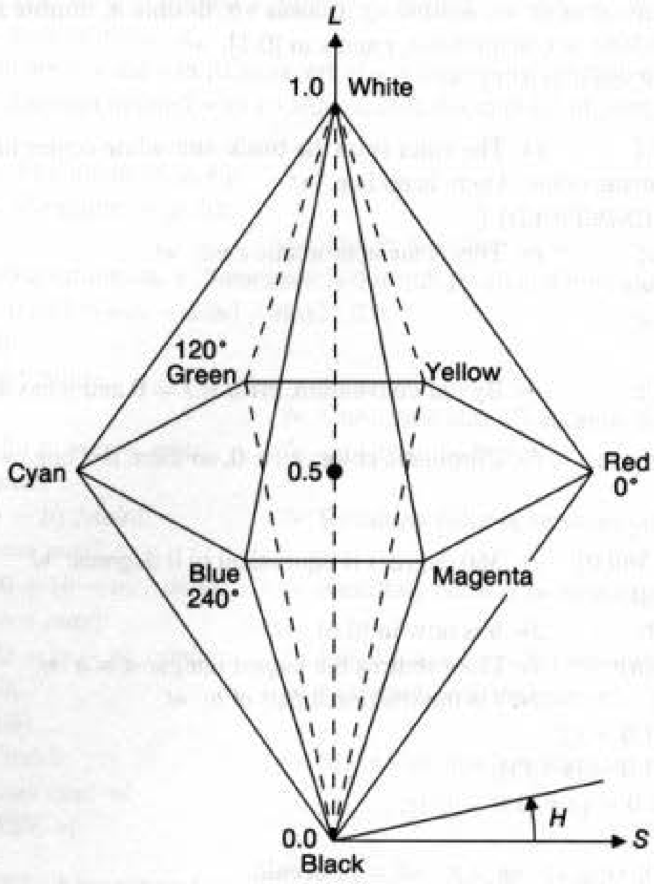


Fig. 13.35 Double-hexcone HLS color model.

The HLS model, like the HSV model, is easy to use. The grays all have $S = 0$, but the maximally saturated hues are at $S = 1, L = 0.5$. If potentiometers are used to specify the color-model parameters, the fact that L must be 0.5 to get the strongest possible colors is a disadvantage over the HSV model, in which $S = 1$ and $V = 1$ achieve the same effect. However, analogously to HSV, the colors of the $L = 0.5$ plane are *not* all of the same perceived brightness. Hence two different colors of equal perceived brightness will generally have different values of L. Furthermore, neither HLS nor any of the other models discussed thus far in this section are perceptually uniform, in the sense discussed in Section 13.2.

The recently developed Tektronix TekHVC (hue, value, chroma) color system, a modification of the CIE LUV perceptually uniform color space discussed in Section 13.2, does provide a color space in which measured and perceived distances between colors are approximately equal. This is an important advantage of both the CIE LUV and TekHVC models. Figure 13.38 shows one view of the HVC color model, and Color Plate II.11 shows another view. Details of the transformations from CIE to TekHVC have not been released. However, we see from Eq. (13.22) that the transformation from CIE XYZ to CIE LUV is computationally straightforward, with the cube root being the most computationally intense element. Thus, we expect that perceptually uniform color spaces will come to be more widely used in the future.

```
void RGB_To_HLS (double r, double g, double b, double *h, double *l, double *s)
/* Given: r, g, b each in [0,1]. */
/* Desired: h in [0,360), l and s in [0,1], except if s = 0, then h = UNDEFINED. */
{
   double max = Maximum (r, g, b);
   double min = Minimum (r, g, b);
   *l = (max + min) / 2.0;               /* This is the lightness. */
   /* Next calculate saturation */
   if (max == min) {                     /* Achromatic case, because r = g = b */
      *s = 0;
      *h = UNDEFINED;
   } else {                              /* Chromatic case */
      double delta = max - min;

      /* First calculate the saturation. */
      *s = (*l <= 0.5) ? (delta / (max + min)) : (delta / (2.0 - (max + min)));
      /* Next calculate the hue. */
      if (r == max)
         *h = (g - b) / delta;           /* Resulting color is between yellow and magenta */
      else if (g == max)
         *h = 2.0 + (b - r) / delta;     /* Resulting color is between cyan and yellow */
      else if (b == max)
         *h = 4.0 + (r - g) / delta;     /* Resulting color is between magenta and cyan */
      *h *= 60.0;                        /* Convert to degrees */
      if (h < 0.0)
         *h += 360.0;                    /* Make degrees be nonnegative */
   } /* Chromatic case */
} /* RGB_To_HLS */
```

Fig. 13.36 Algorithm for converting from RGB to HLS color space.

13.3.6 Interactive Specification of Color

Many application programs allow the user to specify colors of areas, lines, text, and so on. If only a small set of colors is provided, menu selection from samples of the available colors is appropriate. But what if the set of colors is larger than can reasonably be displayed in a menu?

The basic choices are to use English-language names, to specify the numeric coordinates of the color in a color space (either by typing or with slider dials), or to interact directly with a visual representation of the color space. Naming is in general unsatisfactory because it is ambiguous and subjective ("a light navy blue with a touch of green"), and it is also the antithesis of graphic interaction. On the other hand, [BERK82] describes CNS, a fairly well-defined color-naming scheme that uses terms such as "greenish-yellow," "green-yellow," and "yellowish-green" to distinguish three hues between green and

```
void HLS_To_RGB (double *r, double *g, double *b, double h, double l, double s)
/* Given: h in [0,360] or UNDEFINED, l and s in [0,1]. */
/* Desired: r, g, b each in [0,1] */
{
   double m1, m2;

   m2 = (l <= 0.5) ? (l * (l + s)) : (l + s - l * s);
   m1 = 2.0 * l - m2;
   if (s == 0.0) {            /* Achromatic: There is no hue. */
      if (h == UNDEFINED)
         *r = *g = *b = l;    /* This is the achromatic case. */
      else Error ();          /* Error if s = 0 and h has a value */
   } else {       /* Chromatic case, so there is a hue */
      *r = Value (m1, m2, h + 120.0);
      *g = Value (m1, m2, h);
      *b = Value (m1, m2, h - 120.0);
   }
}  /* HLS_To_RGB */

static double Value (double n1, double n2, double hue)
{
   if (hue > 360.0)
      hue -= 360.0;
   else if (hue < 0.0)
      hue += 360.0;
   if (hue < 60.0)
      return n1 + (n2 - n1) * hue / 60.0;
   else if (hue < 180.0)
      return n2;
   else if (hue < 240.0)
      return n1 + (n2 - n1) * (240.0 - hue) / 60.0;
   else
      return n1;
}  /* Value */
```

Fig. 13.37 Algorithm for converting from HLS to RGB color space.

yellow. In an experiment, users of CNS were able to specify colors more precisely than were users who entered numeric coordinates in either RGB or HSV space.

Coordinate specification can be done with slider dials, as in Fig. 13.39, using any of the color models. If the user understands how each dimension affects the color, this technique works well. Probably the best interactive specification method is to let the user interact directly with a representation of the color space, as shown in Fig. 13.40. The line on the circle (representing the $V = 1$ plane) can be dragged around to determine which slice of the HSV volume is displayed in the triangle. The cursor on the triangle can be moved around to specify saturation and value. As the line or the cursor is moved, the numeric readouts change value. When the user types new values directly into the numeric readouts, the line and cursor are repositioned. The color sample box shows the currently selected

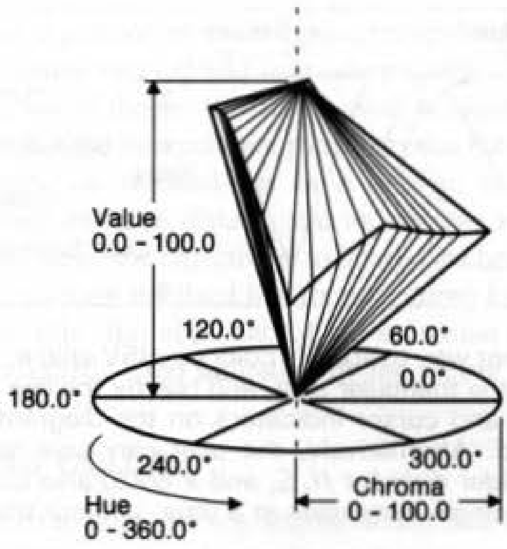


Fig. 13.38 The TekHVC color model. (Courtesy of Tektronix, Inc.)

color. Color Plate II.12 shows a similar method used for the HSV model.

[SCHW87] describes color-matching experiments in which subjects used a data tablet to specify colors in several models including RGB, YIQ, LAB [WYSZ82], and HSV. HSV was found to be slow but accurate, whereas RGB was faster but less accurate. There is a widespread belief that the HSV model is especially tractable, usually making it the model of choice.

Many interactive systems that permit user specification of color show the user the current color settings as a series of adjacent patches of color, as in part (a) of Fig. 13.39, or as the color of the single pixel value currently being set, as in part (b). As the user

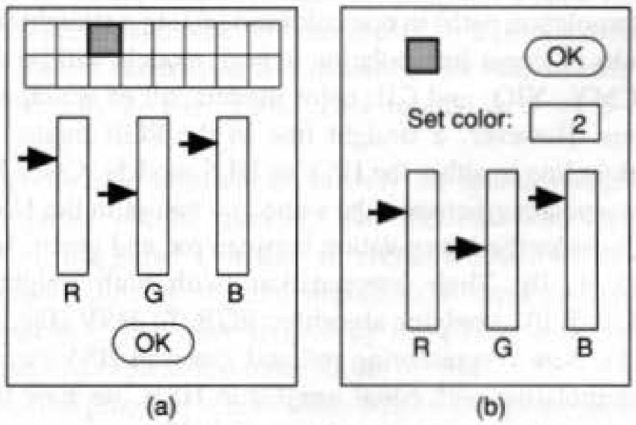


Fig. 13.39 Two common ways of setting colors. In (a), the user selects one of 16 colors to set; the selected color is designated with the thick border. The RGB slider dials control the color; OK is selected to dismiss the color control panel. In (b), the number of the color to be set is typed, and the current color is displayed in the gray-toned box. In both cases, the user must simultaneously see the actual display in order to understand the effects of the color change.

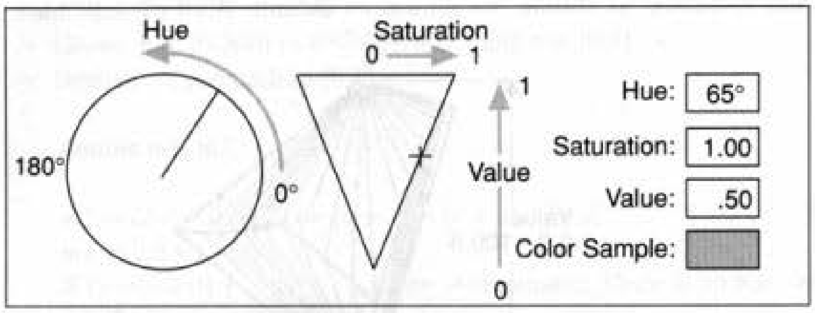


Fig. 13.40 A convenient way to specify colors in HSV space. Saturation and value are shown by the cursor in the triangular area, and hue by the line in the circular area. The user can move the line and cursor indicators on the diagrams, causing the numeric readouts to be updated. Alternatively, the user can type new values, causing the indicators to change. Slider dials for *H, S,* and *V* could also be added, giving the user accurate control over a single dimension at a time, without the need to type values.

manipulates the slider dials, the color sample changes. However, a person's perception of color is affected by surrounding colors and the sizes of colored areas, as shown in Color Plate II.13; hence, the color as perceived in the feedback area will probably differ from the color as perceived in the actual display. It is thus important that the user also see the actual display while the colors are being set.

13.3.7 Interpolating in Color Space

Color interpolation is necessary in at least three situations: for Gouraud shading (Section 16.2.4), for antialiasing (Section 3.17), and in blending two images together as for a fade-in, fade-out sequence. The results of the interpolation depend on the color model in which the colors are interpolated; thus, care must be taken to select an appropriate model.

If the conversion from one color model to another transforms a straight line (representing the interpolation path) in one color model into a straight line in the other color model, then the results of linear interpolation in both models will be the same. This is the case for the RGB, CMY, YIQ, and CIE color models, all of which are related by simple affine transformations. However, a straight line in the RGB model does *not* in general transform into a straight line in either the HSV or HLS models. Color Plate II.14 shows the results of linearly interpolating between the same two colors in the HSV, HSL, RGB, and YIQ color spaces. Consider the interpolation between red and green. In RGB, red = (1, 0, 0) and green = (0, 1, 0). Their interpolation (with both weights equal to 0.5 for convenience) is (0.5, 0.5, 0). Applying algorithm RGB_To_HSV (Fig. 13.33) to this result, we have (60°, 1, 0.5). Now, representing red and green in HSV, we have (0°, 1, 1) and (120°, 1, 1). But interpolating with equal weights in HSV, we have (60°, 1, 1); thus, the value differs by 0.5 from the same interpolation in RGB.

As a second example, consider interpolating red and cyan in the RGB and HSV models. In RGB, we start with (1, 0, 0) and (0, 1, 1), respectively, and interpolate to (0.5, 0.5, 0.5), which in HSV is represented as (UNDEFINED, 0, 0.5). In HSV, red and cyan are (0°, 1, 1) and (180°, 1, 1). Interpolating, we have (90°, 1, 1); a new hue at maximum value and saturation has been introduced, whereas the "right" result of combining equal amounts of complementary colors is a gray value. Here, again, interpolating and then transforming gives different results from transforming and then interpolating.

For Gouraud shading, any of the models can be used, because the two interpolants are generally so close together that the interpolation paths between the colors are close together as well. When two images are blended—as in a fade-in, fade-out sequence or for antialiasing—the colors may be quite distant, and an additive model, such as RGB, is appropriate. If, on the other hand, the objective is to interpolate between two colors of fixed hue (or saturation) and to maintain the fixed hue (or saturation) for all interpolated colors, then HSV or HLS is preferable. But note that a fixed-saturation interpolation in HSV or HLS is *not* seen as having exactly fixed saturation by the viewer [WARE87].

13.4 REPRODUCING COLOR

Color images are reproduced in print in a way similar to that used for monochrome images, but four sets of halftone dots are printed, one for each of the subtractive primaries, and another for black. In a process called *undercolor removal*, black replaces equal amounts of cyan, magenta, and yellow. This creates a darker black than is possible by mixing the three primaries, and hastens drying by decreasing the amounts of cyan, magenta, and yellow ink needed. The orientation of each of the grids of dots is different, so that interference patterns are not created. Color Plate II.15 shows an enlarged halftone color pattern. Our eyes spatially integrate the light reflected from adjacent dots, so that we see the color defined by the proportions of primaries in adjacent dots. This spatial integration of different colors is the same phenomenon we experience when viewing the triads of red, green, and blue dots on a color monitor.

We infer, then, that color reproduction in print and on CRTs depends on the same spatial integration used in monochrome reproduction. The monochrome dithering techniques discussed in Section 13.1.2 can also be used with color to extend the number of available colors, again at the expense of resolution. Consider a color display with 3 bits per pixel, one each for red, green, and blue. We can use a 2×2 pixel pattern area to obtain 125 different colors: each pattern can display five intensities for each of red, green, and blue, by using the halftone patterns in Fig. 13.8. This results in $5 \times 5 \times 5 = 125$ color combinations.

Not all color reproduction depends exclusively on spatial integration. For instance, xerographic color copiers, ink-jet plotters, and thermal color printers actually mix subtractive pigments on the paper's surface to obtain a small set of different colors. In xerography, the colored pigments are first deposited in three successive steps, then are heated and melted together. The inks sprayed by the plotter mix before drying. Spatial integration may be used to expand the color range further.

A related quantization problem occurs when a color image with n bits per pixel is to be displayed on a display of $m < n$ bits per pixel with no loss of spatial resolution. Here, color resolution *must* be sacrificed. In this situation, there are two key questions: Which 2^m colors should be displayed? What is the mapping from the set of 2^n colors in the image to the smaller set of 2^m colors being displayed?

The simple answers are to use a predefined set of display colors and a fixed mapping from image colors to display colors. For instance, with $m = 8$, a typical assignment is 3 bits to red and green and 2 to blue (because of the eye's lower sensitivity to blue). Thus, the 256 displayable colors are all the combinations of eight reds, eight greens, and four blues. The specific values for the red, green, and blue colors would be spaced across the range of 0.0 to 1.0 range on a ratio scale, as discussed in Section 13.1.1. For an image with 6 bits per color and hence 64 levels for each color, the 64 red colors are mapped into one of the eight displayable reds, and similarly for the greens. The 64 blue colors are mapped into just four blues.

With this solution, however, if all the blue image colors are clustered together in color space, they might all be displayed as the same blue, whereas the other three displayable blues might go unused. An adaptive scheme would take this possibility into account and would divide up the blue-value range on the basis of the distribution of values in the range. Heckbert [HECK82] describes two approaches of this type: the popularity algorithm and the median-cut algorithm.

The *popularity algorithm* creates a histogram of the image's colors, and uses the 2^m most frequent colors in the mapping. The *median-cut algorithm* recursively fits a box around the colors used in the image, splitting the box along its longer dimension at the median in that dimension. The recursion ends when 2^m boxes have been created; the centroid of each box is used as the display color for all image colors in the box. The median-cut algorithm is slower than is the popularity algorithm, but produces better results. Another way of splitting the boxes is described in [WAN88].

Creating an *accurate* color reproduction is much more difficult than is approximating colors. Two display monitors can be calibrated to create the same tristimulus values; the multistep process is described in detail in [COWA83; MEYE83]. The steps are to measure the chromaticity coordinates of the monitor phosphors, then to adjust the brightness and contrast controls of each of the monitor guns so that the same white chromaticity is produced whenever $R = G = B$, and to determine the appropriate gamma correction for each gun.

Making slides or movie film that look exactly like the image on a display is difficult, because many variables are involved. They include the gamma correction of the display and of the CRT used in the film recorder; the color of light emitted by the CRT in the film recorder; the filters used in the film recorder; the type of film used; the quality and temperature of the developing chemicals, the length of time the film is in the chemicals, and the color of light emitted by the bulb in the slide or film projector. Fortunately, all these variables can be quantified and controlled, albeit with considerable difficulty.

Controlling the color match on printed materials is also difficult; the printing process, with its cyan, magenta, yellow, and black primaries, requires careful quality control to maintain registration and ink flow. The paper texture, absorbancy, and gloss also affect the result. Complicating matters further, the simple subtractive CMY color model discussed in Section 13.3.2 cannot be used directly, because it does not take into account these complications of the printing process. More detail can be found in [STON88].

Even if extreme care is taken in color reproduction, the results may not seem to match the original. Lighting conditions and reflections from the display can cause colors with the same measured chromaticity coordinates to appear to be different. Fortunately, the purpose of the reproduction is usually (although not always) to maintain color relationships between different parts of the image, rather than to make an exact copy.

13.5 USING COLOR IN COMPUTER GRAPHICS

We use color for aesthetics, to establish a tone or mood, for realism, as a highlight, to identify associated areas as being associated, and for coding. With care, color can be used effectively for these purposes. In addition, users tend to like color, even when there is no quantitative evidence that it helps their performance. Although cost-conscious buyers may scoff at color monitors, we believe that anything that encourages people to use computers is important!

Careless use of color can make the display less useful or less attractive than a corresponding monochrome presentation. In one experiment, introduction of meaningless color reduced user performance to about one-third of what it was without color [KREB79]. Color should be employed conservatively. Any decorative use of color should be subservient to the functional use, so that the color cannot be misinterpreted as having some underlying meaning. Thus, the use of color, like all other aspects of a user–computer interface, must be tested with real users to identify and remedy problems. Of course, some individuals may have other preferences, so it is common practice to provide defaults chosen on the basis of color usage rules, with some means for the user to change the defaults. A conservative approach to color selection is to design first for a monochrome display, to ensure that color use is purely redundant. This avoids creating problems for color-deficient users and also means that the application can be used on a monochrome display. Additional information on color deficiencies is given in [MEYE88]. The color choices used in the window managers shown in Color Plates I.26 through I.29 are quite conservative. Color is not used as a unique code for button status, selected menu item, and so forth.

Many books have been written on the use of color for aesthetic purposes, including [BIRR61]; we state here just a few of the simpler rules that help to produce color harmony. The most fundamental rule of color aesthetics is to select colors according to some method, typically by traversing a smooth path in a color model or by restricting the colors to planes or hexcones in a color space. This might mean using colors of constant lightness or value. Furthermore, colors are best spaced at equal *perceptual* distances (this is not the same as being at equally spaced increments of a coordinate, and can be difficult to implement). Recall too that linear interpolation (as in Gouraud shading) between two colors produces different results in different color spaces (see Exercise 13.10 and Color Plate II.14).

A random selection of different hues and saturations is usually quite garish. Alvy Ray Smith performed an informal experiment in which a 16 × 16 grid was filled with randomly generated colors. Not unexpectedly, the grid was unattractive. Sorting the 256 colors according to their H, S, and V values and redisplaying them on the grid in their new order improved the appearance of the grid remarkably.

More specific instances of these rules suggest that, if a chart contains just a few colors, the complement of one of the colors should be used as the background. A neutral (gray) background should be used for an image containing many different colors, since it is both harmonious and inconspicuous. If two adjoining colors are not particularly harmonious, a thin black border can be used to set them apart. This use of borders is also more effective for the achromatic (black/white) visual channel, since shape detection is facilitated by the black outline. Some of these rules are encoded in ACE (A Color Expert), an expert system for selecting user-interface colors [MEIE88]. In general, it is good to minimize the number of different colors being used (except for shading of realistic images).

Color can be used for coding, as discussed in Chapter 9 and illustrated by Color Plate II.16. However, several cautions are in order. First, color codes can easily carry unintended meanings. Displaying the earnings of company A as red and those of company B as green might well suggest that company A is in financial trouble, because of our learned associations of colors with meanings. Bright, saturated colors stand out more strongly than do dimmer, paler colors, and may give unintended emphasis. Two elements of a display that have the same color may be seen as related by the same color code, even if they are not.

This problem often arises when color is used both to group menu items and to distinguish display elements, such as different layers of a printed circuit board or VLSI chip; for example, green display elements tend to be associated with menu items of the same color. This is one of the reasons that use of color in user-interface elements, such as menus, dialogue boxes, and window borders, should be restrained. (Another reason is to leave as many colors as possible free for the application program itself.)

A number of color usage rules are based on physiological rather than aesthetic considerations. For example, because the eye is more sensitive to spatial variation in intensity than it is to variation in chromaticity, lines, text, and other fine detail should vary from the background not just in chromaticity, but in brightness (perceived intensity) as well—especially for colors containing blue, since relatively few cones are sensitive to blue. Thus, the edge between two equal-brightness colored areas that differ only in the amount of blue will be fuzzy. On the other hand, blue-sensitive cones spread out farther on the retina than do red- and green-sensitive ones, so our peripheral color vision is better for blue (this is why many police-car flashers are now blue instead of red).

Blue and black differ very little in brightness, and are thus a particularly bad combination. Similarly, yellow on white is relatively hard to distinguish, because both colors are quite bright (see Exercise 13.11). Color plates I.28 and I.29 show a very effective use of yellow to highlight black text on a white background. The yellow contrasts very well with the black text and also stands out. In addition, the yellow highlight is not as overpowering as a black highlight with reversed text (that is, with the highlighted text in white on a black highlight), as is common on monochrome displays.

White text on a blue background provides a good contrast that is less harsh than white on black. It is good to avoid reds and greens with low saturation and luminance, as these are the colors confused by those of us who are red–green color blind, the most common form of color-perception deficiency. Meyer and Greenberg describe effective ways to choose colors for color-blind viewers [MEYE88].

The eye cannot distinguish the color of very small objects, as already remarked in connection with the YIQ NTSC color model, so color coding should not be applied to small objects. In particular, judging the color of objects subtending less than 20 to 40 minutes of arc is error-prone [BISH60, HAEU76]. An object 0.1 inches high, viewed from 24 inches (a typical viewing distance) subtends this much arc, which corresponds to about 7 pixels of height on a 1024-line display with a vertical height of 15 inches. It is clear that the color of a single pixel is quite difficult to discern (see Exercise 13.18).

The perceived color of a colored area is affected by the color of the surrounding area, as is very evident in Color Plate II.13; this effect is particularly problematic if colors are used to encode information. The effect is minimized when the surrounding areas are some shade of gray or are relatively unsaturated colors.

The color of an area can actually affect its perceived size. Cleveland and McGill discovered that a red square is perceived as larger than is a green square of equal size [CLEV83]. This effect could well cause the viewer to attach more importance to the red square than to the green ones.

If a user stares at a large area of highly saturated color for several seconds and then looks elsewhere, an afterimage of the large area will appear. This effect is disconcerting, and causes eye strain. Use of large areas of saturated colors is hence unwise. Also, large areas of different colors can appear to be at different distances from the viewer, because the index of refraction of light depends on wavelength. The eye changes its focus as the viewer's gaze moves from one colored area to another, and this change in focus gives the impression of differing depths. Red and blue, which are at opposite ends of the spectrum, have the strongest depth-disparity effect, with red appearing closer and blue more distant. Hence, simultaneously using blue for foreground objects and red for the background is unwise; the converse is fine.

With all these perils and pitfalls of color usage, is it surprising that one of our first-stated rules was to apply color conservatively?

13.6 SUMMARY

The importance of color in computer graphics will continue to increase as color monitors and color hardcopy devices become the norm in many applications. In this chapter, we have introduced those color concepts most relevant to computer graphics; for more information, see the vast literature on color, such as [BILL81; BOYN79; GREG66; HUNT87; JUDD75; WYSZ82]. More background on artistic and aesthetic issues in the use of color in computer graphics can be found in [FROM84; MARC82; MEIE88; MURC85]. The difficult problems of precisely calibrating monitors and matching the colors appearing on monitors with printed colors are discussed in [COWA83; STON88].

EXERCISES

13.1 Derive an equation for the number of intensities that can be represented by $m \times m$ pixel patterns, where each pixel has w bits.

13.2 Write the programs needed to gamma-correct a black-and-white display through a look-up table. Input parameters are γ, I_0, m, the number of intensities desired, and K, the constant in Eq. (13.5).

13.3 Write an algorithm to display a pixel array on a bilevel output device. The inputs to the algorithm are an $m \times m$ array of pixel intensities, with w bits per pixel, and an $n \times n$ growth sequence matrix. Assume that the output device has resolution of $m \cdot n \times m \cdot n$.

13.4 Repeat Exercise 13.3 by using ordered dither. Now the output device has resolution $m \times m$, the same as the input array of pixel intensities.

13.5 Write an algorithm to display a filled polygon on a bilevel device by using an $n \times n$ filling pattern.

13.6 When certain patterns are used to fill a polygon being displayed on an interlaced raster display, all of the "on" bits fall on either the odd or the even scan lines, introducing a slight amount of flicker. Revise the algorithm from Exercise 13.5 to permute rows of the $n \times n$ pattern so that alternate

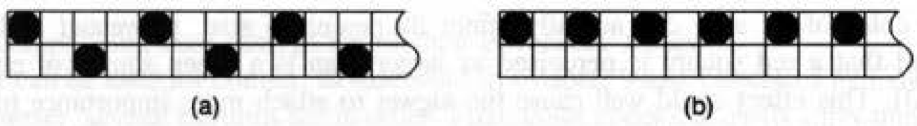


Fig. 13.41 Results obtained by using intensity level 1 from Fig. 13.8 in two ways: (a) with alternation (intensified pixels are on both scan lines), and (b) without alternation (all intensified pixels are on the same scan line).

replications of the pattern will alternate use of the odd and even scan lines. Figure 13.41 shows the results obtained by using intensity level 1 from Fig. 13.8, with and without this alternation.

13.7 Given a spectral energy distribution, how would you find the dominant wavelength, excitation purity, and luminance of the color it represents?

13.8 Plot the locus of points of the constant luminance values 0.25, 0.50, and 0.75, defined by $Y = 0.30R + 0.59G + 0.11B$, on the RGB cube, the HLS double hexcone, and the HSV hexcone.

13.9 Why are the opposite ends of the spectrum in the CIE diagram connected by a *straight* line?

13.10 Express, in terms of R, G, and B: the I of YIQ, the V of HSV, and the L of HSL. Note that I, V, and L are not the same.

13.11 Calculate in YIQ color space the luminances of the additive and subtractive primaries. Rank the primaries by luminance. This ranking gives their relative intensities, both as displayed on a black-and-white television and as perceived by our eyes.

13.12 Discuss the design of a raster display that uses HSV or HLS, instead of RGB, as its color specification.

13.13 In which color models are the rules of color harmony most easily applied?

13.14 Verify that Eq. (13.27) can be rewritten as Eq. (13.29) when $R = G = B = 1$.

13.15 If the white color used to calculate C_r, C_g, and C_b in Eq. (13.29) is given by x_w, y_w, and Y_w rather than by X_w, Y_w, and Z_w, what are the algebraic expressions for C_r, C_g, and C_b?

13.16 Rewrite the HSV-to-RGB conversion algorithm to make it more efficient. Replace the assignment statements for p, q, and t with: $vs = v * s$; $vsf = vs * f$; $p = v - vs$; $q = v - vsf$; $t = p + vsf$. Also assume that R, G, and B are in the interval [0, 255], and see how many of the computations can be converted to integer.

13.17 Write a program that displays, side by side, two 16×16 grids. Fill each grid with colors. The left grid will have 256 colors randomly selected from HSV color space (created by using a random-number generator to choose one out of 10 equally spaced values for each of H, S, and V). The right grid contains the same 256 colors, sorted on H, S, and V. Experiment with the results obtained by varying which of H, S, and V is used as the primary sort key.

13.18 Write a program to display on a gray background small squares colored orange, red, green, blue, cyan, magenta, and yellow. Each square is separated from the others and is of size $n \times n$ pixels, where n is an input variable. How large must n be so that the colors of each square can be unambiguously judged from distances of 24 and of 48 inches? What should be the relation between the two values of n? What effect, if any, do different background colors have on this result?

13.19 Calculate the number of bits of look-up–table accuracy needed to store 256 different intensity levels given dynamic intensity ranges of 50, 100, and 200.

13.20 Write a program to interpolate linearly between two colors in RGB, HSV, and HSL. Accept the two colors as input, allowing them to be specified in any of these three models.

14
The Quest for Visual Realism

In previous chapters, we discussed graphics techniques involving simple 2D and 3D primitives. The pictures that we produced, such as the wireframe houses of Chapter 6, represent objects that in real life are significantly more complex in both structure and appearance. In this chapter, we introduce an increasingly important application of computer graphics: creating realistic images of 3D scenes.

What is a *realistic* image? In what sense a picture, whether painted, photographed, or computer-generated, can be said to be "realistic" is a subject of much scholarly debate [HAGE86]. We use the term rather broadly to refer to a picture that captures many of the effects of light interacting with real physical objects. Thus, we treat realistic images as a continuum and speak freely of pictures, and of the techniques used to create them, as being "more" or "less" realistic. At one end of the continuum are examples of what is often called *photographic realism* (or *photorealism*). These pictures attempt to synthesize the field of light intensities that would be focused on the film plane of a camera aimed at the objects depicted. As we approach the other end of the continuum, we find images that provide successively fewer of the visual cues we shall discuss.

You should bear in mind that a more realistic picture is not necessarily a more desirable or useful one. If the ultimate goal of a picture is to convey information, then a picture that is free of the complications of shadows and reflections may well be more successful than a *tour de force* of photographic realism. In addition, in many applications of the techniques outlined in the following chapters, reality is intentionally altered for aesthetic effect or to fulfill a naive viewer's expectations. This is done for the same reasons that science-fiction films feature the sounds of weapon blasts in outer space—an impossibility in a vacuum. For example, in depicting Uranus in Color Plate II.20, Blinn shined an extra light on the night side of the planet and stretched the contrast to make all features visible simultaneously—the night side of the planet would have been black otherwise. Taking liberties with physics can result in attractive, memorable, and useful pictures!

Creating realistic pictures involves a number of stages that are treated in detail in the following chapters. Although these stages are often thought of as forming a conceptual pipeline, the order in which they are performed can vary, as we shall see, depending on the algorithms used. First, models of the objects are generated using methods discussed in Chapters 11, 12, and 20. Next, a viewing specification (as developed in Chapter 6) and lighting conditions are selected. Those surfaces visible to the viewer are then determined by algorithms discussed in Chapter 15. The color assigned to each pixel in a visible surface's projection is a function of the light reflected and transmitted by the objects and is determined by methods treated in Chapter 16. The resulting picture can then be combined with previously generated ones (e.g., to reuse a complex background) by using the compositing techniques of Chapter 17. Finally, if we are producing an animated sequence, time-varying changes in the models, lighting, and viewing specifications must be defined, as discussed in Chapter 21. The process of creating images from models is often called *rendering*. The term *rasterization* is also used to refer specifically to those steps that involve determining pixel values from input geometric primitives.

This chapter presents realistic rendering from a variety of perspectives. First, we look at some of the applications in which realistic images have been used. Then, we examine, in roughly historical progression, a series of techniques that make it possible to create successively more realistic pictures. Each technique is illustrated by a picture of a standard scene with the new technique applied to it. Next, we examine the problems caused by aliasing, which must be dealt with when images are represented as discrete arrays of pixels. Finally, we conclude with suggestions about how to approach the following chapters.

14.1 WHY REALISM?

The creation of realistic pictures is an important goal in fields such as simulation, design, entertainment and advertising, research and education, and command and control.

Simulation systems present images that not only are realistic, but also change dynamically. For example, a flight simulator shows the view that would be seen from the cockpit of a moving plane. To produce the effect of motion, the system generates and displays a new, slightly different view many times per second. Simulators such as those shown in Color Plate I.5 have been used to train the pilots of spacecraft, airplanes, and boats—and, more recently, drivers of cars.

Designers of 3D objects such as automobiles, airplanes, and buildings want to see how their preliminary designs look. Creating realistic computer-generated images is often an easier, less expensive, and more effective way to see preliminary results than is building models and prototypes, and also allows more alternative designs to be considered. If the design work itself is also computer-based, a digital description of the object may already be available to use in creating the images. Ideally, the designer can also interact with the displayed image to modify the design. Color Plate II.17 shows an image produced by an automotive-design system to determine what a car will look like under a variety of lighting conditions. Realistic graphics is often coupled with programs that analyze other aspects of the object being designed, such as its mass properties or its response to stress.

Computer-generated imagery is used extensively in the entertainment world, both in traditional animated cartoons and in realistic and surrealistic images for logos, advertisements, and science-fiction movies (see Color Plates D, F, I.11, I.12, and II.18). Computer-generated cartoons can mimic traditional animation, but can also transcend manual techniques by introducing more complicated motion and richer or more realistic imagery. Some complex realistic images can be produced at less cost than filming them from physical models of the objects. Other images have been generated that would have been extremely difficult or impossible to stage with real models. Special-purpose hardware and software created for use in entertainment include sophisticated paint systems and real-time systems for generating special effects and for combining images. As technology improves, home and arcade video games generate increasingly realistic images.

Realistic images are becoming an essential tool in research and education. A particularly important example is the use of graphics in molecular modeling, as shown in Color Plate II.19. It is interesting how the concept of realism is stretched here: The realistic depictions are not of "real" atoms, but rather of stylized ball-and-stick and volumetric models that allow larger structures to be built than are feasible with physical models, and that permit special effects, such as animated vibrating bonds and color changes representing reactions. On a macroscopic scale, movies made at JPL show NASA space-probe missions, depicted in Color Plate II.20.

Another application for realistic imagery is in command and control, in which the user needs to be informed about and to control the complex process represented by the picture. Unlike simulations, which attempt to mimic what a user would actually see and feel in the simulated situation, command and control applications often create symbolic displays that emphasize certain data and suppress others to aid in decision making.

14.2 FUNDAMENTAL DIFFICULTIES

A fundamental difficulty in achieving total visual realism is the complexity of the real world. Observe the richness of your environment. There are many surface textures, subtle color gradations, shadows, reflections, and slight irregularities in the surrounding objects. Think of patterns on wrinkled cloth, the texture of skin, tousled hair, scuff marks on the floor, and chipped paint on the wall. These all combine to create a "real" visual experience. The computational costs of simulating these effects can be high: Creating pictures such as those of Color Plates A–H can take many minutes or even hours on powerful computers.

A more easily met subgoal in the quest for realism is to provide sufficient information to let the viewer understand the 3D spatial relationships among several objects. This subgoal can be achieved at a significantly lower cost and is a common requirement in CAD and in many other application areas. Although highly realistic images convey 3D spatial relationships, they usually convey much more as well. For example, Fig. 14.1, a simple line drawing, suffices to persuade us that one building is partially behind the other. There is no need to show building surfaces filled with shingles and bricks, or shadows cast by the buildings. In fact, in some contexts, such extra detail may only distract the viewer's attention from more important information being depicted.

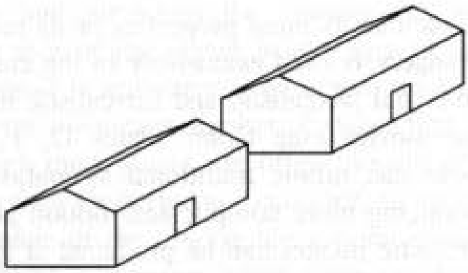


Fig. 14.1 Line drawing of two houses.

One long-standing difficulty in depicting spatial relationships is that most display devices are 2D. Therefore, 3D objects must be projected into 2D, with considerable attendant loss of information—which can sometimes create ambiguities in the image. Some of the techniques introduced in this chapter can be used to add back information of the type normally found in our visual environment, so that human depth-perception mechanisms resolve the remaining ambiguities properly.

Consider the Necker cube illusion of Fig. 14.2(a), a 2D projection of a cube; we do not know whether it represents the cube in part (b) or that in part (c) of this figure. Indeed, the viewer can easily "flip-flop" between the alternatives, because Fig. 14.2(a) does not contain enough visual information for an unambiguous interpretation.

The more the viewers know about the object being displayed, the more readily they can form what Gregory calls an *object hypothesis* [GREG70]. Figure 14.3 shows the Schröder stairway illusion—are we looking down a stairway, or looking up from underneath it? We are likely to choose the former interpretation, probably because we see stairways under our feet more frequently than over our heads and therefore "know" more about stairways viewed from above. With a small stretch of the imagination, however, we can visualize the alternative interpretation of the figure. Nevertheless, with a blink of the eye, a reversal occurs for most viewers and the stairway again appears to be viewed from above. Of course, additional context, such as a person standing on the steps, will resolve the ambiguity.

In the following sections, we list some of the steps along the path toward realistic images. The path has actually been a set of intertwined trails, rather than a single straight road, but we have linearized it for the sake of simplicity, providing a purely descriptive introduction to the detailed treatment in subsequent chapters. We mention first techniques applicable to static line drawings. These methods concentrate on ways to present the 3D spatial relationships among several objects on a 2D display. Next come techniques for

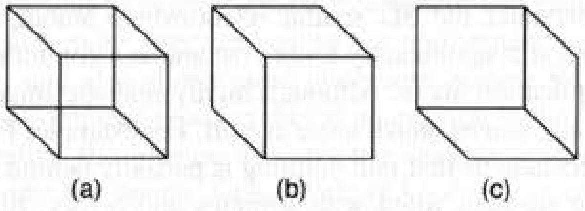


Fig. 14.2 The Necker cube illusion. Is the cube in (a) oriented like the cube in (b) or like that in (c)?

