CHAPTER

9 Three-Dimensional Concepts



When we model and display a three-dimensional scene, there are many more considerations we must take into account besides just including coordinate values for the third dimension. Object boundaries can be constructed with various combinations of plane and curved surfaces, and we sometimes need to specify information about object interiors. Graphics packages often provide routines for displaying internal components or cross-sectional views of solid objects. Also, some geometric transformations are more involved in three-dimensional space than in two dimensions. For example, we can rotate an object about an axis with any spatial orientation in three-dimensional space. Two-dimensional rotations, on the other hand, are always around an axis that is perpendicular to the *xy* plane. Viewing transformations in three dimensions are much more complicated because we have many more parameters to select when specifying how a three-dimensional scene is to be mapped to a display device. The scene description must be processed through viewing-coordinate transformations and projection routines that transform three-dimensional viewing coordinates onto two-dimensional device coordinates. Visible parts of a scene, for a selected view, must be identified; and surface-rendering algorithms must be applied if a realistic rendering of the scene is required.

9-1
THREE-DIMENSIONAL DISPLAY METHODS

To obtain a display of a three-dimensional scene that has been modeled in world coordinates, we must first set up a coordinate reference for the "camera". This coordinate reference defines the position and orientation for the plane of the camera film (Fig. 9-1), which is the plane we want to use to display a view of the objects in the scene. Object descriptions are then transferred to the camera reference coordinates and projected onto the selected display plane. We can then display

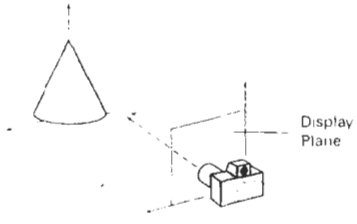


Figure 9-1
Coordinate reference for obtaining a particular view of a three-dimensional scene.

the objects in wireframe (outline) form, as in Fig. 9-2, or we can apply lighting and surface-rendering techniques to shade the visible surfaces.

Parallel Projection

One method for generating a view of a solid object is to project points on the object surface along parallel lines onto the display plane. By selecting different viewing positions, we can project visible points on the object onto the display plane to obtain different two-dimensional views of the object, as in Fig. 9-3. In a *parallel projection*, parallel lines in the world-coordinate scene project into parallel lines on the two-dimensional display plane. This technique is used in engineering and architectural drawings to represent an object with a set of views that maintain relative proportions of the object. The appearance of the solid object can then be reconstructured from the major views.

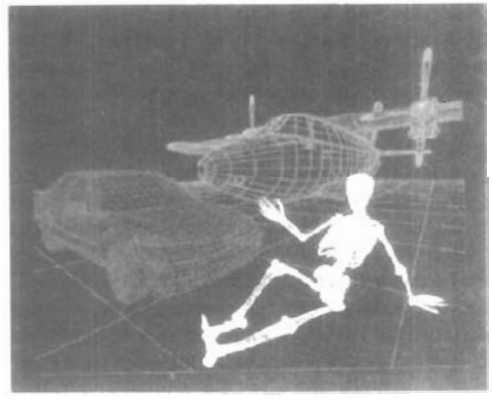


Figure 9-2
Wireframe display of three objects, with back lines removed, from a commercial database of object shapes. Each object in the database is defined as a grid of coordinate points, which can then be viewed in wireframe form or in a surface-rendered form. (*Courtesy of Viewpoint DataLabs.*)

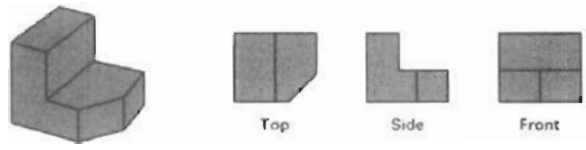


Figure 9-3
Three parallel-projection views of an object, showing relative proportions from different viewing positions.

Perspective Projection

Another method for generating a view of a three-dimensional scene is to project points to the display plane along converging paths. This causes objects farther from the viewing position to be displayed smaller than objects of the same size that are nearer to the viewing position. In a *perspective projection*, parallel lines in a scene that are not parallel to the display plane are projected into converging lines. Scenes displayed using perspective projections appear more realistic, since this is the way that our eyes and a camera lens form images. In the perspective-projection view shown in Fig. 9-4, parallel lines appear to converge to a distant point in the background, and distant objects appear smaller than objects closer to the viewing position.

Depth Cueing

With few exceptions, depth information is important so that we can easily identify, for a particular viewing direction, which is the front and which is the back of displayed objects. Figure 9-5 illustrates the ambiguity that can result when a wireframe object is displayed without depth information. There are several ways in which we can include depth information in the two-dimensional representation of solid objects.

A simple method for indicating depth with wireframe displays is to vary the intensity of objects according to their distance from the viewing position. Figure 9-6 shows a wireframe object displayed with *depth cueing*. The lines closest to

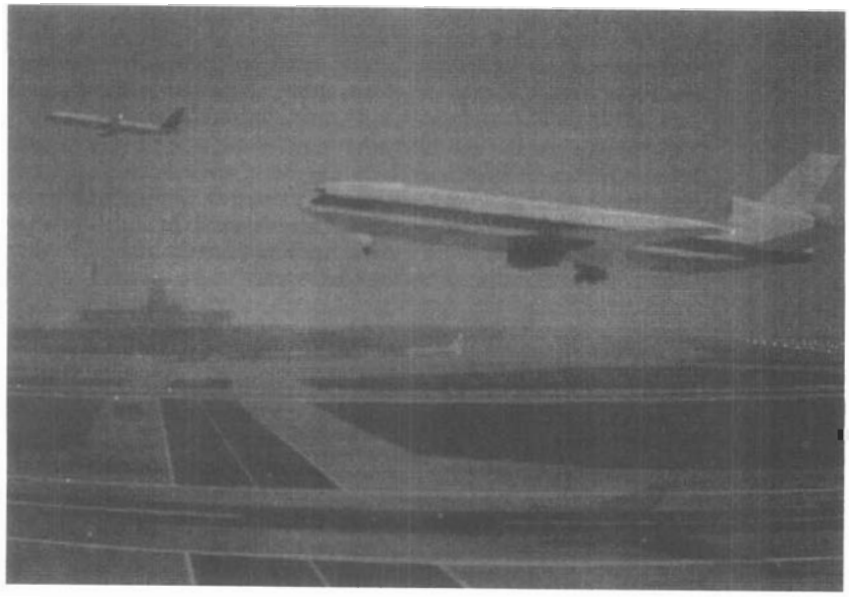


Figure 9-4
A perspective-projection view of an airport scene. (*Courtesy of Evans & Sutherland.*)

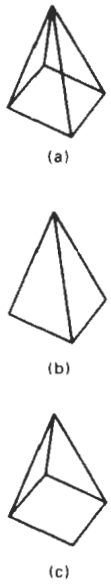


Figure 9-5
The wireframe representation of the pyramid in (a) contains no depth information to indicate whether the viewing direction is (b) downward from a position above the apex or (c) upward from a position below the base.

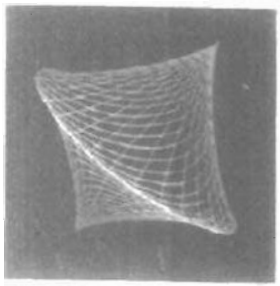


Figure 9-6
A wireframe object displayed with depth cueing, so that the intensity of lines decreases from the front to the back of the object.

the viewing position are displayed with the highest intensities, and lines farther away are displayed with decreasing intensities. Depth cueing is applied by choosing maximum and minimum intensity (or color) values and a range of distances over which the intensities are to vary.

Another application of depth cueing is modeling the effect of the atmosphere on the perceived intensity of objects. More distant objects appear dimmer to us than nearer objects due to light scattering by dust particles, haze, and smoke. Some atmospheric effects can change the perceived color of an object, and we can model these effects with depth cueing.

Visible Line and Surface Identification

We can also clarify depth relationships in a wireframe display by identifying visible lines in some way. The simplest method is to highlight the visible lines or to display them in a different color. Another technique, commonly used for engineering drawings, is to display the nonvisible lines as dashed lines. Another approach is to simply remove the nonvisible lines, as in Figs. 9-5(b) and 9-5(c). But removing the hidden lines also removes information about the shape of the back surfaces of an object. These visible-line methods also identify the visible surfaces of objects.

When objects are to be displayed with color or shaded surfaces, we apply surface-rendering procedures to the visible surfaces so that the hidden surfaces are obscured. Some visible-surface algorithms establish visibility pixel by pixel across the viewing plane; other algorithms determine visibility for object surfaces as a whole.

Surface Rendering

Added realism is attained in displays by setting the surface intensity of objects according to the lighting conditions in the scene and according to assigned surface characteristics. Lighting specifications include the intensity and positions of light sources and the general background illumination required for a scene. Surface properties of objects include degree of transparency and how rough or smooth the surfaces are to be. Procedures can then be applied to generate the correct illumination and shadow regions for the scene. In Fig. 9-7, surface-rendering methods are combined with perspective and visible-surface identification to generate a degree of realism in a displayed scene.

Exploded and Cutaway Views

Many graphics packages allow objects to be defined as hierarchical structures, so that internal details can be stored. Exploded and cutaway views of such objects can then be used to show the internal structure and relationship of the object parts. Figure 9-8 shows several kinds of exploded displays for a mechanical design. An alternative to exploding an object into its component parts is the cutaway view (Fig. 9-9), which removes part of the visible surfaces to show internal structure.

Three-Dimensional and Stereoscopic Views

Another method for adding a sense of realism to a computer-generated scene is to display objects using either three-dimensional or stereoscopic views. As we have seen in Chapter 2, three-dimensional views can be obtained by reflecting a



Figure 9-7
A realistic room display achieved with stochastic ray-tracing methods that apply a perspective projection, surface-texture mapping, and illumination models. (*Courtesy of John Snyder, Jed Lengyel, Devendra Kalra, and Al Barr, California Institute of Technology.* Copyright © 1992 Caltech.)

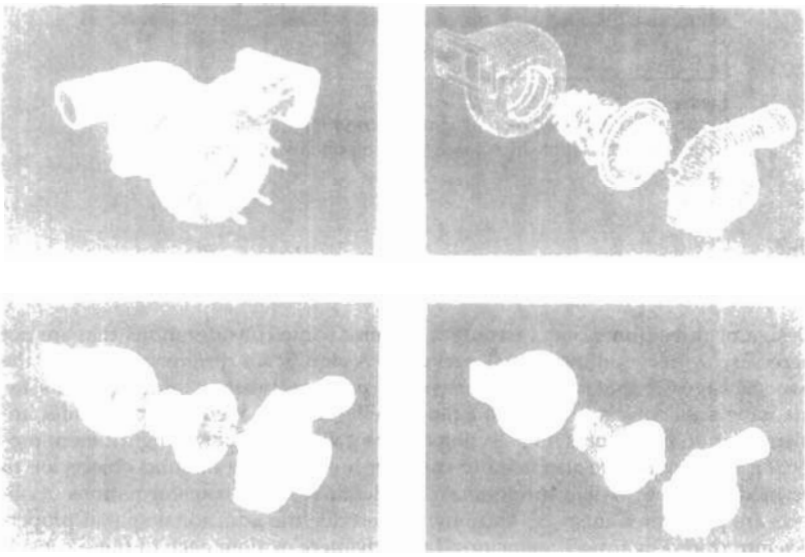


Figure 9-8
A fully rendered and assembled turbine display (a) can also be viewed as (b) an exploded wireframe display, (c) a surface-rendered exploded display, or (d) a surface-rendered, color-coded exploded display. (*Courtesy of Autodesk, Inc.*)

raster image from a vibrating flexible mirror. The vibrations of the mirror are synchronized with the display of the scene on the CRT. As the mirror vibrates, the focal length varies so that each point in the scene is projected to a position corresponding to its depth.