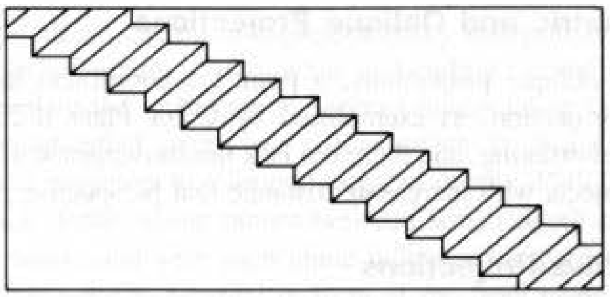


Fig. 14.3 The Schröder stairway illusion. Is the stairway being viewed from above or from below?

shaded images, made possible by raster graphics hardware, that concentrate on the interaction of objects with light. These are followed by the issues of increased model complexity and dynamics, applicable to both line and shaded pictures. Finally, we discuss the possibilities of true 3D images, advances in display hardware, and the future place of picture generation in the context of full, interactive environmental synthesis.

14.3 RENDERING TECHNIQUES FOR LINE DRAWINGS

In this section, we focus on a subgoal of realism: showing 3D depth relationships on a 2D surface. This goal is served by the planar geometric projections defined in Chapter 6.

14.3.1 Multiple Orthographic Views

The easiest projections to create are parallel orthographics, such as plan and elevation views, in which the projection plane is perpendicular to a principal axis. Since depth information is discarded, plan and elevations are typically shown together, as with the top, front, and side views of a block letter "L" in Fig. 14.4. This particular drawing is not difficult to understand; however, understanding drawings of complicated manufactured parts from a set of such views may require many hours of study. Training and experience sharpen one's interpretive powers, of course, and familiarity with the types of objects being represented hastens the formulation of a correct object hypothesis. Still, scenes as complicated as that of our "standard scene" shown in Color Plate II.21 are often confusing when shown in only three such projections. Although a single point may be unambiguously located from three mutually perpendicular orthographics, multiple points and lines may conceal one another when so projected.

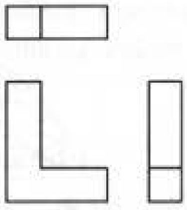


Fig. 14.4 Front, top, and side orthographic projections of the block letter "L."

14.3.2 Axonometric and Oblique Projections

In axonometric and oblique projections, a point's z coordinate influences its x and y coordinates in the projection, as exemplified by Color Plate II.22. These projections provide constant foreshortening, and therefore lack the convergence of parallel lines and the decreasing size of objects with increasing distance that perspective projection provides.

14.3.3 Perspective Projections

In perspective projections, an object's size is scaled in inverse proportion to its distance from the viewer. The perspective projection of a cube shown in Fig. 14.5 reflects this scaling. There is still ambiguity, however; the projection could just as well be a picture frame, or the parallel projection of a truncated pyramid, or the perspective projection of a rectangular parallelepiped with two equal faces. If one's object hypothesis is a truncated pyramid, then the smaller square represents the face closer to the viewer; if the object hypothesis is a cube or rectangular parallelepiped, then the smaller square represents the face farther from the viewer.

Our interpretation of perspective projections is often based on the assumption that a smaller object is farther away. In Fig. 14.6, we would probably assume that the larger house is nearer to the viewer. However, the house that appears larger (a mansion, perhaps) may actually be more distant than the one that appears smaller (a cottage, for example), at least as long as there are no other cues, such as trees and windows. When the viewer knows that the projected objects have many parallel lines, perspective further helps to convey depth, because the parallel lines seem to converge at their vanishing points. This convergence may actually be a stronger depth cue than the effect of decreasing size. Color Plate II.23 shows a perspective projection of our standard scene.

14.3.4 Depth Cueing

The depth (distance) of an object can be represented by the intensity of the image: Parts of objects that are intended to appear farther from the viewer are displayed at lower intensity (see Color Plate II.24). This effect is known as *depth cueing*. Depth cueing exploits the fact that distant objects appear dimmer than closer objects, especially if seen through haze. Such effects can be sufficiently convincing that artists refer to the use of changes in intensity (as well as in texture, sharpness, and color) to depict distance as *aerial perspective*. Thus, depth cueing may be seen as a simplified version of the effects of atmospheric attenuation.

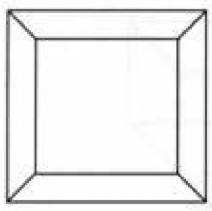


Fig. 14.5 Perspective projection of a cube.

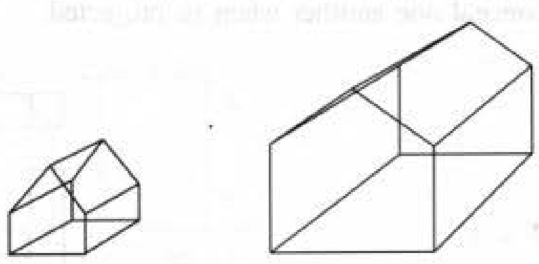


Fig. 14.6 Perspective projection of two houses.

In vector displays, depth cueing is implemented by interpolating the intensity of the beam along a vector as a function of its starting and ending z coordinates. Color graphics systems usually generalize the technique to support interpolating between the color of a primitive and a user-specified depth-cue color, which is typically the color of the background. To restrict the effect to a limited range of depths, PHIGS+ allows the user to specify front and back depth-cueing planes between which depth cueing is to occur. A separate scale factor associated with each plane indicates the proportions of the original color and the depth-cue color to be used in front of the front plane and behind the back plane. The color of points between the planes is linearly interpolated between these two values. The eye's intensity resolution is lower than its spatial resolution, so depth cueing is not useful for accurately depicting small differences in distance. It is quite effective, however, in depicting large differences, or as an exaggerated cue in depicting small ones.

14.3.5 Depth Clipping

Further depth information can be provided by *depth clipping*. The back clipping plane is placed so as to cut through the objects being displayed, as shown in Color Plate II.25. Partially clipped objects are then known by the viewer to be cut by the clipping plane. A front clipping plane may also be used. By allowing the position of one or both planes to be varied dynamically, the system can convey more depth information to the viewer. Back-plane depth clipping can be thought of as a special case of depth cueing: In ordinary depth cueing, intensity is a smooth function of z; in depth clipping, it is a step function. Color Plate II.25 combines both techniques. A technique related to depth clipping is highlighting all points on the object intersected by some plane. This technique is especially effective when the slicing plane is shown moving through the object dynamically, and has even been used to help illustrate depth along a fourth dimension [BANC77].

14.3.6 Texture

Simple vector textures, such as cross-hatching, may be applied to an object. These textures follow the shape of an object and delineate it more clearly. Texturing one of a set of otherwise identical faces can clarify a potentially ambiguous projection. Texturing is especially useful in perspective projections, as it adds yet more lines whose convergence and foreshortening may provide useful depth cues.

14.3.7 Color

Color may be used symbolically to distinguish one object from another, as in Color Plate II.26, in which each object has been assigned a different color. Color can also be used in line drawings to provide other information. For example, the color of each vector of an object may be determined by interpolating colors that encode the temperatures at the vector's endpoints.

14.3.8 Visible-Line Determination

The last line-drawing technique we mention is *visible-line determination* or *hidden-line removal,* which results in the display of only visible (i.e., unobscured) lines or parts of lines. Only surfaces, bounded by edges (lines), can obscure other lines. Thus, objects that are to block others must be modeled either as collections of surfaces or as solids.

Color Plate II.27 shows the usefulness of hidden-line removal. Because hidden-line–removed views conceal *all* the internal structure of opaque objects, they are not necessarily the most effective way to show depth relations. Hidden-line–removed views convey less depth information than do exploded and cutaway views. Showing hidden lines as dashed lines can be a useful compromise.

14.4 RENDERING TECHNIQUES FOR SHADED IMAGES

The techniques mentioned in Section 14.3 can be used to create line drawings on both vector and raster displays. The techniques introduced in this section exploit the ability of raster devices to display shaded areas. When pictures are rendered for raster displays, problems are introduced by the relatively coarse grid of pixels on which smooth contours and shading must be reproduced. The simplest ways to render shaded pictures fall prey to the problem of aliasing, first encountered in Section 3.17. In Section 14.10, we introduce the theory behind aliasing, and explain how to combat aliasing through antialiasing. Because of the fundamental role that antialiasing plays in producing high-quality pictures, all the pictures in this section have been created with antialiasing.

14.4.1 Visible-Surface Determination

By analogy to visible-line determination, *visible-surface determination* or *hidden-surface removal,* entails displaying only those parts of surfaces that are visible to the viewer. As we have seen, simple line drawings can often be understood without visible-line determination. When there are few lines, those in front may not seriously obstruct our view of those behind them. In raster graphics, on the other hand, if surfaces are rendered as opaque areas, then visible-surface determination is essential for the picture to make sense. Color Plate II.28 shows an example in which all faces of an object are painted the same color.

14.4.2 Illumination and Shading

A problem with Color Plate II.28 is that each object appears as a flat silhouette. Our next step toward achieving realism is to shade the visible surfaces. Ultimately, each surface's appearance should depend on the types of light sources illuminating it, its properties (color, texture, reflectance), and its position and orientation with respect to the light sources, viewer, and other surfaces.

In many real visual environments, a considerable amount of *ambient light* impinges from all directions. Ambient light is the easiest kind of light source to model, because in a simple lighting model it is assumed to produce constant illumination on all surfaces, regardless of their position or orientation. Using ambient light by itself produces very unrealistic images, however, since few real environments are illuminated solely by uniform ambient light. Color Plate II.28 is an example of a picture shaded this way.

A *point source,* whose rays emanate from a single point, can approximate a small incandescent bulb. A *directional source,* whose rays all come from the same direction, can be used to represent the distant sun by approximating it as an infinitely distant point source.

Modeling these sources requires additional work because their effect depends on the surface's orientation. If the surface is *normal* (perpendicular) to the incident light rays, it is brightly illuminated; the more oblique the surface is to the light rays, the less its illumination. This variation in illumination is, of course, a powerful cue to the 3D structure of an object. Finally, a *distributed* or *extended source,* whose surface area emits light, such as a bank of fluorescent lights, is even more complex to model, since its light comes from neither a single direction nor a single point. Color Plate II.29 shows the effect of illuminating our scene with ambient and point light sources, and shading each polygon separately.

14.4.3 Interpolated Shading

Interpolated shading is a technique in which shading information is computed for each polygon vertex and interpolated across the polygons to determine the shading at each pixel. This method is especially effective when a polygonal object description is intended to approximate a curved surface. In this case, the shading information computed at each vertex can be based on the surface's actual orientation at that point and is used for all of the polygons that share that vertex. Interpolating among these values across a polygon approximates the smooth changes in shade that occur across a curved, rather than planar, surface.

Even objects that are supposed to be polyhedral, rather than curved, can benefit from interpolated shading, since the shading information computed for each vertex of a polygon may differ, although typically much less dramatically than for a curved object. When shading information is computed for a true polyhedral object, the value determined for a polygon's vertex is used only for that polygon and not for others that share the vertex. Color Plate II.30 shows Gouraud shading, a kind of interpolated shading discussed in Section 16.2.

14.4.4 Material Properties

Realism is further enhanced if the *material properties* of each object are taken into account when its shading is determined. Some materials are dull and disperse reflected light about equally in all directions, like a piece of chalk; others are shiny and reflect light only in certain directions relative to the viewer and light source, like a mirror. Color Plate II.31 shows what our scene looks like when some objects are modeled as shiny. Color Plate II.32 uses Phong shading, a more accurate interpolated shading method (Section 16.2).

14.4.5 Modeling Curved Surfaces

Although interpolated shading vastly improves the appearance of an image, the object geometry is still polygonal. Color Plate II.33 uses object models that include curved surfaces. Full shading information is computed at each pixel in the image.

14.4.6 Improved Illumination and Shading

One of the most important reasons for the "unreal" appearance of most computer graphics images is the failure to model accurately the many ways that light interacts with objects.

Color Plate II.34 uses better illumination models. Sections 16.7–13 discuss progress toward the design of efficient, physically correct illumination models, resulting in pictures such as Color Plates III.19–III.29 and the jacket of this book (Color Plate I.9).

14.4.7 Texture

Object texture not only provides additional depth cues, as discussed in Section 14.3.6, but also can mimic the surface detail of real objects. Color Plates II.35 and II.36 show a variety of ways in which texture may be simulated, ranging from varying the surface's color (as was done with the patterned ball), to actually deforming the surface geometry (as was done with the striated torus and crumpled cone in Color Plate II.36).

14.4.8 Shadows

We can introduce further realism by reproducing shadows cast by objects on one another. Note that this technique is the first we have met in which the appearance of an object's visible surfaces is affected by other objects. Color Plate II.36 shows the shadows cast by the lamp at the rear of the scene. Shadows enhance realism and provide additional depth cues: If object *A* casts a shadow on surface *B*, then we know that *A* is between *B* and a direct or reflected light source. A point light source casts sharp shadows, because from any point it is either totally visible or invisible. An extended light source casts "soft" shadows, since there is a smooth transition from those points that see all of the light source, through those that see only part of it, to those that see none of it.

14.4.9 Transparency and Reflection

Thus far, we have dealt with opaque surfaces only. Transparent surfaces can also be useful in picture making. Simple models of transparency do not include the refraction (bending) of light through a transparent solid. Lack of refraction can be a decided advantage, however, if transparency is being used not so much to simulate reality as to reveal an object's inner geometry. More complex models include refraction, diffuse translucency, and the attenuation of light with distance. Similarly, a model of light reflection may simulate the sharp reflections of a perfect mirror reflecting another object or the diffuse reflections of a less highly polished surface. Color Plate II.37 shows the effect of reflection from the floor and teapot; Color Plates III.7 and III.10 show transparency.

Like modeling shadows, modeling transparency or reflection requires knowledge of other surfaces besides the surface being shaded. Furthermore, refractive transparency is the first effect we have mentioned that requires objects actually to be modeled as solids rather than just as surfaces! We must know something about the materials through which a light ray passes and the distance it travels to model its refraction properly.

14.4.10 Improved Camera Models

All the pictures shown so far are based on a camera model with a pinhole lens and an infinitely fast shutter: All objects are in sharp focus and represent the world at one instant in time. It is possible to model more accurately the way that we (and cameras) see the world.

For example, by modeling the focal properties of lenses, we can produce pictures, such as Color Plates II.38 and II.39, that show *depth of field*: Some parts of objects are in focus, whereas closer and farther parts are out of focus. Other techniques allow the use of special effects, such as fish-eye lenses. The lack of depth-of-field effects is responsible in part for the surreal appearance of many early computer-generated pictures.

Moving objects look different from stationary objects in a picture taken with a regular still or movie camera. Because the shutter is open for a finite period of time, visible parts of moving objects are blurred across the film plane. This effect, called *motion blur*, is simulated in Color Plates III.16 and IV.14. Motion blur not only captures the effects of motion in stills, but is of crucial importance in producing high-quality animation, as described in Chapter 21.

14.5 IMPROVED OBJECT MODELS

Independent of the rendering technology used, the search for realism has concentrated in part on ways of building more convincing models, both static and dynamic. Some researchers have developed models of special kinds of objects such as gases, waves, mountains, and trees; see, for example, Color Plates IV.11–IV.21. Other investigators have concentrated on automating the positioning of large numbers of objects, such as trees in a forest, which would be too tedious to do by hand (Color Plate IV.25). These techniques are covered in Chapter 20.

14.6 DYNAMICS

By *dynamics*, we mean changes that spread across a sequence of pictures, including changes in position, size, material properties, lighting, and viewing specification—indeed, changes in any parts of the scene or the techniques applied to it. The benefits of dynamics can be examined independently of the progression toward more realistic static images.

Perhaps the most popular kind of dynamics is motion dynamics, ranging from simple transformations performed under user control to complex animation, as described in Chapter 21. Motion has been an important part of computer graphics since the field's inception. In the early days of slow raster graphics hardware, motion capability was one of the strong competitive selling points of vector graphics systems. If a series of projections of the same object, each from a slightly different viewpoint around the object, is displayed in rapid succession, then the object appears to rotate. By integrating the information across the views, the viewer creates an object hypothesis.

A perspective projection of a rotating cube, for instance, provides several types of information. There is the series of different projections, which are themselves useful. This is supplemented by the motion effect, in which the maximum linear velocity of points near the center of rotation is lower than that of points distant from the center of rotation. This difference can help to clarify the relative distance of a point from the center of rotation. Also, the changing sizes of different parts of the cube as they change distance under perspective projection provide additional cues about the depth relationships. Motion becomes even more powerful when it is under the interactive control of the viewer. By selectively transforming an object, viewers may be able to form an object hypothesis more quickly.

In contrast to the use of simple transformations to clarify complex models, surprisingly simple models look extremely convincing if they move in a realistic fashion. For example, just a few points positioned at key parts of a human model, when moved naturally, can provide a convincing illusion of a person in motion. The points themselves do not "look like" a person, but they do inform the viewer that a person is present. It is also well known that objects in motion can be rendered with less detail than is needed to represent static objects, because the viewer has more difficulty picking out details when an object is moving. Television viewers, for example, are often surprised to discover how poor and grainy an individual television frame appears.

14.7 STEREOPSIS

All the techniques we have discussed thus far present the same image to both eyes of the viewer. Now conduct an experiment: Look at your desk or table top first with one eye, then with the other. The two views differ slightly because our eyes are separated from each other by a few inches, as shown in Fig. 14.7. The *binocular disparity* caused by this separation provides a powerful depth cue called *stereopsis* or *stereo vision*. Our brain fuses the two separate images into one that is interpreted as being in 3D. The two images are called a *stereo pair*; stereo pairs were used in the stereo viewers popular around the turn of the century, and are used today in the common toy, the View-Master. Color Plate II.19 shows a stereo pair of a molecule. You can fuse the two images into one 3D image by viewing them such that each eye sees only one image; you can do this, for example, by placing a stiff piece of paper between the two images perpendicular to the page. Some people can see the effect without any need for the piece of paper, and a small number of people cannot see it at all.

A variety of other techniques exists for providing different images to each eye, including glasses with polarizing filters and holography. Some of these techniques make possible true 3D images that occupy space, rather than being projected on a single plane. These displays can provide an additional 3D depth cue: Closer objects actually are closer,

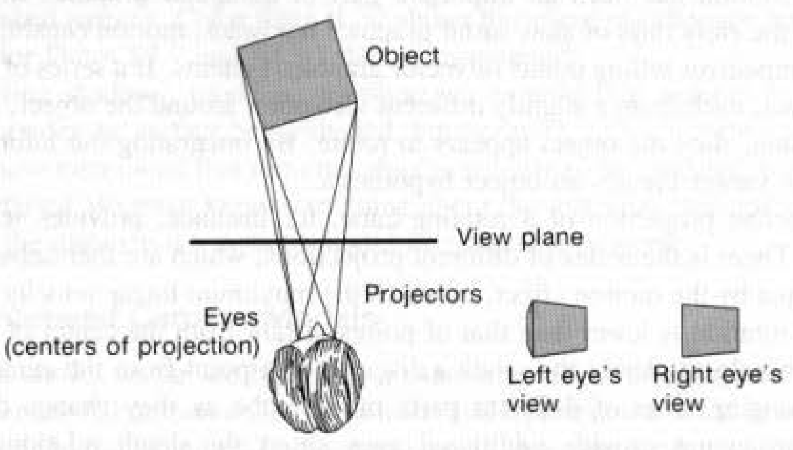


Fig. 14.7 Binocular disparity.

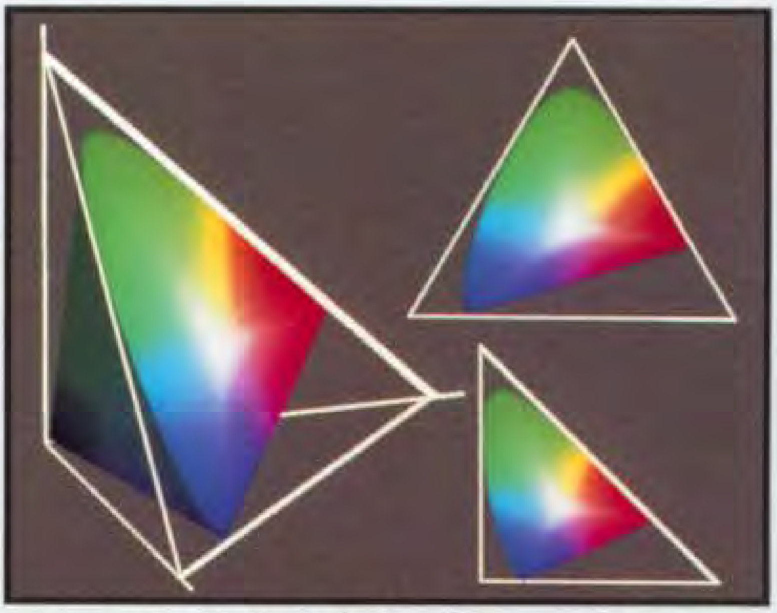
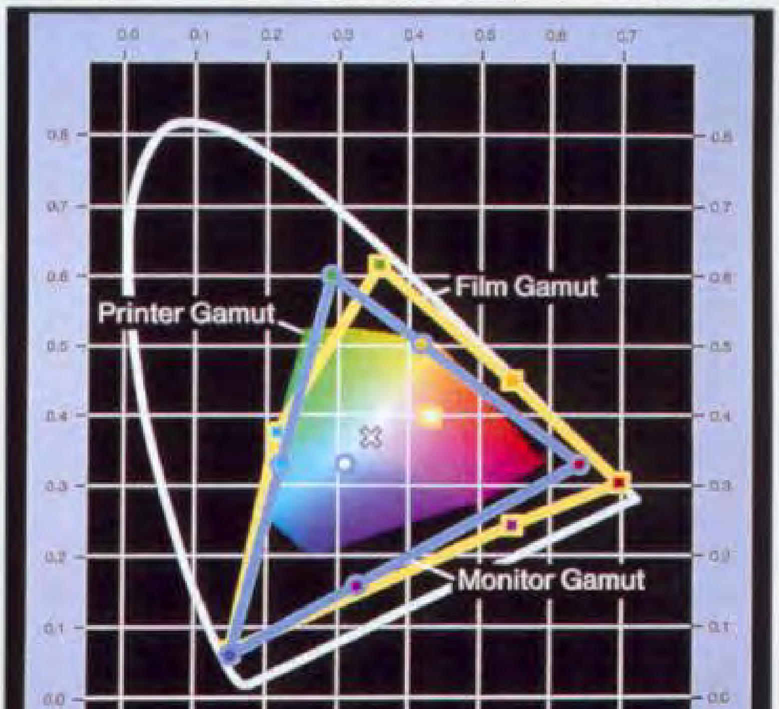


Plate II.1 Several views of the $X + Y + Z = 1$ plane of CIE space. Left: the plane embedded in CIE space. Top right: a view perpendicular to the plane. Bottom right: the projection onto the (X, Y) plane (that is, the $Z = 0$ plane), which is the chromaticity diagram. (Courtesy of Barbara Meier, Brown University.)

Plate II.2 The CIE chromaticity diagram, showing typical color gamuts for an offset printing press, a color monitor, and for slide film. The print colors represent the Graphics Arts Technical Foundation S.W.O.P. standard colors measured under a graphic arts light with a color temperature of 5000° K. The color monitor is a Barco CTVM 3/51 with a white point set to 6500° K and the slide film is Kodak Ektachrome 5017 ISO 64 as characterized under CIE source A: a 2653° K black body that closely approximates a Tungsten lamp. The ×, circle, and square indicate the white points for the print, color monitor, and film gamuts, respectively. (Courtesy of M. Stone, Xerox Palo Alto Research Center. Film gamut measured by A. Paeth, Computer Graphics Lab, University of Waterloo: see also the first appendix of [PAET89].)



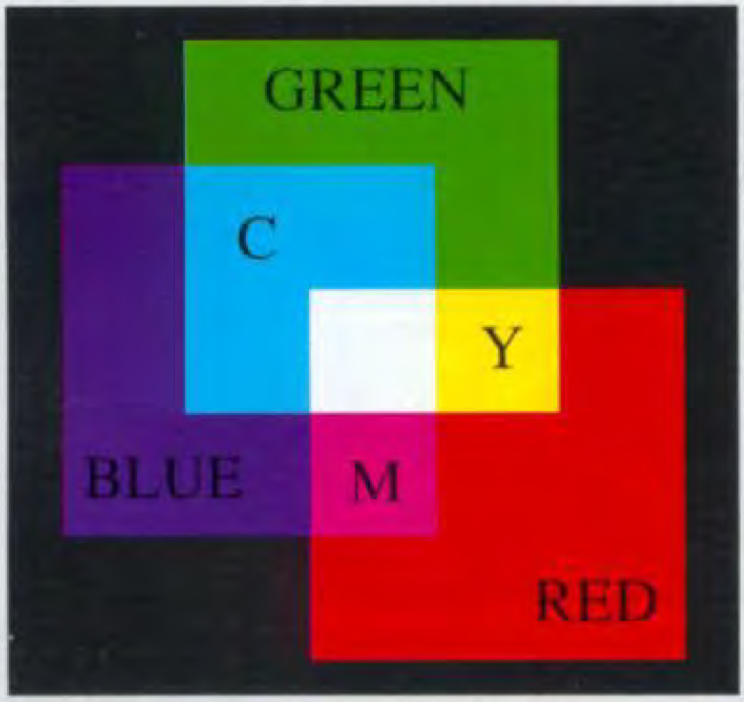
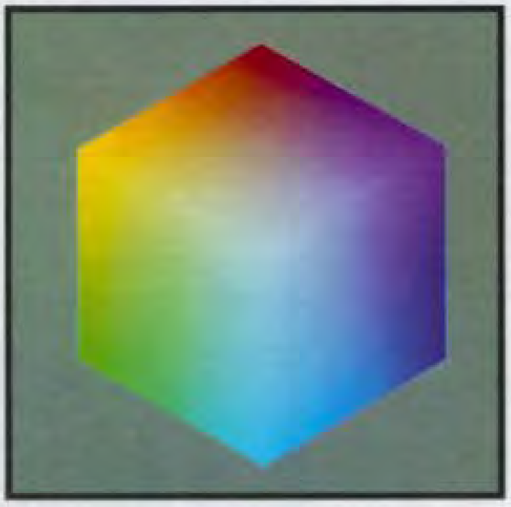


Plate II.3 Additive colors. Red plus green form yellow, red plus blue form magenta, green plus blue form cyan, red plus green plus blue form white.

Plate II.4 The RGB color space, viewed looking along the main diagonal from white to black. Only the black vertex is invisible. (Courtesy of David Small, Visible Language Workshop, MIT Media Lab, Cambridge, MA 02139. © MIT, 1989.)



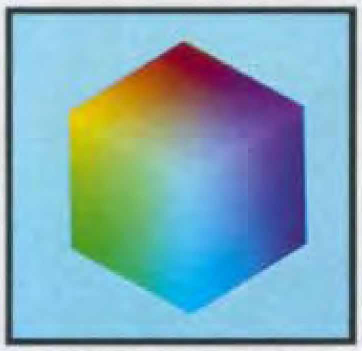


Plate II.5 An interior subcube of the RGB color space. The gray vertex is at (0.5, 0.5, 0.5) Hence the subcube is half the height, width, and depth of the entire space shown in Color Plate II.4. (Courtesy of David Small, Visible Language Workshop, MIT Media Lab, Cambridge, MA 02139. © MIT, 1989.)

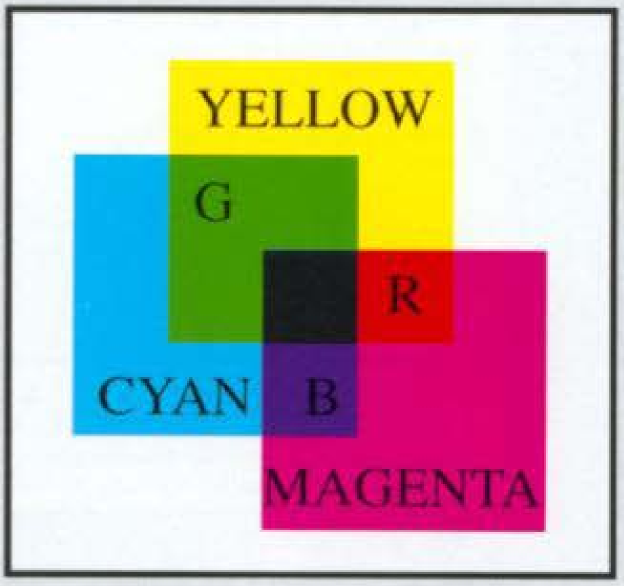
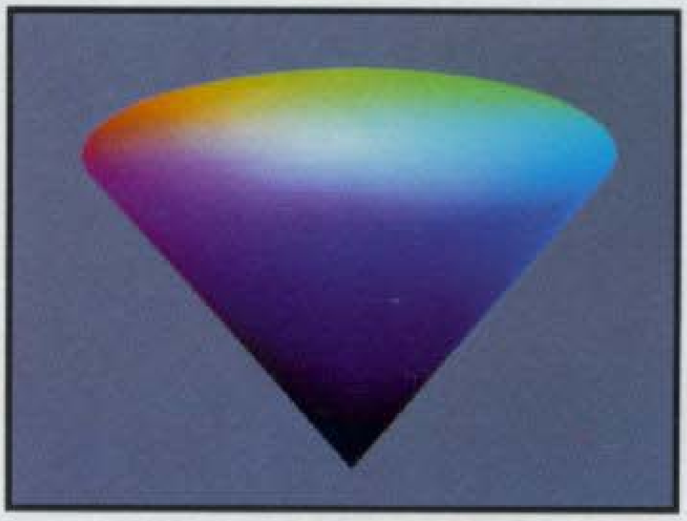


Plate II.6 Subtractive colors. Yellow and magenta subtracted from white form red, yellow and cyan subtracted from white form green, cyan and magenta subtracted from white form blue.

Plate II.7 The HSV color space. (Courtesy of David Small, Visible Language Workshop, MIT Media Lab, Cambridge, MA 02139. © MIT, 1989.)



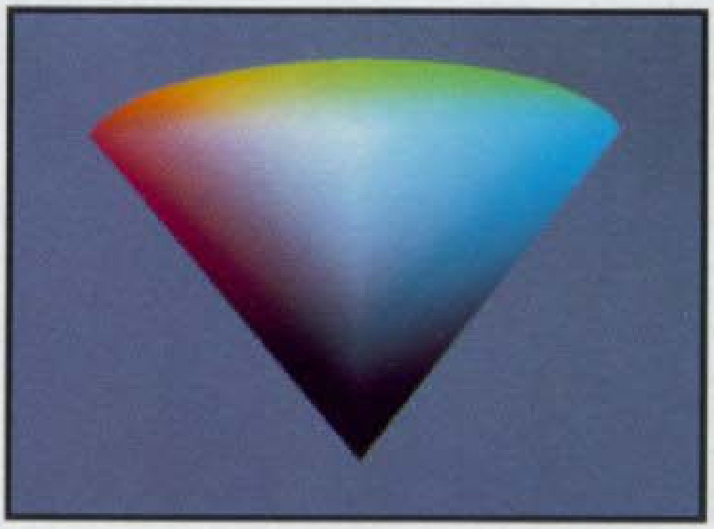


Plate II.8 A vertical cross-section slice of HSV space along the V axis. (Courtesy of David Small, Visible Language Workshop, MIT Media Lab, Cambridge, MA 02139. © MIT, 1989.)

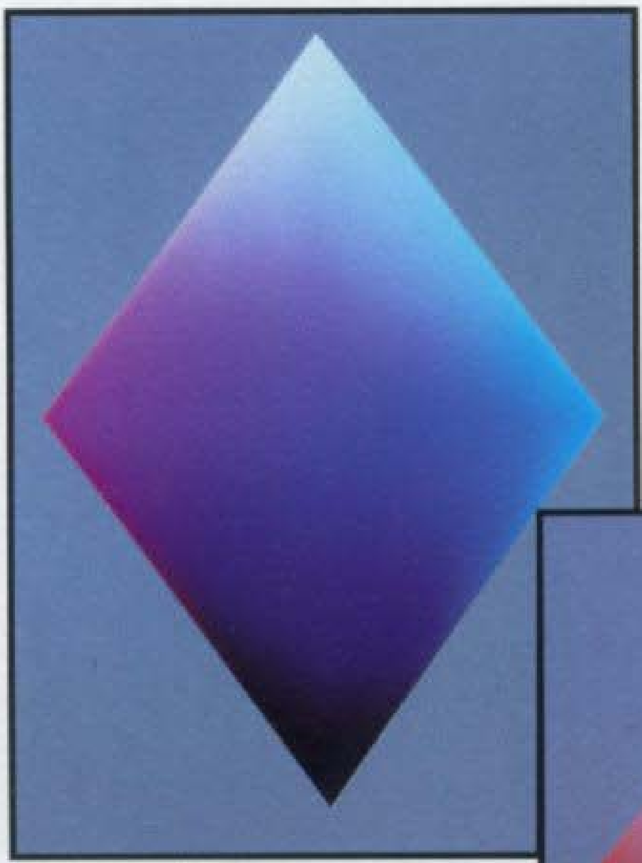


Plate II.9 The HLS color space. (Courtesy of David Small, Visible Language Workshop, MIT Media Lab, Cambridge, MA 02139. © MIT, 1989.)

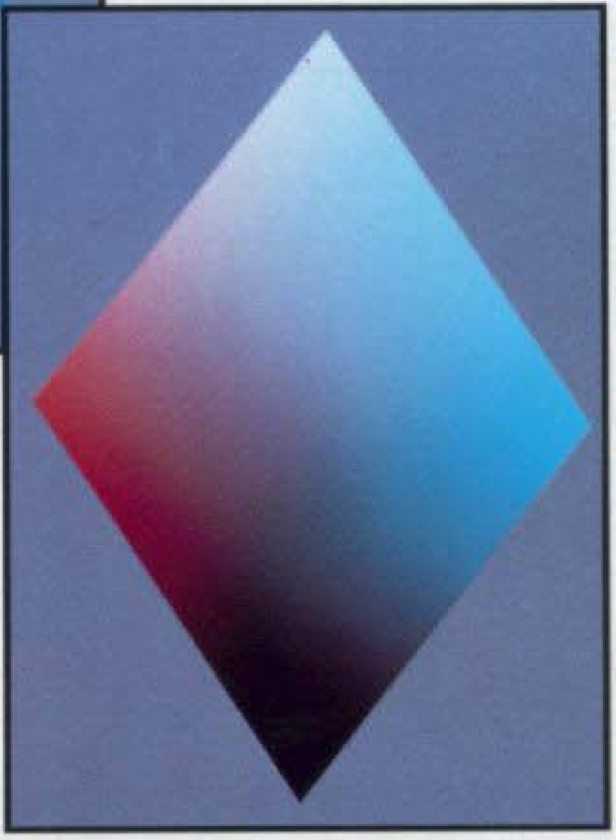


Plate II.10 A vertical cross-section slice of HLS space along the L axis. (Courtesy of David Small, Visible Language Workshop, MIT Media Lab, Cambridge, MA 02139. © MIT, 1989.)

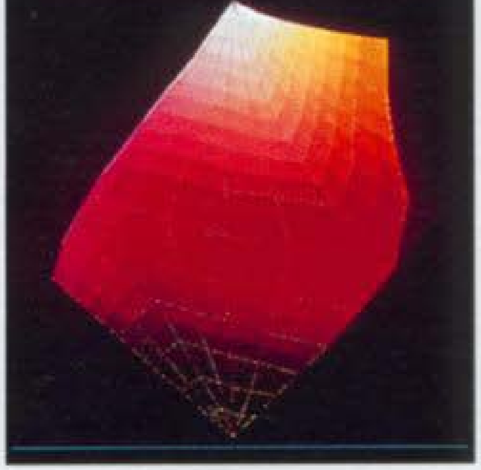
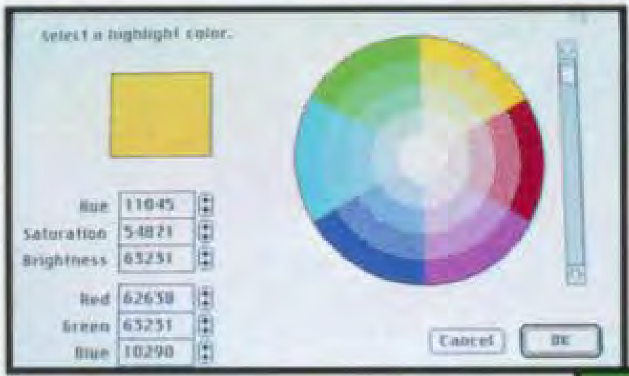


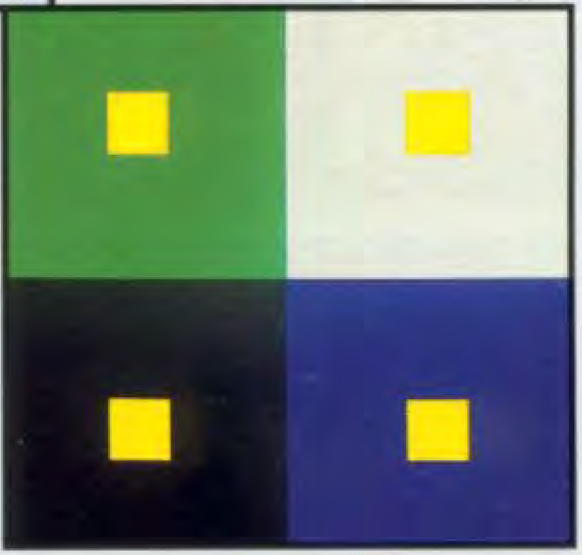
Plate II.11 The HVC color space. (Courtesy of Tektronix, Inc.)

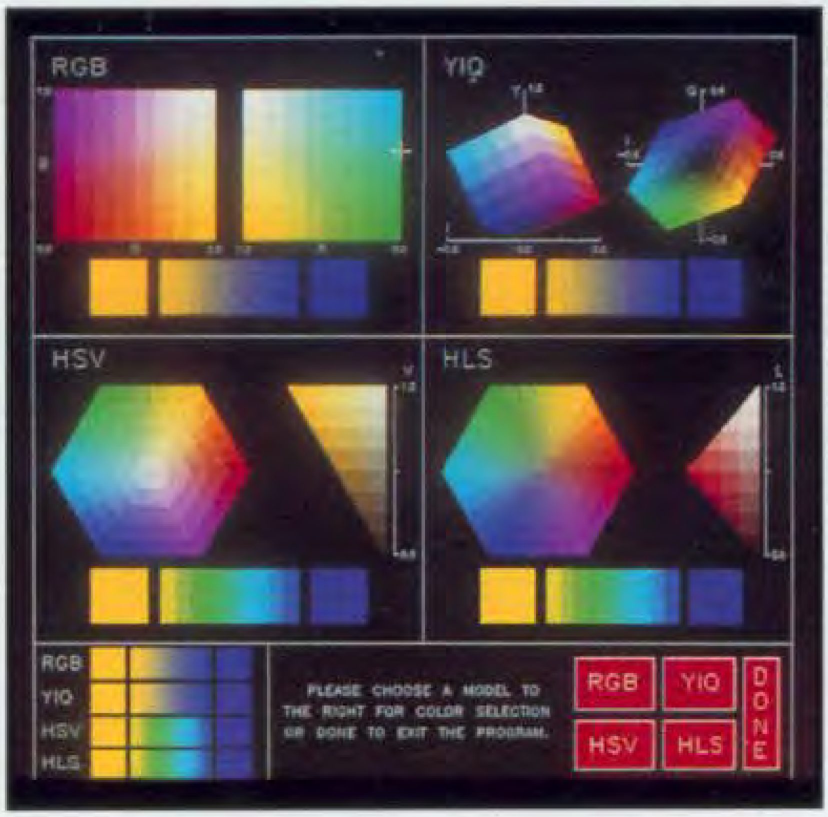

Select a highlight color.