Stereoscopic devices present two views of a scene: one for the left eye and the other for the right eye. The two views are generated by selecting viewing positions that correspond to the two eye positions of a single viewer. These two views then can be displayed on alternate refresh cycles of a raster monitor, and viewed through glasses that alternately darken first one lens then the other in synchronization with the monitor refresh cycles.

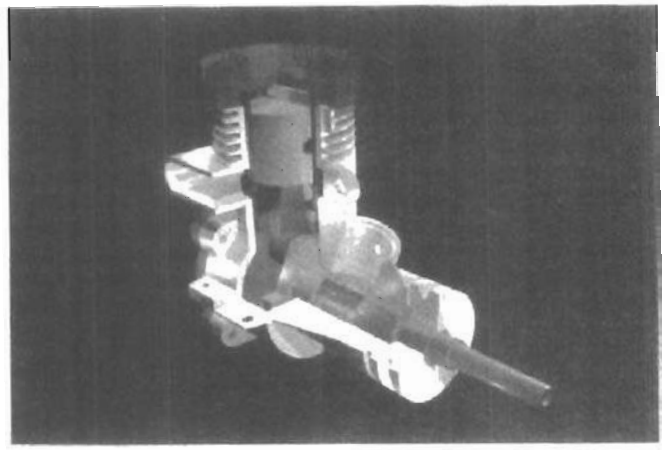


Figure 9-9
Color-coded cutaway view of a lawn mower engine showing the structure and relationship of internal components. (*Courtesy of Autodesk, Inc.*)

9-2
THREE-DIMENSIONAL GRAPHICS PACKAGES

Design of three-dimensional packages requires some considerations that are not necessary with two-dimensional packages. A significant difference between the two packages is that a three-dimensional package must include methods for mapping scene descriptions onto a flat viewing surface. We need to consider implementation procedures for selecting different views and for using different projection techniques. We also need to consider how surfaces of solid objects are to be modeled, how visible surfaces can be identified, how transformations of objects are performed in space, and how to describe the additional spatial properties introduced by three dimensions. Later chapters explore each of these considerations in detail.

Other considerations for three-dimensional packages are straightforward extensions from two-dimensional methods. World-coordinate descriptions are extended to three dimensions, and users are provided with output and input routines accessed with specifications such as

```
polyline3 (n, wcPoints)
fillarea3 (n, wcPoints)
text3 (wcPoint, string)
getLocator3 (wcPoint)
translate3(translateVector, matrixTranslate)
```

where points and vectors are specified with three components, and transformation matrices have four rows and four columns.

Two-dimensional attribute functions that are independent of geometric considerations can be applied in both two-dimensional and three-dimensional applications. No new attribute functions need be defined for colors, line styles, marker



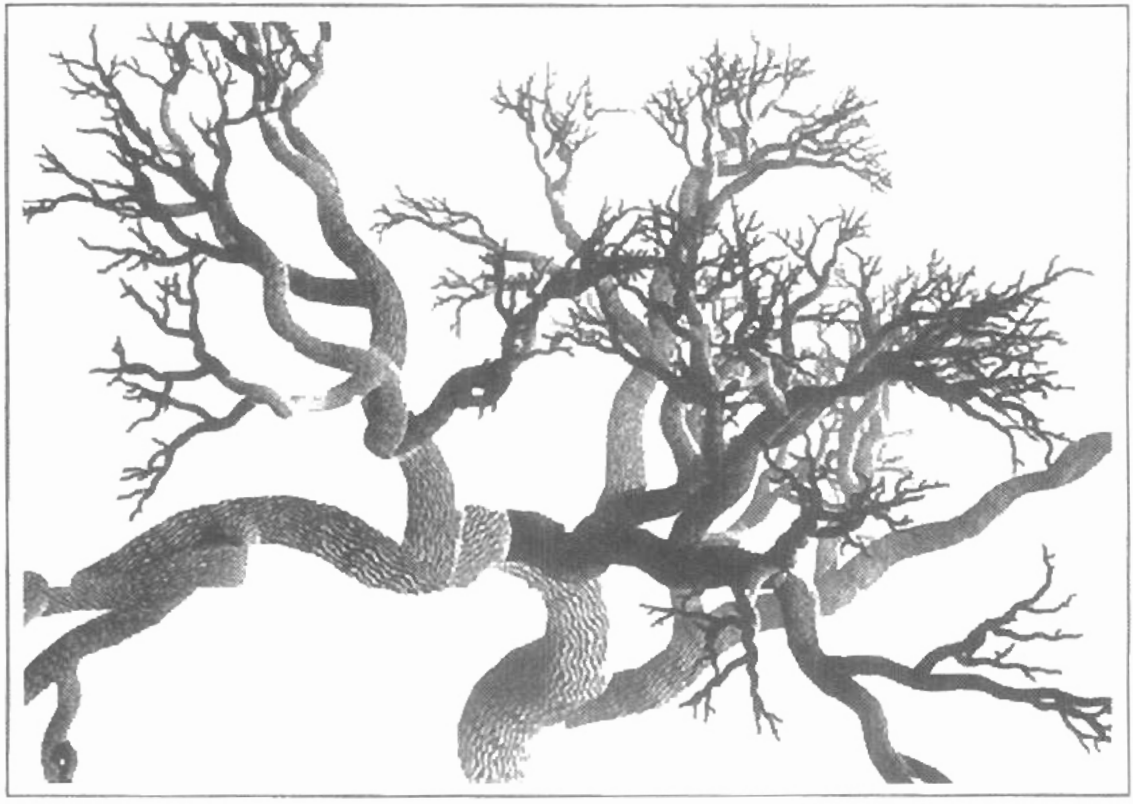
Figure 9-10
Pipeline for transforming a view of a world-coordinate scene to device coordinates.

attributes, or text fonts. Attribute procedures for orienting character strings, however, need to be extended to accommodate arbitrary spatial orientations. Text-attribute routines associated with the up vector require expansion to include z-coordinate data so that strings can be given any spatial orientation. Area-filling routines, such as those for positioning the pattern reference point and for mapping patterns onto a fill area, need to be expanded to accommodate various orientations of the fill-area plane and the pattern plane. Also, most of the two-dimensional structure operations discussed in earlier chapters can be carried over to a three-dimensional package.

Figure 9-10 shows the general stages in a three-dimensional transformation pipeline for displaying a world-coordinate scene. After object definitions have been converted to viewing coordinates and projected to the display plane, scan-conversion algorithms are applied to store the raster image.

CHAPTER

10 Three-Dimensional Object Representations



Graphics scenes can contain many different kinds of objects: trees, flowers, clouds, rocks, water, bricks, wood paneling, rubber, paper, marble, steel, glass, plastic, and cloth, just to mention a few. So it is probably not too surprising that there is no one method that we can use to describe objects that will include all characteristics of these different materials. And to produce realistic displays of scenes, we need to use representations that accurately model object characteristics.

Polygon and quadric surfaces provide precise descriptions for simple Euclidean objects such as polyhedrons and ellipsoids; spline surfaces and construction techniques are useful for designing aircraft wings, gears, and other engineering structures with curved surfaces; procedural methods, such as fractal constructions and particle systems, allow us to give accurate representations for clouds, clumps of grass, and other natural objects; physically based modeling methods using systems of interacting forces can be used to describe the nonrigid behavior of a piece of cloth or a glob of jello; octree encodings are used to represent internal features of objects, such as those obtained from medical CT images; and isosurface displays, volume renderings, and other visualization techniques are applied to three-dimensional discrete data sets to obtain visual representations of the data.

Representation schemes for solid objects are often divided into two broad categories, although not all representations fall neatly into one or the other of these two categories. **Boundary representations** (**B-reps**) describe a three-dimensional object as a set of surfaces that separate the object interior from the environment. Typical examples of boundary representations are polygon facets and spline patches. **Space-partitioning representations** are used to describe interior properties, by partitioning the spatial region containing an object into a set of small, nonoverlapping, contiguous solids (usually cubes). A common space-partitioning description for a three-dimensional object is an octree representation. In this chapter, we consider the features of the various representation schemes and how they are used in applications.

10-1 POLYGON SURFACES

The most commonly used boundary representation for a three-dimensional graphics object is a set of surface polygons that enclose the object interior. Many graphics systems store all object descriptions as sets of surface polygons. This simplifies and speeds up the surface rendering and display of objects, since all surfaces are described with linear equations. For this reason, polygon descrip-

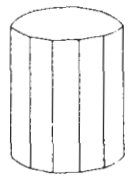


Figure 10-1
Wireframe representation of a cylinder with back (hidden) lines removed.

tions are often referred to as "standard graphics objects." In some cases, a polygonal representation is the only one available, but many packages allow objects to be described with other schemes, such as spline surfaces, that are then converted to polygonal representations for processing.

A polygon representation for a polyhedron precisely defines the surface features of the object. But for other objects, surfaces are *tesselated* (or *tiled*) to produce the polygon-mesh approximation. In Fig. 10-1, the surface of a cylinder is represented as a polygon mesh. Such representations are common in design and solid-modeling applications, since the wireframe outline can be displayed quickly to give a general indication of the surface structure. Realistic renderings are produced by interpolating shading patterns across the polygon surfaces to eliminate or reduce the presence of polygon edge boundaries. And the polygon-mesh approximation to a curved surface can be improved by dividing the surface into smaller polygon facets.

Polygon Tables

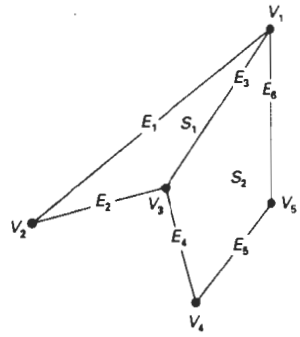
We specify a polygon surface with a set of vertex coordinates and associated attribute parameters. As information for each polygon is input, the data are placed into tables that are to be used in the subsequent processing, display, and manipulation of the objects in a scene. Polygon data tables can be organized into two groups: geometric tables and attribute tables. Geometric data tables contain vertex coordinates and parameters to identify the spatial orientation of the polygon surfaces. Attribute information for an object includes parameters specifying the degree of transparency of the object and its surface reflectivity and texture characteristics.

A convenient organization for storing geometric data is to create three lists: a vertex table, an edge table, and a polygon table. Coordinate values for each vertex in the object are stored in the vertex table. The edge table contains pointers back into the vertex table to identify the vertices for each polygon edge. And the polygon table contains pointers back into the edge table to identify the edges for each polygon. This scheme is illustrated in Fig. 10-2 for two adjacent polygons on an object surface. In addition, individual objects and their component polygon faces can be assigned object and facet identifiers for easy reference.

Listing the geometric data in three tables, as in Fig. 10-2, provides a convenient reference to the individual components (vertices, edges, and polygons) of each object. Also, the object can be displayed efficiently by using data from the edge table to draw the component lines. An alternative arrangement is to use just two tables: a vertex table and a polygon table. But this scheme is less convenient, and some edges could get drawn twice. Another possibility is to use only a polygon table, but this duplicates coordinate information, since explicit coordinate values are listed for each vertex in each polygon. Also edge information would have to be reconstructed from the vertex listings in the polygon table.

We can add extra information to the data tables of Fig. 10-2 for faster information extraction. For instance, we could expand the edge table to include forward pointers into the polygon table so that common edges between polygons could be identified more rapidly (Fig. 10-3). This is particularly useful for the rendering procedures that must vary surface shading smoothly across the edges from one polygon to the next. Similarly, the vertex table could be expanded so that vertices are cross-referenced to corresponding edges.

Additional geometric information that is usually stored in the data tables includes the slope for each edge and the coordinate extents for each polygon. As vertices are input, we can calculate edge slopes, and we can scan the coordinate



VERTEX TABLE	
V_1:	x_1, y_1, z_1
V_2:	x_2, y_2, z_2
V_3:	x_3, y_3, z_3
V_4:	x_4, y_4, z_4
V_5:	x_5, y_5, z_5

EDGE TABLE	
E_1:	V_1, V_2
E_2:	V_2, V_3
E_3:	V_3, V_1
E_4:	V_3, V_4
E_5:	V_4, V_5
E_6:	V_5, V_1

POLYGON-SURFACE TABLE	
S_1:	E_1, E_2, E_3
S_2:	E_3, E_4, E_5, E_6

Figure 10-2
Geometric data table representation for two adjacent polygon surfaces, formed with six edges and five vertices.

values to identify the minimum and maximum x, y, and z values for individual polygons. Edge slopes and bounding-box information for the polygons are needed in subsequent processing, for example, surface rendering. Coordinate extents are also used in some visible-surface determination algorithms.

Since the geometric data tables may contain extensive listings of vertices and edges for complex objects, it is important that the data be checked for consistency and completeness. When vertex, edge, and polygon definitions are specified, it is possible, particularly in interactive applications, that certain input errors could be made that would distort the display of the object. The more information included in the data tables, the easier it is to check for errors. Therefore, error checking is easier when three data tables (vertex, edge, and polygon) are used, since this scheme provides the most information. Some of the tests that could be performed by a graphics package are (1) that every vertex is listed as an endpoint for at least two edges, (2) that every edge is part of at least one polygon, (3) that every polygon is closed, (4) that each polygon has at least one shared edge, and (5) that if the edge table contains pointers to polygons, every edge referenced by a polygon pointer has a reciprocal pointer back to the polygon.

E_1:	V_1, V_2, S_1
E_2:	V_2, V_3, S_1
E_3:	V_3, V_1, S_1, S_2
E_4:	V_3, V_4, S_2
E_5:	V_4, V_5, S_2
E_6:	V_5, V_1, S_2

Figure 10-3
Edge table for the surfaces of Fig. 10-2 expanded to include pointers to the polygon table.

Plane Equations

To produce a display of a three-dimensional object, we must process the input data representation for the object through several procedures. These processing steps include transformation of the modeling and world-coordinate descriptions to viewing coordinates, then to device coordinates; identification of visible surfaces; and the application of surface-rendering procedures. For some of these processes, we need information about the spatial orientation of the individual

surface components of the object. This information is obtained from the vertex-coordinate values and the equations that describe the polygon planes.

The equation for a plane surface can be expressed in the form

$$Ax + By + Cz + D = 0 \tag{10-1}$$

where (x, y, z) is any point on the plane, and the coefficients A, B, C, and D are constants describing the spatial properties of the plane. We can obtain the values of A, B, C, and D by solving a set of three plane equations using the coordinate values for three noncollinear points in the plane. For this purpose, we can select three successive polygon vertices, (x_1, y_1, z_1), (x_2, y_2, z_2), and (x_3, y_3, z_3), and solve the following set of simultaneous linear plane equations for the ratios A/D, B/D, and C/D:

$$(A/D)x_k + (B/D)y_k + (C/D)z_k = -1, \qquad k = 1, 2, 3 \tag{10-2}$$

The solution for this set of equations can be obtained in determinant form, using Cramer's rule, as

$$A = \begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{vmatrix} \qquad B = \begin{vmatrix} x_1 & 1 & z_1 \\ x_2 & 1 & z_2 \\ x_3 & 1 & z_3 \end{vmatrix}$$

$$C = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \qquad D = -\begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix} \tag{10-3}$$

Expanding the determinants, we can write the calculations for the plane coefficients in the form

$$\begin{aligned} A &= y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2) \\ B &= z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2) \\ C &= x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2) \\ D &= -x_1(y_2z_3 - y_3z_2) - x_2(y_3z_1 - y_1z_3) - x_3(y_1z_2 - y_2z_1) \end{aligned} \tag{10-4}$$

As vertex values and other information are entered into the polygon data structure, values for A, B, C, and D are computed for each polygon and stored with the other polygon data.

Orientation of a plane surface in space can be described with the normal vector to the plane, as shown in Fig. 10-4. This surface normal vector has Cartesian components (A, B, C), where parameters A, B, and C are the plane coefficients calculated in Eqs. 10-4.

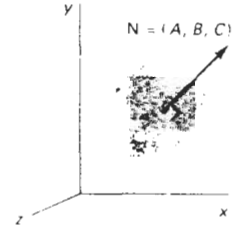


Figure 10-4
The vector N, normal to the surface of a plane described by the equation $Ax + By + Cz + D = 0$, has Cartesian components (A, B, C)

Since we are usually dealing with polygon surfaces that enclose an object interior, we need to distinguish between the two sides of the surface. The side of the plane that faces the object interior is called the "inside" face, and the visible or outward side is the "outside" face. If polygon vertices are specified in a counterclockwise direction when viewing the outer side of the plane in a right-handed coordinate system, the direction of the normal vector will be from inside to outside. This is demonstrated for one plane of a unit cube in Fig. 10-5.

To determine the components of the normal vector for the shaded surface shown in Fig. 10-5, we select three of the four vertices along the boundary of the polygon. These points are selected in a counterclockwise direction as we view from outside the cube toward the origin. Coordinates for these vertices, in the order selected, can be used in Eqs. 10-4 to obtain the plane coefficients: $A = 1$, $B = 0, C = 0, D = -1$. Thus, the normal vector for this plane is in the direction of the positive x axis.

The elements of the plane normal can also be obtained using a vector cross-product calculation. We again select three vertex positions, $\mathbf{V}_1$, $\mathbf{V}_2$, and $\mathbf{V}_3$, taken in counterclockwise order when viewing the surface from outside to inside in a right-handed Cartesian system. Forming two vectors, one from $\mathbf{V}_1$ to $\mathbf{V}_2$ and the other from $\mathbf{V}_1$ to $\mathbf{V}_3$, we calculate $\mathbf{N}$ as the vector cross product:

$$\mathbf{N} = (\mathbf{V}_2 - \mathbf{V}_1) \times (\mathbf{V}_3 - \mathbf{V}_1) \tag{10-5}$$

This generates values for the plane parameters A, B, and C. We can then obtain the value for parameter D by substituting these values and the coordinates for one of the polygon vertices in plane equation 10-1 and solving for D. The plane equation can be expressed in vector form using the normal $\mathbf{N}$ and the position $\mathbf{P}$ of any point in the plane as

$$\mathbf{N} \cdot \mathbf{P} = -D \tag{10-6}$$

Plane equations are used also to identify the position of spatial points relative to the plane surfaces of an object. For any point (x, y, z) not on a plane with parameters A, B, C, D, we have

$$Ax + By + Cz + D \neq 0$$

We can identify the point as either inside or outside the plane surface according to the sign (negative or positive) of $Ax + By + Cz + D$:

if $Ax + By + Cz + D < 0$, the point (x, y, z) is inside the surface

if $Ax + By + Cz + D > 0$, the point (x, y, z) is outside the surface

These inequality tests are valid in a right-handed Cartesian system, provided the plane parameters A, B, C, and D were calculated using vertices selected in a counterclockwise order when viewing the surface in an outside-to-inside direction. For example, in Fig. 10-5, any point outside the shaded plane satisfies the inequality $x - 1 > 0$, while any point inside the plane has an x-coordinate value less than 1.

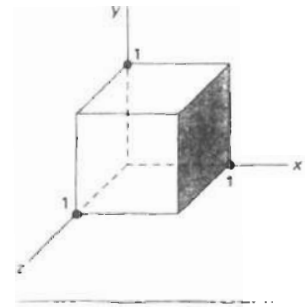


Figure 10-5
The shaded polygon surface of the unit cube has plane equation $x - 1 = 0$ and normal vector $\mathbf{N} = (1, 0, 0)$.

Polygon Meshes

Some graphics packages (for example, PHIGS) provide several polygon functions for modeling objects. A single plane surface can be specified with a function such as `fillArea`. But when object surfaces are to be tiled, it is more convenient to specify the surface facets with a mesh function. One type of polygon mesh is the *triangle strip*. This function produces $n - 2$ connected triangles, as shown in Fig. 10-6, given the coordinates for n vertices. Another similar function is the *quadrilateral mesh*, which generates a mesh of $(n - 1)$ by $(m - 1)$ quadrilaterals, given

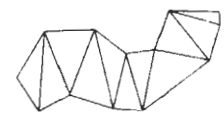


Figure 10-6
A triangle strip formed with 11 triangles connecting 13 vertices.

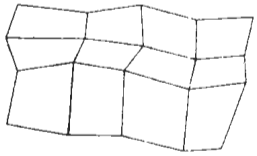


Figure 10-7
A quadrilateral mesh containing 12 quadrilaterals constructed from a 5 by 4 input vertex array.

the coordinates for an n by m array of vertices. Figure 10-7 shows 20 vertices forming a mesh of 12 quadrilaterals.

When polygons are specified with more than three vertices, it is possible that the vertices may not all lie in one plane. This can be due to numerical errors or errors in selecting coordinate positions for the vertices. One way to handle this situation is simply to divide the polygons into triangles. Another approach that is sometimes taken is to approximate the plane parameters A, B, and C. We can do this with averaging methods or we can project the polygon onto the coordinate planes. Using the projection method, we take A proportional to the area of the polygon projection on the yz plane, B proportional to the projection area on the xz plane, and C proportional to the projection area on the xy plane.

High-quality graphics systems typically model objects with polygon meshes and set up a database of geometric and attribute information to facilitate processing of the polygon facets. Fast hardware-implemented polygon renderers are incorporated into such systems with the capability for displaying hundreds of thousands to one million or more shaded polygons per second (usually triangles), including the application of surface texture and special lighting effects.

10-2
CURVED LINES AND SURFACES

Displays of three-dimensional curved lines and surfaces can be generated from an input set of mathematical functions defining the objects or from a set of user-specified data points. When functions are specified, a package can project the defining equations for a curve to the display plane and plot pixel positions along the path of the projected function. For surfaces, a functional description is often tesselated to produce a polygon-mesh approximation to the surface. Usually, this is done with triangular polygon patches to ensure that all vertices of any polygon are in one plane. Polygons specified with four or more vertices may not have all vertices in a single plane. Examples of display surfaces generated from functional descriptions include the quadrics and the superquadrics.

When a set of discrete coordinate points is used to specify an object shape, a functional description is obtained that best fits the designated points according to the constraints of the application. Spline representations are examples of this class of curves and surfaces. These methods are commonly used to design new object shapes, to digitize drawings, and to describe animation paths. Curve-fitting methods are also used to display graphs of data values by fitting specified curve functions to the discrete data set, using regression techniques such as the least-squares method.