Hue 11045
Saturation 54821
Brightness 65251
Red 62658
Green 65251
Blue 10290

Plate II.12 An interaction technique used on the Macintosh to specify colors in HSV space. Hue and saturation are shown in the circular area, and value by the slider dial. The user can move the mark in the circular area and change the slider dial, or can type in new HSV or RGB values. The square color area (upper left) shows the current color and the new color. In this picture, taken from a color monitor driven by a 4-bit-per-pixel bit map, many of the colors are created by dithering (see Section 13.4). (Courtesy of Apple Computer, Inc., © 1984.)

Plate II.13 The same yellow surrounded by different background colors appears to be different shades of yellow.





PLEASE CHOOSE A MODEL TO THE RIGHT FOR COLOR SELECTION OR DONE TO EXIT THE PROGRAM.

Plate II.14 An interactive program that allows the user to specify and interpolate colors in four different color spaces: RGB, YIQ, HSV, and HLS. The starting and ending colors for a linear interpolation are specified by pointing at the various projections of the color spaces. The interpolation is shown below each color space, and together for comparison in the lower left. (Courtesy of Paul Charlton, The George Washington University.)

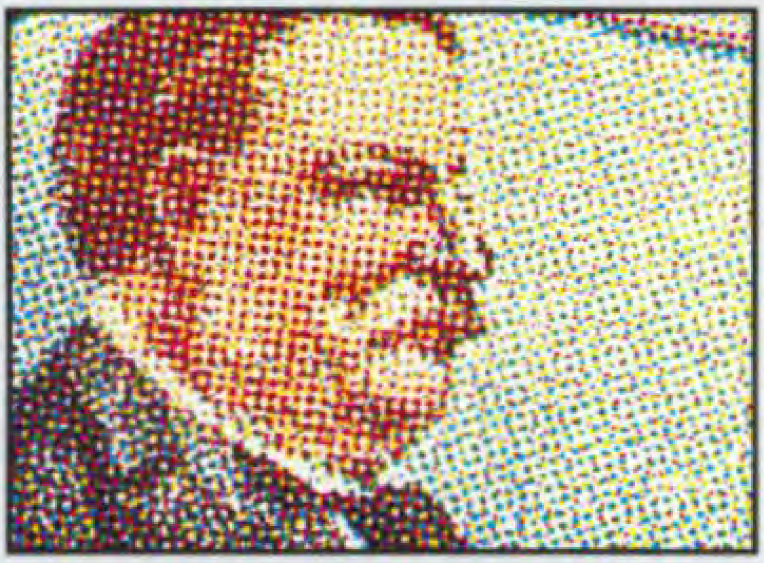
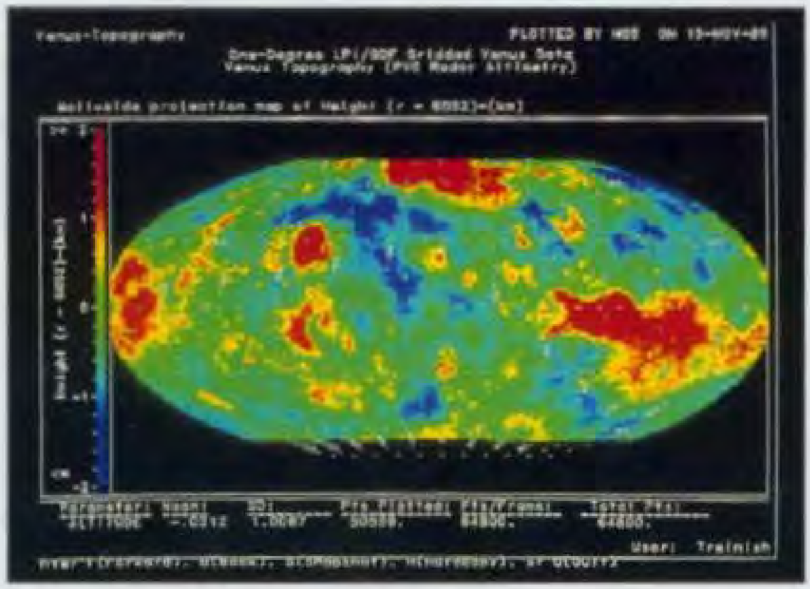


Plate II.15 An enlarged halftone color picture. Individual dots of cyan, magenta, yellow, and black combine to create a broad gamut of colors.

Plate II.16 A pseudo-color image showing the topography of Venus. The color scale on the left indicates altitudes from –2 km to + 2 km above or below an average radius for Venus of 6052 km. Data were calculated by the Lunar and Planetary Institute from radar altimetry observations by NASA's Pioneer Venus Orbiter spacecraft. The image was created with the National Space Science Data Center Graphics System. (Courtesy of Lloyd Treinish, NASA Goddard Space Flight Center.)


Venus-Topography
One-Degree LPI/SDF Gridded Venus Data
Venus Topography (PVO Radar Altimetry)
User: Treinish

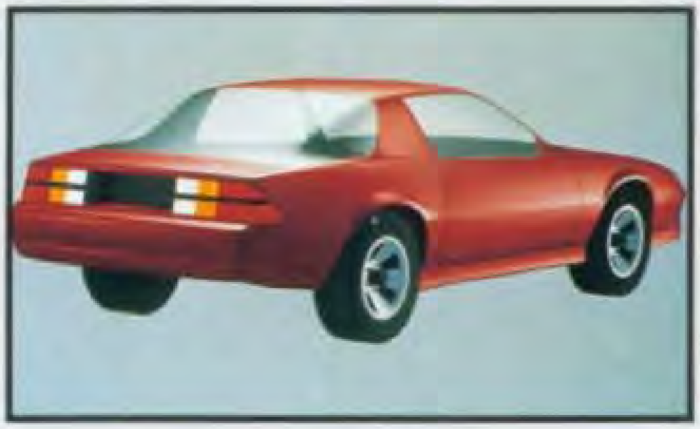


Plate II.17 Chevrolet Camaro lit by five lights with Warn's lighting controls. (Courtesy of David R. Warn, General Motors Research Laboratories.)

COME
HOME
to

Plate II.18 *'87–'88 NBC Network Package.* By James Dixon (animator) and Glenn Entis (producer), Pacific Data Images, Sunnyvale, CA, for Marks Communications.

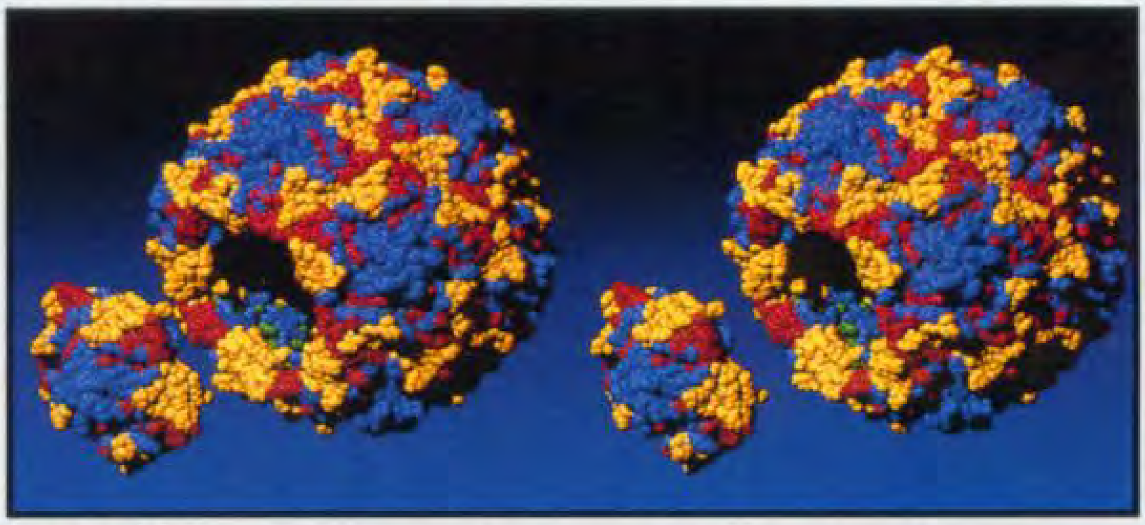


Plate II.19 Stereo pair of Polio virus capsid, imaged by placing a sphere of 0.5 nm radius at each alpha carbon position. One pentamer is removed to reveal the interior. Coordinates courtesy of J. Hogle. (Courtesy of David Goodsell and Arthur J. Olson. Copyright © 1989, Research Institute of Scripps Clinic.)

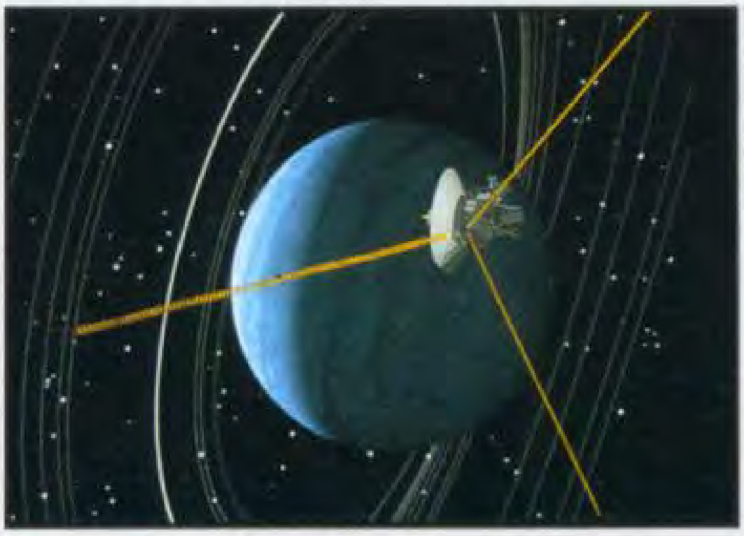


Plate II.20 Simulated flyby of Uranus with rings and orbit. (Courtesy of Jim Blinn, Computer Graphics Lab, Jet Propulsion Lab, California Institute of Technology.)

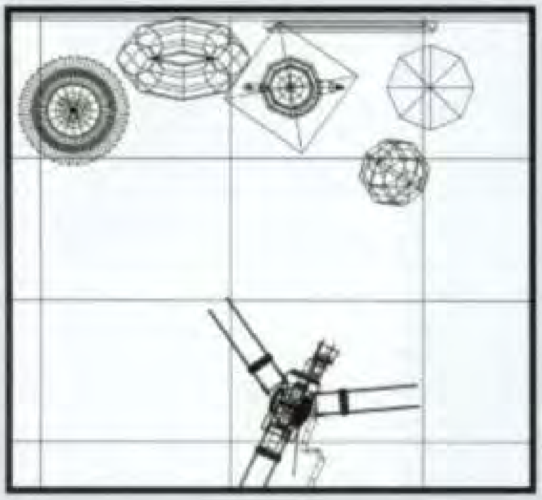
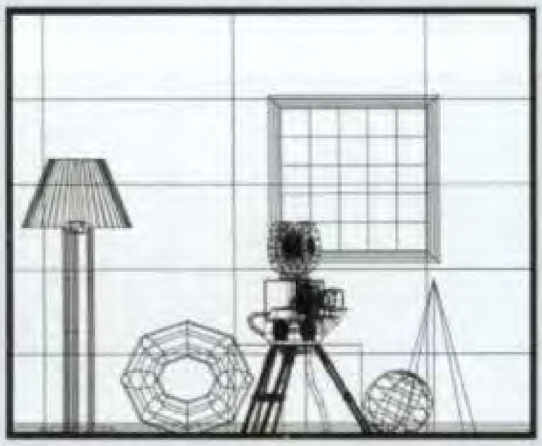

(a)

Plate II.21 *Shutterbug.* Living room scene with movie camera. Orthographic projections (Sections 6.1.2 and 14.3.1). (a) Plan view. (b) Front view. (c) Side view. Polygonal models generated from spline patches. Note the "patch cracks" (Section 11.3.5) visible along the entire right front side of the teapot, and how they cause shading discontinuities in the polygon-mesh interpolated-shading models used in Color Plates II.30–II.32. (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)


(b)

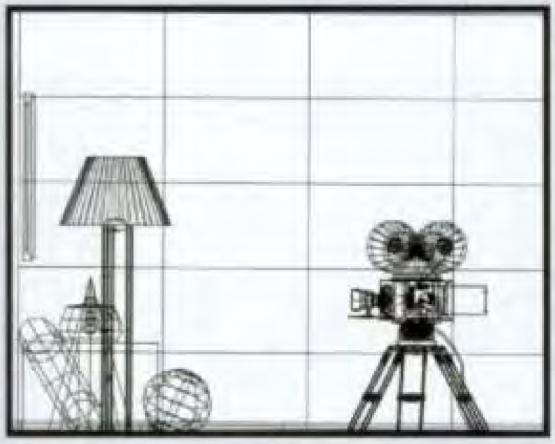
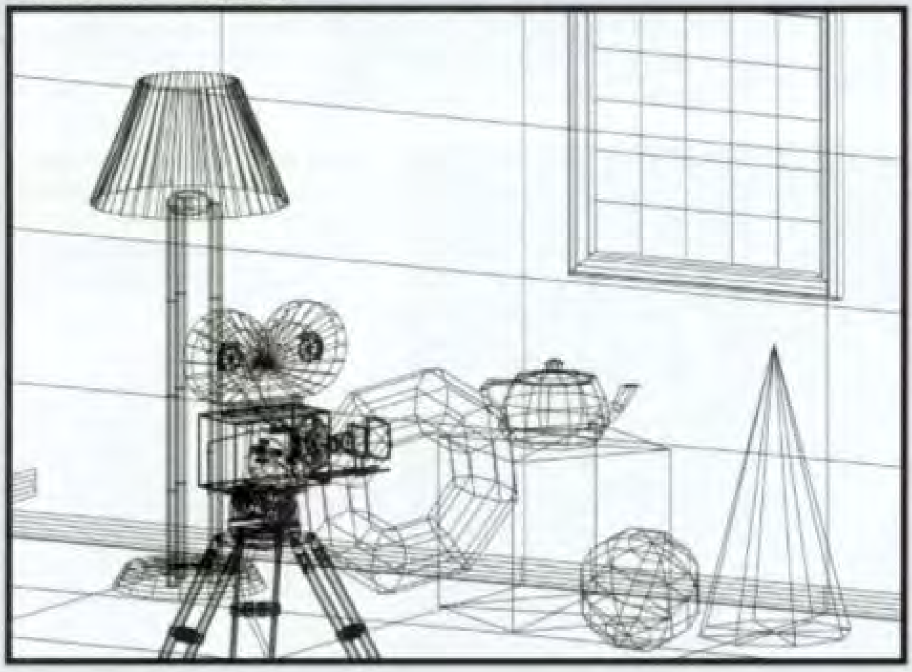

(c)

Plate II.22 *Shutterbug.* Axonometric projection (Sections 6.1.2 and 14.3.2). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)



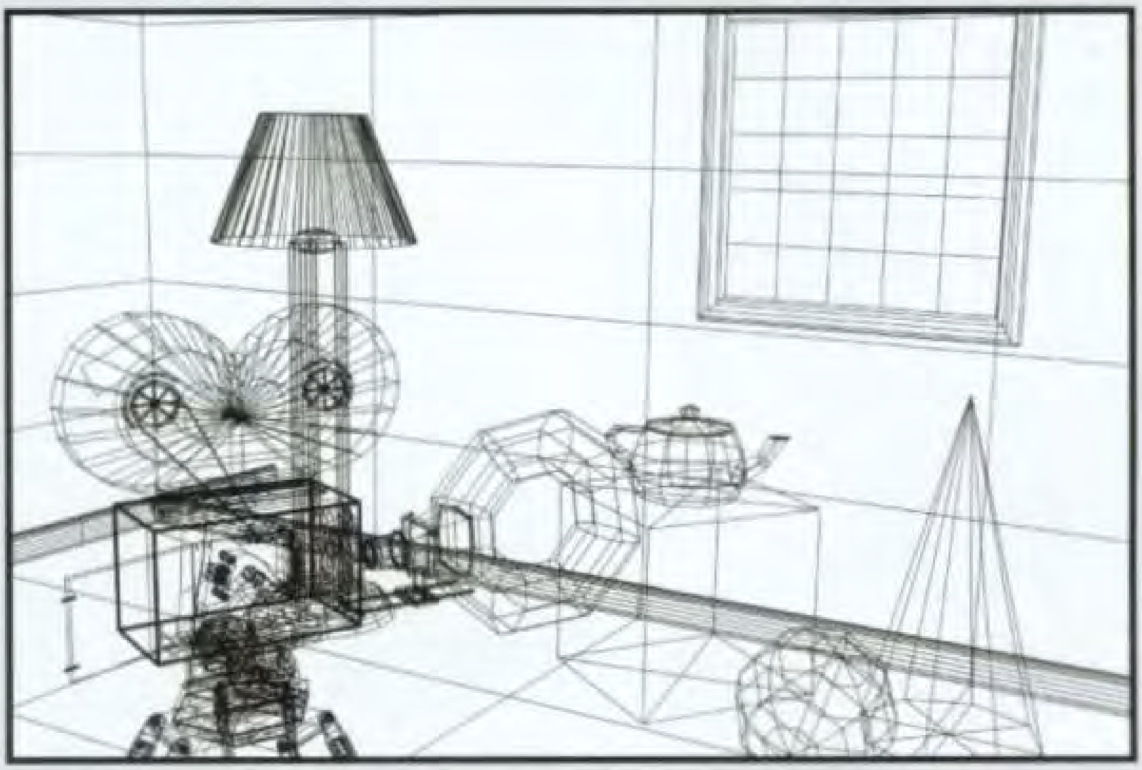
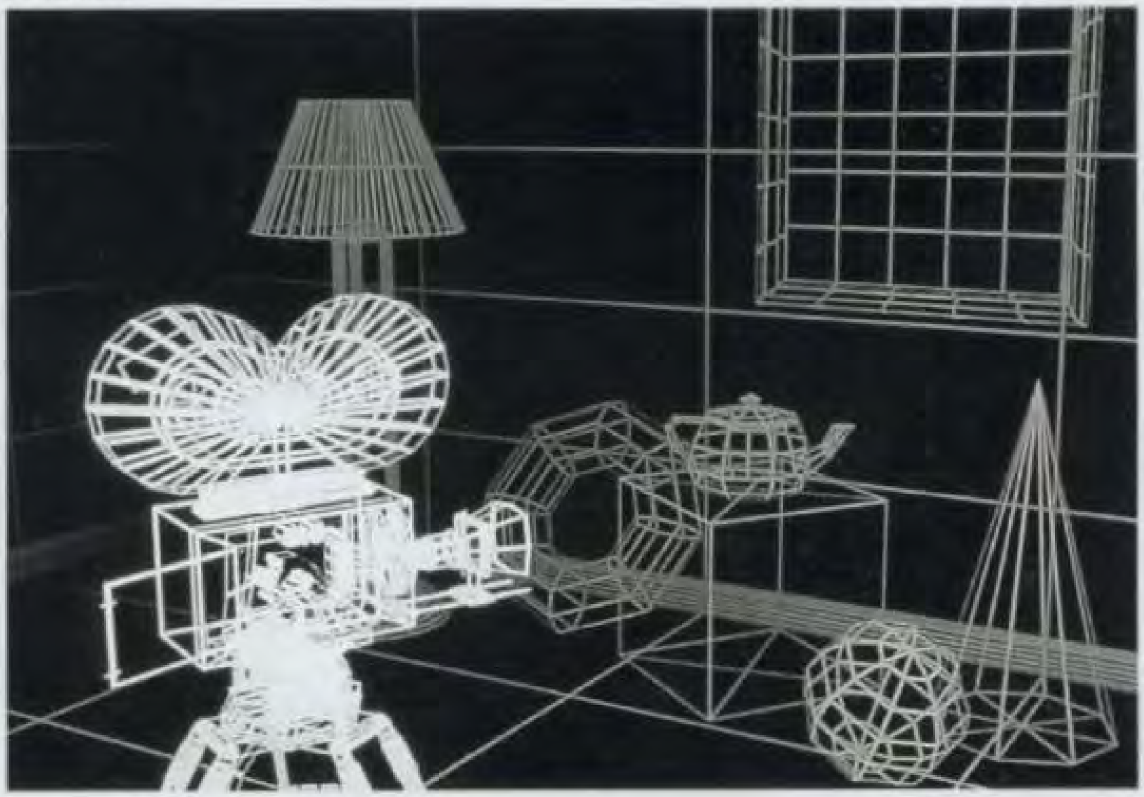


Plate II.23 *Shutterbug.* Perspective projection (Sections 6.1.1 and 14.3.3). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

Plate II.24 *Shutterbug.* Depth cueing (Sections 14.3.4 and 16.1.3). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)



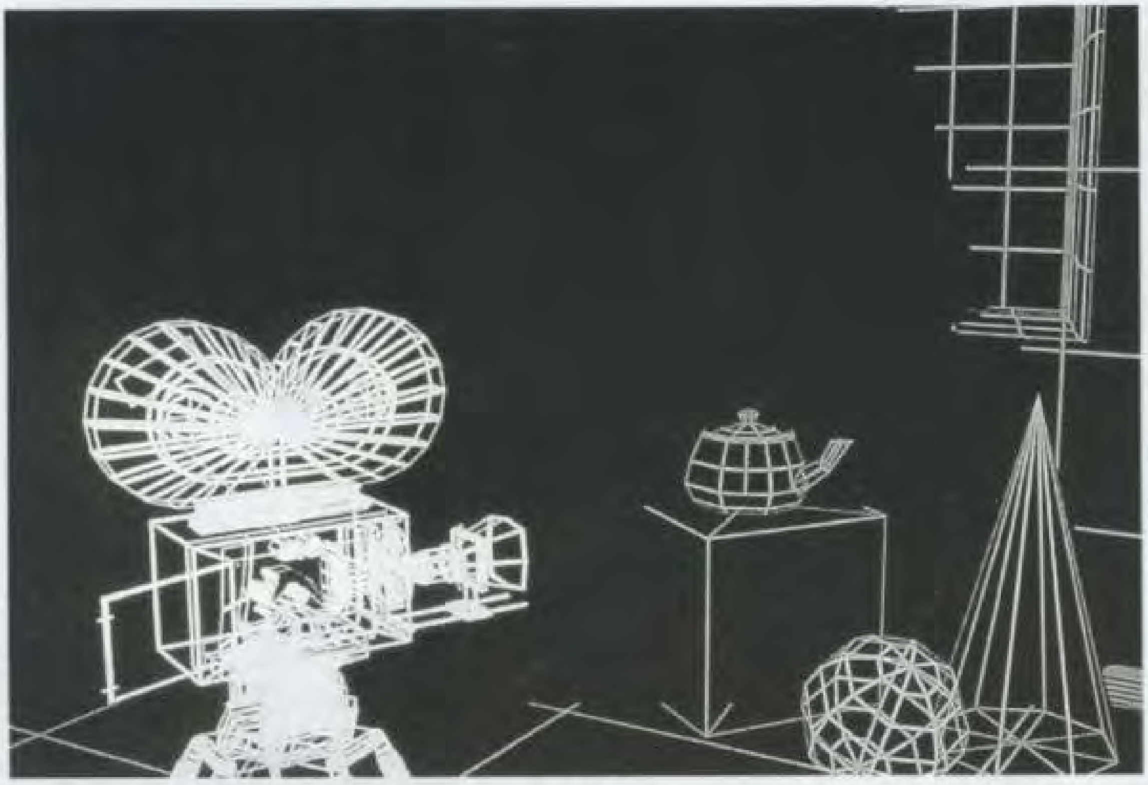
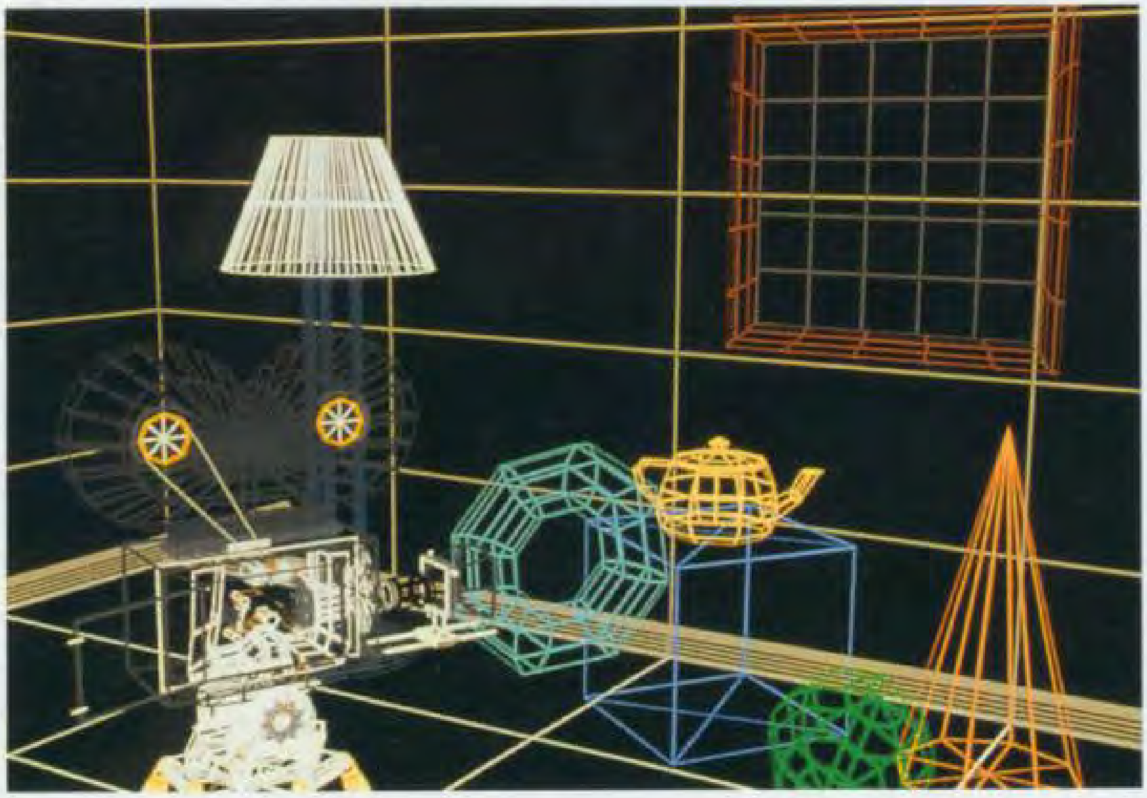


Plate II.25 *Shutterbug.* Depth clipping (Section 14.3.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

Plate II.26 *Shutterbug.* Colored vectors (Section 14.3.7). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)



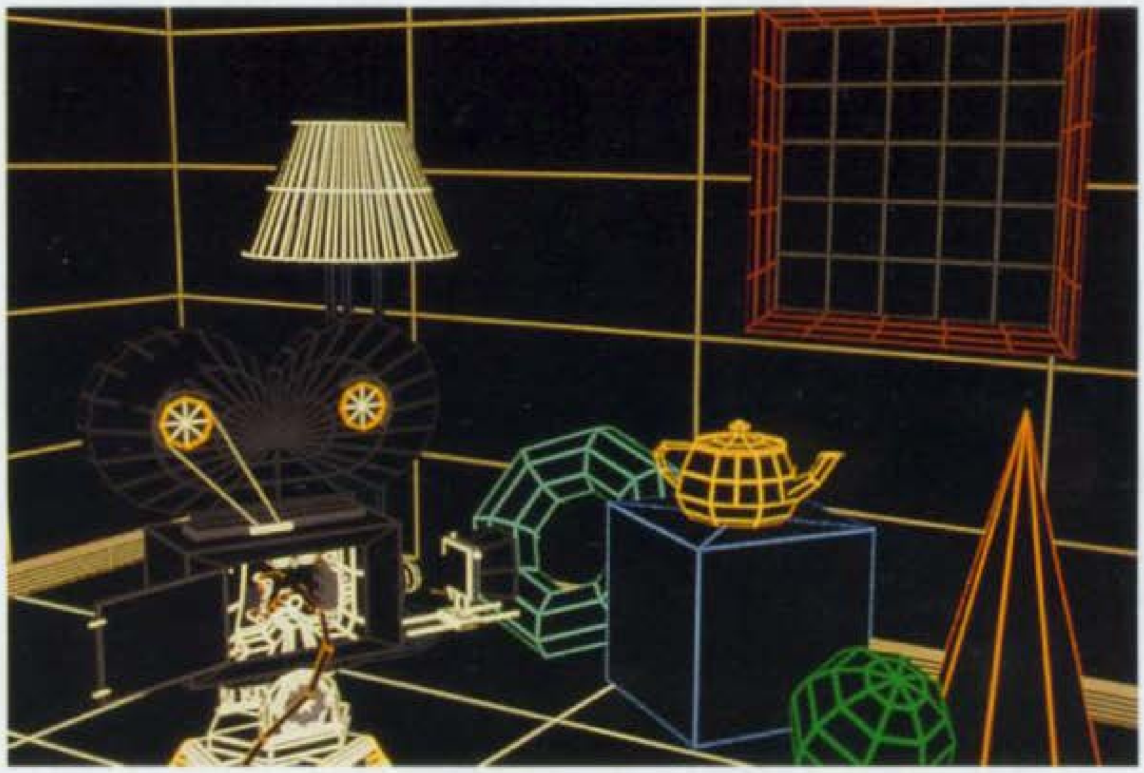
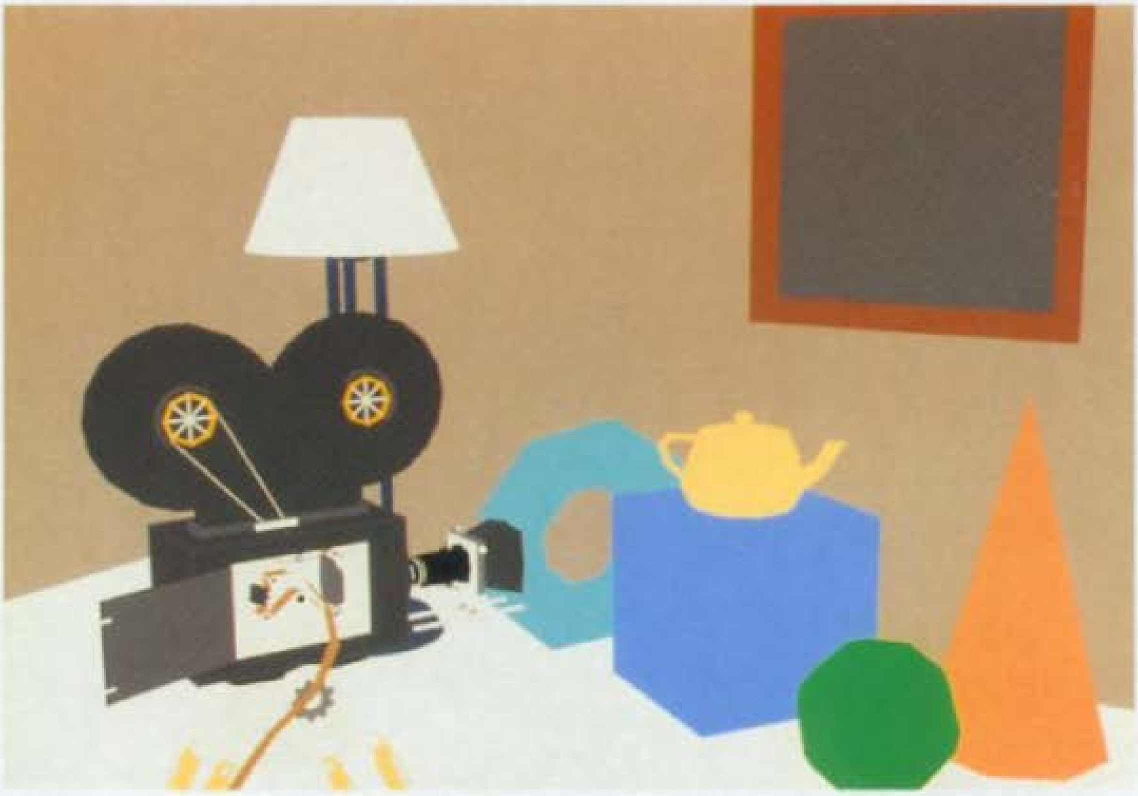


Plate II.27 *Shutterbug.* Visible-line determination (Section 14.3.8). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

Plate II.28 *Shutterbug.* Visible-surface determination with ambient illumination only (Sections 14.4.1 and 16.1.1). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)



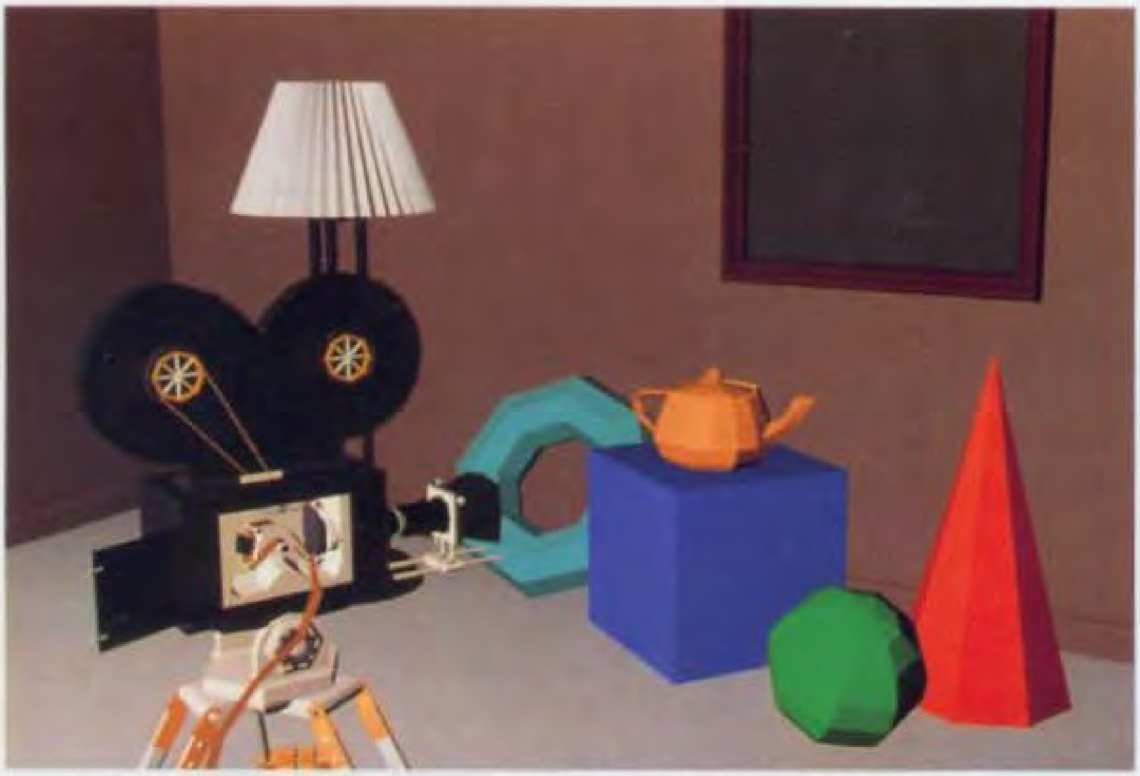
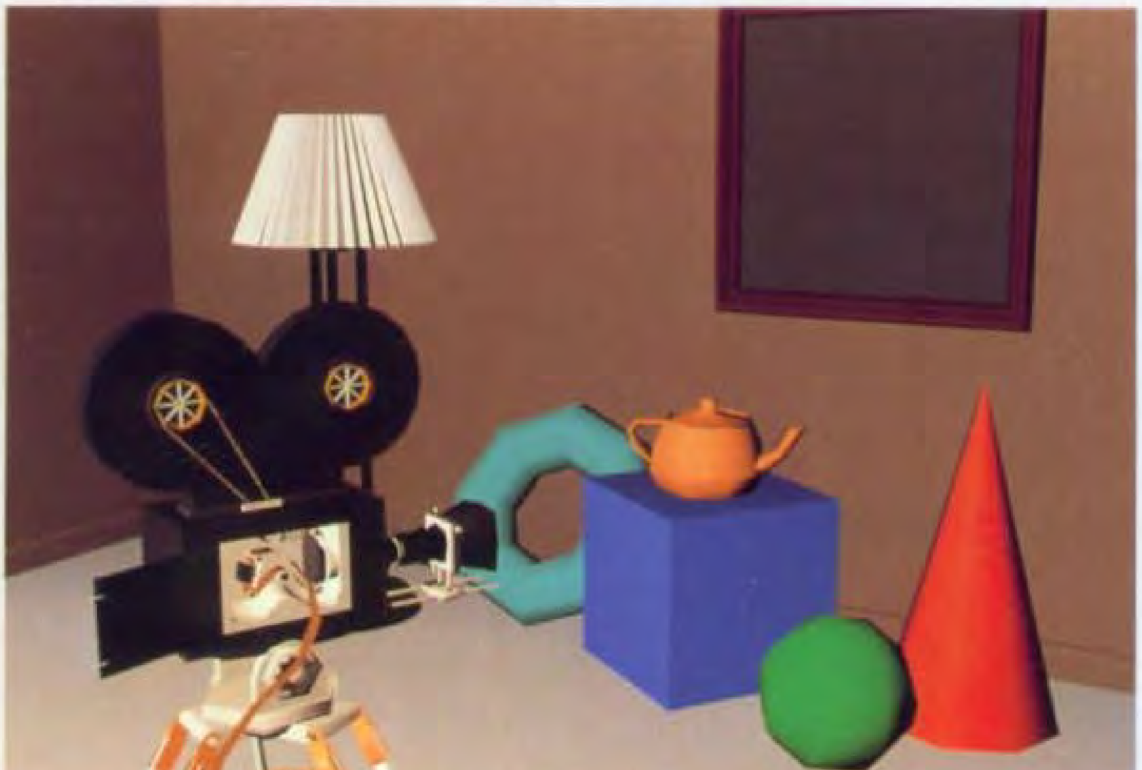


Plate II.29 *Shutterbug.* Individually shaded polygons with diffuse reflection (Sections 14.4.2 and 16.2.3). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

Plate II.30 *Shutterbug.* Gouraud shaded polygons with diffuse reflection (Sections 14.4.3 and 16.2.4). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)



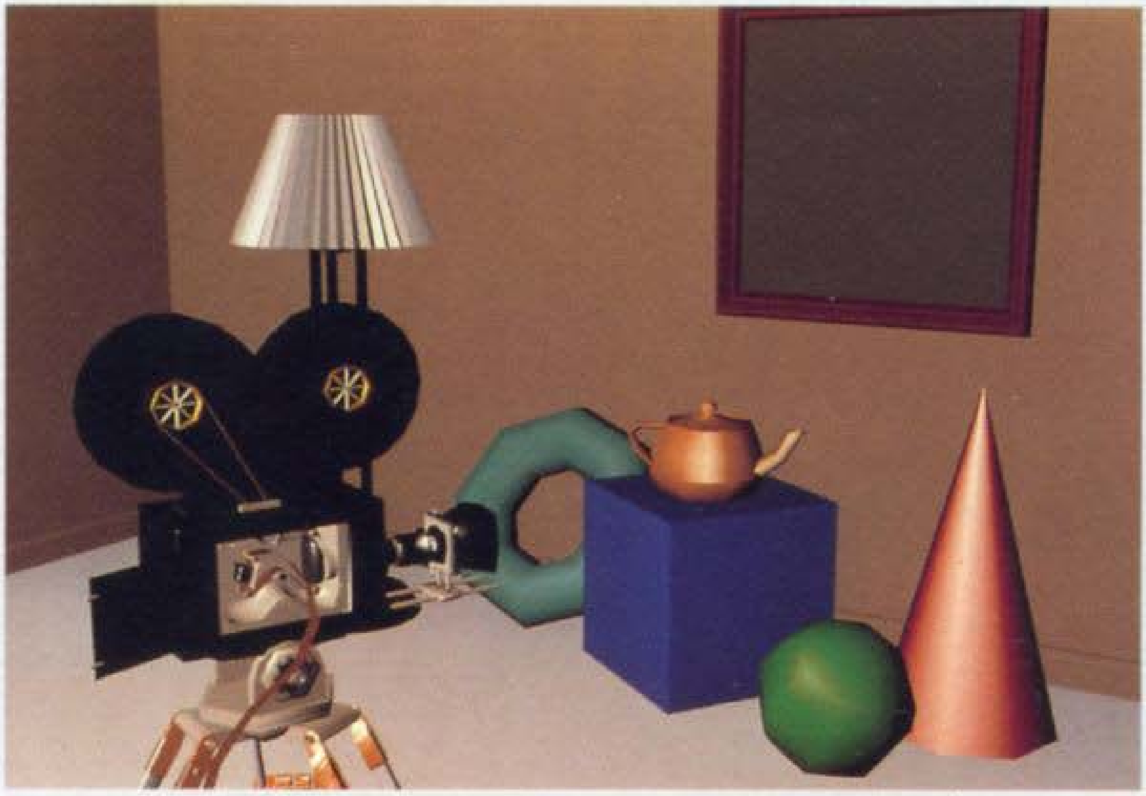
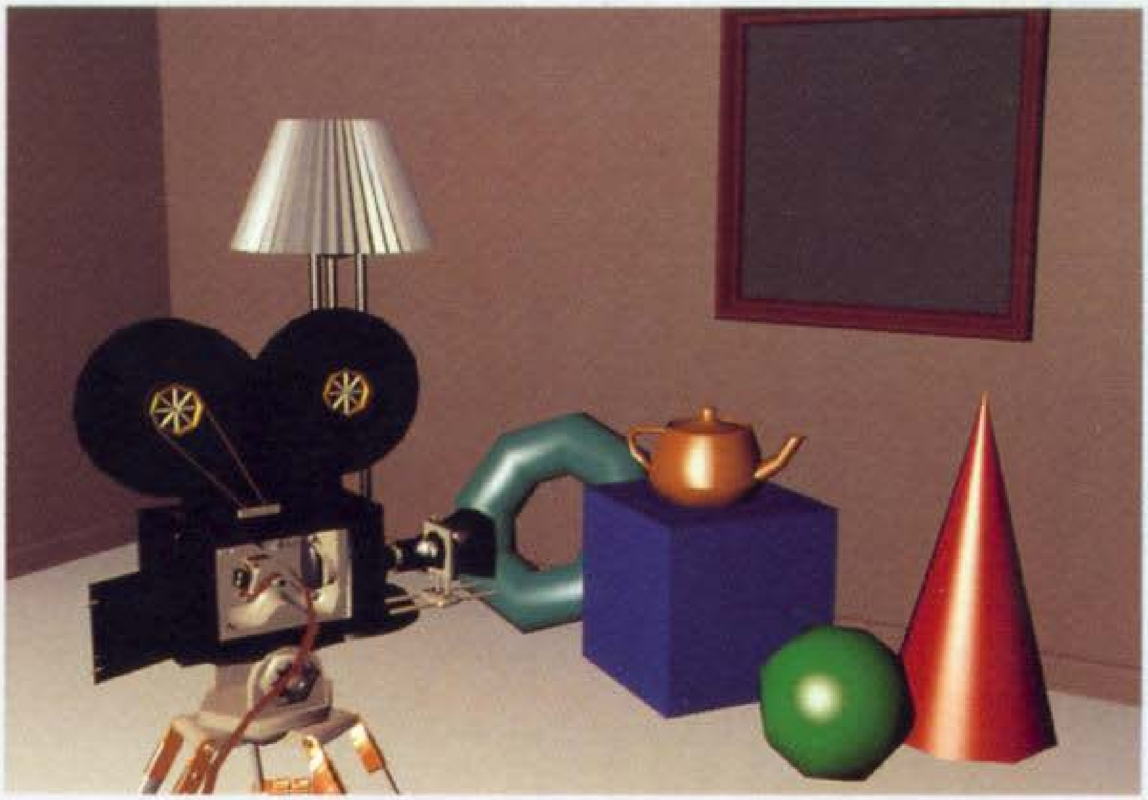


Plate II.31 *Shutterbug.* Gouraud shaded polygons with specular reflection (Sections 14.4.4 and 16.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

Plate II.32 *Shutterbug.* Phong shaded polygons with specular reflection (Sections 14.4.4 and 16.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)



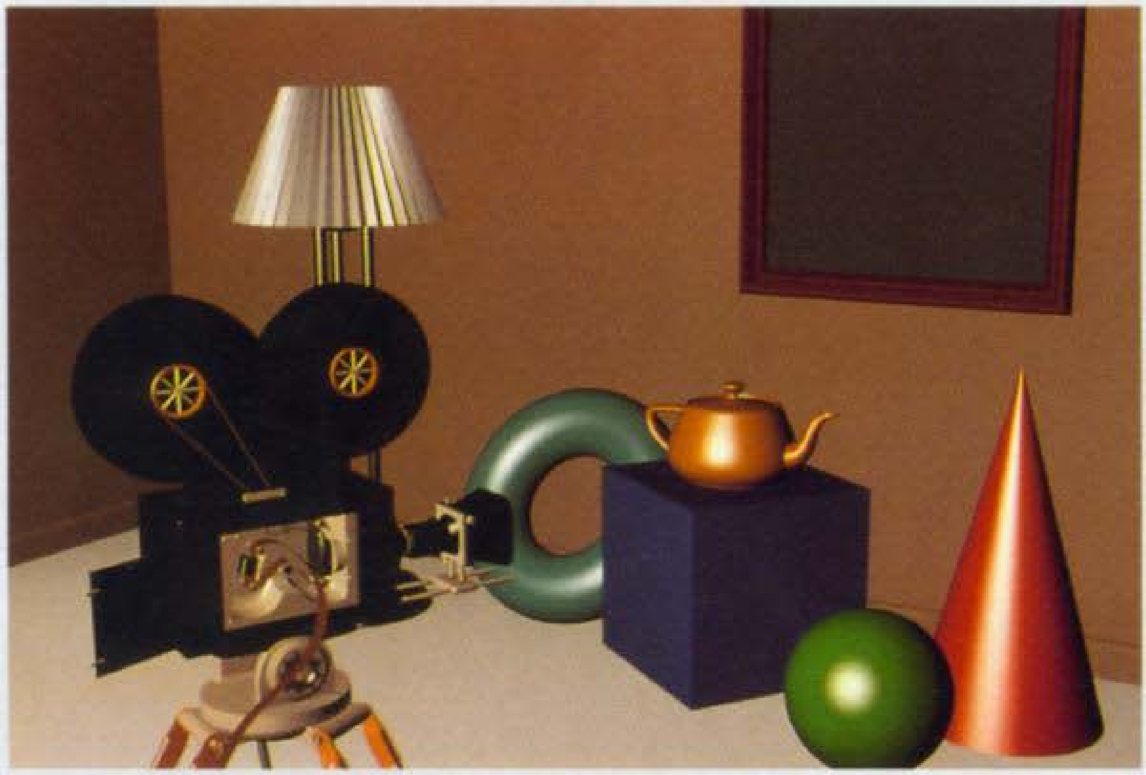
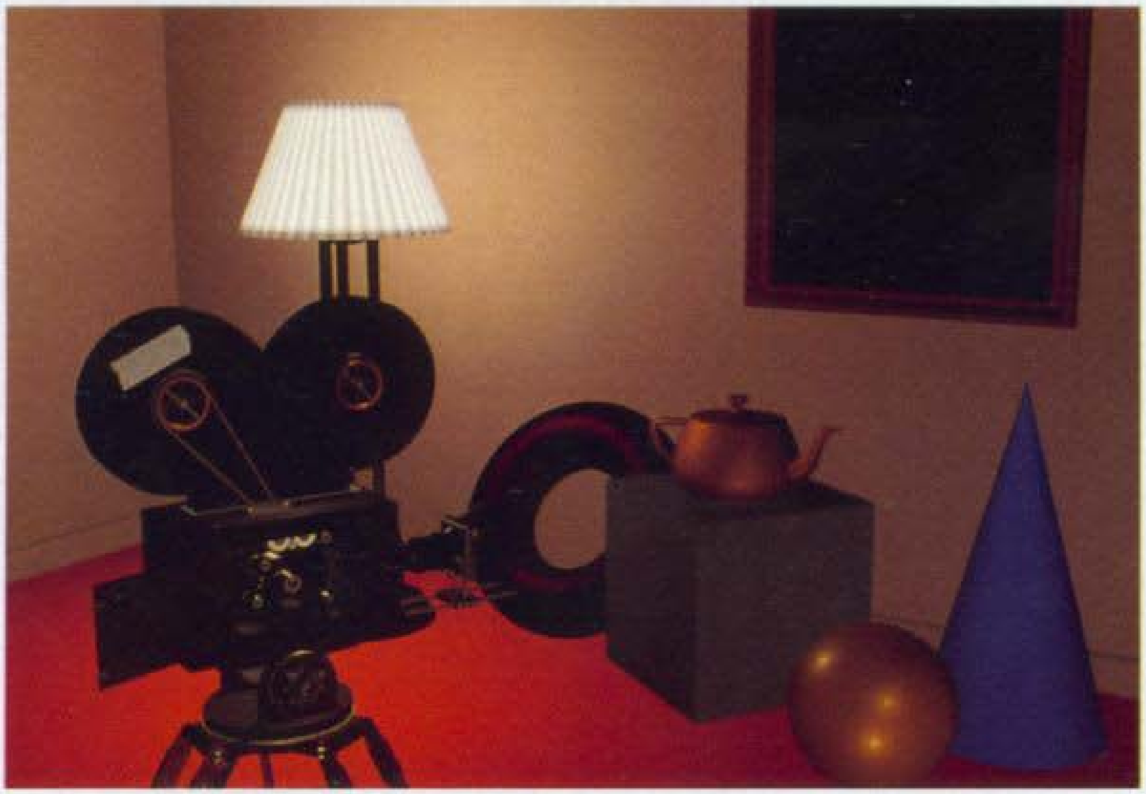


Plate II.33 *Shutterbug.* Curved surfaces with specular reflection (Section 14.4.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

Plate II.34 *Shutterbug.* Improved illumination model and multiple lights (Sections 14.4.6 and 16.1). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)



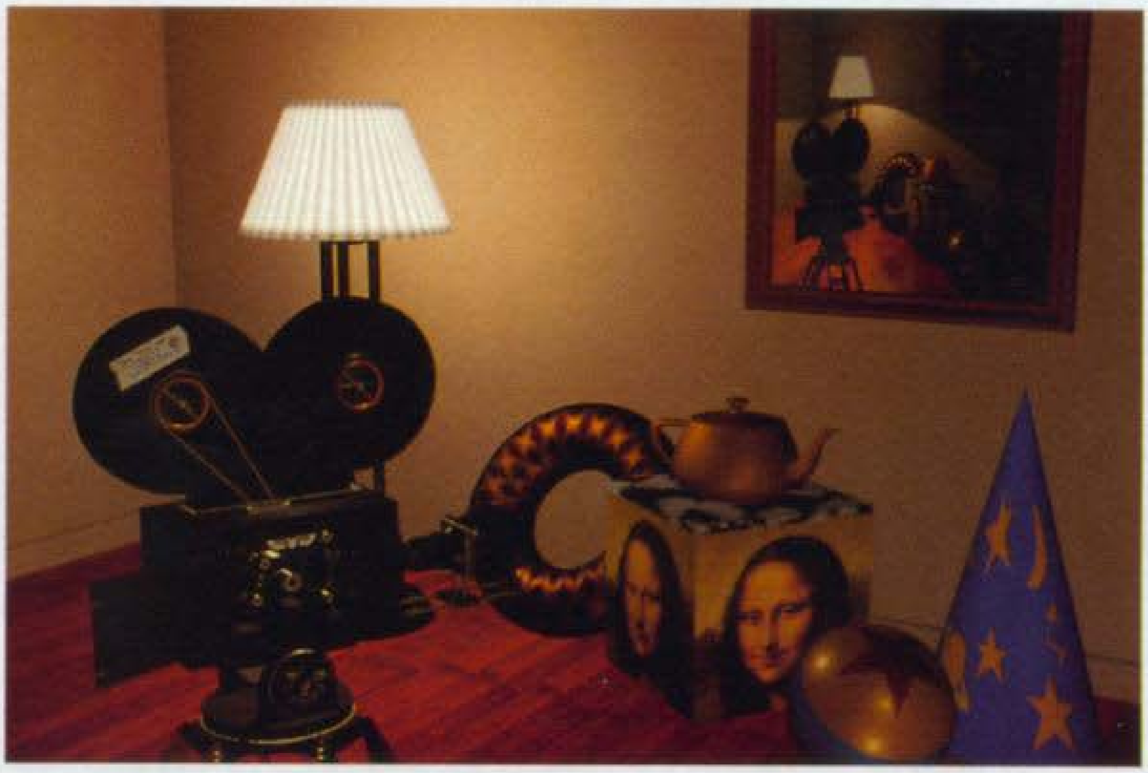
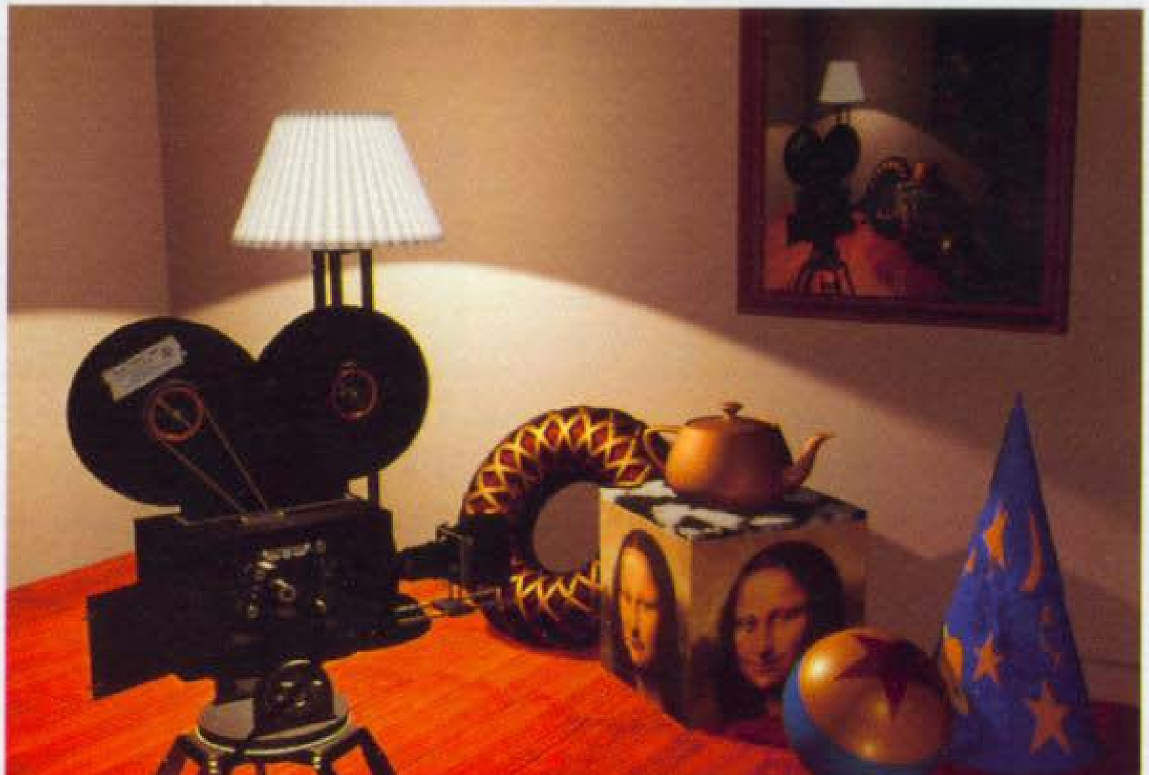


Plate II.35 *Shutterbug.* Texture mapping (Sections 14.4.7, 16.3.2, 17.4.2, and 17.4.3). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

Plate II.36 *Shutterbug.* Displacement mapping (Sections 14.4.7 and 16.3.4) and shadows (Sections 14.4.8 and 16.4). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)



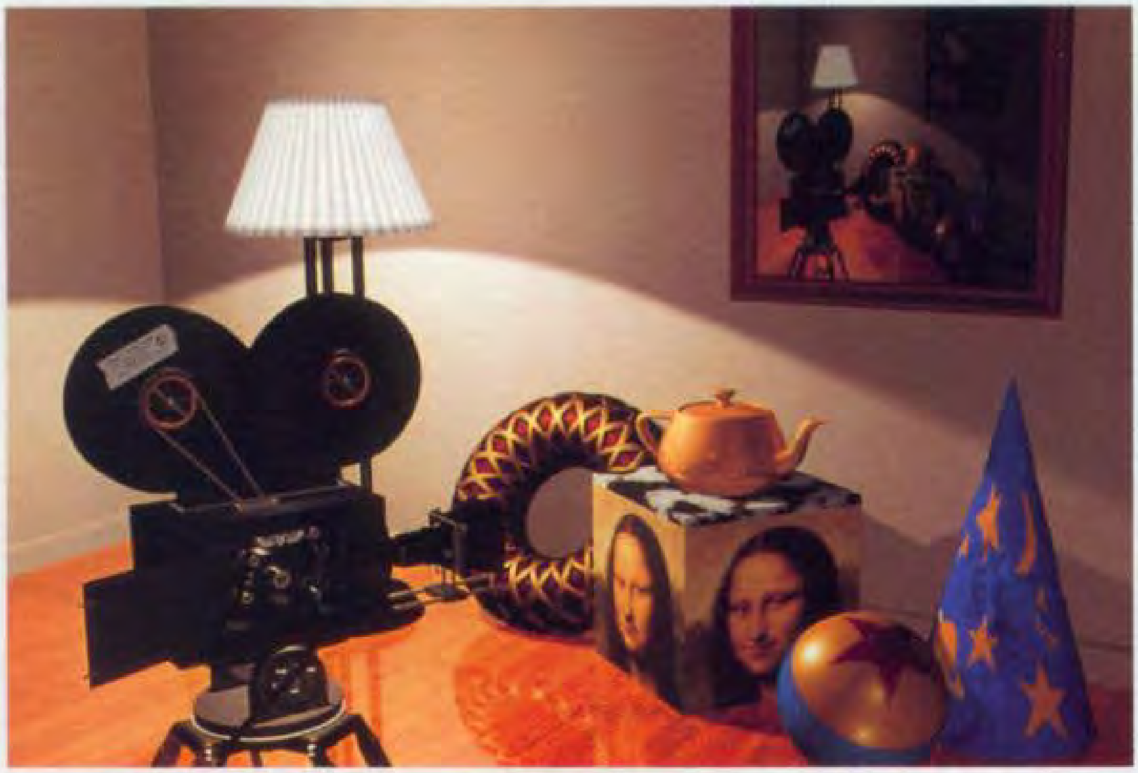


Plate II.37 Shutterbug. Reflection mapping (Sections 14.4.9 and 16.6). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

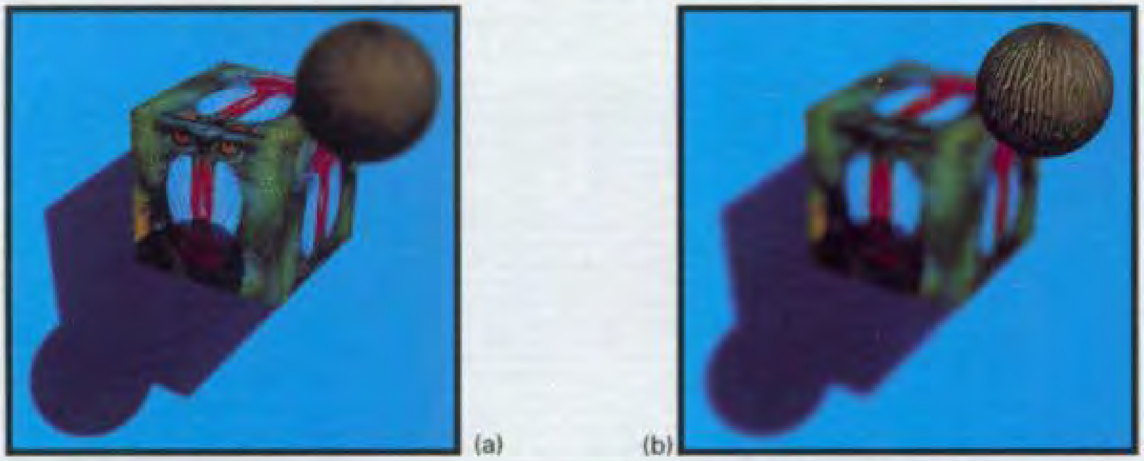

(a)
(b)

Plate II.38 Depth of field, implemented by postprocessing (Sections 14.4.10 and 16.10). (a) Focused at cube (550 mm), f/11 aperture. (b) Focused at sphere (290 mm), f/11 aperture. (Courtesy of Michael Potmesil and Indranil Chakravarty, RPI.)

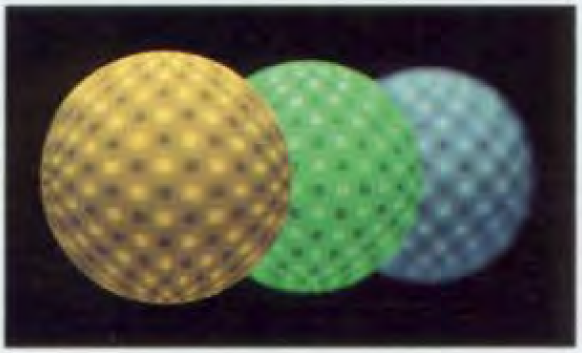


Plate II.39 Depth of field, implemented by distributed ray tracing (Sections 14.4.10 and 16.12.4). (By Robert Cook, Thomas Porter, and Loren Carpenter. Copyright © Pixar 1984. All rights reserved.)

just as in real life, so the viewer's eyes focus differently on different objects, depending on each object's proximity. Methods for producing and viewing stereo images are examined in more detail in Section 18.11.5; the mathematics of stereo projection is described in Exercise 6.27.

14.8 IMPROVED DISPLAYS

In addition to improvements in the software used to design and render objects, improvements in the displays themselves have heightened the illusion of reality. The history of computer graphics is in part that of a steady improvement in the visual quality achieved by display devices. Still, a modern monitor's color gamut and its dynamic intensity range are both a small subset of what we can see. We have a long way to go before the image on our display can equal the crispness and contrast of a well-printed professional photograph! Limited display resolution makes it impossible to reproduce extremely fine detail. Artifacts such as a visible phosphor pattern, glare from the screen, geometric distortion, and the stroboscopic effect of frame-rate flicker are ever-present reminders that we are viewing a display. The display's relatively small size, compared with our field of vision, also helps to remind us that the display is a window on a world, rather than a world itself.

14.9 INTERACTING WITH OUR OTHER SENSES

Perhaps the final step toward realism is the integration of realistic imagery with information presented to our other senses. Computer graphics has a long history of programs that rely on a variety of input devices to allow user interaction. Flight simulators are a current example of the coupling of graphics with realistic engine sounds and motion, all offered in a mocked-up cockpit to create an entire environment. The head-worn simulator of Color Plate I.16 monitors head motion, making possible another important 3D depth cue called *head-motion parallax*: when the user moves her head from side to side, perhaps to try to see more of a partially hidden object, the view changes as it would in real life. Other active work on head-mounted displays centers on the exploration of *virtual worlds*, such as the insides of molecules or of buildings that have not yet been constructed [CHUN89].

Many current arcade games feature a car or plane that the player rides, moving in time to a simulation that includes synthesized or digitized images, sound, and force feedback, as shown in Color Plate I.7. This use of additional output and input modalities points the way to systems of the future that will provide complete immersion of all the senses, including hearing, touch, taste, and smell.

14.10 ALIASING AND ANTIALIASING

In Section 3.17, we introduced the problem of aliasing and discussed some basic techniques for generating antialiased 2D primitives. Here we examine aliasing in more detail so that we can understand when and why it occurs, laying the groundwork for incorporating antialiasing into the visible-surface and shading algorithms covered in the following chapters. Additional material may be found in [CROW77b; CROW81]; an excellent set of examples is included in [BLIN89a; BLIN89b].

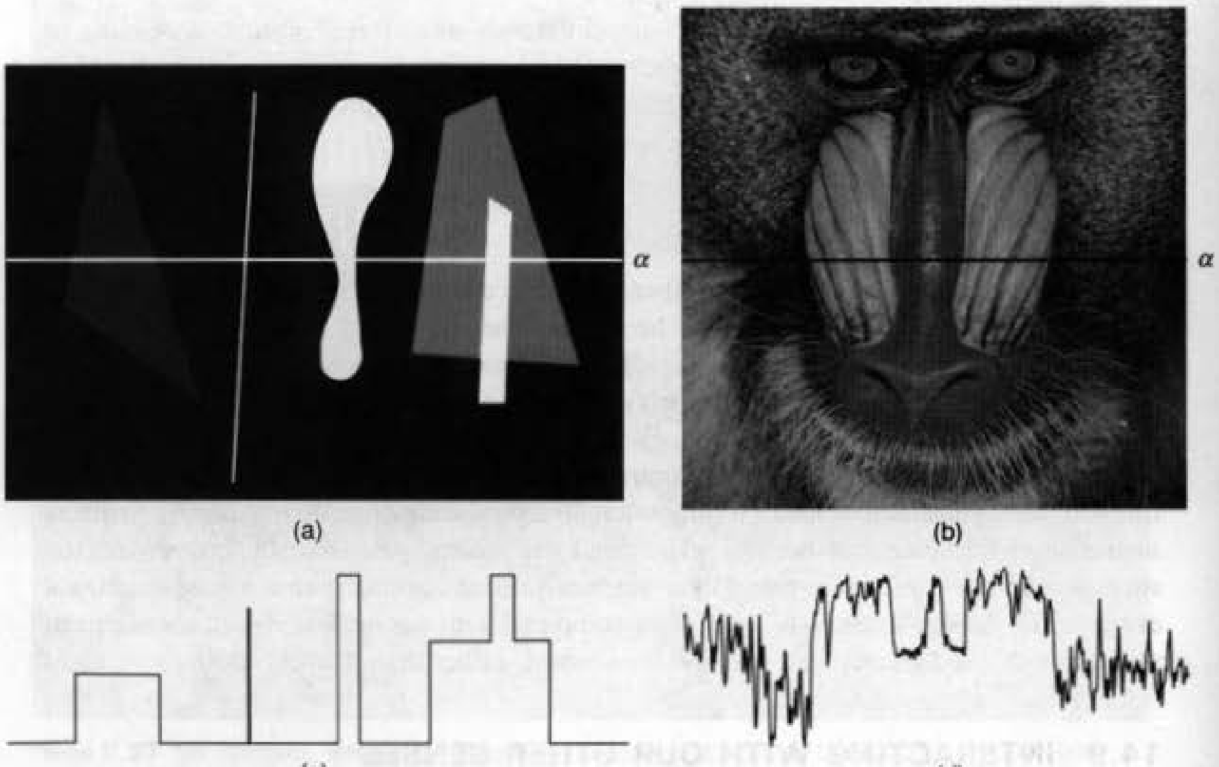


Fig. 14.8 Image. (a) Graphical primitives. (b) Mandrill. (c) Intensity plot of scan line α in (a). (d) Intensity plot of scan line α in (b). (Part d is courtesy of George Wolberg, Columbia University.)

To understand aliasing, we have to introduce some basic concepts from the field of signal processing. We start with the concept of a *signal,* which is a function that conveys information. Signals are often thought of as functions of time, but can equally well be functions of other variables. Since we can think of images as intensity variations over space, we will refer to signals in the *spatial domain* (as functions of spatial coordinates), rather than in the *temporal domain* (as functions of time). Although images are 2D functions of two independent spatial variables (x and y), for convenience our examples will often use the 1D case of a single spatial variable x. This case can be thought of as an infinitesimally thin slice through the image, representing intensity along a single horizontal line. Figure 14.8(a) and (b) show 2D signals, and parts (c) and (d) of the figure show plots of the intensity along the horizontal line α.

Signals can be classified by whether or not they have values at all points in the spatial domain. A *continuous signal*[1] is defined at a continuum of positions in space; a *discrete signal* is defined at a set of discrete points in space. Before scan conversion, the projection of our 3D objects onto the view plane may be treated as a continuous 2D signal whose value

[1]Not to be confused with the definition of continuity in calculus.

at each infinitesimal point in the plane indicates the intensity at that point. In contrast, the array of pixel values in the graphics system's frame buffer is a discrete 2D signal whose value is defined only at the positions in the array. Our rendering algorithms must determine the intensities of the finite number of pixels in the array so that they best represent the continuous 2D signal defined by the projection. The precise meaning of "best represent" is not at all obvious, however. We shall discuss this problem further.

A continuous signal may contain arbitrarily fine detail in the form of very rapid (high-frequency) variations in its value as its continuous parameter is changed. Since a discrete signal can change value only at discrete points, it clearly has a maximum rate of variation. Therefore, it should be clear that converting a continuous signal to a finite array of values may result in a loss of information. Our goal is to ensure that as little information as possible is lost, so that the resulting pixel array can be used to display a picture that looks as much as possible like the original signal would look if we were able to display it directly. The process of selecting a finite set of values from a signal is known as *sampling*, and the selected values are called *samples*. Once we have selected these samples, we must then display them using a process, known as *reconstruction*, that attempts to recreate the original continuous signal from the samples. The array of pixels in the frame buffer is reconstructed by the graphics system's display hardware, which converts these discrete intensity values to continuous, analog voltages that are applied to the CRT's electron gun (see Chapter 4). An idealized version of this pipeline is shown in Fig. 14.9. Signal-processing theory [GONZ87] establishes the minimum frequency at which samples must be selected from a given signal to reconstruct an exact copy of the signal, and specifies how to perform the reconstruction process. As we show later, however, this minimum sampling frequency will be infinite for many kinds of signals in which we are interested, so perfect reconstruction will often be impossible. Furthermore, as described in Section 14.10.5, the reconstruction method typically used by the display hardware differs from the approach prescribed by theory. Therefore, even properly sampled signals will not be reconstructed perfectly.

14.10.1 Point Sampling

The most straightforward way to select each pixel's value is known as *point sampling*. In point sampling, we select one point for each pixel, evaluate the original signal at this point, and assign its value to the pixel. The points that we select are typically arranged in a regular grid, as shown in Fig. 14.10. Unlike the scan-conversion algorithms of Chapter 3, projected vertices are not constrained to lie on integer grid points. Because the signal's values at a finite set of points are sampled, however, important features of the signal may be missed. For example, objects *A* and *C* in Fig. 14.10 are represented by the samples, whereas objects *B* and *D* are not. To make matters worse, if the viewing specification changes slightly or if the objects move, objects may pop in or out of visibility. What if we sample at a higher rate? The more samples we collect from the signal, the more we know about it. For example, we can see easily that, by increasing sufficiently the number of samples taken horizontally and vertically in Fig. 14.10, we can make sure that no object is missed in that particular picture. This is a necessary, but *not* a sufficient condition for adequate sampling. Nevertheless, sampling at a higher rate, we can generate images with more pixels representing each portion of the picture. We can also generate an image with

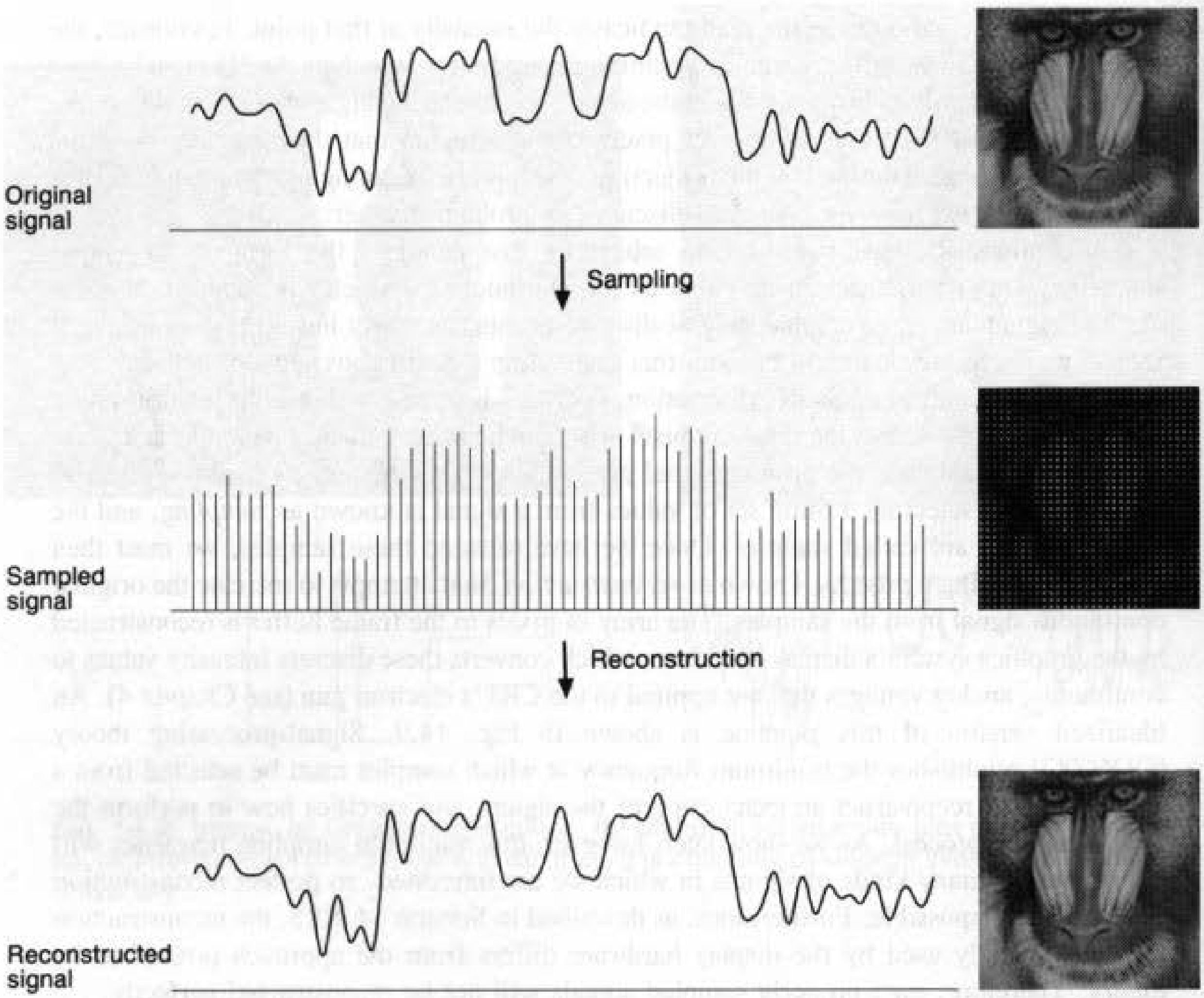


Fig. 14.9 The original signal is sampled, and the samples are used to reconstruct the signal. (Sampled 2D image is an approximation, since point samples have no area.) (Courtesy of George Wolberg, Columbia University.)

fewer pixels, by combining several adjacent samples (e.g., by averaging) to determine the value of each pixel of the smaller image. This means that all the features that would be present in the larger image at least contribute to the smaller one.

The approach of taking more than one sample for each pixel and combining them is known as *supersampling*. It actually corresponds to reconstructing the signal and resampling the reconstructed signal. For reasons described later, sampling the reconstructed signal is often better than sampling the original signal. This technique is popular in computer graphics precisely because it is so easy and often achieves good results, despite the obvious increase in computation. But, how many samples are enough? How do we know that there are no features that our samples are missing? Merely testing whether every object's projection is sampled is not sufficient. The projection may have a complex shape or variations in shading intensity that the samples do not reflect. We would like some way to guarantee that the samples we take are spaced close enough to reconstruct the original

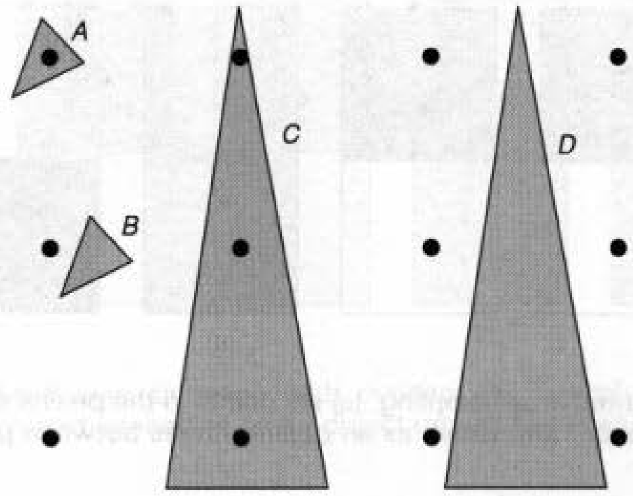


Fig. 14.10 Point-sampling problems. Samples are shown as black dots (●). Objects *A* and *C* are sampled, but corresponding objects *B* and *D* are not.

signal. As we shall see, sampling theory tells us that, on the basis of a particular signal's properties, we can compute a minimum sampling rate that will be adequate. Unfortunately, the rate turns out to be infinite for certain kinds of signals, including the signal shown in Fig. 14.10! We shall explain the reason for this in more detail later; for now, we can see that taking a finite number of samples cannot guarantee to capture the exact x coordinate at which the intensity jumps from one value to another in the figure. Furthermore, even if we find a finite sampling rate at which all of the current objects are sampled, we can always imagine adding just one more object positioned between samples that will be missed entirely.

14.10.2 Area Sampling

The problem of objects "falling between" samples and being missed suggests another approach: integrating the signal over a square centered about each grid point, dividing by the square's area, and using this average intensity as that of the pixel. This technique, called *unweighted area sampling,* was introduced in Chapter 3. The array of nonoverlapping squares is typically thought of as representing the pixels. Each object's projection, no matter how small, contributes to those pixels that contain it, in strict proportion to the amount of each pixel's area it covers, and without regard to the location of that area in the pixel, as shown by the equal weighting function of Fig. 14.11(a). No objects are missed, as may happen with point sampling. The definition of the definite integral requires evaluating a function at many points of an interval, and then taking the limit as the number of points increases. Thus, integrating amounts to a kind of infinite sampling process.

Unweighted area sampling has drawbacks caused by this evenhandedness with which objects are treated. Consider a small black object wholly contained inside of one of the pixels and surrounded by a white background, as in Fig. 14.11(b). This small object may move freely inside the pixel, and for each position the value computed for the pixel (shown as the pixel's shade) remains the same. As soon as the object crosses over into an adjoining pixel, however, the values of the original pixel and the adjoining pixel are both affected.