Curve and surface equations can be expressed in either a parametric or a nonparametric form. Appendix A gives a summary and comparison of parametric and nonparametric equations. For computer graphics applications, parametric representations are generally more convenient.

10-3
QUADRIC SURFACES

A frequently used class of objects are the *quadric surfaces*, which are described with second-degree equations (quadratics). They include spheres, ellipsoids, tori,

paraboloids, and hyperboloids. Quadric surfaces, particularly spheres and ellipsoids, are common elements of graphics scenes, and they are often available in graphics packages as primitives from which more complex objects can be constructed.

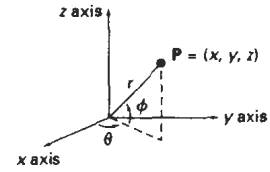


Figure 10-8
Parametric coordinate position (r, θ, ϕ) on the surface of a sphere with radius r.

Sphere

In Cartesian coordinates, a spherical surface with radius r centered on the coordinate origin is defined as the set of points (x, y, z) that satisfy the equation

$$x^2 + y^2 + z^2 = r^2 \tag{10-7}$$

We can also describe the spherical surface in parametric form, using latitude and longitude angles (Fig. 10-8):

$$\begin{aligned} x &= r\cos\phi\cos\theta, \qquad -\pi/2 \le \phi \le \pi/2 \\ y &= r\cos\phi\sin\theta, \qquad -\pi \le \theta \le \pi \\ z &= r\sin\phi \end{aligned} \tag{10-8}$$

The parametric representation in Eqs. 10-8 provides a symmetric range for the angular parameters θ and ϕ. Alternatively, we could write the parametric equations using standard spherical coordinates, where angle ϕ is specified as the colatitude (Fig. 10-9). Then, ϕ is defined over the range $0 \le \phi \le \pi$, and θ is often taken in the range $0 \le \theta \le 2\pi$. We could also set up the representation using parameters u and v defined over the range from 0 to 1 by substituting $\phi = \pi u$ and $\theta = 2\pi v$.

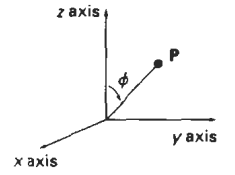


Figure 10-9
Spherical coordinate parameters (r, θ, ϕ), using colatitude for angle ϕ.

Ellipsoid

An ellipsoidal surface can be described as an extension of a spherical surface, where the radii in three mutually perpendicular directions can have different values (Fig. 10-10). The Cartesian representation for points over the surface of an ellipsoid centered on the origin is

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1 \tag{10-9}$$

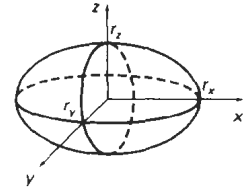


Figure 10-10
An ellipsoid with radii r_x, r_y, and r_z centered on the coordinate origin.

And a parametric representation for the ellipsoid in terms of the latitude angle ϕ and the longitude angle θ in Fig. 10-8 is

$$\begin{aligned} x &= r_x\cos\phi\cos\theta, \qquad -\pi/2 \le \phi \le \pi/2 \\ y &= r_y\cos\phi\sin\theta, \qquad -\pi \le \theta \le \pi \\ z &= r_z\sin\phi \end{aligned} \tag{10-10}$$

Torus

A torus is a doughnut-shaped object, as shown in Fig. 10-11. It can be generated by rotating a circle or other conic about a specified axis. The Cartesian represen-

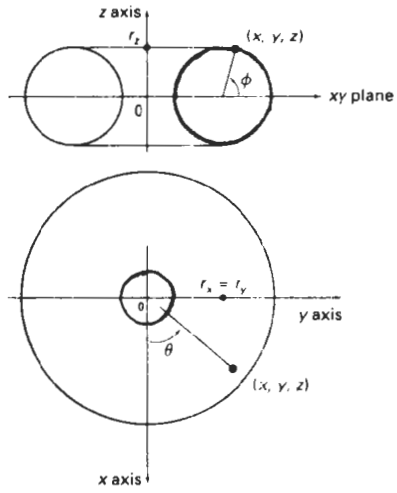


Figure 10-11
A torus with a circular cross section centered on the coordinate origin.

tation for points over the surface of a torus can be written in the form

$$\left[r - \sqrt{\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2}\right]^2 + \left(\frac{z}{r_z}\right)^2 = 1 \tag{10-11}$$

where r is any given offset value. Parametric representations for a torus are similar to those for an ellipse, except that angle ϕ extends over 360°. Using latitude and longitude angles ϕ and θ, we can describe the torus surface as the set of points that satisfy

$$\begin{aligned} x &= r_x(r + \cos\phi)\cos\theta, \qquad -\pi \le \phi \le \pi \\ y &= r_y(r + \cos\phi)\sin\theta, \qquad -\pi \le \theta \le \pi \\ z &= r_z \sin\phi \end{aligned} \tag{10-12}$$

10-4 SUPERQUADRICS

This class of objects is a generalization of the quadric representations. **Superquadrics** are formed by incorporating additional parameters into the quadric equations to provide increased flexibility for adjusting object shapes. The number of additional parameters used is equal to the dimension of the object: one parameter for curves and two parameters for surfaces.

Superellipse

We obtain a Cartesian representation for a superellipse from the corresponding equation for an ellipse by allowing the exponent on the x and y terms to be vari-

able. One way to do this is to write the Cartesian superellipse equation in the form

$$\left(\frac{x}{r_x}\right)^{2/s} + \left(\frac{y}{r_y}\right)^{2/s} = 1 \tag{10-13}$$

where parameter s can be assigned any real value. When $s = 1$, we get an ordinary ellipse.

Corresponding parametric equations for the superellipse of Eq. 10-13 can be expressed as

$$\begin{aligned} x &= r_x \cos^s \theta, \qquad -\pi \le \theta \le \pi \\ y &= r_y \sin^s \theta \end{aligned} \tag{10-14}$$

Figure 10-12 illustrates supercircle shapes that can be generated using various values for parameter s.

Superellipsoid

A Cartesian representation for a superellipsoid is obtained from the equation for an ellipsoid by incorporating two exponent parameters:

$$\left[\left(\frac{x}{r_x}\right)^{2/s_2} + \left(\frac{y}{r_y}\right)^{2/s_2}\right]^{s_2/s_1} + \left(\frac{z}{r_z}\right)^{2/s_1} = 1 \tag{10-15}$$

For $s_1 = s_2 = 1$, we have an ordinary ellipsoid.

We can then write the corresponding parametric representation for the superellipsoid of Eq. 10-15 as

$$\begin{aligned} x &= r_x \cos^{s_1}\phi \cos^{s_2}\theta, \qquad -\pi/2 \le \phi \le \pi/2 \\ y &= r_y \cos^{s_1}\phi \sin^{s_2}\theta, \qquad -\pi \le \theta \le \pi \\ z &= r_z \sin^{s_1}\phi \end{aligned} \tag{10-16}$$

Figure 10-13 illustrates supersphere shapes that can be generated using various values for parameters s_1 and s_2. These and other superquadric shapes can be combined to create more complex structures, such as furniture, threaded bolts, and other hardware.



Figure 10-12
Superellipses plotted with different values for parameter s and with $r_x = r_y$.

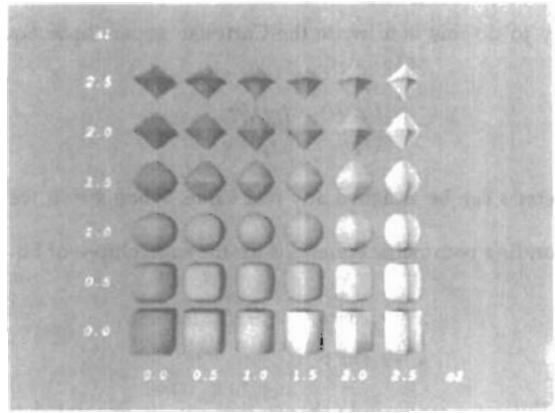


Figure 10-13
Superellipsoids plotted with different values for parameters s_1 and s_2, and with $r_x = r_y = r_z$.

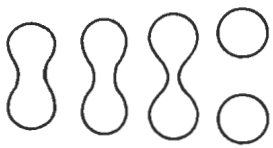


Figure 10-14
Molecular bonding. As two molecules move away from each other, the surface shapes stretch, snap, and finally contract into spheres.

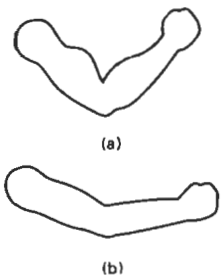


Figure 10-15
Blobby muscle shapes in a human arm.

10-5 BLOBBY OBJECTS

Some objects do not maintain a fixed shape, but change their surface characteristics in certain motions or when in proximity to other objects. Examples in this class of objects include molecular structures, water droplets and other liquid effects, melting objects, and muscle shapes in the human body. These objects can be described as exhibiting "blobbiness" and are often simply referred to as **blobby objects**, since their shapes show a certain degree of fluidity.

A molecular shape, for example, can be described as spherical in isolation, but this shape changes when the molecule approaches another molecule. This distortion of the shape of the electron density cloud is due to the "bonding" that occurs between the two molecules. Figure 10-14 illustrates the stretching, snapping, and contracting effects on molecular shapes when two molecules move apart. These characteristics cannot be adequately described simply with spherical or elliptical shapes. Similarly, Fig. 10-15 shows muscle shapes in a human arm, which exhibit similar characteristics. In this case, we want to model surface shapes so that the total volume remains constant.

Several models have been developed for representing blobby objects as distribution functions over a region of space. One way to do this is to model objects as combinations of Gaussian density functions, or "bumps" (Fig. 10-16). A surface function is then defined as

$$f(x, y, z) = \sum_k b_k e^{-a_k r_k^2} - T = 0 \qquad (10\text{-}17)$$

where $r_k^2 = \sqrt{x_k^2 + y_k^2 + z_k^2}$, parameter T is some specified threshold, and parameters a and b are used to adjust the amount of blobbiness of the individual objects. Negative values for parameter b can be used to produce dents instead of bumps. Figure 10-17 illustrates the surface structure of a composite object modeled with four Gaussian density functions. At the threshold level, numerical root-finding

techniques are used to locate the coordinate intersection values. The cross sections of the individual objects are then modeled as circles or ellipses. If two cross sections are near to each other, they are merged to form one blobby shape, as in Figure 10-14, whose structure depends on the separation of the two objects.

Other methods for generating blobby objects use density functions that fall off to 0 in a finite interval, rather than exponentially. The "metaball" model describes composite objects as combinations of quadratic density functions of the form

$$f(r) = \begin{cases} b(1 - 3r^2/d^2), & \text{if } 0 < r \le d/3 \\ \frac{3}{2}b(1 - r/d)^2, & \text{if } d/3 < r \le d \\ 0, & \text{if } r > d \end{cases} \tag{10-18}$$

And the "soft object" model uses the function

$$f(r) = \begin{cases} 1 - \frac{22r^2}{9d^2} + \frac{17r^4}{9d^4} - \frac{4r^6}{9d^6}, & \text{if } 0 < r \le d \\ 0, & \text{if } r > d \end{cases} \tag{10-19}$$

Some design and painting packages now provide blobby function modeling for handling applications that cannot be adequately modeled with polygon or spline functions alone. Figure 10-18 shows a user interface for a blobby object modeler using metaballs.

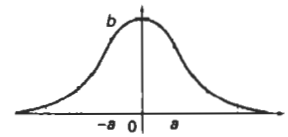


Figure 10-16
A three-dimensional Gaussian bump centered at position 0, with height b and standard deviation a.

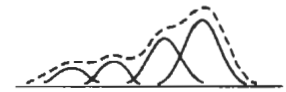


Figure 10-17
A composite blobby object formed with four Gaussian bumps.

10-6 SPLINE REPRESENTATIONS

In drafting terminology, a spline is a flexible strip used to produce a smooth curve through a designated set of points. Several small weights are distributed along the length of the strip to hold it in position on the drafting table as the curve is drawn. The term *spline curve* originally referred to a curve drawn in this manner. We can mathematically describe such a curve with a piecewise cubic

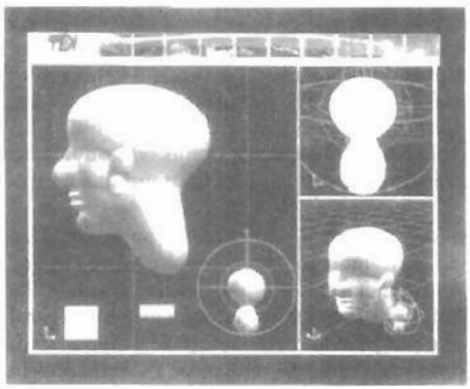


Figure 10-18
A screen layout, used in the Blob Modeler and the Blob Animator packages, for modeling objects with metaballs. (*Courtesy of Thomson Digital Image.*)

polynomial function whose first and second derivatives are continuous across the various curve sections. In computer graphics, the term **spline curve** now refers to any composite curve formed with polynomial sections satisfying specified continuity conditions at the boundary of the pieces. A **spline surface** can be described with two sets of orthogonal spline curves. There are several different kinds of spline specifications that are used in graphics applications. Each individual specification simply refers to a particular type of polynomial with certain specified boundary conditions.

Splines are used in graphics applications to design curve and surface shapes, to digitize drawings for computer storage, and to specify animation paths for the objects or the camera in a scene. Typical CAD applications for splines include the design of automobile bodies, aircraft and spacecraft surfaces, and ship hulls.



Figure 10-19
A set of six control points interpolated with piecewise continuous polynomial sections.



Figure 10-20
A set of six control points approximated with piecewise continuous polynomial sections

Interpolation and Approximation Splines

We specify a spline curve by giving a set of coordinate positions, called **control points**, which indicates the general shape of the curve. These control points are then fitted with piecewise continuous parametric polynomial functions in one of two ways. When polynomial sections are fitted so that the curve passes through each control point, as in Fig. 10-19, the resulting curve is said to **interpolate** the set of control points. On the other hand, when the polynomials are fitted to the general control-point path without necessarily passing through any control point, the resulting curve is said to **approximate** the set of control points (Fig. 10-20).

Interpolation curves are commonly used to digitize drawings or to specify animation paths. Approximation curves are primarily used as design tools to structure object surfaces. Figure 10-21 shows an approximation spline surface created for a design application. Straight lines connect the control-point positions above the surface.

A spline curve is defined, modified, and manipulated with operations on the control points. By interactively selecting spatial positions for the control points, a designer can set up an initial curve. After the polynomial fit is displayed for a given set of control points, the designer can then reposition some or all of the control points to restructure the shape of the curve. In addition, the curve can be translated, rotated, or scaled with transformations applied to the control points. CAD packages can also insert extra control points to aid a designer in adjusting the curve shapes.

The convex polygon boundary that encloses a set of control points is called the **convex hull**. One way to envision the shape of a convex hull is to imagine a rubber band stretched around the positions of the control points so that each control point is either on the perimeter of the hull or inside it (Fig. 10-22). Convex hulls provide a measure for the deviation of a curve or surface from the region bounding the control points. Some splines are bounded by the convex hull, thus ensuring that the polynomials smoothly follow the control points without erratic oscillations. Also, the polygon region inside the convex hull is useful in some algorithms as a clipping region.

A polyline connecting the sequence of control points for an approximation spline is usually displayed to remind a designer of the control-point ordering. This set of connected line segments is often referred to as the **control graph** of the curve. Other names for the series of straight-line sections connecting the control points in the order specified are **control polygon** and **characteristic polygon**. Figure 10-23 shows the shape of the control graph for the control-point sequences in Fig. 10-22.

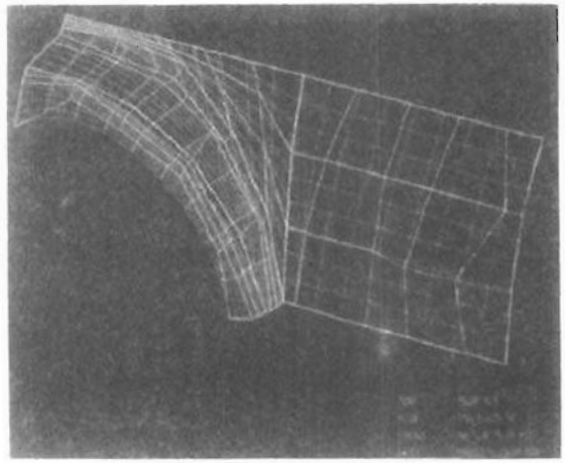


Figure 10-21
An approximation spline surface for a CAD application in automotive design. Surface contours are plotted with polynomial curve sections, and the surface control points are connected with straight-line segments. (*Courtesy of Evans & Sutherland.*)

Parametric Continuity Conditions

To ensure a smooth transition from one section of a piecewise parametric curve to the next, we can impose various **continuity conditions** at the connection points. If each section of a spline is described with a set of parametric coordinate functions of the form

$$x = x(u), \quad y = y(u), \quad z = z(u), \quad u_1 \leq u \leq u_2 \tag{10-20}$$

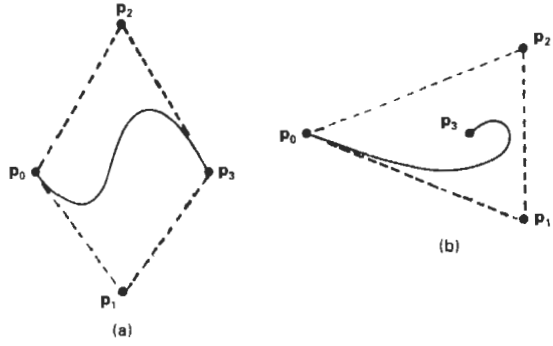


Figure 10-22
Convex-hull shapes (dashed lines) for two sets of control points.

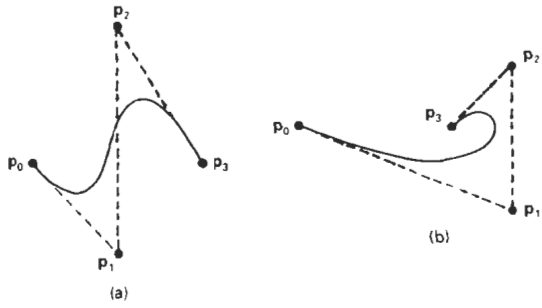


Figure 10-23
Control-graph shapes (dashed lines) for two different sets of control points.

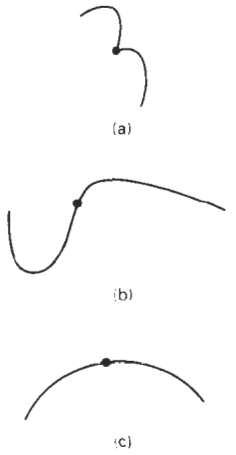


Figure 10-24
Piecewise construction of a curve by joining two curve segments using different orders of continuity: (a) zero-order continuity only, (b) first-order continuity, and (c) second-order continuity.

we set **parametric continuity** by matching the parametric derivatives of adjoining curve sections at their common boundary.

Zero-order parametric continuity, described as C^0 continuity, means simply that the curves meet. That is, the values of x, y, and z evaluated at u_2 for the first curve section are equal, respectively, to the values of x, y, and z evaluated at u_1 for the next curve section. **First-order parametric continuity**, referred to as C^1 continuity, means that the first parametric derivatives (tangent lines) of the coordinate functions in Eq. 10-20 for two successive curve sections are equal at their joining point. **Second-order parametric continuity**, or C^2 continuity, means that both the first and second parametric derivatives of the two curve sections are the same at the intersection. Higher-order parametric continuity conditions are defined similarly. Figure 10-24 shows examples of C^0, C^1, and C^2 continuity.

With second-order continuity, the rates of change of the tangent vectors for connecting sections are equal at their intersection. Thus, the tangent line transitions smoothly from one section of the curve to the next (Fig. 10-24(c)). But with first-order continuity, the rates of change of the tangent vectors for the two sections can be quite different (Fig. 10-24(b)), so that the general shapes of the two adjacent sections can change abruptly. First-order continuity is often sufficient for digitizing drawings and some design applications, while second-order continuity is useful for setting up animation paths for camera motion and for many precision CAD requirements. A camera traveling along the curve path in Fig. 10-24(b) with equal steps in parameter u would experience an abrupt change in acceleration at the boundary of the two sections, producing a discontinuity in the motion sequence. But if the camera were traveling along the path in Fig. 10-24(c), the frame sequence for the motion would smoothly transition across the boundary.

Geometric Continuity Conditions

An alternate method for joining two successive curve sections is to specify conditions for **geometric continuity**. In this case, we only require parametric derivatives of the two sections to be proportional to each other at their common boundary instead of equal to each other.

Zero-order geometric continuity, described as G^0 continuity, is the same as zero-order parametric continuity. That is, the two curves sections must have the

same coordinate position at the boundary point. **First-order geometric continuity**, or G^1 continuity, means that the parametric first derivatives are proportional at the intersection of two successive sections. If we denote the parametric position on the curve as $\mathbf{P}(u)$, the direction of the tangent vector $\mathbf{P}'(u)$, but not necessarily its magnitude, will be the same for two successive curve sections at their joining point under G^1 continuity. **Second-order geometric continuity**, or G^2 continuity, means that both the first and second parametric derivatives of the two curve sections are proportional at their boundary. Under G^2 continuity, curvatures of two curve sections will match at the joining position.

A curve generated with geometric continuity conditions is similar to one generated with parametric continuity, but with slight differences in curve shape. Figure 10-25 provides a comparison of geometric and parametric continuity. With geometric continuity, the curve is pulled toward the section with the greater tangent vector.

Spline Specifications

There are three equivalent methods for specifying a particular spline representation: (1) We can state the set of boundary conditions that are imposed on the spline; or (2) we can state the matrix that characterizes the spline; or (3) we can state the set of **blending functions** (or **basis functions**) that determine how specified geometric constraints on the curve are combined to calculate positions along the curve path.

To illustrate these three equivalent specifications, suppose we have the following parametric cubic polynomial representation for the x coordinate along the path of a spline section:

$$x(u) = a_x u^3 + b_x u^2 + c_x u + d_x, \qquad 0 \le u \le 1 \tag{10-21}$$

Boundary conditions for this curve might be set, for example, on the endpoint coordinates $x(0)$ and $x(1)$ and on the parametric first derivatives at the endpoints $x'(0)$ and $x'(1)$. These four boundary conditions are sufficient to determine the values of the four coefficients a_x, b_x, c_x, and d_x.