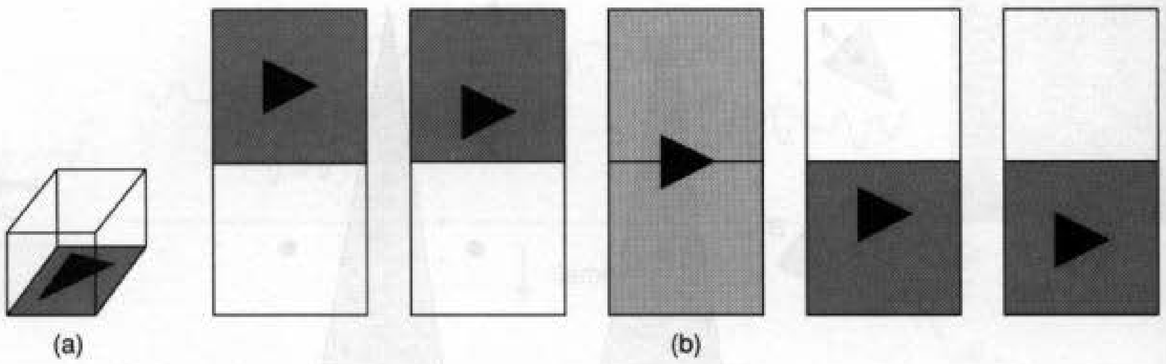


Fig. 14.11 Unweighted area sampling. (a) All points in the pixel are weighted equally. (b) Changes in computed intensities as an object moves between pixels.

Thus, the object causes the image to change only when it crosses pixel boundaries. As the object moves farther from the center of one pixel and closer to the center of another, however, we would like this change to be represented in the image. In other words, we would like the object's contribution to the pixel's intensity to be *weighted* by its distance from the pixel's center: the farther away it is, the less it should contribute.

In Chapter 3, we noted that *weighted area sampling* allows us to assign different weights to different parts of the pixel, and we suggested that the weighting functions of adjacent pixels should overlap. To see why the overlap is needed, we consider a weighting function consisting of an upright pyramid erected over a single pixel, as shown in Fig. 14.12(a). Under this weighting, as desired, an object contributes less to a pixel as it moves away from the pixel's center. But a drawback of unweighted area sampling still remains: An object contributes to only the single pixel that contains it. Consider a subpixel-sized black object moving over a white background from the center of one pixel to the center of an adjacent pixel, as shown in Fig. 14.12(b). As the object moves away from the center of the first pixel, its contribution to the first pixel decreases as it nears its edge. It begins to contribute to the pixel it enters only after it has crossed its border, and reaches its maximum contribution when it reaches the center of the new pixel. Thus, even though the black object has constant intensity, the first pixel increases in intensity before the second pixel decreases in intensity.

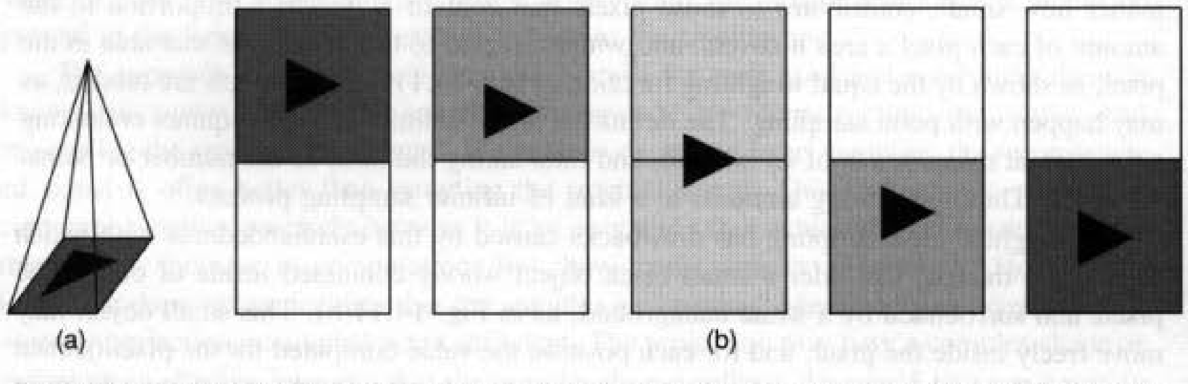


Fig. 14.12 Weighted area sampling. (a) Points in the pixel are weighted differently. (b) Changes in computed intensities as an object moves between pixels.

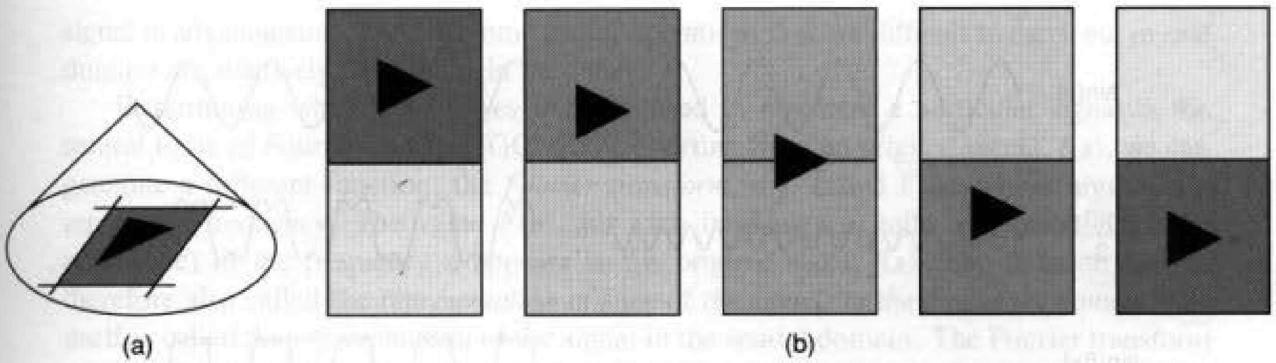


Fig. 14.13 Weighted area sampling with overlap. (a) Typical weighting function. (b) Changes in computed intensities as an object moves between pixels.

The net effect is that the display changes in intensity depending on the object's position, a change that gives rise to flickering as the object moves across the screen. It is clear that, to correct this problem, we must allow our weighting functions to overlap, so that a point on an object can simultaneously influence more than one pixel, as shown in Fig. 14.13. This figure also uses a radially symmetric weighting function. Here, it is appropriate to turn to sampling theory to discover the underlying reasons for increasing the weighting function's size, and to find out exactly what we need to do to sample and reconstruct a signal.

14.10.3 Sampling Theory

Sampling theory provides an elegant mathematical framework to describe the relationship between a continuous signal and its samples. So far, we have considered signals in the *spatial domain*; that is, we have represented each of them as a plot of amplitude against spatial position. A signal may also be considered in the *frequency domain*; that is, we may represent it as a sum of sine waves, possibly offset from each other (the offset is called *phase shift*), and having different frequencies and amplitudes. Each sine wave represents a component of the signal's *frequency spectrum*. We sum these components in the spatial domain by summing their values at each point in space.

Periodic signals, such as those shown in Fig. 14.14, can each be represented as the sum of phase-shifted sine waves whose frequencies are integral multiples (*harmonics*) of the signal's *fundamental* frequency. But what of nonperiodic signals such as images? Since an image is of finite size, we can define its signal to have a value of zero outside the area of the image. Such a signal, which is nonzero over a finite domain, and, more generally, any signal $f(x)$ that tapers off sufficiently fast (faster than $1/x$ for large values of x) can also be represented as a sum of phase-shifted sine waves. Its frequency spectrum, however, will not consist of integer multiples of some fundamental frequency, but may contain any frequency at all. The original signal cannot be represented as a sum of countably many sine waves, but instead must be represented by an integral over a continuum of frequencies. It is often the case, however, that an image (perhaps padded with surrounding zeros) is treated as one cycle of a periodic signal. This was done in Fig. 14.14(b), which shows the first ten components of Fig. 14.8(d). Each signal in the spatial domain has one representation in the frequency domain, and vice versa. As we shall see later, using two representations for a

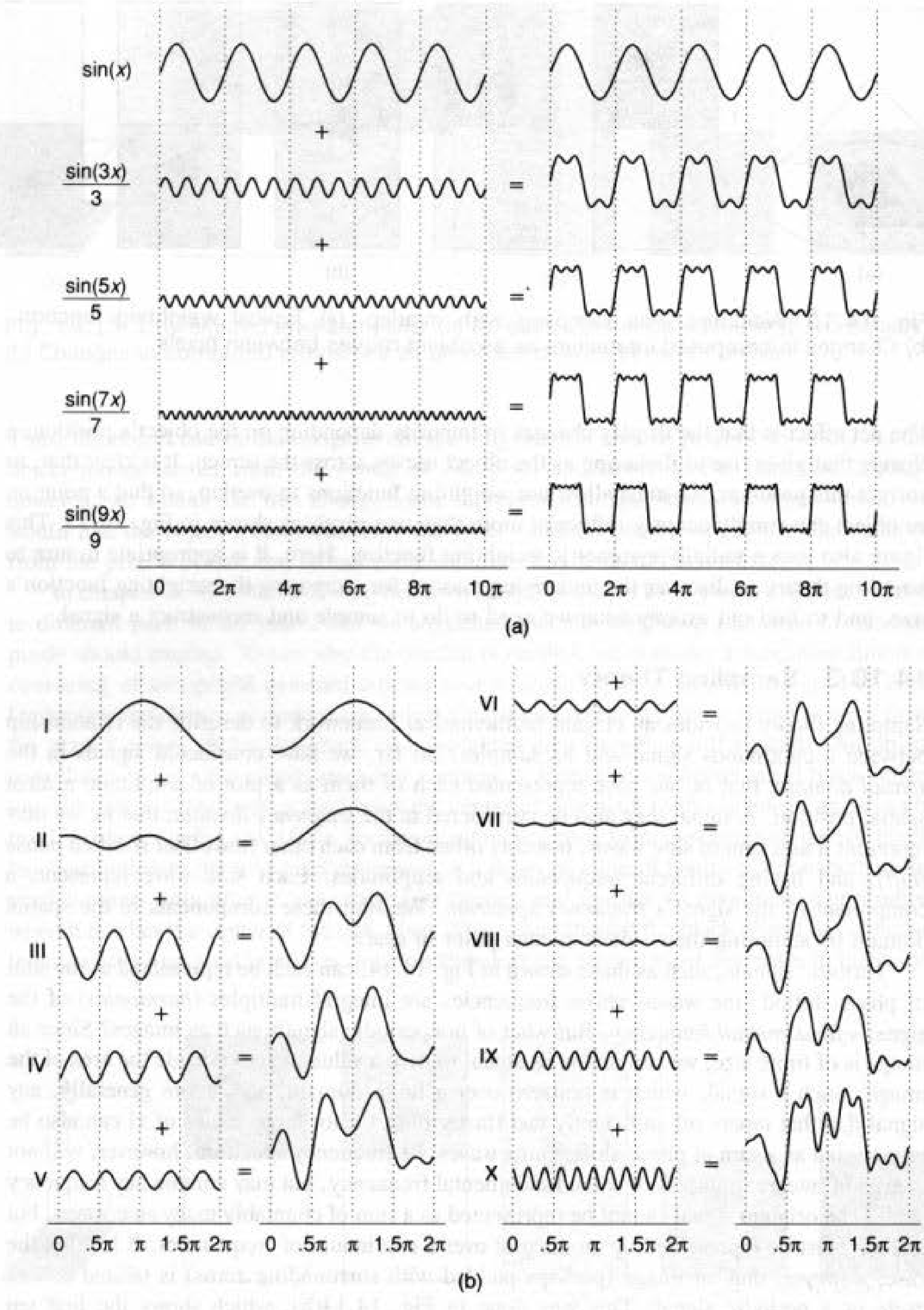


Fig. 14.14 A signal in the spatial domain is the sum of phase-shifted sines. Each component is shown with its effect on the signal shown at its right. (a) Approximation of a square wave. (b) Approximation of Fig. 14.8(d). (Courtesy of George Wolberg, Columbia University.)

signal is advantageous, because some useful operations that are difficult to carry out in one domain are relatively easy to do in the other.

Determining which sine waves must be used to represent a particular signal is the central topic of *Fourier analysis* [GONZ87]. Starting from an original signal, $f(x)$, we can generate a different function, the *Fourier transform* of f, called $F(u)$, whose argument u represents frequency. The value $F(u)$, for each frequency u, tells how much (i.e., the amplitude) of the frequency u appears in the original signal $f(x)$. The function $F(u)$ is therefore also called the *representation of f* (or of *the signal*) *in the frequency domain;* $f(x)$ itself is called the representation of the signal in the spatial domain. The Fourier transform of a continuous, integrable signal $f(x)$ from the spatial domain to the frequency domain is defined by

$$F(u) = \int_{-\infty}^{+\infty} f(x)[\cos 2\pi ux - i \sin 2\pi ux]dx, \tag{14.1}$$

where $i = \sqrt{-1}$ and u represents the frequency of a sine and cosine pair. (Note that this applies only to functions that taper off sufficiently fast.) Recall that the cosine is just the sine, phase shifted by $\pi/2$. Together they can be used to determine the amplitude and phase shift of their frequency's component. For each u, the value of $F(u)$ is therefore a complex number. This is a clever way of encoding the phase shift and amplitude of the frequency u component of the signal: The value $F(u)$ may be written as $R(u) + iI(u)$, where $R(u)$ and $I(u)$ are the real and imaginary parts, respectively. The amplitude (or magnitude) of $F(u)$ is defined by

$$|F(u)| = \sqrt{R^2(u) + I^2(u)}, \tag{14.2}$$

and the phase shift (also known as the *phase angle*) is given by

$$\phi(u) = \tan^{-1}\left[\frac{I(u)}{R(u)}\right]. \tag{14.3}$$

In turn, an integrable signal $F(u)$ may be transformed from the frequency domain to the spatial domain by the *inverse Fourier transform*

$$f(x) = \int_{-\infty}^{+\infty} F(u)[\cos 2\pi ux + i \sin 2\pi ux]du. \tag{14.4}$$

The Fourier transform of a signal is often plotted as magnitude against frequency, ignoring phase angle. Figure 14.15 shows representations of several signals in both domains. In the spatial domain, we label the abscissa with numbered pixel centers; in the frequency domain, we label the abscissa with cycles per pixel (or more precisely, cycles per interval between pixel centers). In each case, the spike at $u = 0$ represents the DC (direct current) component of the spectrum. Substituting $\cos 0 = 1$ and $\sin 0 = 0$ in Eq. (14.1) reveals that this corresponds to integrating $f(x)$. If .5 were subtracted from each value of $f(x)$ in Fig. 14.15 (a) or (b), the magnitude of the signal's DC component would be 0.

Most of the figures in this chapter that show signals and their Fourier transforms were actually computed using discrete versions of Eqs. (14.1) and (14.4) that operate on signals represented by N regularly spaced samples. The *discrete Fourier transform* is

$$F(u) = \sum_{0 \le x \le N-1} f(x)[\cos(2\pi ux/N) - i \sin(2\pi ux/N)], \quad 0 \le u \le N-1, \tag{14.5}$$

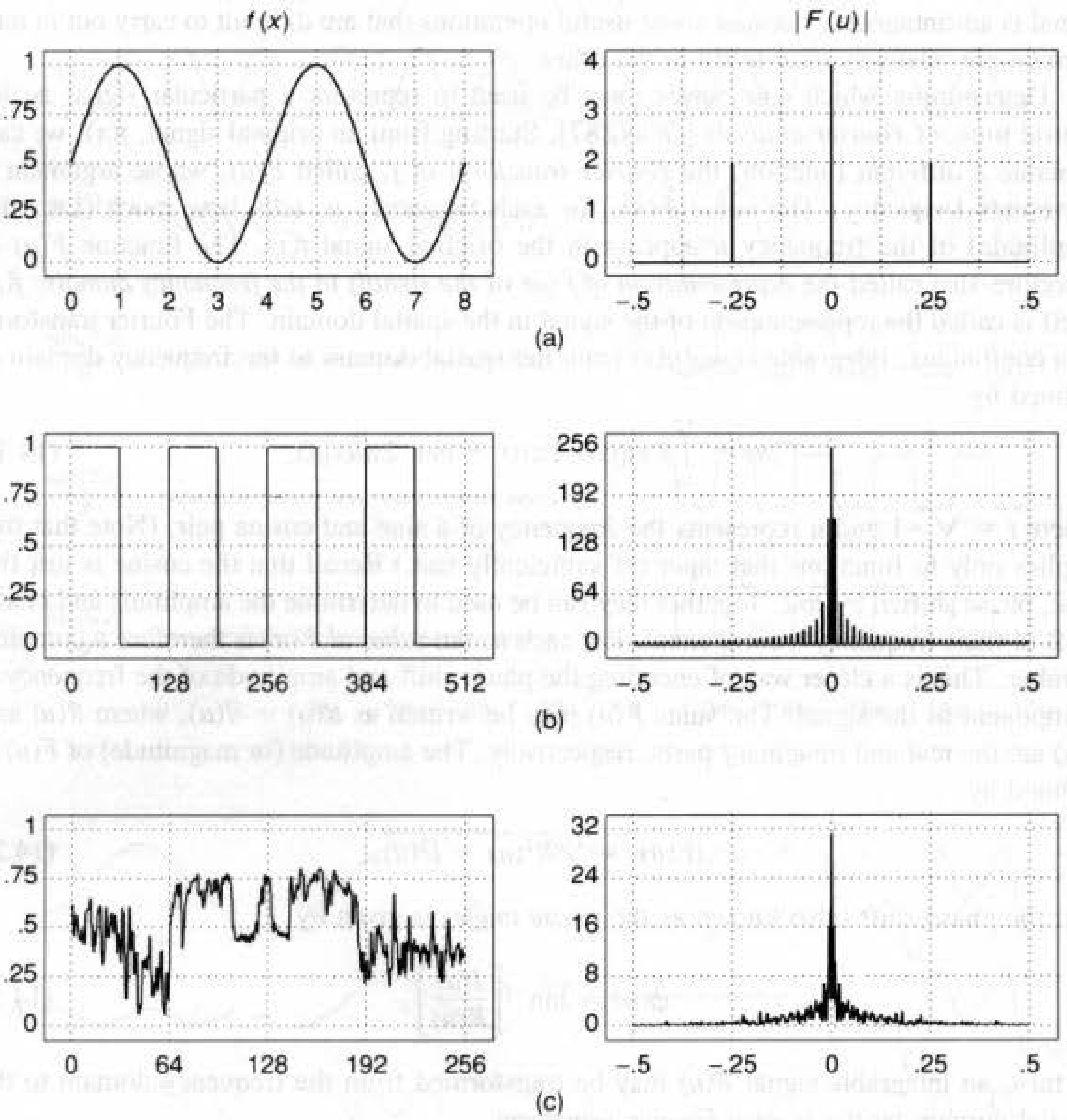


Fig. 14.15 Signals in the spatial and frequency domains. (a) Sine. (b) Square Wave. (c) Mandrill. The DC value in the frequency domain is truncated to make the other values legible and should be 129. (Courtesy of George Wolberg, Columbia Univeristy.)

and the *inverse discrete Fourier transform* is

$$f(x) = \frac{1}{N}\sum_{0 \le u \le N-1} F(u)[\cos(2\pi ux/N) + i\sin(2\pi ux/N)],\ 0 \le x \le N-1. \quad (14.6)$$

By choosing a sufficiently high sampling rate, a good approximation to the behavior of the continuous Fourier transform is obtained for most signals. (The discrete Fourier transform may also be computed more efficiently than Eqs. (14.5) and (14.6) would imply, by using a clever reformulation known as the *fast Fourier transform* [BRIG74].) The discrete Fourier transform always yields a finite spectrum. Note that, if a signal is symmetric about the origin, then $I(u) = 0$. This is true because the contribution of each sine term on one side of the origin is canceled by its equal and opposite contribution on the other side. In this case, following [BLIN89a], we will plot the signed function $R(u)$, instead of the magnitude $|F(u)|$.

Sampling theory tells us that a signal can be properly reconstructed from its samples if the original signal is sampled at a frequency that is greater than twice f_h, the highest-frequency component in its spectrum. This lower bound on the sampling rate is known as the *Nyquist rate*. Although we do not give the formal proof of the adequacy of sampling above the Nyquist rate, we can provide an informal justification. Consider one cycle of a signal whose highest-frequency component is at frequency f_h. This component is a sine wave with f_h maxima and f_h minima, as shown in Fig. 14.16. Therefore, at least $2f_h$ samples are required to capture the overall shape of the signal's highest-frequency component. Note that exactly $2f_h$ samples is, in fact, a special case that succeeds only if the samples are taken precisely at the maxima and minima (Fig. 14.16a). If they are taken anywhere else, then the amplitude will not be represented correctly (Fig. 14.16b) and may even be determined to be zero if the samples are taken at the zero crossings (Fig. 14.16c). If we sample below the Nyquist rate, the samples we obtain may be identical to what would have been obtained from sampling a lower-frequency signal, as demonstrated in Fig. 14.17. This phenomenon of high frequencies masquerading as low frequencies in the reconstructed signal is known as *aliasing*: The high-frequency components appear as though they were actually lower-frequency components. Another example of aliasing is demonstrated in Fig. 14.18. Figure 14.18(a) shows an image and a plot of its intensity across a horizontal line, representing a set of intensity fluctuations that increase in spatial frequency from left to right. The image in Fig. 14.18(b) was created by selecting every 8th pixel from each line of Fig. 14.18(a) and replicating it eight times. It shows aliasing as the bands increase in spatial frequency.

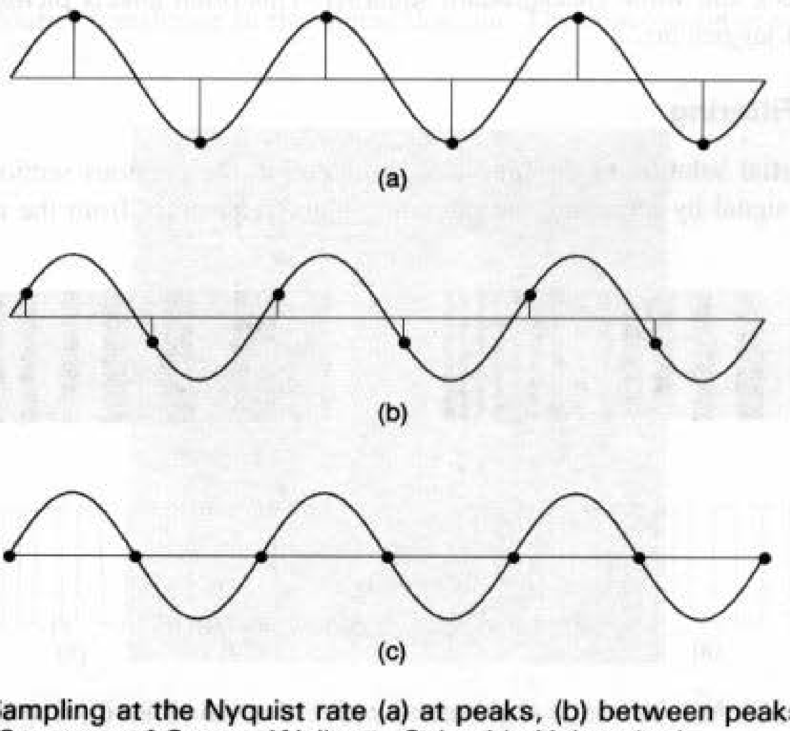


Fig. 14.16 Sampling at the Nyquist rate (a) at peaks, (b) between peaks, (c) at zero crossings. (Courtesy of George Wolberg, Columbia University.)

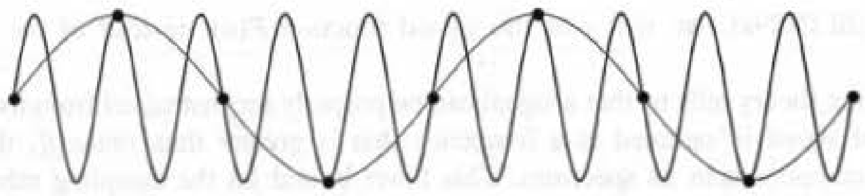


Fig. 14.17 Sampling below the Nyquist rate. (Courtesy of George Wolberg, Columbia University.)

A signal's shape is determined by its frequency spectrum. The sharper and more angular a waveform is, the richer it is in high-frequency components; signals with discontinuities have an infinite frequency spectrum. Figure 14.10 reveals the sharp edges of the objects' projections that our algorithms attempt to represent. This signal has an infinite frequency spectrum, since the image intensity changes discontinuously at object boundaries. Therefore, the signal cannot be represented properly with a finite number of samples. Computer graphics images thus exhibit two major kinds of aliasing. First, "jaggies" along edges are caused by discontinuities at the projected edges of objects: a point sample either does or does not lie in an object's projection. Even the presence of a single such edge in an environment's projection means that the projection has an infinite frequency spectrum. The frequency spectrum tapers off quite rapidly, however, like those of Fig. 14.15(b) and (c). Second, textures and objects seen in perspective may cause arbitrarily many discontinuities and fluctuations in the environment's projection, making it possible for objects whose projections are too small and too close together to be alternately missed and sampled, as in the right hand side of Fig. 14.18(b). The high frequency components representing the frequency at which these projections cross a scan line may have high amplitude (e.g., alternating black and white checkerboard squares). This often affects picture quality more seriously than jaggies do.

14.10.4 Filtering

There is a partial solution to the problems discussed in the previous section. If we could create a new signal by removing the offending high frequencies from the original signal,

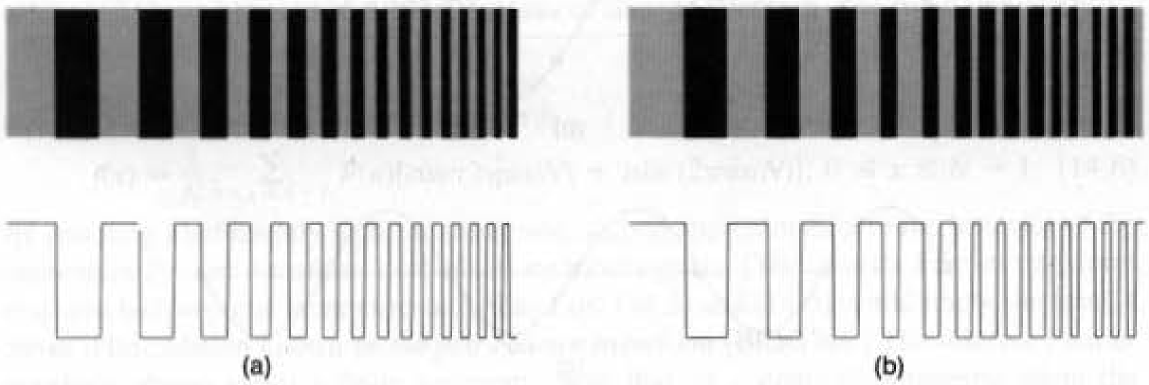


Fig. 14.18 Aliasing. (a) Image and intensity plot of a scan line. (b) Sampled image and intensity plot of a scan line. (Courtesy of George Wolberg, Columbia University.)

then the new signal could be reconstructed properly from a finite number of samples. The more high frequencies we remove, the lower the sampling frequency needed, but the less the signal resembles the original. This process is known as *bandwidth limiting* or *band limiting* the signal. It is also known as *low-pass filtering,* since filtering a signal changes its frequency spectrum; in this case, high frequencies are filtered out and only low frequencies are allowed to pass. Low-pass filtering causes blurring in the spatial domain, since fine visual detail is captured in the high frequencies that are attenuated by low-pass filtering, as shown in Fig. 14.19. We shall revise the pipeline of Fig. 14.9 to include an optional filter, as shown in Fig. 14.20.

A perfect low-pass filter completely suppresses all frequency components above some specified cut-off point, and lets those below the cut-off point pass untouched. We can easily do this filtering in the frequency domain by multiplying the signal's spectrum by a *pulse function,* as shown in Fig. 14.21. We can multiply two signals by taking their product at each point along the paired signals. The pulse function

$$S(u) = \begin{cases} 1, \text{ when } -k \le u \le k, \\ 0, \text{ elsewhere.} \end{cases} \tag{14.7}$$

cuts off all components of frequency higher than k. Therefore, if we were to low-pass filter the signal so as to remove all variation, we would be left with only its DC value.

So far, it would seem that a recipe for low-pass filtering a signal in the spatial domain would involve transforming the signal into the frequency domain, multiplying it by an appropriate pulse function, and then transforming the product back into the spatial domain. Some important relationships between signals in the two domains, however, make this procedure unnecessary. It can be shown that multiplying two Fourier transforms in the frequency domain corresponds exactly to performing an operation called *convolution* on their inverse Fourier transforms in the spatial domain. The *convolution* of two signals $f(x)$

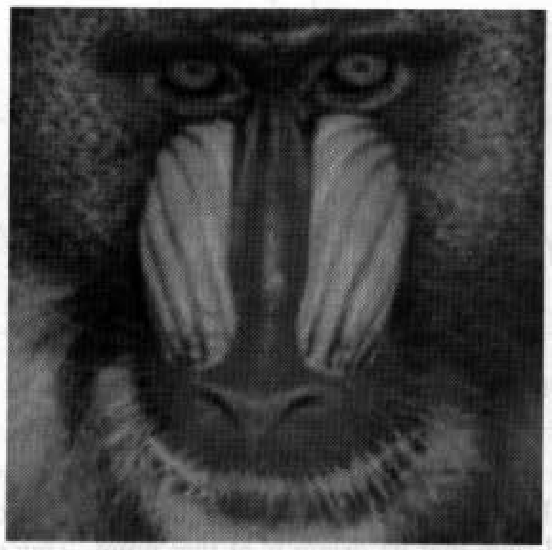


Fig. 14.19 Figure 14.8(b) after low-pass filtering. (Courtesy of George Wolberg, Columbia University.)

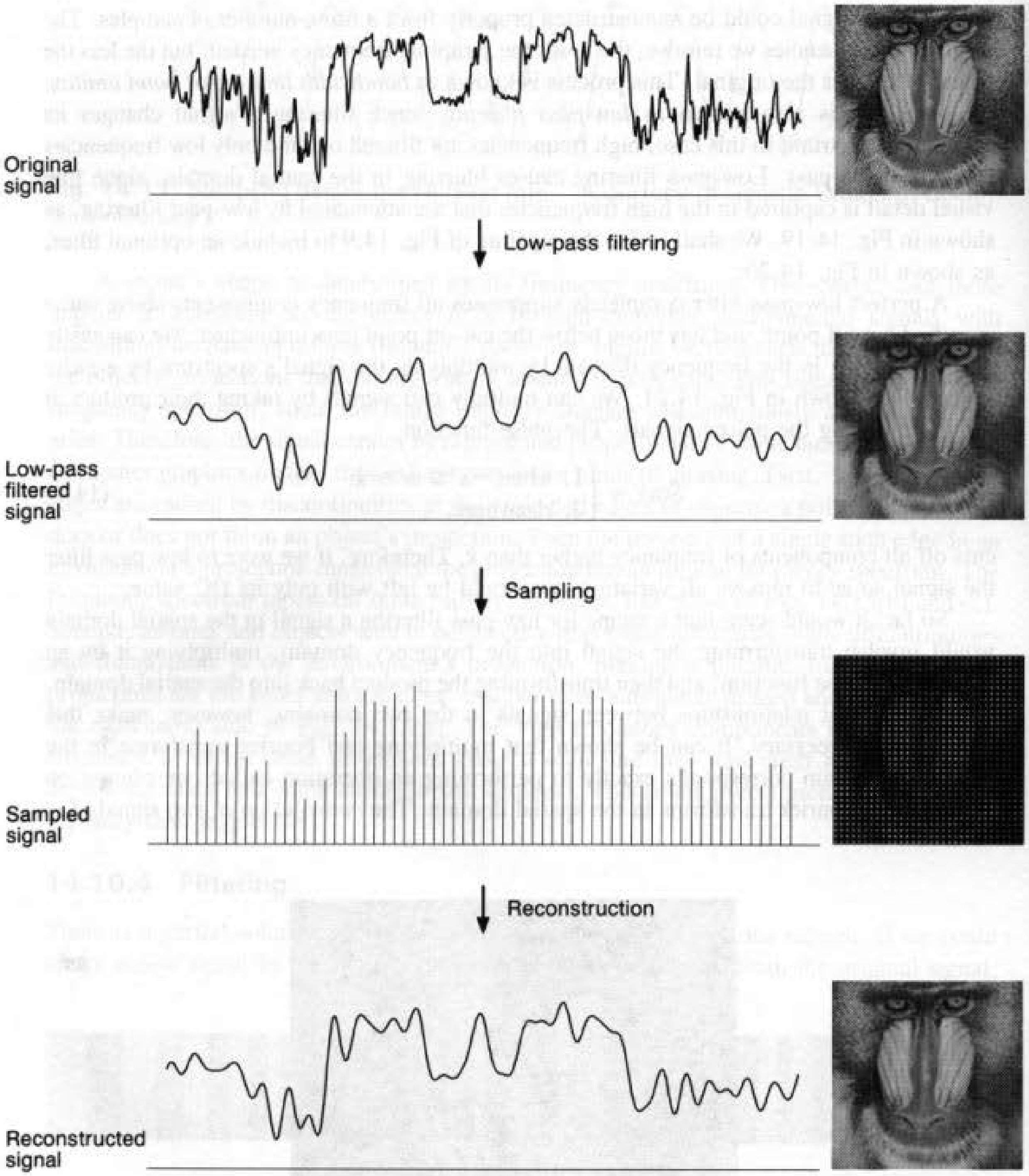


Fig. 14.20 The sampling pipeline with filtering. (Courtesy of George Wolberg, Columbia University.)

and $g(x)$, written as $f(x) * g(x)$, is a new signal $h(x)$ defined as follows. The value of $h(x)$ at each point is the integral of the product of $f(x)$ with the filter function $g(x)$ flipped about its vertical axis and shifted such that its origin is at that point. This corresponds to taking a weighted average of the neighborhood around each point of the signal $f(x)$—weighted by a flipped copy of filter $g(x)$ positioned at the point—and using it for the value of $h(x)$ at the point. The size of the neighborhood is determined by the size of the domain over which the filter is nonzero. This is known as the filter's *support*, and a filter that is nonzero over a finite domain is said to have *finite support*. We use τ as a dummy variable of integration when defining the convolution. Thus,

$$h(x) = f(x) * g(x) \overset{\Delta}{=} \int_{-\infty}^{+\infty} f(\tau)g(x - \tau)d\tau. \tag{14.8}$$

Conversely, convolving two Fourier transforms in the frequency domain corresponds exactly to multiplying their inverse Fourier transforms in the spatial domain. The filter function is often called the *convolution kernel* or *filter kernel*.

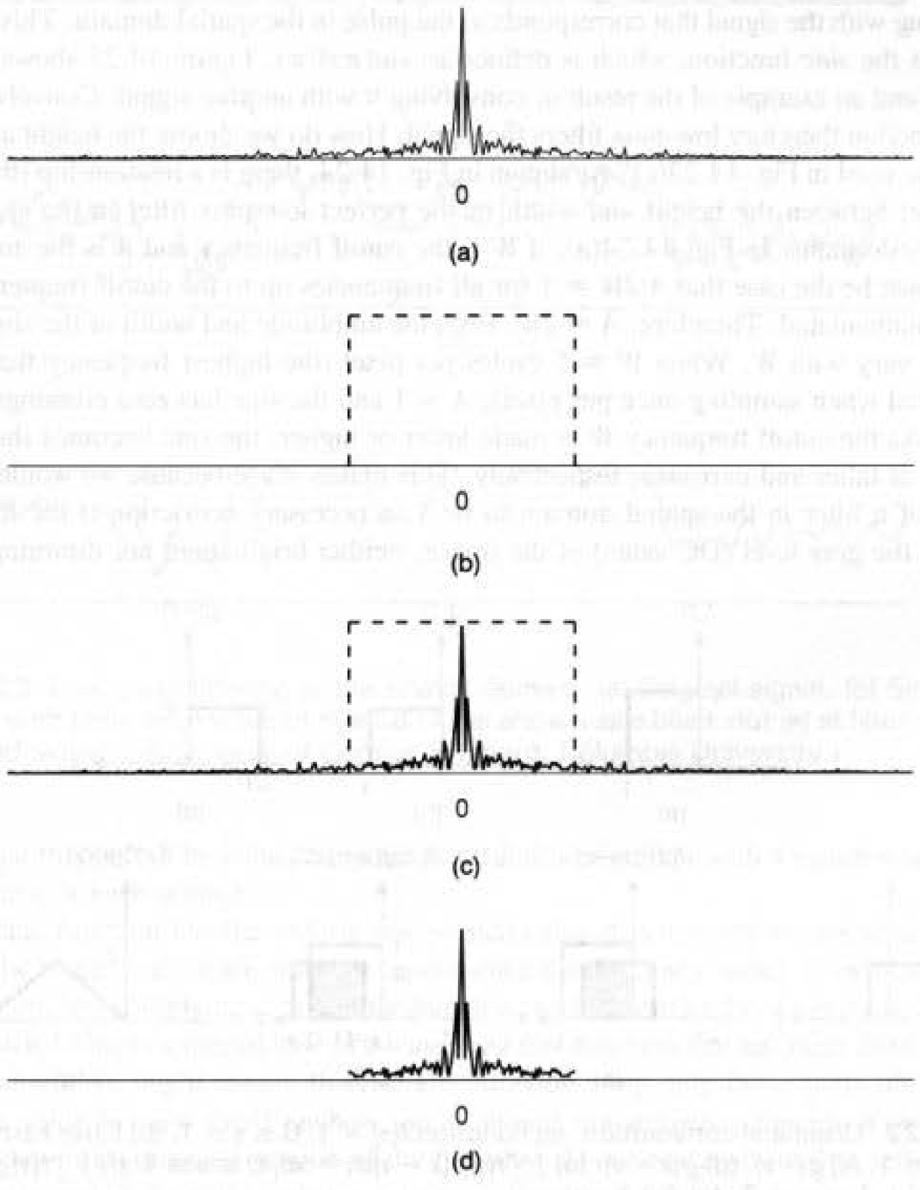


Fig. 14.21 Low-pass filtering in the frequency domain. (a) Original spectrum. (b) Low-pass filter. (c) Spectrum with filter. (d) Filtered spectrum. (Courtesy of George Wolberg, Columbia University.)

Convolution can be illustrated graphically. We will convolve the function $f(x) = 1$, $0 \le x \le 1$, with the filter kernel $g(x) = c$, $0 \le x \le 1$, shown in Figs. 14.22(a) and (b). By using functions of τ, we can vary x to move the filter relative to the signal being filtered. To create the function $g(x - \tau)$, we first flip $g(\tau)$ about the origin to yield $g(-\tau)$, and then offset it by x to form $g(x - \tau)$, as depicted in Figs. 14.22(c) and (d). The integral, with respect to τ, of the product $f(\tau)g(x - \tau)$, which is the area of the shaded portions of the figures, is 0 for $-\infty \le x < 0$, xc for $0 \le x \le 1$ (Fig. 14.22e), $(2 - x)c$ for $1 \le x \le 2$ (Fig. 14.22f), and 0 for $2 < x \le \infty$. The convolution $f(x) * g(x)$ is illustrated in Fig. 14.22(g). Note how convolution with this kernel smooths the discontinuities of $f(x)$ while it widens the area over which $f(x)$ is nonzero.