From the boundary conditions, we can obtain the matrix that characterizes this spline curve by first rewriting Eq. 10-21 as the matrix product

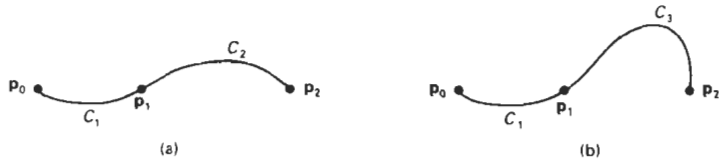


Figure 10-25
Three control points fitted with two curve sections joined with (a) parametric continuity and (b) geometric continuity, where the tangent vector of curve C_3 at point $\mathbf{p}_1$ has a greater magnitude than the tangent vector of curve C_1 at $\mathbf{p}_1$.

$$x(u) = [u^3\ u^2\ u\ 1]\begin{bmatrix} a_x \\ b_x \\ c_x \\ d_x \end{bmatrix} \tag{10-22}$$
$$= \mathbf{U} \cdot \mathbf{C}$$

where **U** is the row matrix of powers of parameter u, and **C** is the coefficient column matrix. Using Eq. 10-22, we can write the boundary conditions in matrix form and solve for the coefficient matrix **C** as

$$\mathbf{C} = \mathbf{M}_{\text{spline}} \cdot \mathbf{M}_{\text{geom}} \tag{10-23}$$

where $\mathbf{M}_{\text{geom}}$ is a four-element column matrix containing the geometric constraint values (boundary conditions) on the spline, and $\mathbf{M}_{\text{spline}}$ is the 4-by-4 matrix that transforms the geometric constraint values to the polynomial coefficients and provides a characterization for the spline curve. Matrix $\mathbf{M}_{\text{geom}}$ contains control-point coordinate values and other geometric constraints that have been specified. Thus, we can substitute the matrix representation for **C** into Eq. 10-22 to obtain

$$x(u) = \mathbf{U} \cdot \mathbf{M}_{\text{spline}} \cdot \mathbf{M}_{\text{geom}} \tag{10-24}$$

The matrix, $\mathbf{M}_{\text{spline}}$, characterizing a spline representation, sometimes called the *basis matrix*, is particularly useful for transforming from one spline representation to another.

Finally, we can expand Eq. 10-24 to obtain a polynomial representation for coordinate x in terms of the geometric constraint parameters

$$x(u) = \sum_{k=0}^{3} g_k \cdot BF_k(u) \tag{10-25}$$

where g_k are the constraint parameters, such as the control-point coordinates and slope of the curve at the control points, and $BF_k(u)$ are the polynomial blending functions. In the following sections, we discuss some commonly used splines and their matrix and blending-function specifications.

10-7 CUBIC SPLINE INTERPOLATION METHODS

This class of splines is most often used to set up paths for object motions or to provide a representation for an existing object or drawing, but interpolation splines are also used sometimes to design object shapes. Cubic polynomials offer a reasonable compromise between flexibility and speed of computation. Compared to higher-order polynomials, cubic splines require less calculations and memory and they are more stable. Compared to lower-order polynomials, cubic splines are more flexible for modeling arbitrary curve shapes.

Given a set of control points, cubic interpolation splines are obtained by fitting the input points with a piecewise cubic polynomial curve that passes through every control point. Suppose we have $n + 1$ control points specified with coordinates

$$\mathbf{p}_k = (x_k, y_k, z_k), \qquad k = 0, 1, 2, \ldots, n$$

A cubic interpolation fit of these points is illustrated in Fig. 10-26. We can describe the parametric cubic polynomial that is to be fitted between each pair of control points with the following set of equations:

$$\begin{aligned} x(u) &= a_x u^3 + b_x u^2 + c_x u + d_x \\ y(u) &= a_y u^3 + b_y u^2 + c_y u + d_y, \qquad (0 \le u \le 1) \\ z(u) &= a_z u^3 + b_z u^2 + c_z u + d_z \end{aligned} \tag{10-26}$$

For each of these three equations, we need to determine the values of the four coefficients a, b, c, and d in the polynomial representation for each of the n curve sections between the $n + 1$ control points. We do this by setting enough boundary conditions at the "joints" between curve sections so that we can obtain numerical values for all the coefficients. In the following sections, we discuss common methods for setting the boundary conditions for cubic interpolation splines.

Natural Cubic Splines

One of the first spline curves to be developed for graphics applications is the **natural cubic spline**. This interpolation curve is a mathematical representation of the original drafting spline. We formulate a natural cubic spline by requiring that two adjacent curve sections have the same first and second parametric derivatives at their common boundary. Thus, natural cubic splines have C^2 continuity.

If we have $n + 1$ control points to fit, as in Fig. 10-26, then we have n curve sections with a total of $4n$ polynomial coefficients to be determined. At each of the $n - 1$ interior control points, we have four boundary conditions: The two curve sections on either side of a control point must have the same first and second parametric derivatives at that control point, and each curve must pass through that control point. This gives us $4n - 4$ equations to be satisfied by the $4n$ polynomial coefficients. We get an additional equation from the first control point $\mathbf{p}_0$, the position of the beginning of the curve, and another condition from control point $\mathbf{p}_n$, which must be the last point on the curve. We still need two more conditions to be able to determine values for all coefficients. One method for obtaining the two additional conditions is to set the second derivatives at $\mathbf{p}_0$ and $\mathbf{p}_n$ to 0. Another approach is to add two extra "dummy" control points, one at each end of the original control-point sequence. That is, we add a control point $\mathbf{p}_{-1}$ and a control point $\mathbf{p}_{n+1}$. Then all of the original control points are interior points, and we have the necessary $4n$ boundary conditions.

Although natural cubic splines are a mathematical model for the drafting spline, they have a major disadvantage. If the position of any one control point is altered, the entire curve is affected. Thus, natural cubic splines allow for no "local control", so that we cannot restructure part of the curve without specifying an entirely new set of control points.

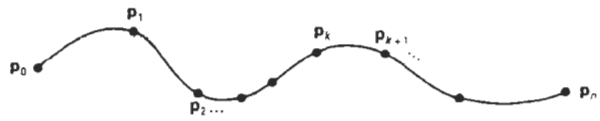


Figure 10-26
A piecewise continuous cubic-spline interpolation of $n + 1$ control points.

Hermite Interpolation

A **Hermite spline** (named after the French mathematician Charles Hermite) is an interpolating piecewise cubic polynomial with a specified tangent at each control point. Unlike the natural cubic splines, Hermite splines can be adjusted locally because each curve section is only dependent on its endpoint constraints.

If $\mathbf{P}(u)$ represents a parametric cubic point function for the curve section between control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$, as shown in Fig. 10-27, then the boundary conditions that define this Hermite curve section are

$$\begin{aligned} \mathbf{P}(0) &= \mathbf{p}_k \\ \mathbf{P}(1) &= \mathbf{p}_{k+1} \\ \mathbf{P}'(0) &= \mathbf{D}\mathbf{p}_k \\ \mathbf{P}'(1) &= \mathbf{D}\mathbf{p}_{k+1} \end{aligned} \tag{10-27}$$

with $\mathbf{D}\mathbf{p}_k$ and $\mathbf{D}\mathbf{p}_{k+1}$ specifying the values for the parametric derivatives (slope of the curve) at control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$, respectively.

We can write the vector equivalent of Eqs. 10-26 for this Hermite-curve section as

$$\mathbf{P}(u) = \mathbf{a}u^3 + \mathbf{b}u^2 + \mathbf{c}u + \mathbf{d}, \qquad 0 \le u \le 1 \tag{10-28}$$

where the x component of $\mathbf{P}$ is $x(u) = a_x u^3 + b_x u^2 + c_x u + d_x$, and similarly for the y and z components. The matrix equivalent of Eq. 10-28 is

$$\mathbf{P}(u) = [u^3\ u^2\ u\ 1] \cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix} \tag{10-29}$$

and the derivative of the point function can be expressed as

$$\mathbf{P}'(u) = [3u^2\ 2u\ 1\ 0] \cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix} \tag{10-30}$$

Substituting endpoint values 0 and 1 for parameter u into the previous two equations, we can express the Hermite boundary conditions 10-27 in the matrix form:

$$\begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix} \tag{10-31}$$

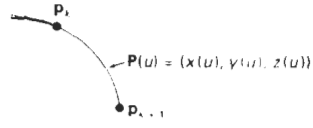


Figure 10-27
Parametric point function $\mathbf{P}(u)$ for a Hermite curve section between control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$.

Solving this equation for the polynomial coefficients, we have

$$\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k-1} \end{bmatrix}$$

$$= \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix} \tag{10-32}$$

$$= \mathbf{M}_H \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix}$$

where $\mathbf{M}_H$, the Hermite matrix, is the inverse of the boundary constraint matrix. Equation 10-29 can thus be written in terms of the boundary conditions as

$$\mathbf{P}(u) = [u^3\ u^2\ u\ 1] \cdot \mathbf{M}_H \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix} \tag{10-33}$$

Finally, we can determine expressions for the Hermite blending functions by carrying out the matrix multiplications in Eq. 10-33 and collecting coefficients for the boundary constraints to obtain the polynomial form:

$$\begin{aligned} \mathbf{P}(u) &= \mathbf{p}_k(2u^3 - 3u^2 + 1) + \mathbf{p}_{k+1}(-2u^3 + 3u^2) - \mathbf{D}\mathbf{p}_k(u^3 - 2u^2 + u) \\ &\quad + \mathbf{D}\mathbf{p}_{k+1}(u^3 - u^2) \\ &= \mathbf{p}_k H_0(u) + \mathbf{p}_{k+1} H_1(u) + \mathbf{D}\mathbf{p}_k H_2(u) + \mathbf{D}\mathbf{p}_{k+1} H_3(u) \end{aligned} \tag{10-34}$$

The polynomials $H_k(u)$ for $k = 0, 1, 2, 3$ are referred to as blending functions because they blend the boundary constraint values (endpoint coordinates and slopes) to obtain each coordinate position along the curve. Figure 10-28 shows the shape of the four Hermite blending functions.

Hermite polynomials can be useful for some digitizing applications where it may not be too difficult to specify or approximate the curve slopes. But for most problems in computer graphics, it is more useful to generate spline curves without requiring input values for curve slopes or other geometric information, in addition to control-point coordinates. Cardinal splines and Kochanek–Bartels splines, discussed in the following two sections, are variations on the Hermite splines that do not require input values for the curve derivatives at the control points. Procedures for these splines compute parametric derivatives from the coordinate positions of the control points.

Cardinal Splines

As with Hermite splines, **cardinal splines** are interpolating piecewise cubics with specified endpoint tangents at the boundary of each curve section. The difference

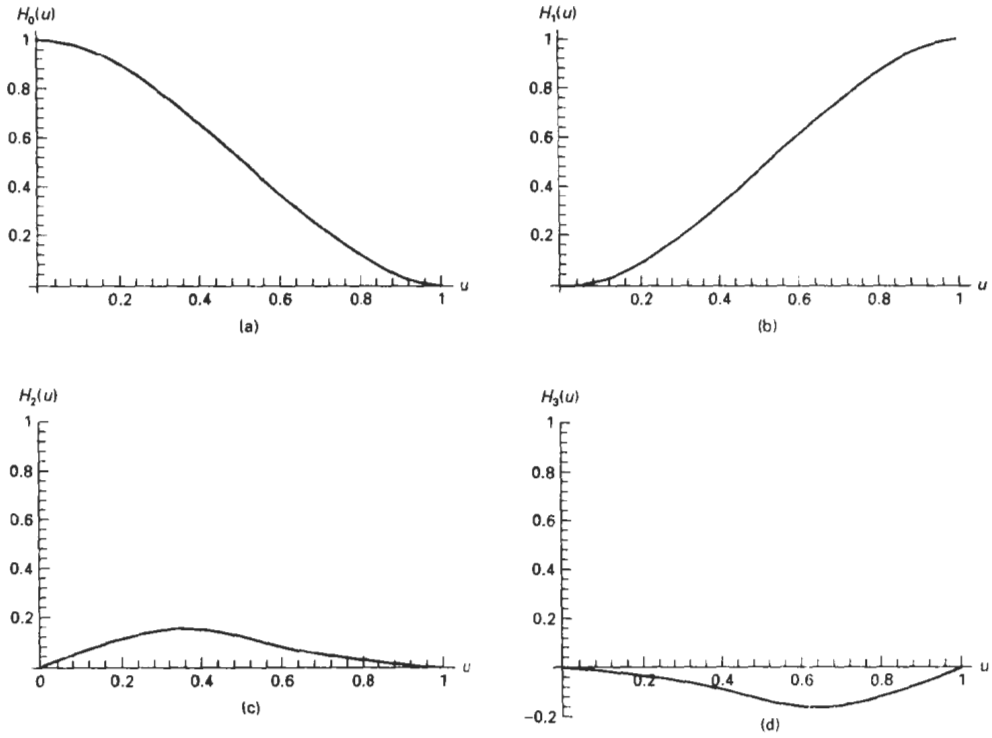


Figure 10-28
The Hermite blending functions.

is that we do not have to give the values for the endpoint tangents. For a cardinal spline, the value for the slope at a control point is calculated from the coordinates of the two adjacent control points.

A cardinal spline section is completely specified with four consecutive control points. The middle two control points are the section endpoints, and the other two points are used in the calculation of the endpoint slopes. If we take $\mathbf{P}(u)$ as the representation for the parametric cubic point function for the curve section between control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$, as in Fig. 10-29, then the four control points from $\mathbf{p}_{k-1}$ to $\mathbf{p}_{k+1}$ are used to set the boundary conditions for the cardinal-spline section as

$$\begin{aligned} \mathbf{P}(0) &= \mathbf{p}_k \\ \mathbf{P}(1) &= \mathbf{p}_{k+1} \\ \mathbf{P}'(0) &= \tfrac{1}{2}(1-t)(\mathbf{p}_{k+1} - \mathbf{p}_{k-1}) \\ \mathbf{P}'(1) &= \tfrac{1}{2}(1-t)(\mathbf{p}_{k+2} - \mathbf{p}_k) \end{aligned} \qquad (10\text{-}35)$$

Thus, the slopes at control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$ are taken to be proportional, respectively, to the chords $\overline{\mathbf{p}_{k-1}\mathbf{p}_{k+1}}$ and $\overline{\mathbf{p}_k\,\mathbf{p}_{k+2}}$ (Fig. 10-30). Parameter t is called the **tension** parameter since it controls how loosely or tightly the cardinal spline fits

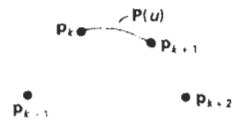


Figure 10-29
Parametric point function $\mathbf{P}(u)$ for a cardinal-spline section between control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$.

the input control points. Figure 10-31 illustrates the shape of a cardinal curve for very small and very large values of tension t. When $t = 0$, this class of curves is referred to as **Catmull–Rom splines**, or **Overhauser splines**.

Using methods similar to those for Hermite splines, we can convert the boundary conditions 10-35 into the matrix form

$$\mathbf{P}(u) = [u^3\ u^2\ u\ 1] \cdot \mathbf{M}_C \cdot \begin{bmatrix} \mathbf{p}_{k-1} \\ \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{p}_{k+2} \end{bmatrix} \tag{10-36}$$

where the cardinal matrix is

$$\mathbf{M}_C = \begin{bmatrix} -s & 2-s & s-2 & s \\ 2s & s-3 & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{10-37}$$

with $s = (1 - t)/2$.

Expanding matrix equation 10-36 into polynomial form, we have

$$\begin{aligned} \mathbf{P}(u) &= \mathbf{p}_{k-1}(-su^3 + 2su^2 - su) + \mathbf{p}_k[(2-s)u^3 + (s-3)u^2 + 1] \\ &\quad + \mathbf{p}_{k+1}[(s-2)u^3 + (3-2s)u^2 + su] + \mathbf{p}_{k+2}(su^3 - su^2) \\ &= \mathbf{p}_{k-1}CAR_0(u) + \mathbf{p}_kCAR_1(u) + \mathbf{p}_{k+1}CAR_2(u) + \mathbf{p}_{k+2}CAR_3(u) \end{aligned} \tag{10-38}$$

where the polynomials $CAR_k(u)$ for $k = 0, 1, 2, 3$ are the cardinal blending functions. Figure 10-32 gives a plot of the basis functions for cardinal splines with $t = 0$.

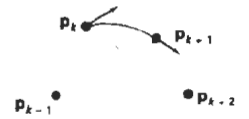


Figure 10-30
Tangent vectors at the endpoints of a cardinal-spline section are proportional to the chords formed with neighboring control points (dashed lines).

Kochanek–Bartels Splines

These interpolating cubic polynomials are extensions of the cardinal splines. Two additional parameters are introduced into the constraint equations defining **Kochanek–Bartels splines** to provide for further flexibility in adjusting the shape of curve sections.

Given four consecutive control points, labeled $\mathbf{p}_{k-1}$, $\mathbf{p}_k$, $\mathbf{p}_{k+1}$, and $\mathbf{p}_{k+2}$, we define the boundary conditions for a Kochanek–Bartels curve section between $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$ as

$$\begin{aligned} \mathbf{P}(0) &= \mathbf{p}_k \\ \mathbf{P}(1) &= \mathbf{p}_{k+1} \\ \mathbf{P}'(0)_{\text{in}} &= \tfrac{1}{2}(1-t)[(1+b)(1-c)(\mathbf{p}_k - \mathbf{p}_{k-1}) \\ &\qquad + (1-b)(1+c)(\mathbf{p}_{k+1} - \mathbf{p}_k)] \\ \mathbf{P}'(1)_{\text{out}} &= \tfrac{1}{2}(1-t)[(1+b)(1+c)(\mathbf{p}_{k+1} - \mathbf{p}_k) \\ &\qquad + (1-b)(1-c)(\mathbf{p}_{k+2} - \mathbf{p}_{k+1})] \end{aligned} \tag{10-39}$$

where t is the **tension** parameter, b is the **bias** parameter, and c is the **continuity** parameter. In the Kochanek-Bartels formulation, parametric derivatives may not be continuous across section boundaries.

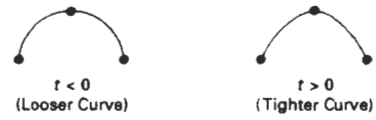


Figure 10-31
Effect of the tension parameter on the shape of a cardinal spline section.

Tension parameter t has the same interpretation as in the cardinal-spline formulation; that is, it controls the looseness or tightness of the curve sections. Bias (b) is used to adjust the amount that the curve bends at each end of a section, so that curve sections can be skewed toward one end or the other (Fig. 10-33). Parameter c controls the continuity of the tangent vector across the boundaries of sections. If c is assigned a nonzero value, there is a discontinuity in the slope of the curve across section boundaries.

Kochanek–Bartel splines were designed to model animation paths. In particular, abrupt changes in motion of a object can be simulated with nonzero values for parameter c.

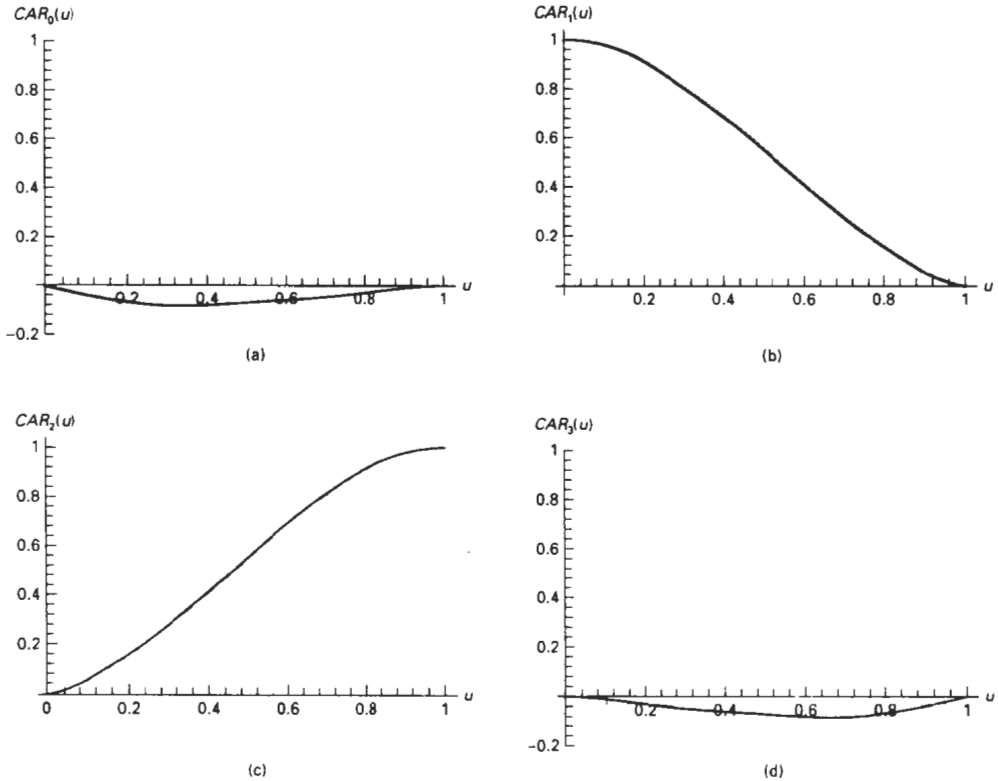


Figure 10-32
The cardinal blending functions for $t = 0$ and $s = 0.5$.

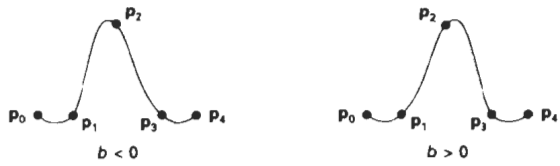


Figure 10-33
Effect of the bias parameter on the shape of a Kochanek–Bartels spline section.

10-8
BÉZIER CURVES AND SURFACES

This spline approximation method was developed by the French engineer Pierre Bézier for use in the design of Renault automobile bodies. **Bézier splines** have a number of properties that make them highly useful and convenient for curve and surface design. They are also easy to implement. For these reasons, Bézier splines are widely available in various CAD systems, in general graphics packages (such as GL on Silicon Graphics systems), and in assorted drawing and painting packages (such as Aldus SuperPaint and Cricket Draw).

Bézier Curves

In general, a Bézier curve section can be fitted to any number of control points. The number of control points to be approximated and their relative position determine the degree of the Bézier polynomial. As with the interpolation splines, a Bézier curve can be specified with boundary conditions, with a characterizing matrix, or with blending functions. For general Bézier curves, the blending-function specification is the most convenient.