Multiplying by a pulse function in the frequency domain has the same effect as convolving with the signal that corresponds to the pulse in the spatial domain. This signal is known as the *sinc* function, which is defined as $\sin(\pi x)/\pi x$. Figure 14.23 shows the sinc function and an example of the result of convolving it with another signal. Convolving with a sinc function therefore low-pass filters the signal. How do we choose the height and width of the sinc used in Fig. 14.23(c)? As shown in Fig. 14.24, there is a relationship (that we do not prove) between the height and width of the perfect low-pass filter in the spatial and frequency domains. In Fig. 14.24(a), if W is the cutoff frequency and A is the amplitude, then it must be the case that $A/2W = 1$ for all frequencies up to the cutoff frequency to be passed unattenuated. Therefore, $A = 2W$. Both the amplitude and width of the sinc in Fig. 14.24(a) vary with W. When $W = .5$ cycles per pixel (the highest frequency that can be represented when sampling once per pixel), $A = 1$ and the sinc has zero crossings at pixel centers. As the cutoff frequency W is made lower or higher, the sinc becomes shorter and broader, or taller and narrower, respectively. This makes sense because we would like the integral of a filter in the spatial domain to be 1, a necessary restriction if the filter is to maintain the gray level (DC value) of the image, neither brightening nor dimming it. (We

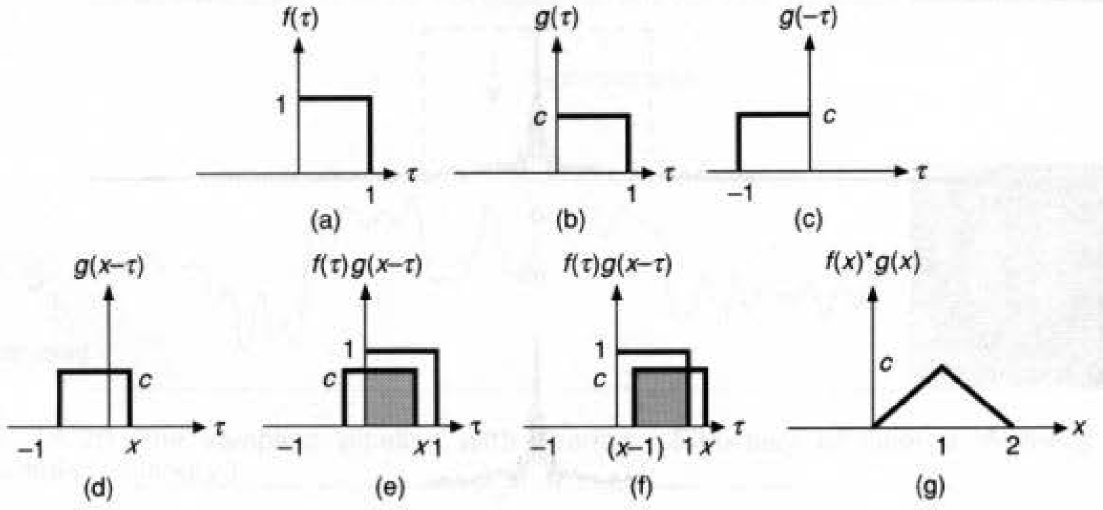


Fig. 14.22 Graphical convolution. (a) Function $f(\tau) = 1$, $0 \le \tau \le 1$. (b) Filter kernel $g(\tau) = c$, $0 \le \tau \le 1$. (c) $g(-\tau)$. (d) $g(x - \tau)$. (e) $\int_{-\infty}^{+\infty} f(\tau)g(x - \tau)d\tau = xc$, $0 \le x \le 1$. (f) $\int_{-\infty}^{+\infty} f(\tau)g(x - \tau)d\tau = (2 - x)c$, $1 \le x \le 2$. (g) $f(x) * g(x)$.

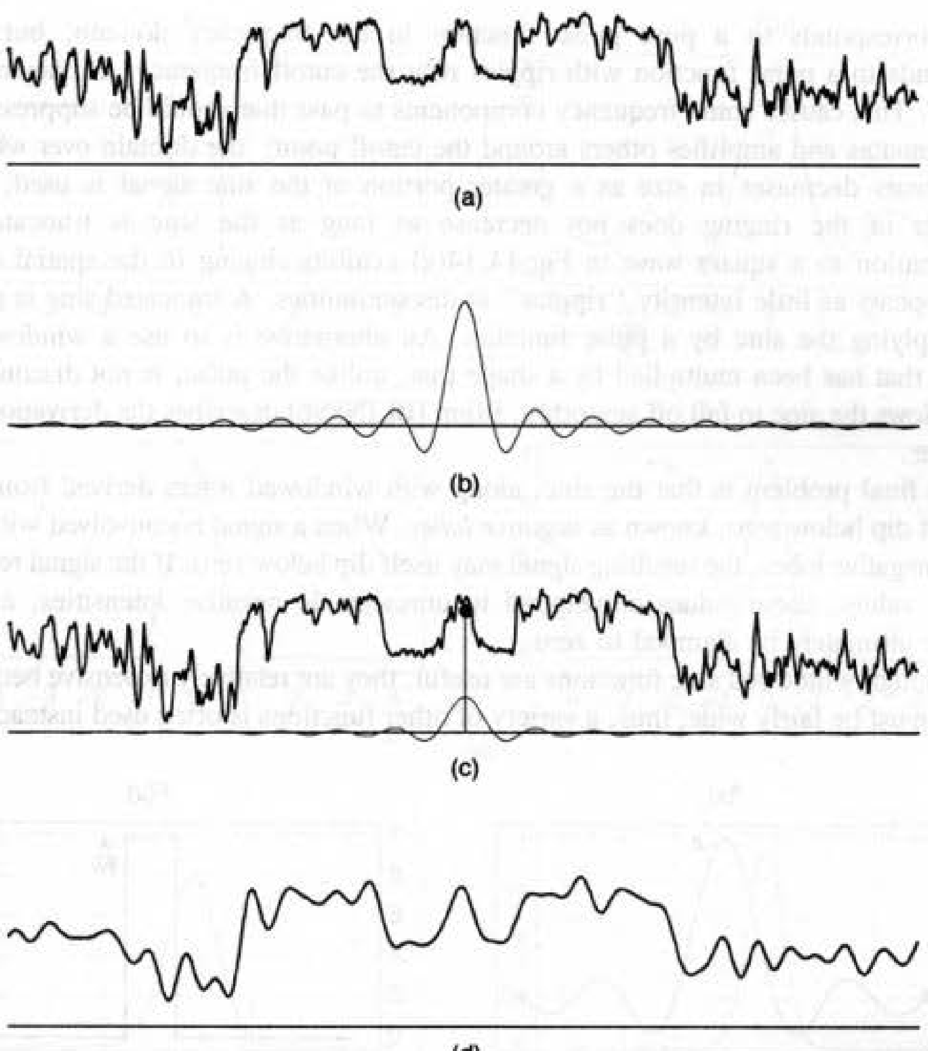


Fig. 14.23 Low-pass filtering in the spatial domain. (a) Original signal. (b) Sinc filter. (c) Signal with filter, with value of filtered signal shown as a black dot (●) at filter's origin. (d) Filtered signal. (Courtesy of George Wolberg, Columbia University.)

can see that this is true by considering the convolution of a filter with a signal that has the same value c at each point.)

The sinc function has the unfortunate property that it is nonzero on points arbitrarily far from the origin (i.e., it has *infinite support* since it is infinitely wide). If we truncate the sinc function, by multiplying it by a pulse function, we can restrict the support, as shown in Fig. 14.24(b). This is a special case of a *windowed* sinc function that has been restricted to a finite window. We might reason that we are throwing away only those parts of the filter where the value is very small anyhow, so it should not influence the result too much. Unfortunately, the truncated version of the filter has a Fourier transform that suffers from *ringing* (also called the *Gibbs phenomenon*): A truncated sinc in the spatial domain no longer corresponds to a pure pulse function in the frequency domain, but instead corresponds to a pulse function with ripples near the cutoff frequency, as shown in Fig. 14.24(b). This causes some frequency components to pass that should be suppressed, and both attenuates and amplifies others around the cutoff point; the domain over which this effect occurs decreases in size as a greater portion of the sinc signal is used, but the amplitude of the ringing does not decrease as long as the sinc is truncated. The approximation to a square wave in Fig.14.14(a) exhibits ringing in the spatial domain, which appears as little intensity "ripples" at discontinuities. A truncated sinc is obtained by multiplying the sinc by a pulse function. An alternative is to use a windowed sinc function that has been multiplied by a shape that, unlike the pulse, is not discontinuous, which allows the sinc to fall off smoothly. Blinn [BLIN89b] describes the derivation of one such filter.

One final problem is that the sinc, along with windowed filters derived from it, has parts that dip below zero, known as *negative lobes*. When a signal is convolved with a filter that has negative lobes, the resulting signal may itself dip below zero. If the signal represents intensity values, these values correspond to unrealizable negative intensities, and must therefore ultimately be clamped to zero.

Although windowed sinc functions are useful, they are relatively expensive because the window must be fairly wide; thus, a variety of other functions is often used instead. Filters

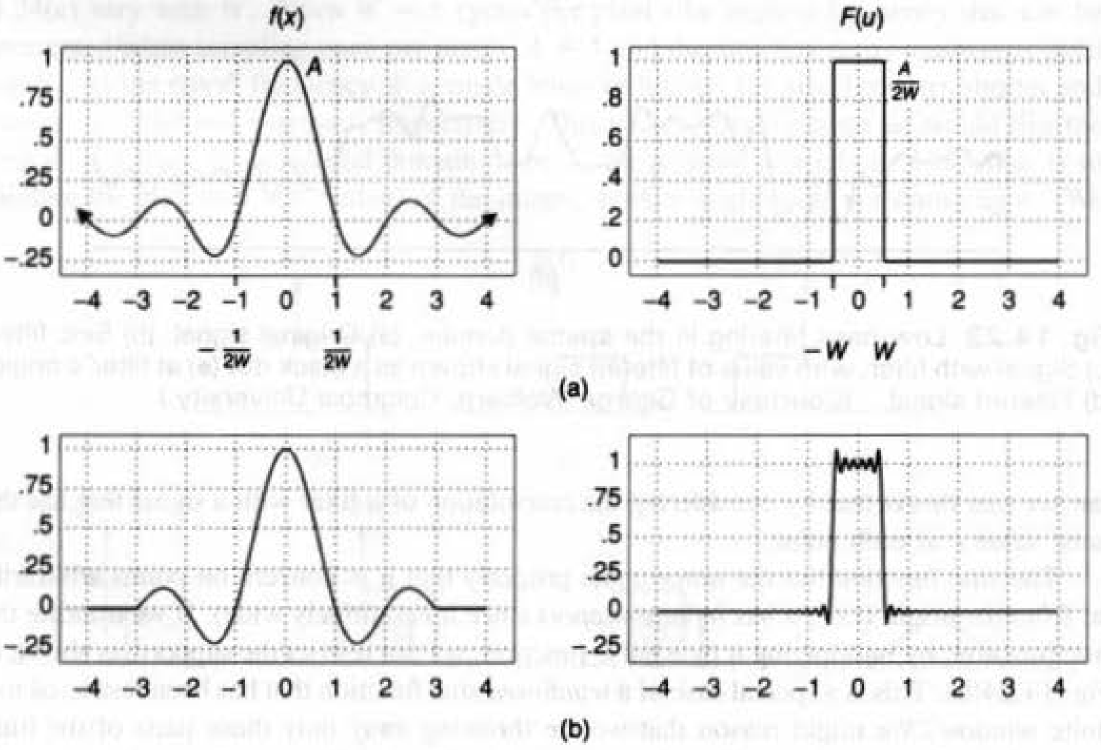


Fig. 14.24 (a) Sinc in spatial domain corresponds to pulse in frequency domain. (b) Truncated sinc in spatial domain corresponds to ringing pulse in frequency domain. (Courtesy of George Wolberg, Columbia University.)

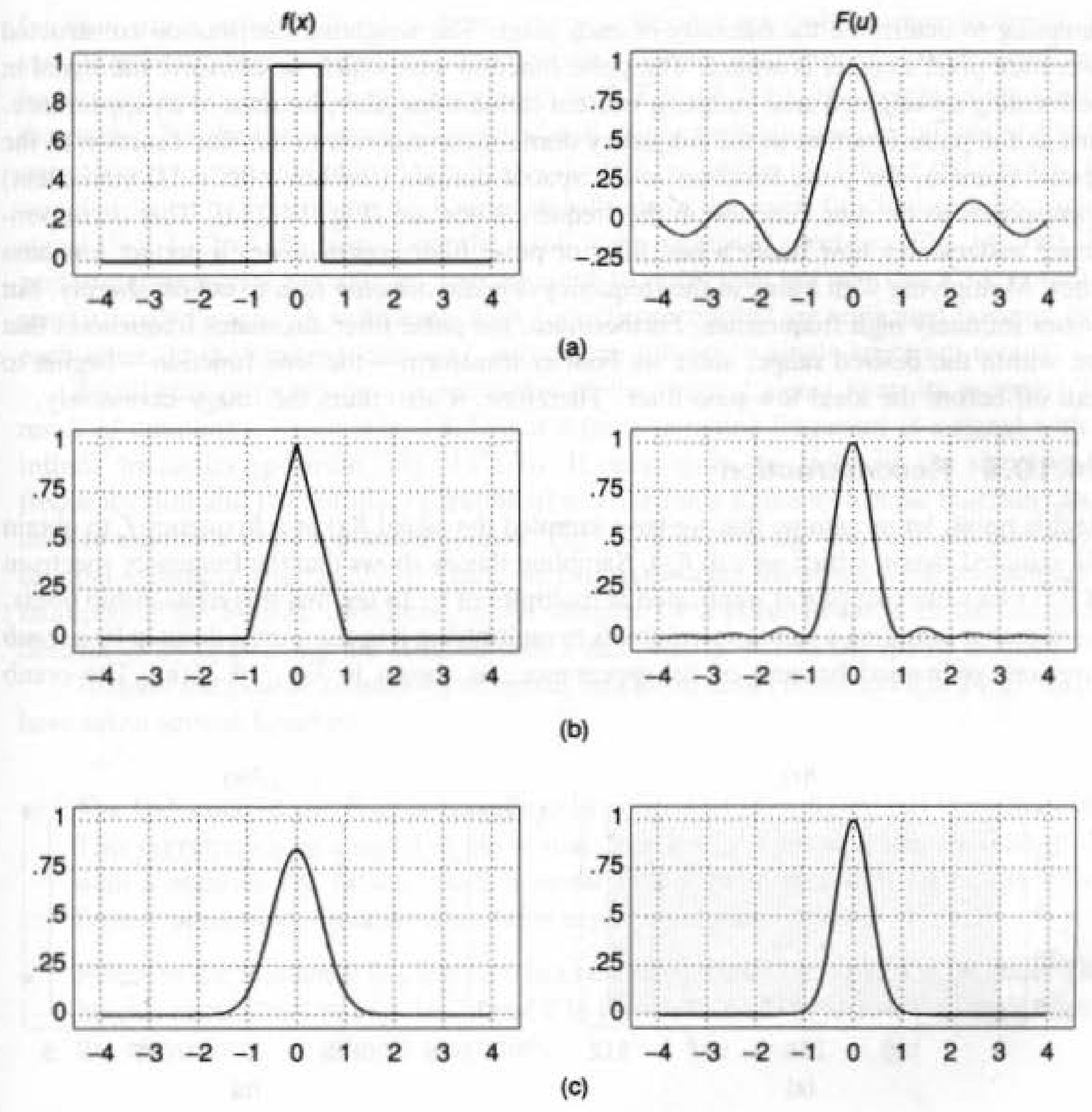


Fig. 14.25 Filters in spatial and frequency domains. (a) Pulse—sinc. (b) Triangle—sinc^2. (c) Gaussian—Gaussian. (Courtesy of George Wolberg, Columbia University.)

with finite support are known as finite impulse-response (FIR) filters, in contrast to the untruncated sinc filter, which is an infinite impulse-response (IIR) filter. Figure 14.25 shows several popular filters in both spatial and frequency domains.

We have now reduced the sampling problem to one of convolving the signal with a suitable filter and then sampling the filtered signal. Notice, however, that if our only use of the filtered signal is to sample it, then the work done filtering the signal anywhere but at the sample points is wasted. If we know in advance exactly where the samples will be taken, we need only to evaluate the convolution integral (Eq. 14.8) at each sample point to determine the sample's value. This is precisely how we perform the weighting operation in using area sampling to determine the intensity of each pixel. The weighting distribution constructed over each pixel's center is a filter. The pulse function with which we convolve the signal in performing unweighted area sampling is often called a *box filter*, because of its appearance. Just as the pulse function in the frequency domain corresponds to the sinc function in the spatial domain, the pulse function in the spatial domain (the box filter's 1D equivalent) corresponds to the sinc function in the frequency domain (Fig. 14.25a). This correspondence underscores how badly a box filter or pulse filter approximates a perfect low-pass filter. Multiplying with a sinc in the frequency domain not only fails to cut off sharply, but passes infinitely high frequencies. Furthermore, the pulse filter attenuates frequencies that are within the desired range, since its Fourier transform—the sinc function—begins to trail off before the ideal low-pass filter. Therefore, it also blurs the image excessively.

14.10.5 Reconstruction

At this point, let us assume that we have sampled the signal $f(x)$ at a frequency f_s to obtain the sampled signal, which we call $\hat{f}(x)$. Sampling theory shows that the frequency spectrum of $\hat{f}(x)$ looks like that of $f(x)$, replicated at multiples of f_s. To see that this relationship holds, we note that sampling a signal corresponds to multiplying it in the spatial domain by a *comb* function, so named because of its appearance, as shown in Fig. 14.26(a). The comb

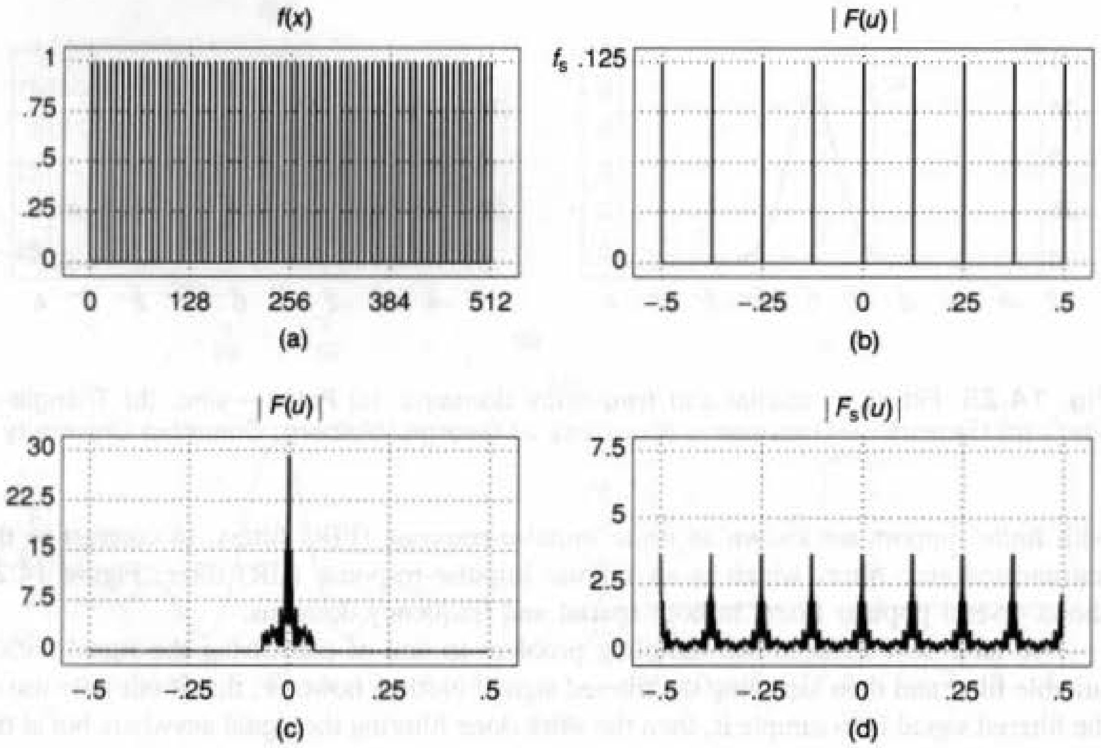


Fig. 14.26 (a) Comb function and (b) its Fourier transform. Convolving the comb's Fourier transform with (c) a signal's Fourier transform in the frequency domain yields (d) the replicated spectrum of the sampled signal. (Courtesy of George Wolberg, Columbia University.)

function has a value of 0 everywhere, except at regular intervals, corresponding to the sample points, where its value is 1. The (discrete) Fourier transform of a comb turns out to be just another comb with teeth at multiples of f_s (Fig. 14.26b). The height of the teeth in the comb's Fourier transform is f_s in cycles/pixel. Since multiplication in the spatial domain corresponds to convolution in the frequency domain, we obtain the Fourier transform of the sampled signal by convolving the Fourier transforms of the comb function and the original signal (Fig. 14.26c). By inspection, the result is the replicated spectrum (Fig. 14.26d). Try performing graphical convolution with the comb to verify this, but note that $F(u)$, not $|F(u)|$ must actually be used. A sufficiently high f_s yields spectra that are replicated far apart from each other. In the limiting case, as f_s approaches infinity, a single spectrum results.

Recall that *reconstruction* is recreation of the original signal from its samples. The result of sampling a signal (Fig. 14.27a) at a finite sampling frequency is a signal with an infinite frequency spectrum (Fig. 14.27b). If once again we deal with the signal in the frequency domain, the familiar operation of multiplying a signal by a pulse function can be used to eliminate these replicated spectra (Fig. 14.27c), leaving only a single copy of the original spectrum (Fig. 14.27d). Thus, we can reconstruct the signal from its samples by multiplying the Fourier transform of the samples by a pulse function in the frequency domain or by convolving the samples with a sinc with $A = 1$ in the spatial domain.

To make the Fourier transforms of signals and filters easier to see in Figs. 14.27–29, we have taken several liberties:

- The DC value of the Fourier transform in part (a) of each figure has been truncated. This corresponds to a signal in the spatial domain with the same shape as shown, but with a negative DC offset. (Such a signal cannot be displayed as an image without further processing, because it contains negative intensity values.)
- Filters in the frequency domain have not been drawn with the correct magnitude. Their heights should be 1 in Fig. 14.27 and 2 in Figs. 14.28–29 to restore the single copy of the spectrum to its original magnitude.

Figure 14.27(e) and (f) show the result of reconstructing the samples with a triangle filter (also known as a Bartlett filter). Convolving with this filter is equivalent to linearly interpolating the samples.

If the sampling frequency is too low, the replicated copies of the frequency spectra overlap, as in Fig. 14.28. In this case, the reconstruction process will fail to remove those parts of the replicated spectra that overlapped the original signal's spectrum. High-frequency components from the replicated spectra are mixed in with low-frequency components from the original spectrum, and therefore are treated like low frequencies during the reconstruction process. Note how an inadequate sampling rate causes aliasing by making a higher frequency appear identical to a lower one before and after reconstruction. There are two ways to resolve this problem. We may choose to sample at a high enough frequency, an approach that is sufficient only if the signal does not have an infinite spectrum. Alternatively, we may filter the signal before sampling to remove all components above $f_s/2$, as shown in Fig. 14.29.

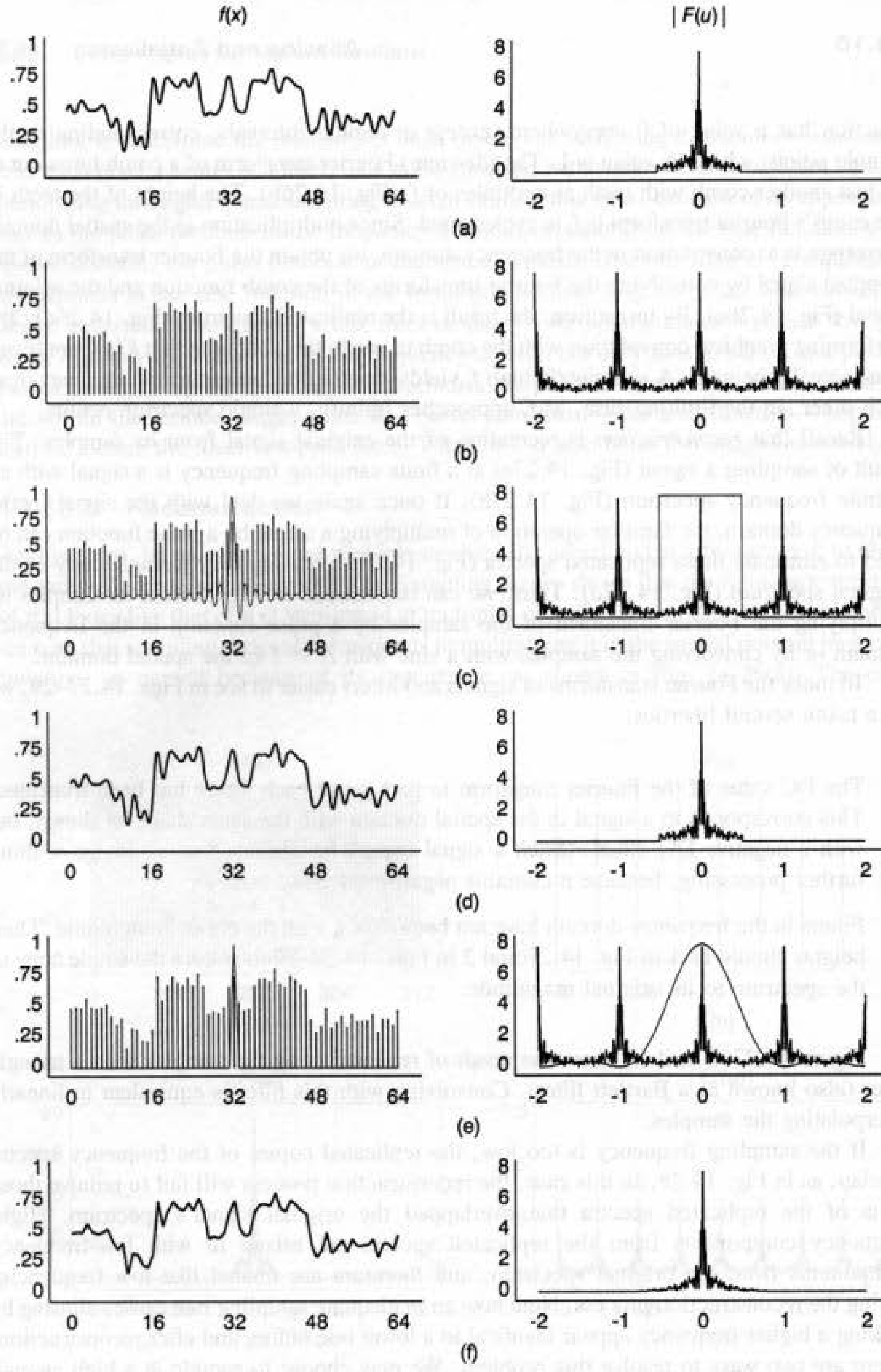


Fig. 14.27 Sampling and reconstruction: Adequate sampling rate. (a) Original signal. (b) Sampled signal. (c) Sampled signal ready to be reconstructed with sinc. (d) Signal reconstructed with sinc. (e) Sampled signal ready to be reconstructed with triangle. (f) Signal reconstructed with triangle. (Courtesy of George Wolberg, Columbia University.)

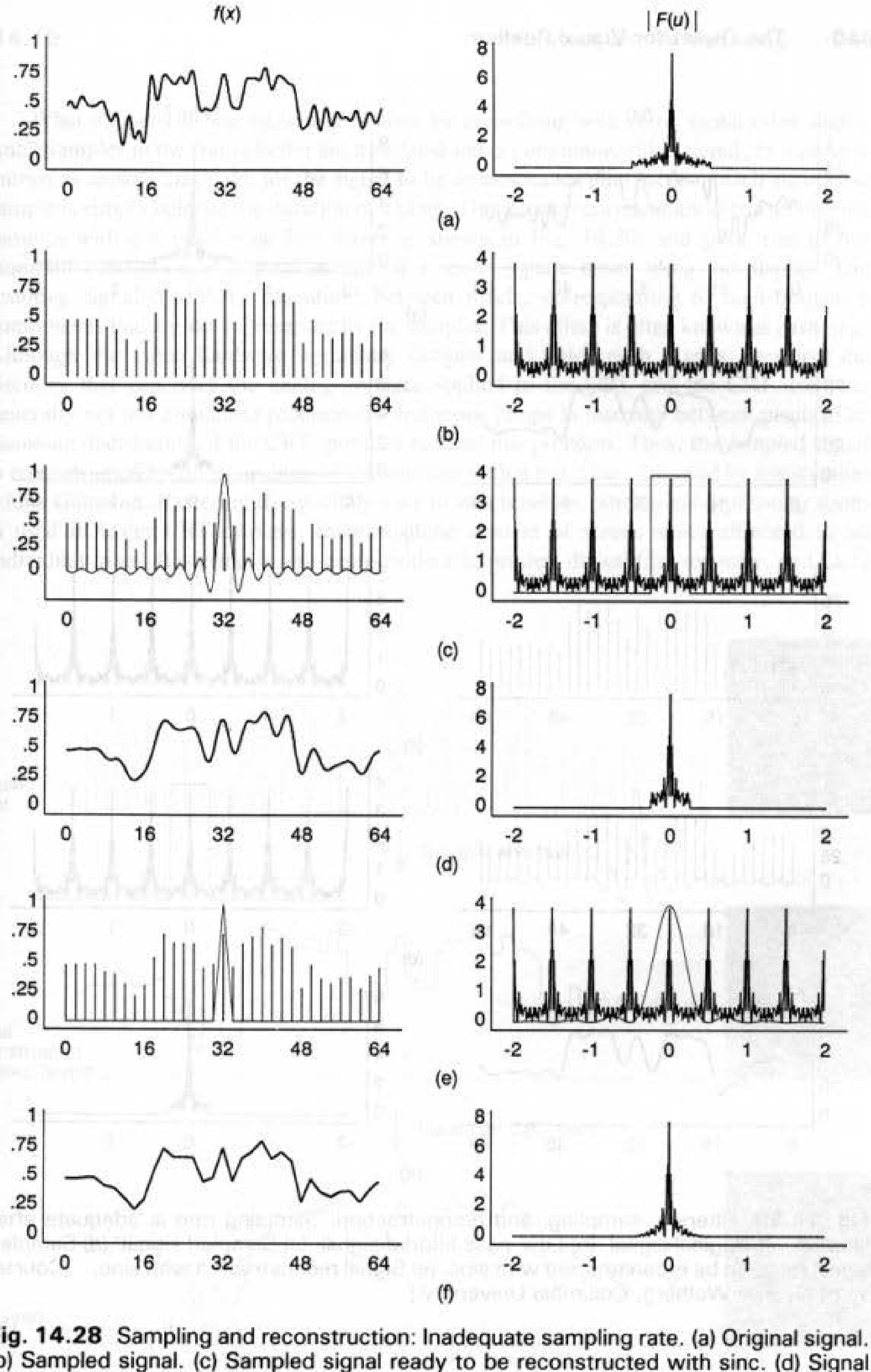


Fig. 14.28 Sampling and reconstruction: Inadequate sampling rate. (a) Original signal. (b) Sampled signal. (c) Sampled signal ready to be reconstructed with sinc. (d) Signal reconstructed with sinc. (e) Sampled signal ready to be reconstructed with triangle. (f) Signal reconstructed with triangle. (Courtesy of George Wolberg, Columbia University.)

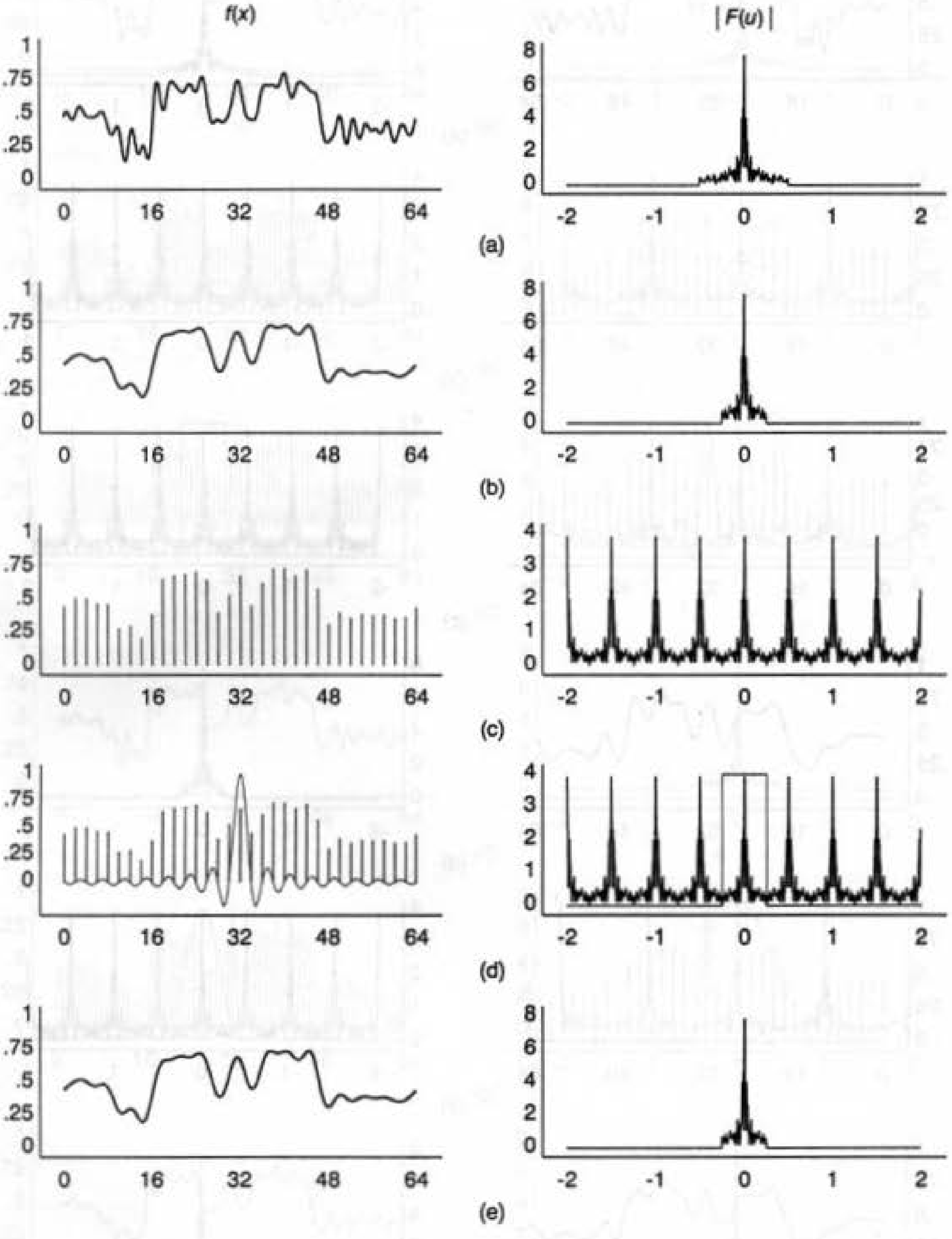


Fig. 14.29 Filtering, sampling, and reconstruction: Sampling rate is adequate after filtering. (a) Original signal. (b) Low-pass filtered signal. (c) Sampled signal. (d) Sampled signal ready to be reconstructed with sinc. (e) Signal reconstructed with sinc. (Courtesy of George Wolberg, Columbia University.)

What happens if reconstruction is done by convolving with some signal other than a sinc? Samples in the frame buffer are translated into a continuous video signal, by a process known as *sample and hold*: for the signal to be reconstructed, the value of each successive sample is simply held for the duration of a pixel. This process corresponds to convolving the samples with a 1-pixel-wide box filter, as shown in Fig. 14.30, and gives rise to our common conception of a pixel as one of a set of square boxes tiling the display. The resulting signal has sharp transitions between pixels, corresponding to high-frequency components that are not represented by the samples. This effect is often known as *rastering*. Although the video hardware nominally samples and holds each pixel's intensity, the circuitry that generates the analog voltages applied to the CRT and the CRT itself are generally not fast enough to produce discontinuous jumps in intensity between pixels. The Gaussian distribution of the CRT spot also reduces this problem. Thus, the sampled signal is reconstructed by the equivalent of convolution with a box filter, followed by convolution with a Gaussian. Rastering is especially easy to see, however, when pixel-replicating zoom is used in raster CRT displays, increasing the amount of screen space allocated to an individual pixel. Rastering is also more evident in printer, digital film recorder, and LCD

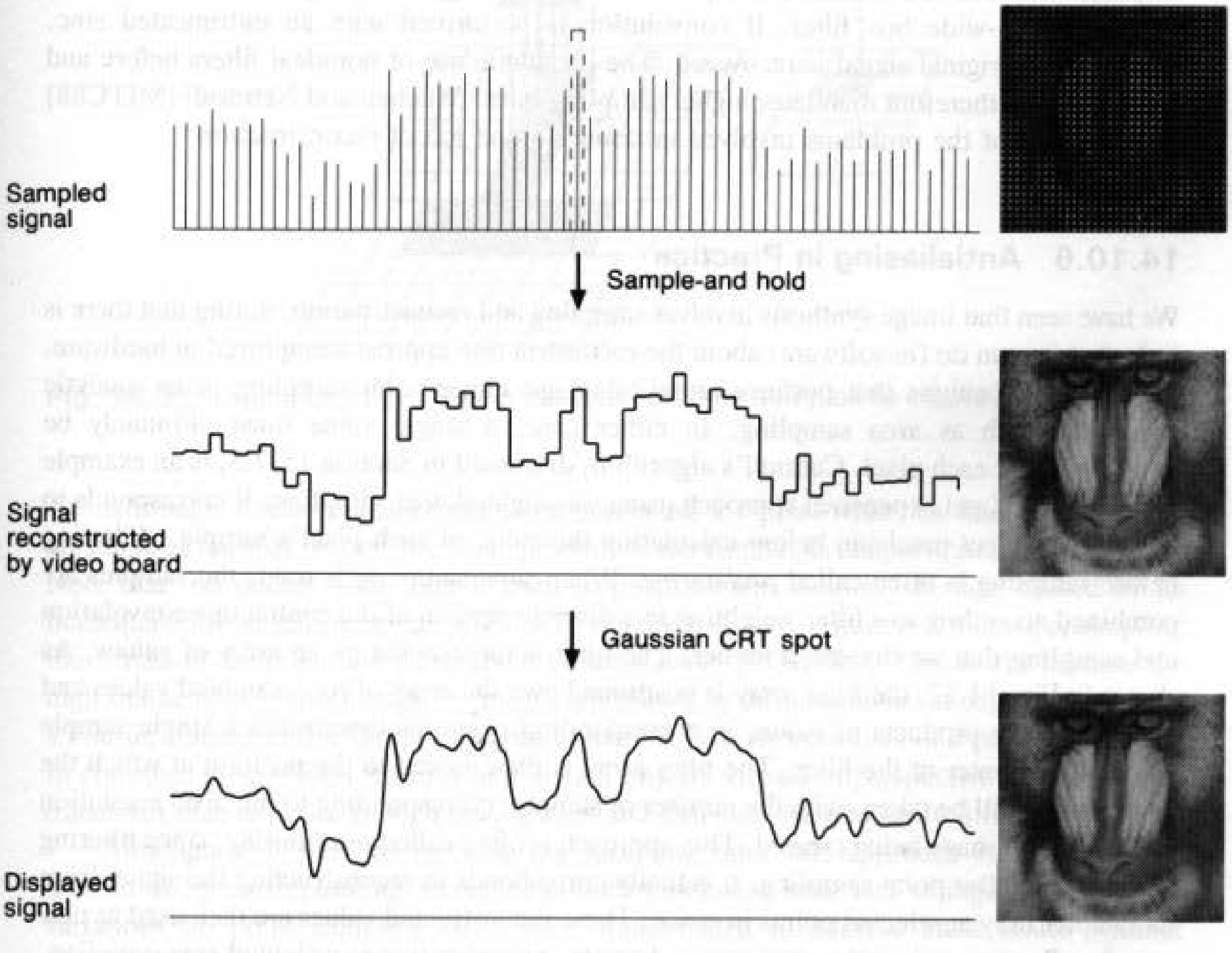


Fig. 14.30 Reconstruction by sample and hold and Gaussian CRT spot. (Courtesy of George Wolberg, Columbia University.)

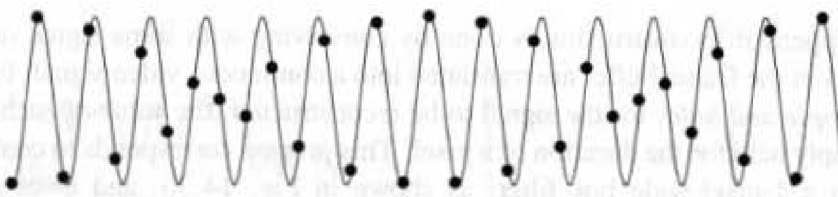


Fig. 14.31 A signal sampled at slightly over the Nyquist rate. (Courtesy of George Wolberg, Columbia University.)

technologies, in which pixel-to-pixel transitions are much sharper and produce relatively hard-edged square pixels of constant intensity.

We noted earlier that a signal must be sampled at a frequency greater than $2f_h$ to make perfect reconstruction possible. If the filter used to reconstruct the samples is not an untruncated sinc, as is always the case when displaying an image, then the sampling frequency must be even higher! Consider, for example, a sampling frequency slightly greater than $2f_h$. The resulting samples trace out the original signal modulated by (multiplied by) a low-frequency sine wave, as shown in Fig. 14.31. The low-frequency amplitude modulation remains, compounded by rastering, if the signal is reconstructed with a 1-pixel-wide box filter. If convolution is performed with an untruncated sinc, however, the original signal is recovered. The inevitable use of nonideal filters before and after sampling therefore mandates higher sampling rates. Mitchell and Netravali [MITC88] discuss some of the problems involved in doing a good job of reconstruction.

14.10.6 Antialiasing in Practice

We have seen that image synthesis involves sampling and reconstruction, noting that there is little that we can do (in software) about the reconstruction approach employed in hardware. Rendering algorithms that perform antialiasing use either point sampling or an analytic approach, such as area sampling. In either case, a single value must ultimately be determined for each pixel. Catmull's algorithm, discussed in Section 15.7.3, is an example of an analytic (and expensive) approach using unweighted area sampling. It corresponds to filtering at object precision before calculating the value of each pixel's sample. Filtering before sampling is often called *prefiltering*. When supersampling is used, the samples are combined according to a filter weighting in a discrete version of the continuous convolution and sampling that we discussed earlier. The filter is represented by an array of values. As shown in Fig. 14.32, the filter array is positioned over the array of supersampled values and the sum of the products of values in corresponding positions determines a single sample taken at the center of the filter. The filter array is then moved to the position at which the next sample will be taken, with the number of samples corresponding to the pixel resolution of the filtered image being created. This approach is often called *postfiltering*, since filtering is performed after point sampling. It actually corresponds to reconstructing the signal from its samples only at selected points in space. These reconstructed values are then used as new samples. Supersampling thus performs a discrete approximation to weighted area sampling.

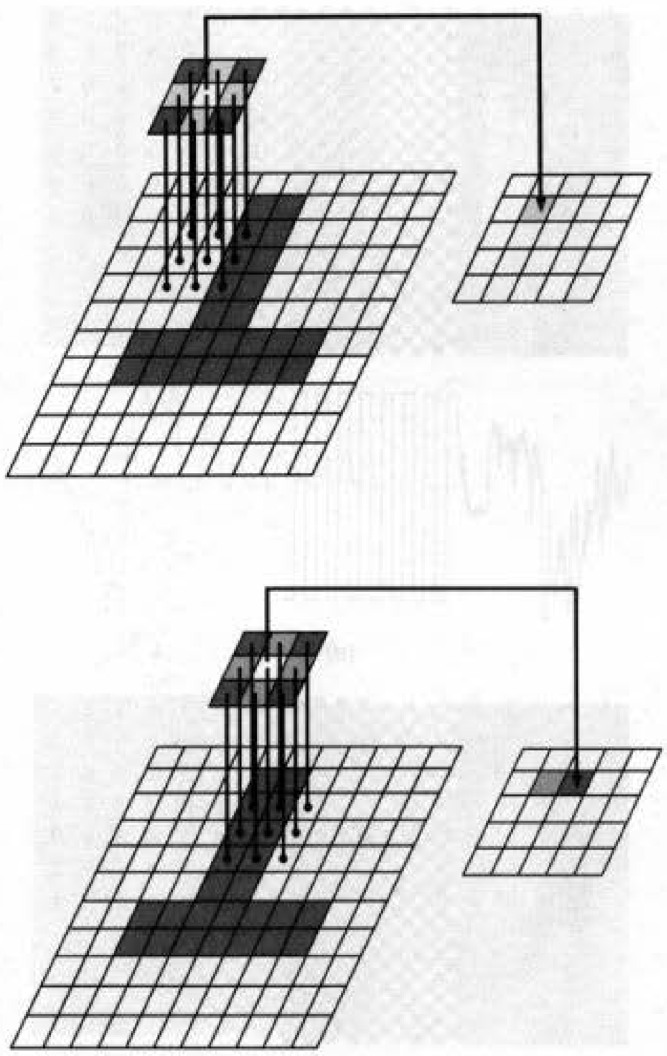
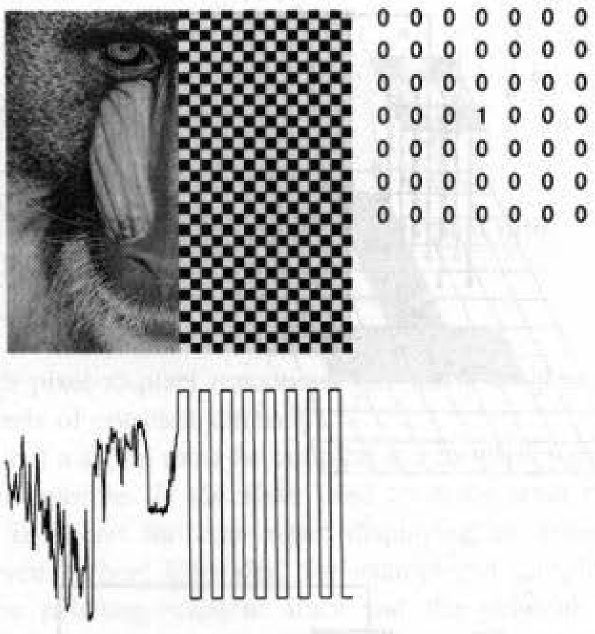


Fig. 14.32 Digital filtering. Filter is used to combine samples to create a new sample.

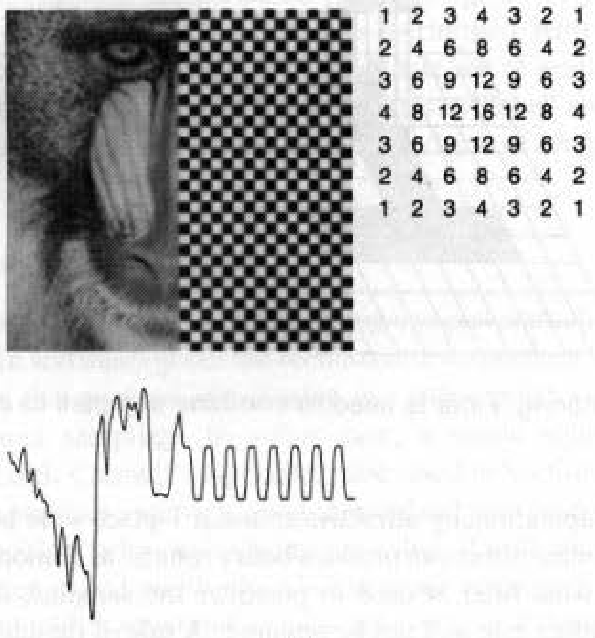
Although it is computationally attractive to use a 1-pixel-wide box filter that averages all subpixel samples, better filters can produce better results, as demonstrated in Fig. 14.33. Note that, no matter what filter is used to postfilter the samples, damage caused by an inadequate initial sampling rate will not be repaired. A rule of thumb is that supersampling four times in each of x and y often will be satisfactory [WHIT85]. This works because the high frequencies in most graphics images are caused by discontinuities at edges, which have a Fourier transform that tapers off rapidly (like the Fourier transform of a pulse—the sinc). In contrast, images with textures and distant objects viewed in perspective have a Fourier transform that is richer in high frequencies and that may be arbitrarily difficult to filter.

Although it is easy to increase the sampling rate, this approach is limited in its usefulness by corresponding increases in both processing time and storage. A number of variations on point sampling have been implemented to address these issues without sacrificing the conceptually simple mechanism of point sampling itself. In *adaptive*



0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

(a)

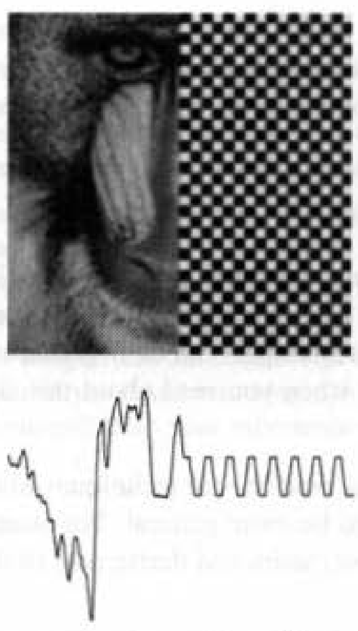


1	2	3	4	3	2	1
2	4	6	8	6	4	2
3	6	9	12	9	6	3
4	8	12	16	12	8	4
3	6	9	12	9	6	3
2	4	6	8	6	4	2
1	2	3	4	3	2	1

(c)

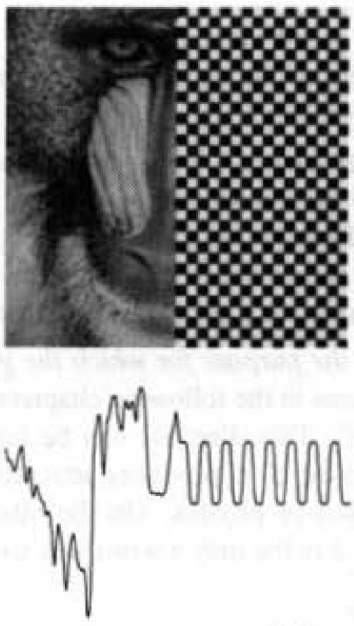
Fig. 14.33 *(Cont'd.)*

supersampling, an example of which is discussed in Section 15.10.4, the sampling rate is varied across the image, with additional samples taken when the system determines that they are needed. *Stochastic supersampling*, discussed in Section 16.12.4, places samples at stochastically determined positions, rather than in a regular grid. This approach produces



$$\begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{matrix}$$

(b)



$$\begin{matrix} 1 & 4 & 8 & 10 & 8 & 4 & 1 \\ 4 & 12 & 25 & 29 & 25 & 12 & 4 \\ 8 & 25 & 49 & 58 & 49 & 25 & 8 \\ 10 & 29 & 58 & 67 & 58 & 29 & 10 \\ 8 & 25 & 49 & 58 & 49 & 25 & 8 \\ 4 & 12 & 25 & 29 & 25 & 12 & 4 \\ 1 & 4 & 8 & 10 & 8 & 4 & 1 \end{matrix}$$

(d)

Fig. 14.33 Filtered images with intensity plot of middle scan line and filter kernel. (a) Original image. (b) Box filter. (c) Bartlett filter. (d) Gaussian filter. Images are 512 × 512, and filters are 7 × 7. Middle scan line is at bottom of a checkerboard row. Because 2D filter covers light and dark squares above and below scan line, amplitude of filtered checkerboard signal along middle scan line is greatly diminished. (Courtesy of George Wolberg, Columbia University.)

aliasing in the form of noise, which our visual system finds less irritating than the clearly defined frequency components of regular aliasing. These two approaches can be combined, allowing the determination of where to place new samples to be based on the statistical properties of those that have already been obtained.

When the original source signal is itself a sampled image, postfiltering followed by resampling may be used to create a new image that has been scaled, rotated, or distorted in a variety of ways. These *image transformations* are discussed in Chapter 17.

14.11 SUMMARY

In this chapter, we provided a high-level introduction to the techniques used to produce realistic images. We then examined the causes of and cures for aliasing. In the following chapters, we discuss in detail how these techniques can be implemented. There are five key questions that you should bear in mind when you read about the algorithms presented in later chapters:

1. *Is the algorithm general or special purpose?* Some techniques work best only in specific circumstances; others are designed to be more general. For example, some algorithms assume that all objects are convex polyhedra and derive part of their speed and relative simplicity from this assumption.
2. *Can antialiasing be incorporated?* Some algorithms may not accommodate antialiasing as easily as others do.
3. *What is the algorithm's space–time performance?* How is the algorithm affected by factors such as the size or complexity of the database, or the resolution at which the picture is rendered?
4. *How convincing are the effects generated?* For example, is refraction modeled correctly, does it look right only in certain special cases, or is it not modeled at all? Can additional effects, such as shadows or specular reflection, be added? How convincing will they be? Sacrificing the accuracy with which an effect is rendered may make possible significant improvements in a program's space or time requirements.
5. *Is the algorithm appropriate, given the purpose for which the picture is created?* The philosophy behind many of the pictures in the following chapters can be summed up by the credo, "If it looks good, do it!" This directive can be interpreted two ways. A simple or fast algorithm may be used if it produces attractive effects, even if no justification can be found in the laws of physics. On the other hand, a shockingly expensive algorithm may be used if it is the only known way to render certain effects.

EXERCISES

14.1 Suppose you had a graphics system that could draw any of the color plates referenced in this chapter in real time. Consider several application areas with which you are (or would like to be) familiar. For each area, list those effects that would be most useful, and those that would be least useful.

14.2 Show that you cannot infer the direction of rotation from orthographic projections of a monochrome, rotating, wireframe cube. Explain how additional techniques can help to make the direction of rotation clear without changing the projection.

14.3 Consider the pulse function $f(x) = 1$ for $-1 \le x \le 1$, and $f(x) = 0$ elsewhere. Show that the Fourier transform of $f(x)$ is a multiple of the sinc function. Hint: The Fourier transform of $f(x)$ can be computed as

$$F(u) = \int_{-1}^{+1} f(x)[\cos 2\pi ux - i\sin 2\pi ux]dx,$$

because the regions where $f(x) = 0$ contribute nothing to the integral, and $f(x) = 1$ in the remaining region. (Apply the inverse Fourier transform to your answer. You should get the original function.)

14.4 Prove that reconstructing a signal with a triangle filter of width 2 corresponds to linearly interpolating its samples. What happens if the filter is wider?

14.5 Write a program that allows you to convolve an image with a filter kernel. Use different filter kernels to create images of the same size from original images with 2, 4, and 8 times the number of pixels in x or y as the new images. You can obtain original images by saving the frame buffer generated by the graphics packages that you have been using. Do your filtered images look better than original images of the same resolution? Does your experience corroborate the rule of thumb mentioned in Section 14.10.6?

15 Visible-Surface Determination

Given a set of 3D objects and a viewing specification, we wish to determine which lines or surfaces of the objects are visible, either from the center of projection (for perspective projections) or along the direction of projection (for parallel projections), so that we can display only the visible lines or surfaces. This process is known as *visible-line* or *visible-surface determination*, or *hidden-line* or *hidden-surface elimination*. In visible-line determination, lines are assumed to be the edges of opaque surfaces that may obscure the edges of other surfaces farther from the viewer. Therefore, we shall refer to the general process as *visible-surface determination*.

Although the statement of this fundamental idea is simple, its implementation requires significant processing power, and consequently involves large amounts of computer time on conventional machines. These requirements have encouraged the development of numerous carefully structured visible-surface algorithms, many of which are described in this chapter. In addition, many special-purpose architectures have been designed to address the problem, some of which are discussed in Chapter 18. The need for this attention can be seen from an analysis of two fundamental approaches to the problem. In both cases, we can think of each object as comprising one or more polygons (or more complex surfaces).

The first approach determines which of *n* objects is visible at each pixel in the image. The pseudocode for this approach looks like this:

```
for (each pixel in the image) {
    determine the object closest to the viewer that is pierced by
        the projector through the pixel;
    draw the pixel in the appropriate color;
}
```

A straightforward, brute-force way of doing this for 1 pixel requires examining all n objects to determine which is closest to the viewer along the projector passing through the pixel. For p pixels, the effort is proportional to np, where p is over 1 million for a high-resolution display.

The second approach is to compare objects directly with each other, eliminating entire objects or portions of them that are not visible. Expressed in pseudocode, this becomes

```
for (each object in the world) {
    determine those parts of the object whose view is unobstructed
        by other parts of it or any other object;
    draw those parts in the appropriate color;
}
```

We can do this naively by comparing each of the n objects to itself and to the other objects, and discarding invisible portions. The computational effort here is proportional to n^2. Although this second approach might seem superior for $n < p$, its individual steps are typically more complex and time consuming, as we shall see, so it is often slower and more difficult to implement.

We shall refer to these prototypical approaches as *image-precision* and *object-precision* algorithms, respectively. Image-precision algorithms are typically performed at the resolution of the display device, and determine the visibility at each pixel. Object-precision algorithms are performed at the precision with which each object is defined, and determine the visibility of each object.[1] Since object-precision calculations are done without regard to a particular display resolution, they must be followed by a step in which the objects are actually displayed at the desired resolution. Only this final display step needs to be repeated if the size of the finished image is changed; for example, to cover a different number of pixels on a raster display. This is because the geometry of each visible object's projection is represented at the full object database resolution. In contrast, consider enlarging an image created by an image-precision algorithm. Since visible-surface calculations were performed at the original lower resolution, they must be done again if we wish to reveal further detail. Recalling our discussion of sampling in Chapter 14, we can think of object-precision algorithms as operating on the original continuous object data, and image-precision algorithms as operating on sampled data; thus, image-precision algorithms fall prey to aliasing in computing visibility, whereas object-precision algorithms do not.

Object-precision algorithms were first developed for vector graphics systems. On these devices, hidden-line removal was most naturally accomplished by turning the initial list of

[1]The terms *image space* and *object space*, popularized by Sutherland, Sproull, and Schumacker [SUTH74a], are often used to draw the same distinction. Unfortunately, these terms have also been used quite differently in computer graphics. For example, *image-space* has been used to refer to objects after perspective transformation [CATM75] or after projection onto the view plane [GILO78], but still at their original precision. To avoid confusion, we have opted for our slightly modified terms. We refer explicitly to an object's perspective transformation or projection, when appropriate, and reserve the terms *image precision* and *object precision* to indicate the precision with which computations are performed. For example, intersecting two objects' projections on the view plane is an object-precision operation if the precision of the original object definitions is maintained.

lines into one in which lines totally hidden by other surfaces were removed, and partially hidden lines were clipped to one or more visible line segments. All processing was performed at the precision of the original list and resulted in a list in the same format. In contrast, image-precision algorithms were first written for raster devices to take advantage of the relatively small number of pixels for which the visibility calculations had to be performed. This was an understandable partitioning. Vector displays had a large address space (4096 by 4096 even in early systems) and severe limits on the number of lines and objects that could be displayed. Raster displays, on the other hand, had a limited address space (256 by 256 in early systems) and the ability to display a potentially unbounded number of objects. Later algorithms often combine both object- and image-precision calculations, with object-precision calculations chosen for accuracy, and image-precision ones chosen for speed.

In this chapter, we first introduce a subset of the visible-surface problem, displaying single-valued functions of two variables. Next, we discuss a variety of ways to increase the efficiency of general visible-surface algorithms. Then, we present the major approaches to determining visible surfaces.

15.1 FUNCTIONS OF TWO VARIABLES

One of the most common uses of computer graphics has been to plot single-valued continuous functions of two variables, such as $y = f(x, z)$. These functions define surfaces like that shown in Fig. 15.1(a). They present an interesting special case of the hidden-surface problem, for which especially fast solutions are possible.

A function of x and z may be approximated by an m by n array Y of values. Each array element may be thought of as representing a height at one point in a regular grid of samples. We assume that each column of the array corresponds to a single x coordinate, and that each row of the array corresponds to a single z coordinate. In other words, the rectangular array is aligned with the x and z axes. The indices of an array element and the element's value together specify the coordinates of a 3D point. A wireframe drawing of the surface may be constructed as a piecewise linear approximation by drawing one set of polylines through the points defined by each row of the array Y (polylines of constant z) and an orthogonal set of polylines through the points defined by each column (polylines of constant x). A

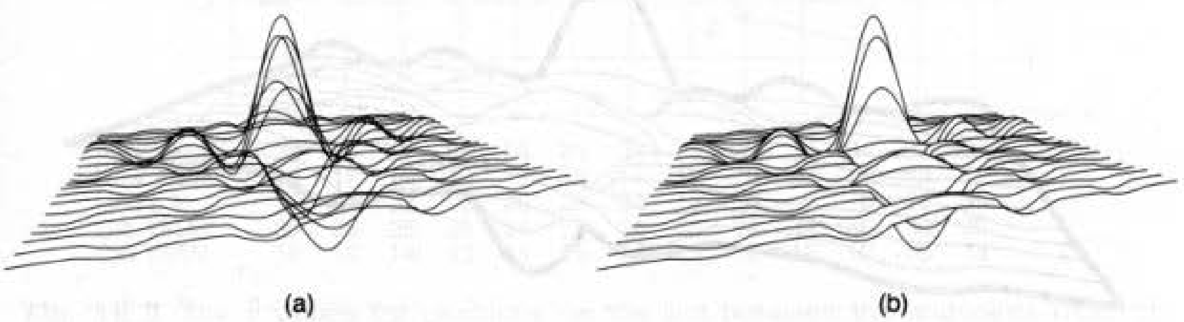

(a) (b)

Fig. 15.1 A single-valued function of two variables: (a) without and (b) with hidden lines removed. (By Michael Hatzitheodorou, Columbia University.)

hidden-line algorithm must suppress all parts of the lines that would be obscured from the viewpoint by other parts of the surface, as shown in Fig. 15.1(b).

To see how we might accomplish this, we first consider the problem of plotting only polylines of constant z, assuming that the closest polyline to the viewpoint is an edge of the surface. An efficient class of solutions to this problem is based on the recognition that each polyline of constant z lies in a separate, parallel plane of constant z [WILL72; WRIG73; BUTL79]. Since none of these planes (and hence none of the polylines) can intersect any of the others, each polyline cannot be obscured by any polyline of constant z farther from the viewpoint. Therefore, we will draw polylines of constant z in order of increasing distance from the viewpoint. This establishes a *front-to-back* order. Drawing each polyline of constant z correctly requires only that we not draw those parts of the polyline that are obscured by parts of the surface already drawn.

Consider the "silhouette boundary" of the polylines drawn thus far on the view plane, shown as thick lines in Fig. 15.2. When a new polyline is drawn, it should be visible only where its projection rises above the top or dips below the bottom of the old silhouette. Since each new polyline has a constant z that is farther than that of any of the preceding polylines, it cannot pierce any part of the surface already drawn. Therefore, to determine what parts are to be drawn, we need only to compare the current polyline's projected y values with those of the corresponding part of the surface's silhouette computed thus far. When only enough information is encoded to represent a minimum and maximum silhouette y for each x, the algorithm is known as an *horizon line algorithm*.

One way to represent this silhouette, implemented by Wright [WRIG73], uses two 1D arrays, *YMIN* and *YMAX*, to hold the minimum and maximum projected y values for a finite set of projected x values. These are image-precision data structures because they have a finite number of entries. *YMIN* and *YMAX* are initialized with y values that are, respectively, above and below all the projected y values of the surface. When a new polyline is drawn, the projected y values of each pair of adjacent vertices are compared with the values at the corresponding locations in the silhouette arrays. As shown in Fig. 15.3, a vertex whose value is above that in the corresponding position in *YMAX* (A, B, G) or below that in *YMIN* (E, F) is visible; otherwise, it is invisible (C, D). If both vertices are invisible, then the line segment is wholly invisible (CD) and the silhouette arrays remain unchanged.

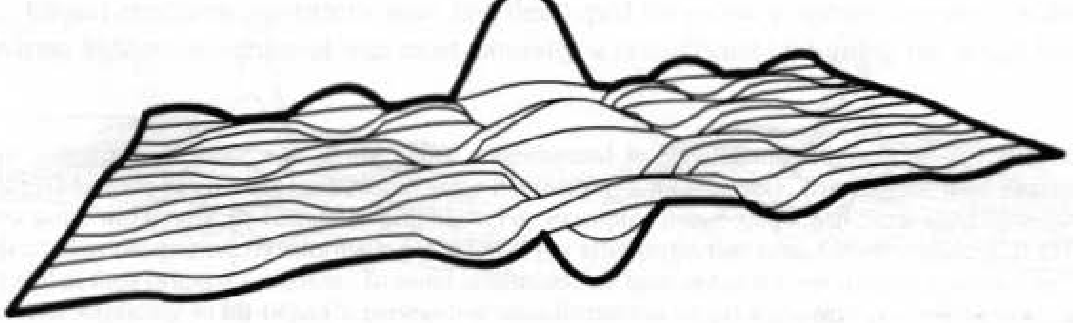
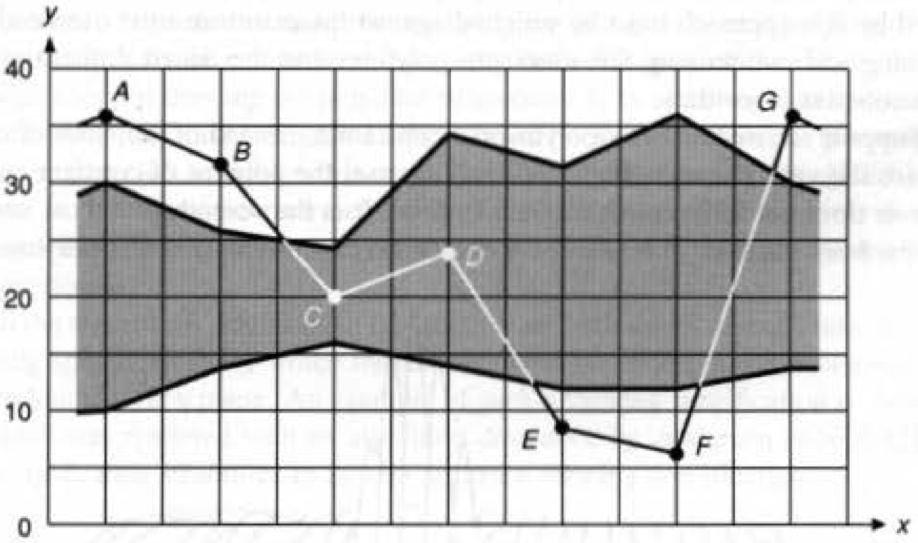


Fig. 15.2 Silhouette of lines drawn. (By Michael Hatzitheodorou, Columbia University.)

If both vertices are visible with regard to the same silhouette array (*AB*, *EF*), then the line segment is wholly visible and should be drawn in its entirety, and that silhouette array should be updated. The x coordinates of two adjacent vertices in a polyline of constant z usually map to nonadjacent locations in the silhouette arrays. In this situation, values of y to insert in the intervening silhouette array locations can be obtained by linearly interpolating between the projected y values of the two adjacent elements of Y.

Finally, we must consider the case of a partially visible line, in which both vertices are not visible with regard to the same silhouette array. Although this typically means that one of the vertices is visible and the other is invisible (*BC*, *DE*), it may be the case that both vertices are visible, one above *YMAX* and the other below *YMIN* (*FG*). Interpolated y values can be compared with those at the intervening locations in the silhouette arrays to determine the point(s) of intersection. The line should not be visible at those places where an interpolated y value falls inside the silhouette. Only the visible parts of the line segment outside the silhouette should be drawn, and the silhouette array should be updated, as shown in Fig. 15.3. When the two adjacent silhouette elements are found between which the line changes visibility, the line can be intersected with the line defined by the x and y values of those elements to determine an endpoint for a line-drawing algorithm.

Unfortunately, the image-precision *YMIN* and *YMAX* silhouette arrays make this algorithm prone to aliasing problems. If the line-drawing primitive used has higher resolution than do the silhouette arrays, aliasing can manifest itself as hidden segments that are drawn or as visible segments that are not. For example, Fig. 15.4 shows three polylines being drawn in front-to-back order. The two polylines of parts (a) and (b) form a gap when



New *YMAX*	36	34	32	26	24	29	34	33	32	34	36	33	36
New *YMIN*	10	12	14	15	16	15	14	13	8	7	6	13	14
Line's *Y*	**36**	34	**32**	26	**20**	22	**24**	16	**8**	7	**6**	21	**36**
Old *YMAX*	30	28	26	25	24	29	34	33	32	34	36	33	30
Old *YMIN*	10	12	14	15	16	15	14	13	12	12	12	13	14

Fig. 15.3 The y values for positions on the line between the endpoints must be interpolated and compared with those in the silhouette arrays. Endpoint values are shown in boldface type.

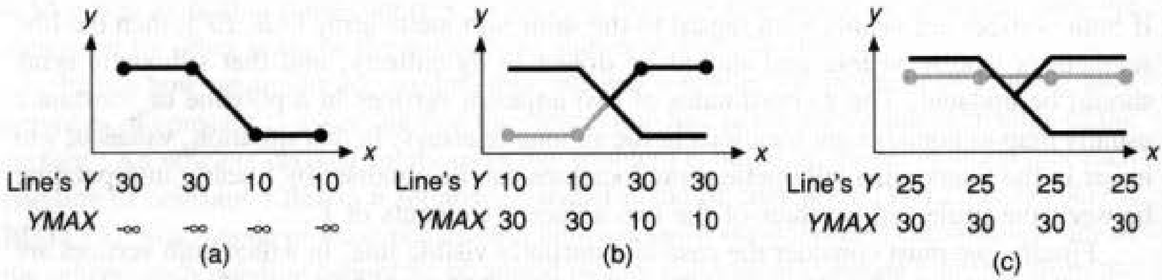


Fig. 15.4 Aliasing problem arising from the use of image-precision silhouette arrays.

they cross. The second line can be correctly hidden by the first by interpolating the values in *YMAX* to determine the intersection of the two lines. After the second line has been drawn, however, both values in *YMAX* are the same, so the third line, drawn in part (c), incorrectly remains hidden when it passes over their intersection. Using higher-resolution silhouette arrays can reduce such problems at the expense of increased processing time.

An alternative to image-precision *YMIN* and *YMAX* arrays is to use object-precision data structures [WILL72]. The *YMIN* and *YMAX* arrays can be replaced by two object-precision polylines representing the silhouette boundaries. As each segment of a polyline of constant z is inspected, only those parts projecting above the *YMAX* polyline or below the *YMIN* polyline are drawn. The projected lines representing these visible parts are linked into the silhouette polyline beginning at the point at which they intersect it, replacing the subsequent parts of the silhouette polyline until the next intersection. The accuracy gained by this approach must be weighed against the extra run-time overhead incurred by searching and maintaining the silhouette polylines and the added difficulties of coding a more complex algorithm.

Suppose we wish to draw polylines of constant x, instead of polylines of constant z, to produce the view shown in Fig. 15.5. In this case, the polyline of constant x closest to the observer does not form an edge of the surface. It is the seventh (the most nearly vertical) polyline from the left. To render the surface correctly, we must render polylines to the right

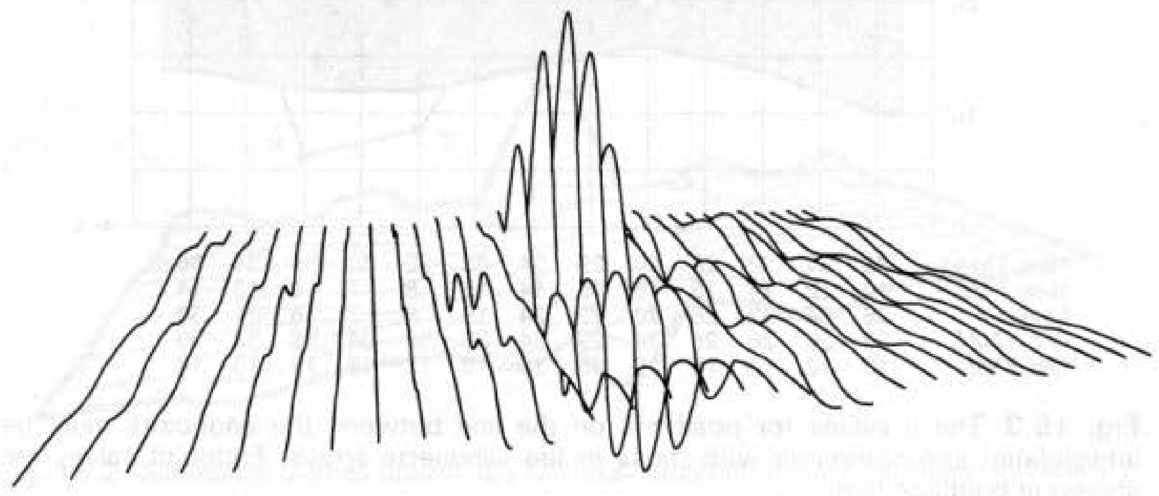


Fig. 15.5 Surface drawn using polylines of constant x, instead of polylines of constant z. (By Michael Hatzitheodorou, Columbia University.)

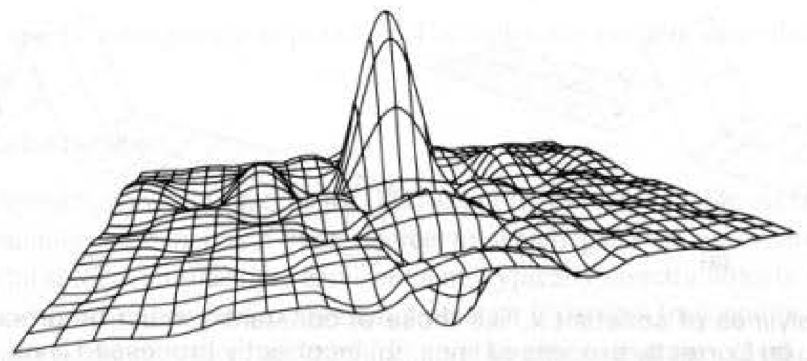


Fig. 15.6 Surface of Fig. 15.5 with lines of constant *x* and *z*. (By Michael Hatzitheodorou, Columbia University.)

of the closest one from left to right, and those to the left of the closest one from right to left. In both cases, polylines are rendered in front-to-back order relative to the observer.

The image-precision algorithm can be easily extended to plot lines of constant x as well as of constant z, as shown in Fig. 15.6. Although it is tempting to assume that superimposing the correctly plotted polylines of constant x and polylines of constant z would work, this does not allow lines from each set to hide those in the other, as shown in Fig. 15.7. Wright [WRIG73] points out that the correct solution can be obtained by interleaving the processing of the two sets of polylines. The set of those polylines that are most nearly parallel to the view plane (e.g., those of constant z) is processed in the same order as before. After each polyline of constant z is processed, the segments of each polyline of constant x between the just-processed polyline of z and the next polyline of z are drawn. The line segments of x must be drawn using the same copy of the silhouette data structure as was used for drawing the polylines of constant z. In addition, they too must be processed in front-to-back order. Figure 15.8(a) shows lines of constant x as processed in the correct order, from left to right in this case. The lines of constant x in Fig. 15.8(b) have been processed in the opposite, incorrect order, from right to left. The incorrect drawing order causes problems, because each successive line is shadowed in the *YMAX* array by the lines drawn previously.

Although the algorithms described in this section are both useful and efficient, they fail for any viewing specification for which the silhouette of the object is not a function of x when projected on the view plane. An example of such a viewing specification is shown in Fig. 15.9, which was rendered with an algorithm developed by Anderson [ANDE82] that uses more complex data structures to handle arbitrary viewing specifications.

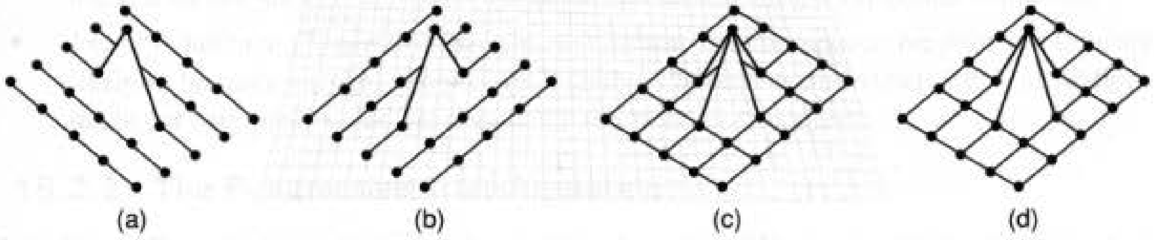


Fig. 15.7 (a) Lines of constant *z*. (b) Lines of constant *x*. (c) Superposition of parts (a) and (b). (d) The correct solution. (Based on [WRIG73].)

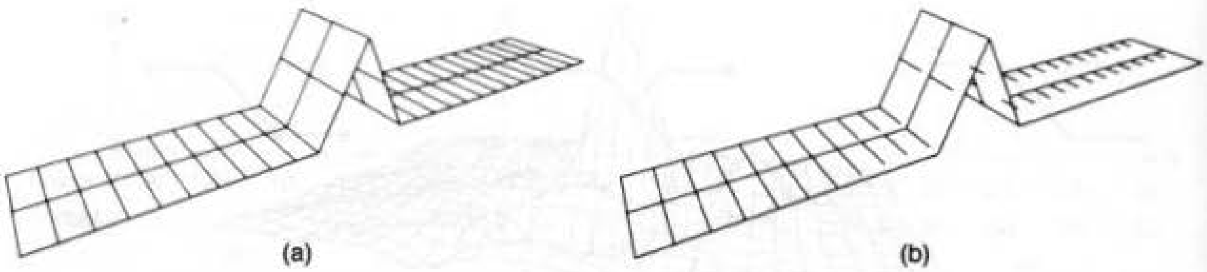

(a) (b)

Fig. 15.8 Polylines of constant *x*, like those of constant *z*, must be processed in the correct order. (a) Correctly processed lines. (b) Incorrectly processed lines. (By Michael Hatzitheodorou, Columbia University.)

15.2 TECHNIQUES FOR EFFICIENT VISIBLE-SURFACE ALGORITHMS

As we have just seen, a restricted version of the visible-line problem for functions of two variables can be solved efficiently by using clever data structures. What can be done for the general problem of visible-surface determination? The simple formulations of prototypical image-precision and object-precision algorithms given at the beginning of this chapter require a number of potentially costly operations. These include determining for a projector and an object, or for two objects' projections, whether or not they intersect and where they intersect. Then, for each set of intersections, it is necessary to compute the object that is closest to the viewer and therefore visible. To minimize the time that it takes to create a picture, we must organize visible-surface algorithms so that costly operations are performed

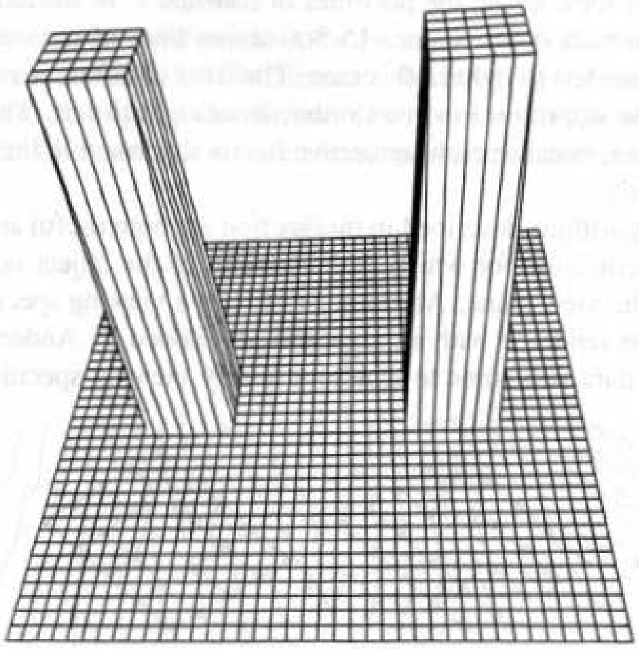


Fig. 15.9 A projection not handled by the algorithm discussed here. (Courtesy of David P. Anderson, University of California, Berkeley.)

as efficiently and as infrequently as possible. The following sections describe some general ways to do this.

15.2.1 Coherence

Sutherland, Sproull, and Schumacker [SUTH74a] point out how visible-surface algorithms can take advantage of *coherence*—the degree to which parts of an environment or its projection exhibit local similarities. Environments typically contain objects whose properties vary smoothly from one part to another. In fact, it is the less frequent discontinuities in properties (such as depth, color, and texture) and the effects that they produce in pictures, that let us distinguish between objects. We exploit coherence when we reuse calculations made for one part of the environment or picture for other nearby parts, either without changes or with incremental changes that are more efficient to make than recalculating the information from scratch. Many different kinds of coherence have been identified [SUTH74a], which we list here and refer to later:

- *Object coherence*. If one object is entirely separate from another, comparisons may need to be done only between the two objects, and not between their component faces or edges. For example, if all parts of object A are farther from the viewer than are all parts of object B, none of A's faces need be compared with B's faces to determine whether they obscure B's faces.
- *Face coherence*. Surface properties typically vary smoothly across a face, allowing computations for one part of a face to be modified incrementally to apply to adjacent parts. In some models, faces can be guaranteed not to interpenetrate.
- *Edge coherence*. An edge may change visibility only where it crosses behind a visible edge or penetrates a visible face.
- *Implied edge coherence*. If one planar face penetrates another, their line of intersection (the implied edge) can be determined from two points of intersection.
- *Scan-line coherence*. The set of visible object spans determined for one scan line of an image typically differs little from the set on the previous line.
- *Area coherence*. A group of adjacent pixels is often covered by the same visible face. A special case of area coherence is *span coherence*, which refers to a face's visibility over a span of adjacent pixels on a scan line.
- *Depth coherence*. Adjacent parts of the same surface are typically close in depth, whereas different surfaces at the same screen location are typically separated farther in depth. Once the depth at one point of the surface is calculated, the depth of points on the rest of the surface can often be determined by a simple difference equation.
- *Frame coherence*. Pictures of the same environment at two successive points in time are likely to be quite similar, despite small changes in objects and viewpoint. Calculations made for one picture can be reused for the next in a sequence.

15.2.2 The Perspective Transformation

Visible-surface determination clearly must be done in a 3D space prior to the projection into 2D that destroys the depth information needed for depth comparisons. Regardless of the kind of projection chosen, the basic depth comparison at a point can be typically reduced to the following question: Given points $P_1 = (x_1, y_1, z_1)$ and $P_2 = (x_2, y_2, z_2)$, does either point obscure the other? This question is the same: Are P_1 and P_2 on the same projector (see Fig. 15.10)? If the answer is yes, z_1 and z_2 are compared to determine which point is closer to the viewer. If the answer is no, then neither point can obscure the other.

Depth comparisons are typically done after the normalizing transformation (Chapter 6) has been applied, so that projectors are parallel to the z axis in parallel projections or emanate from the origin in perspective projections. For a parallel projection, the points are on the same projector if $x_1 = x_2$ and $y_1 = y_2$. For a perspective projection, we must unfortunately perform four divisions to determine whether $x_1 / z_1 = x_2 / z_2$ and $y_1 / z_1 = y_2 / z_2$, in which case the points are on the same projector, as shown in Fig. 15.10. Moreover, if P_1 is later compared against some P_3, two more divisions are required.

Unnecessary divisions can be avoided by first transforming a 3D object into the 3D screen-coordinate system, so that the parallel projection of the transformed object is the same as the perspective projection of the untransformed object. Then the test for one point obscuring another is the same as for parallel projections. This perspective transformation distorts the objects and moves the center of projection to infinity on the positive z axis, making the projectors parallel (see Fig. 6.56). Figure 15.11 shows the effect of this transformation on the perspective view volume; Fig. 15.12 shows how a cube is distorted by the transformation.

The essence of such a transformation is that it preserves relative depth, straight lines, and planes, and at the same time performs the perspective foreshortening. As discussed in Chapter 6, the division that accomplishes the foreshortening is done just once per point, rather than each time two points are compared. The matrix from Eq. (6.48)

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{1}{1 + z_{\min}} & \dfrac{-z_{\min}}{1 + z_{\min}} \\ 0 & 0 & -1 & 0 \end{bmatrix}, \quad z_{\min} \neq -1 \tag{15.1}$$

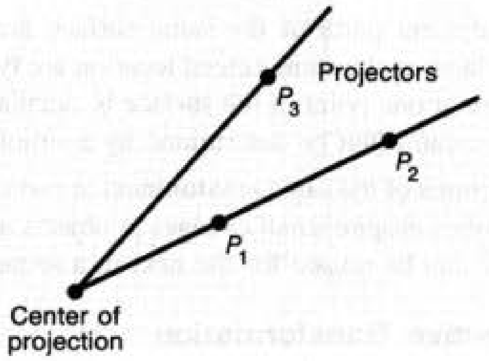


Fig. 15.10 If two points P_1 and P_2 are on the same projector, then the closer one obscures the other; otherwise, it does not (e.g., P_1 does not obscure P_3).

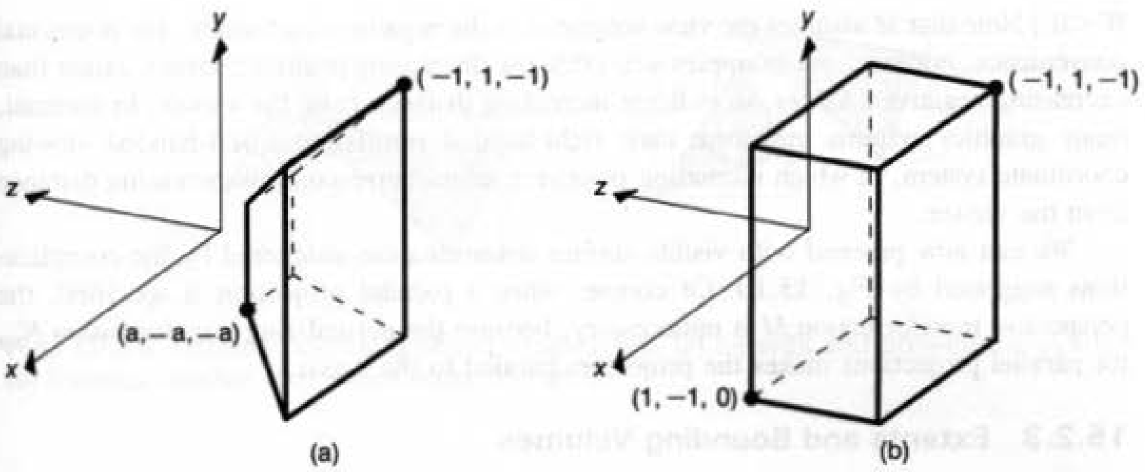


Fig. 15.11 The normalized perspective view volume (a) before and (b) after perspective transformation.

transforms the normalized perspective view volume into the rectangular parallelepiped bounded by

$$-1 \le x \le 1, \qquad -1 \le y \le 1, \qquad -1 \le z \le 0. \tag{15.2}$$

Clipping can be done against the normalized truncated-pyramid view volume before M is applied, but then the clipped results must be multiplied by M. A more attractive alternative is to incorporate M into the perspective normalizing transformation N_{per} from Chapter 6, so that just a single matrix multiplication is needed, and then to clip in homogeneous coordinates prior to the division. If we call the results of that multiplication (X, Y, Z, W), then, for $W > 0$, the clipping limits become

$$-W \le X \le W, \qquad -W \le Y \le W, \qquad -W \le Z \le 0. \tag{15.3}$$

These limits are derived from Eq. (15.2) by replacing x, y, and z by X/W, Y/W, and Z/W, respectively, to reflect the fact that x, y, and z in Eq. (15.2) result from division by W. After clipping, we divide by W to obtain (x_p, y_p, z_p).(See Section 6.5.4. for what to do when

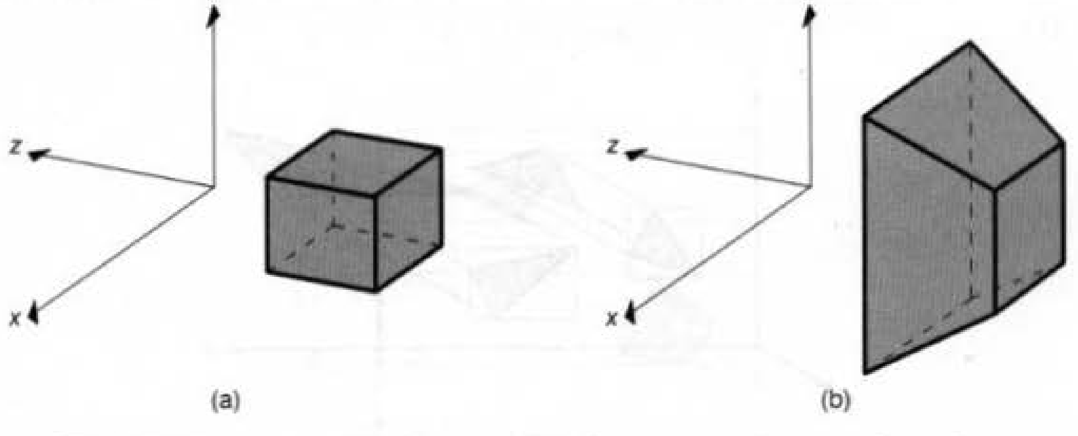


Fig. 15.12 A cube (a) before and (b) after perspective transformation.

$W < 0$.) Note that M assumes the view volume is in the negative z half-space. For notational convenience, however, our examples will often use decreasing positive z values, rather than decreasing negative z values, to indicate increasing distance from the viewer. In contrast, many graphics systems transform their right-handed world into a left-handed viewing coordinate system, in which increasing positive z values correspond to increasing distance from the viewer.

We can now proceed with visible-surface determination unfettered by the complications suggested by Fig. 15.10. Of course, when a parallel projection is specified, the perspective transformation M is unnecessary, because the normalizing transformation N_{par} for parallel projections makes the projectors parallel to the z axis.

15.2.3 Extents and Bounding Volumes

Screen extents, introduced in Chapter 3 as a way to avoid unnecessary clipping, are also commonly used to avoid unnecessary comparisons between objects or their projections. Figure 15.13 shows two objects (3D polygons, in this case), their projections, and the upright rectangular screen extents surrounding the projections. The objects are assumed to have been transformed by the perspective transformation matrix M of Section 15.2.2. Therefore, for polygons, orthographic projection onto the (x, y) plane is done trivially by ignoring each vertex's z coordinate. In Fig. 15.13, the extents do not overlap, so the projections do not need to be tested for overlap with one another. If the extents overlap, one of two cases occurs, as shown in Fig. 15.14: either the projections also overlap, as in part (a), or they do not, as in part (b). In both cases, more comparisons must be performed to determine whether the projections overlap. In part (b), the comparisons will establish that the two projections really do not intersect; in a sense, the overlap of the extents was a false alarm. Extent testing thus provides a service similar to that of trivial reject testing in clipping.

Rectangular-extent testing is also known as *bounding-box* testing. Extents can be used as in Chapter 7 to surround the objects themselves rather than their projections: in this case, the extents become solids and are also known as *bounding volumes*. Alternatively, extents can be used to bound a single dimension, in order to determine, say, whether or not two objects overlap in z. Figure 15.15 shows the use of extents in such a case; here, an extent is the infinite volume bounded by the minimum and maximum z values for each object. There

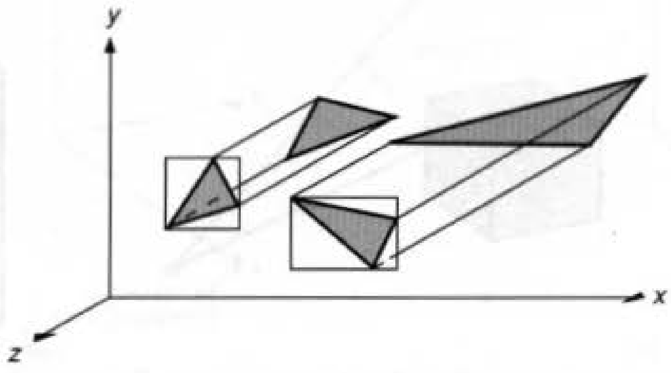


Fig. 15.13 Two objects, their projections onto the (x, y) plane, and the extents surrounding the projections.

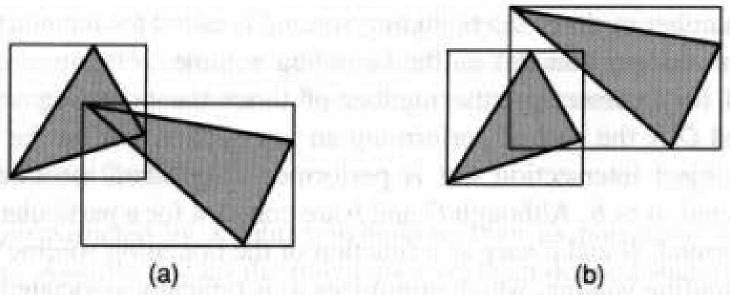


Fig. 15.14 Extents bounding object projections. (a) Extents and projections overlap. (b) Extents overlap, but projections do not.

is no overlap in z if

$$z_{\max2} < z_{\min1} \quad or \quad z_{\max1} < z_{\min2}. \tag{15.4}$$

Comparing against minimum and maximum bounds in one or more dimensions is also known as *minmax* testing. When comparing minmax extents, the most complicated part of the job is finding the extent itself. For polygons (or for other objects that are wholly contained within the convex hull of a set of defining points), an extent may be computed by iterating through the list of point coordinates and recording the largest and smallest values for each coordinate.

Extents and bounding volumes are used not only to compare two objects or their projections with each other, but also to determine whether or not a projector intersects an object. This involves computing the intersection of a point with a 2D projection or a vector with a 3D object, as described in Section 15.10.

Although we have discussed only minmax extents so far, other bounding volumes are possible. What is the best bounding volume to use? Not surprisingly, the answer depends on both the expense of performing tests on the bounding volume itself and on how well the volume protects the enclosed object from tests that do not yield an intersection. Weghorst, Hooper, and Greenberg [WEGH84] treat bounding-volume selection as a matter of minimizing the total cost function T of the intersection test for an object. This may be expressed as

$$T = bB + oO, \tag{15.5}$$

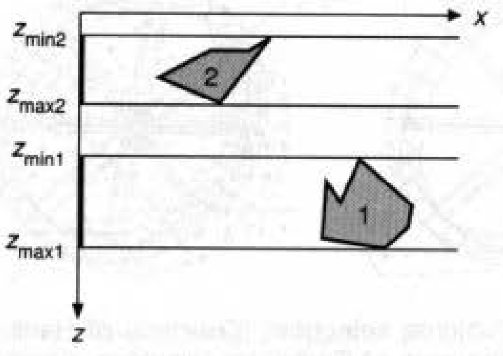


Fig. 15.15 Using 1D extents to determine whether objects overlap.

where b is the number of times the bounding volume is tested for intersection, B is the cost of performing an intersection test on the bounding volume, o is the number of times the object is tested for intersection (the number of times the bounding volume is actually intersected), and O is the cost of performing an intersection test on the object.

Since the object intersection test is performed only when the bounding volume is actually intersected, $o \leq b$. Although O and b are constant for a particular object and set of tests to be performed, B and o vary as a function of the bounding volume's shape and size. A "tighter" bounding volume, which minimizes o, is typically associated with a greater B. A bounding volume's effectiveness may also depend on an object's orientation or the kind of objects with which that object will be intersected. Compare the two bounding volumes for the wagon wheel shown in Fig. 15.16. If the object is to be intersected with projectors perpendicular to the (x, y) plane, then the tighter bounding volume is the sphere. If

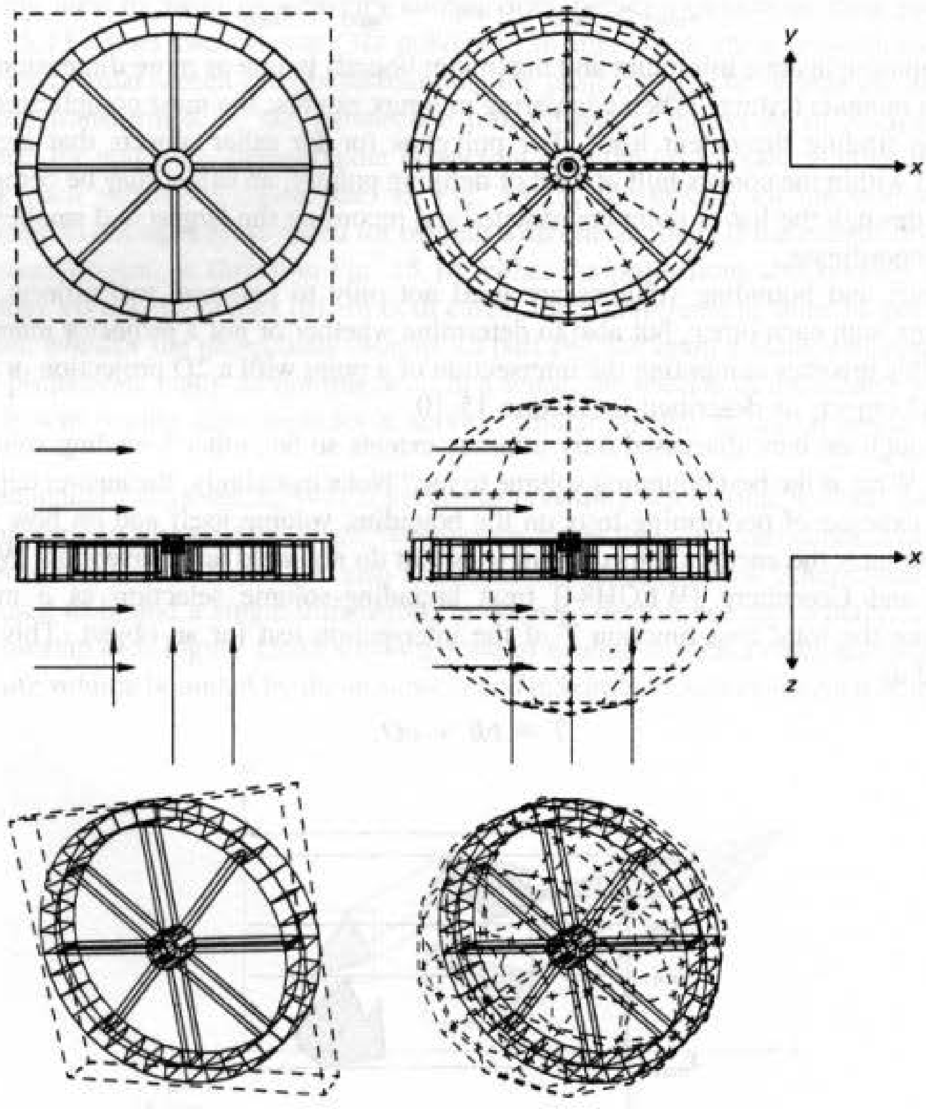


Fig. 15.16 Bounding volume selection. (Courtesy of Hank Weghorst, Gary Hooper, Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1984.)

projectors are perpendicular to the (x, z) or (y, z) planes, then the rectangular extent is the tighter bounding volume. Therefore, multiple bounding volumes may be associated with an object and an appropriate one selected depending on the circumstances.

15.2.4 Back-Face Culling

If an object is approximated by a solid polyhedron, then its polygonal faces completely enclose its volume. Assume that all the polygons have been defined such that their surface normals point out of their polyhedron. If none of the polyhedron's interior is exposed by the front clipping plane, then those polygons whose surface normals point away from the observer lie on a part of the polyhedron whose visibility is completely blocked by other closer polygons, as shown in Fig. 15.17. Such invisible *back-facing* polygons can be eliminated from further processing, a technique known as *back-face culling*. By analogy, those polygons that are not back-facing are often called *front-facing*.

In eye coordinates, a back-facing polygon may be identified by the nonnegative dot product that its surface normal forms with the vector from the center of projection to any point on the polygon. (Strictly speaking, the dot product is positive for a back-facing polygon; a zero dot product indicates a polygon being viewed on edge.) Assuming that the perspective transformation has been performed or that an orthographic projection onto the (x, y) plane is desired, then the direction of projection is $(0, 0, -1)$. In this case, the dot-product test reduces to selecting a polygon as back-facing only if its surface normal has a negative z coordinate. If the environment consists of a single convex polyhedron, back-face culling is the only visible-surface calculation that needs to be performed. Otherwise, there may be front-facing polygons, such as C and E in Fig. 15.17, that are partially or totally obscured.

If the polyhedra have missing or clipped front faces, or if the polygons are not part of polyhedra at all, then back-facing polygons may still be given special treatment. If culling is not desired, the simplest approach is to treat a back-facing polygon as though it were front-facing, flipping its normal in the opposite direction. In PHIGS+, the user can specify a completely separate set of properties for each side of a surface.

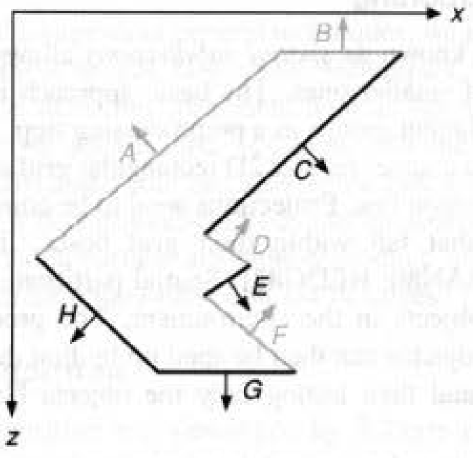


Fig. 15.17 Back-face culling. Back-facing polygons (A,B,D,F) shown in gray are eliminated, whereas front-facing polygons (C,E,G,H) are retained.

Extrapolating from Section 7.12.2's parity-check algorithm for determining whether a point is contained in a polygon, note that a projector passing through a polyhedron intersects the same number of back-facing polygons as of front-facing ones. Thus, a point in a polyhedron's projection lies in the projections of as many back-facing polygons as front-facing ones. Back-face culling therefore halves the number of polygons to be considered for each pixel in an image-precision visible-surface algorithm. On average, approximately one-half of a polyhedron's polygons are back-facing. Thus, back-face culling also typically halves the number of polygons to be considered by the remainder of an object-precision visible-surface algorithm. (Note, however, that this is true only on average. For example, a pyramid's base may be that object's only back- or front-facing polygon.)

As described so far, back-face culling is an object-precision technique that requires time linear in the number of polygons. Sublinear performance can be obtained by preprocessing the objects to be displayed. For example, consider a cube centered about the origin of its own object coordinate system, with its faces perpendicular to the coordinate system's axes. From any viewpoint outside the cube, at most three of its faces are visible. Furthermore, each octant of the cube's coordinate system is associated with a specific set of three potentially visible faces. Therefore, the position of the viewpoint relative to the cube's coordinate system can be used to select one of the eight sets of three potentially visible faces. For objects with a relatively small number of faces, a table may be made up in advance to allow visible-surface determination without processing all the object's faces for each change of viewpoint.

A table of visible faces indexed by viewpoint equivalence class may be quite large, however, for an object with many faces. Tanimoto [TANI77] suggests as an alternative a graph-theoretic approach that takes advantage of frame coherence. A graph is constructed with a node for each face of a convex polyhedron, and a graph edge connecting each pair of nodes whose faces share a polygon edge. The list of edges separating visible faces from invisible ones is then computed for an initial viewpoint. This list contains all edges on the object's silhouette. Tanimoto shows that, as the viewpoint changes between frames, only the visibilities of faces lying between the old and new silhouettes need to be recomputed.

15.2.5 Spatial Partitioning

Spatial partitioning (also known as *spatial subdivision*) allows us to break down a large problem into a number of smaller ones. The basic approach is to assign objects or their projections to spatially coherent groups as a preprocessing step. For example, we can divide the projection plane with a coarse, regular 2D rectangular grid and determine in which grid spaces each object's projection lies. Projections need to be compared for overlap with only those other projections that fall within their grid boxes. This technique is used by [ENCA72; MAHN73; FRAN80; HEDG82]. Spatial partitioning can be used to impose a regular 3D grid on the objects in the environment. The process of determining which objects intersect with a projector can then be sped up by first determining which partitions the projector intersects, and then testing only the objects lying within those partitions (Section 15.10).

If the objects being depicted are unequally distributed in space, it may be more efficient to use *adaptive partitioning*, in which the size of each partition varies. One approach to adaptive partitioning is to subdivide space recursively until some termination criterion is

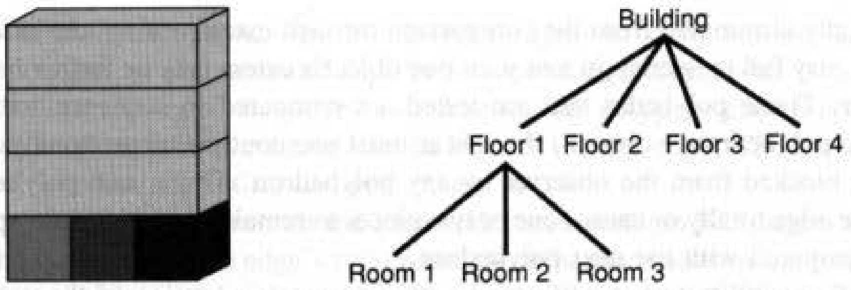


Fig. 15.18 Hierarchy can be used to restrict the number of object comparisons needed. Only if a projector intersects the building and floor 1 does it need to be tested for intersection with rooms 1 through 3.

fulfilled for each partition. For example, subdivision may stop when there are fewer than some maximum number of objects in a partition [TAMM82]. The quadtree, octree, and BSP-tree data structures of Section 12.6 are particularly attractive for this purpose.

15.2.6 Hierarchy

As we saw in Chapter 7, hierarchies can be useful for relating the structure and motion of different objects. A nested hierarchical model, in which each child is considered part of its parent, can also be used to restrict the number of object comparisons needed by a visible-surface algorithm [CLAR76; RUBI80; WEGH84]. An object on one level of the hierarchy can serve as an extent for its children if they are entirely contained within it, as shown in Fig. 15.18. In this case, if two objects in the hierarchy fail to intersect, the lower-level objects of one do not need to be tested for intersection with those of the other. Similarly, only if a projector is found to penetrate an object in the hierarchy must it be tested against the object's children. This use of hierarchy is an important instance of object coherence. A way to automate the construction of hierarchies is discussed in Section 15.10.2.

15.3 ALGORITHMS FOR VISIBLE-LINE DETERMINATION

Now that we have discussed a number of general techniques, we introduce some visible-line and visible-surface algorithms to see how these techniques are used. We begin with visible-line algorithms. The algorithms presented here all operate in object precision and produce as output a list of visible line segments suitable for vector display. The visible-surface algorithms discussed later can also be used for visible-line determination by rendering each surface as a background-colored interior surrounded by a border of the desired line color; most visible-surface algorithms produce an image-precision array of pixels, however, rather than an object-precision list of edges.

15.3.1 Roberts's Algorithm

The earliest visible-line algorithm was developed by Roberts [ROBE63]. It requires that each edge be part of the face of a convex polyhedron. First, back-face culling is used to remove all edges shared by a pair of a polyhedron's back-facing polygons. Next, each remaining edge is compared with each polyhedron that might obscure it. Many polyhedra can be trivially eliminated from the comparison through extent testing: the extents of their projections may fail to overlap in *x* or *y*, or one object's extent may be farther back in *z* than is the other. Those polyhedra that are tested are compared in sequence with the edge. Because the polyhedra are convex, there is at most one contiguous group of points on any line that is blocked from the observer by any polyhedron. Thus, each polyhedron either obscures the edge totally or causes one or two pieces to remain. Any remaining pieces of the edge are compared with the next polyhedron.

Roberts's visibility test is performed with a parametric version of the projector from the eye to a point on the edge being tested. He uses a linear-programming approach to solve for those values of the line equation that cause the projector to pass through a polyhedron, resulting in the invisibility of the endpoint. The projector passes through a polyhedron if it contains some point that is inside all the polyhedron's front faces. Rogers [ROGE85] provides a detailed explanation of Roberts's algorithm and discusses ways in which that algorithm can be further improved.

15.3.2 Appel's Algorithm

Several more general visible-line algorithms [APPE67; GALI69; LOUT70] require only that lines be the edges of polygons, not polyhedra. These algorithms also consider only lines that bound front-facing polygons, and take advantage of edge-coherence in a fashion typified by Appel's algorithm. Appel [APPE67] defines the *quantitative invisibility* of a point on a line as the number of front-facing polygons that obscure that point. When a line passes behind a front-facing polygon, its quantitative invisibility is incremented by 1; when it passes out from behind that polygon, its quantitative invisibility is decremented by 1. A line is visible only when its quantitative invisibility is 0. Line *AB* in Fig. 15.19 is annotated with the quantitative invisibility of each of its segments. If interpenetrating polygons are not allowed, a line's quantitative invisibility changes only when it passes behind what Appel

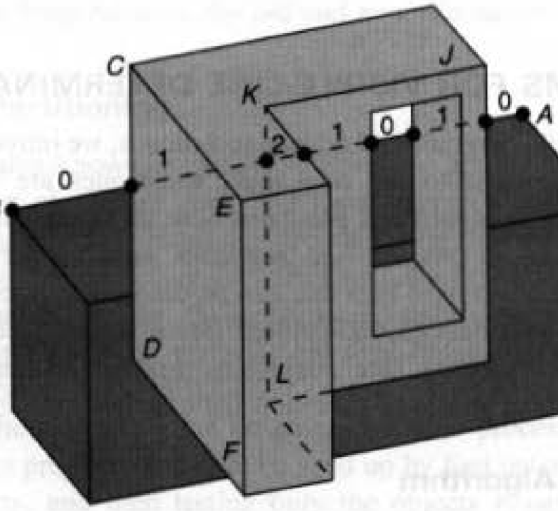


Fig. 15.19 Quantitative invisibility of a line. Dashed lines are hidden. Intersections of *AB*'s projection with projections of contour lines are shown as large dots (●), and each segment of *AB* is marked with its quantitative invisibility.

calls a *contour line*. A contour line is either an edge shared by a front-facing and a back-facing polygon, or the unshared edge of a front-facing polygon that is not part of a closed polyhedron. An edge shared by two front-facing polygons causes no change in visibility and therefore is not a contour line. In Fig. 15.19, edges *AB*, *CD*, *DF*, and *KL* are contour lines, whereas edges *CE*, *EF*, and *JK* are not.

A contour line passes in front of the edge under consideration if it pierces the triangle formed by the eyepoint and the edge's two endpoints. Whether it does so can be determined by a point-in-polygon containment test, such as that discussed in Section 7.12.2. The projection of such a contour line on the edge can be found by clipping the edge against the plane determined by the eyepoint and the contour line. Appel's algorithm requires that all polygon edges be drawn in a consistent direction about the polygon, so that the sign of the change in quantitative invisibility is determined by the sign of the cross-product of the edge with the contour line.

The algorithm first computes the quantitative invisibility of a "seed" vertex of an object by determining the number of front-facing polygons that hide it. This can be done by a brute-force computation of all front-facing polygons whose intersection with the projector to the seed vertex is closer than is the seed vertex itself. The algorithm then takes advantage of edge coherence by propagating this value along the edges emanating from the point, incrementing or decrementing the value at each point at which an edge passes behind a contour line. Only sections of edges whose quantitative invisibility is zero are drawn. When each line's other endpoint is reached, the quantitative invisibility associated with that endpoint becomes the initial quantitative invisibility of all lines emanating in turn from it.

At vertices through which a contour line passes, there is a complication that requires us to make a correction when propagating the quantitative invisibility. One or more lines emanating from the vertex may be hidden by one or more front-facing polygons sharing the vertex. For example, in Fig. 15.19, edge *JK* has a quantitative invisibility of 0, while edge *KL* has a quantitative invisibility of 1 because it is hidden by the object's top face. This change in quantitative invisibility at a vertex can be taken into account by testing the edge against the front-facing polygons that share the vertex.

For an algorithm such as Appel's to handle intersecting polygons, it is necessary to compute the intersections of edges with front-facing polygons and to use each such intersection to increment or decrement the quantitative invisibility. Since visible-line algorithms typically compare whole edges with other edges or objects, they can benefit greatly from spatial-partitioning approaches. Each edge then needs to be compared with only the other edges or objects in the grid boxes containing its projection.

15.3.3 Haloed Lines

Any visible-line algorithm can be easily adapted to show hidden lines as dotted, as dashed, of lower intensity, or with some other rendering style supported by the display device. The program then outputs the hidden line segments in the line style selected, instead of suppressing them. In contrast, Appel, Rohlf, and Stein [APPE79] describe an algorithm for rendering haloed lines, as shown in Fig. 15.20. Each line is surrounded on both sides by a halo that obscures those parts of lines passing behind it. This algorithm, unlike those

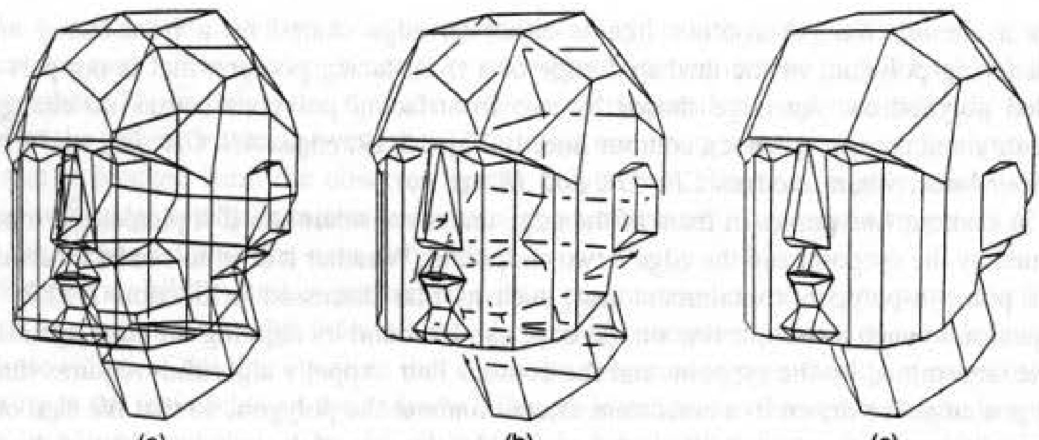


Fig. 15.20 Three heads rendered (a) without hidden lines eliminated, (b) with hidden lines haloed, and (c) with hidden lines eliminated. (Courtesy of Arthur Appel, IBM T.J. Watson Research Center.)

discussed previously, does not require each line to be part of an opaque polygonal face. Lines that pass behind others are obscured only around their intersection on the view plane. The algorithm intersects each line with those passing in front of it, keeps track of those sections that are obscured by halos, and draws the visible sections of each line after the intersections have been calculated. If the halos are wider than the spacing between lines, then an effect similar to conventional hidden-line elimination is achieved, except that a line's halo extends outside a polygon of which it may be an edge.

In the rest of this chapter, we discuss the rich variety of algorithms developed for visible-surface determination. We concentrate here on computing which parts of an object's surfaces are visible, leaving the determination of surface color to Chapter 16. In describing each algorithm, we emphasize its application to polygons, but point out when it can be generalized to handle other objects.

15.4 THE *z*-BUFFER ALGORITHM

The *z-buffer* or *depth-buffer* image-precision algorithm, developed by Catmull [CATM74b], is one of the simplest visible-surface algorithms to implement in either software or hardware. It requires that we have available not only a frame buffer F in which color values are stored, but also a *z-buffer* Z, with the same number of entries, in which a z-value is stored for each pixel. The z-buffer is initialized to zero, representing the z-value at the back clipping plane, and the frame buffer is initialized to the background color. The largest value that can be stored in the z-buffer represents the z of the front clipping plane. Polygons are scan-converted into the frame buffer in arbitrary order. During the scan-conversion process, if the polygon point being scan-converted at (x, y) is no farther from the viewer than is the point whose color and depth are currently in the buffers, then the new point's color and depth replace the old values. The pseudocode for the z-buffer algorithm is shown in Fig. 15.21. The WritePixel and ReadPixel procedures introduced in Chapter 3 are supplemented here by WriteZ and ReadZ procedures that write and read the z-buffer.

```
void zBuffer(void)
{
    int x, y;

    for (y = 0; y < YMAX; y++) {        /* Clear frame buffer and z-buffer */
        for (x = 0; x < XMAX; x++) {
            WritePixel (x, y, BACKGROUND_VALUE);
            WriteZ (x, y, 0);
        }
    }

    for (each polygon) {                /* Draw polygons */
        for (each pixel in polygon's projection) {
            double pz = polygon's z-value at pixel coords (x, y);
            if (pz >= ReadZ (x, y)) {   /* New point is not farther */
                WriteZ (x, y, pz);
                WritePixel (x, y, polygon's color at pixel coords (x, y));
            }
        }
    }
} /* zBuffer */
```

Fig. 15.21 Pseudocode for the *z*-buffer algorithm.

No presorting is necessary and no object–object comparisons are required. The entire process is no more than a search over each set of pairs $\{Z_i(x,y), F_i(x,y)\}$ for fixed x and y, to find the largest Z_i. The *z*-buffer and the frame buffer record the information associated with the largest z encountered thus far for each (x, y). Thus, polygons appear on the screen in the order in which they are processed. Each polygon may be scan-converted one scan line at a time into the buffers, as described in Section 3.6. Figure 15.22 shows the addition of two polygons to an image. Each pixel's shade is shown by its color; its z is shown as a number.

Remembering our discussion of depth coherence, we can simplify the calculation of z for each point on a scan line by exploiting the fact that a polygon is planar. Normally, to calculate z, we would solve the plane equation $Ax + By + Cz + D = 0$ for the variable z:

$$z = \frac{-D - Ax - By}{C}. \tag{15.6}$$

Now, if at (x, y) Eq. (15.6) evaluates to z_1, then at $(x + \Delta x, y)$ the value of z is

$$z_1 - \frac{A}{C}(\Delta x). \tag{15.7}$$

Only one subtraction is needed to calculate $z(x + 1, y)$ given $z(x, y)$, since the quotient A/C is constant and $\Delta x = 1$. A similar incremental calculation can be performed to determine the first value of z on the next scan line, decrementing by B/C for each Δy. Alternatively, if