Suppose we are given $n + 1$ control-point positions: $\mathbf{p}_k = (x_k, y_k, z_k)$, with k varying from 0 to n. These coordinate points can be blended to produce the following position vector $\mathbf{P}(u)$, which describes the path of an approximating Bézier polynomial function between $\mathbf{p}_0$ and $\mathbf{p}_n$.

$$\mathbf{P}(u) = \sum_{k=0}^{n} \mathbf{p}_k BEZ_{k,n}(u), \qquad 0 \leq u \leq 1 \tag{10-40}$$

The Bézier blending functions $BEZ_{k,n}(u)$ are the *Bernstein polynomials*:

$$BEZ_{k,n}(u) = C(n, k)u^k(1 - u)^{n-k} \tag{10-41}$$

where the $C(n, k)$ are the binomial coefficients:

$$C(n, k) = \frac{n!}{k!(n - k)!} \tag{10-42}$$

Equivalently, we can define Bézier blending functions with the recursive calculation

$$BEZ_{k,n}(u) = (1 - u)\,BEZ_{k,n-1}(u) + uBEZ_{k-1,n-1}(u), \qquad n > k \geq 1 \tag{10-43}$$

with $BEZ_{k,k} = u^k$, and $BEZ_{0,k} = (1 - u)^k$. Vector equation 10-40 represents a set of three parametric equations for the individual curve coordinates:

$$x(u) = \sum_{k=0}^{n} x_k \, BEZ_{k,n}(u)$$
$$y(u) = \sum_{k=0}^{n} y_k \, BEZ_{k,n}(u) \qquad (10\text{-}44)$$
$$z(u) = \sum_{k=0}^{n} z_k \, BEZ_{k,n}(u)$$

As a rule, a Bézier curve is a polynomial of degree one less than the number of control points used: Three points generate a parabola, four points a cubic curve, and so forth. Figure 10-34 demonstrates the appearance of some Bezier curves for various selections of control points in the xy plane ($z = 0$). With certain control-point placements, however, we obtain degenerate Bézier polynomials. For example, a Bézier curve generated with three collinear control points is a straight-line segment. And a set of control points that are all at the same coordinate position produces a Bézier "curve" that is a single point.

Bézier curves are commonly found in painting and drawing packages, as well as CAD systems, since they are easy to implement and they are reasonably powerful in curve design. Efficient methods for determining coordinate positions along a Bézier curve can be set up using recursive calculations. For example, successive binomial coefficients can be calculated as

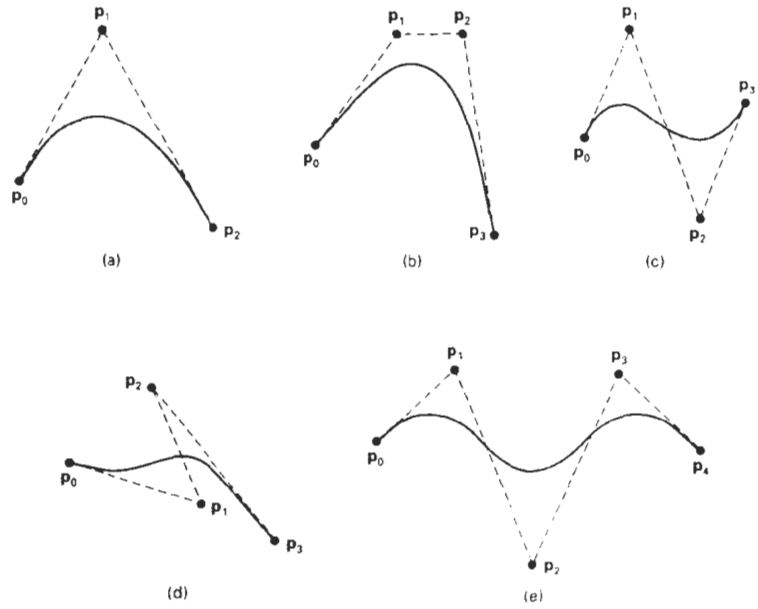


Figure 10-34
Examples of two-dimensional Bézier curves generated from three, four, and five control points. Dashed lines connect the control-point positions.

$$C(n,k) = \frac{n-k+1}{k} C(n,k-1) \qquad (10\text{-}45)$$

for $n \geq k$. The following example program illustrates a method for generating Bézier curves.

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

void computeCoefficients (int n, int * c)
{
  int k, i;

  for (k=0; k<=n; k++) {
    /* Compute n!/(k!(n-k)!) */
    c[k] = 1;
    for (i=n; i>=k+1; i--)
      c[k] *= i;
    for (i=n-k; i>=2; i--)
      c[k] /= i;
  }
}

void computePoint
  (float u, wcPt3 * pt, int nControls, wcPt3 * controls, int * c)
{
  int k, n = nControls - 1;
  float blend;

  pt->x = 0.0; pt->y = 0.0; pt->z = 0.0;

  /* Add in influence of each control point */
  for (k=0; k<nControls; k++) {
    blend = c[k] * powf (u,k) * powf (1-u,n-k);
    pt->x += controls[k].x * blend;
    pt->y += controls[k].y * blend;
    pt->z += controls[k].z * blend;
  }
}

void bezier (wcPt3 * controls, int nControls, int m, wcPt3 * curve)
{
  /* Allocate space for the coefficients */
  int * c = (int *) malloc (nControls * sizeof (int));
  int i;

  computeCoefficients (nControls-1, c);
  for (i=0; i<=m; i++)
    computePoint (i / (float) m, &curve[i], nControls, controls, c);
  free (c);
}
```

Properties of Bézier Curves

A very useful property of a Bézier curve is that it always passes through the first and last control points. That is, the boundary conditions at the two ends of the curve are

$$\begin{aligned} \mathbf{P}(0) &= \mathbf{p}_0 \\ \mathbf{P}(1) &= \mathbf{p}_n \end{aligned} \qquad (10\text{-}46)$$

Values of the parametric first derivatives of a Bézier curve at the endpoints can be calculated from control-point coordinates as

$$\begin{aligned} \mathbf{P}'(0) &= -n\mathbf{p}_0 + n\mathbf{p}_1 \\ \mathbf{P}'(1) &= -n\mathbf{p}_{n-1} + n\mathbf{p}_n \end{aligned} \tag{10-47}$$

Thus, the slope at the beginning of the curve is along the line joining the first two control points, and the slope at the end of the curve is along the line joining the last two endpoints. Similarly, the parametric second derivatives of a Bézier curve at the endpoints are calculated as

$$\begin{aligned} \mathbf{P}''(0) &= n(n-1)[(\mathbf{p}_2 - \mathbf{p}_1) - (\mathbf{p}_1 - \mathbf{p}_0)] \\ \mathbf{P}''(1) &= n(n-1)[(\mathbf{p}_{n-2} - \mathbf{p}_{n-1}) - (\mathbf{p}_{n-1} - \mathbf{p}_n)] \end{aligned} \tag{10-48}$$

Another important property of any Bézier curve is that it lies within the convex hull (convex polygon boundary) of the control points. This follows from the properties of Bézier blending functions: They are all positive and their sum is always 1,

$$\sum_{k=0}^{n} BEZ_{k,n}(u) = 1 \tag{10-49}$$

so that any curve position is simply the weighted sum of the control-point positions. The convex-hull property for a Bézier curve ensures that the polynomial smoothly follows the control points without erratic oscillations.

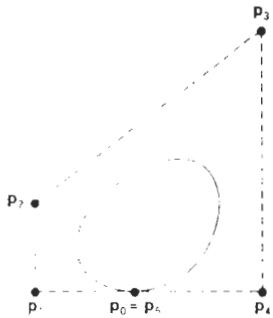


Figure 10-35
A closed Bézier curve generated by specifying the first and last control points at the same location

Design Techniques Using Bézier Curves

Closed Bézier curves are generated by specifying the first and last control points at the same position, as in the example shown in Fig. 10-35. Also, specifying multiple control points at a single coordinate position gives more weight to that position. In Fig. 10-36, a single coordinate position is input as two control points, and the resulting curve is pulled nearer to this position.

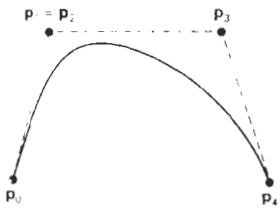


Figure 10-36
A Bézier curve can be made to pass closer to a given coordinate position by assigning multiple control points to that position

We can fit a Bézier curve to any number of control points, but this requires the calculation of polynomial functions of higher degree. When complicated curves are to be generated, they can be formed by piecing several Bézier sections of lower degree together. Piecing together smaller sections also gives us better control over the shape of the curve in small regions. Since Bézier curves pass through endpoints, it is easy to match curve sections (zero-order continuity). Also, Bézier curves have the important property that the tangent to the curve at an endpoint is along the line joining that endpoint to the adjacent control point. Therefore, to obtain first-order continuity between curve sections, we can pick control points $\mathbf{p}'_0$ and $\mathbf{p}'_1$ of a new section to be along the same straight line as control points $\mathbf{p}_{n-1}$ and $\mathbf{p}_n$ of the previous section (Fig. 10-37). When the two curve sections have the same number of control points, we obtain C^1 continuity by choosing the first control point of the new section as the last control point of the previous section and by positioning the second control point of the new section at position

$$\mathbf{p}'_1 = \mathbf{p}_n + (\mathbf{p}_n - \mathbf{p}_{n-1})$$

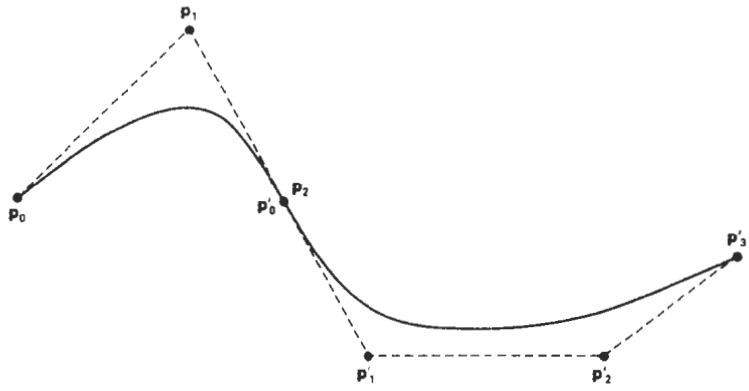


Figure 10-37
Piecewise approximation curve formed with two Bézier sections. Zero-order and first-order continuity are attained between curve sections by setting $\mathbf{p}_0' = \mathbf{p}_2$ and by making points $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_1'$ collinear.

Thus, the three control points are collinear and equally spaced.

We obtain C^2 continuity between two Bézier sections by calculating the position of the third control point of a new section in terms of the positions of the last three control points of the previous section as

$$\mathbf{p}_{n-2} + 4(\mathbf{p}_n - \mathbf{p}_{n-1})$$

Requiring second-order continuity of Bézier curve sections can be unnecessarily restrictive. This is especially true with cubic curves, which have only four control points per section. In this case, second-order continuity fixes the position of the first three control points and leaves us only one point that we can use to adjust the shape of the curve segment.

Cubic Bézier Curves

Many graphics packages provide only cubic spline functions. This gives reasonable design flexibility while avoiding the increased calculations needed with higher-order polynomials. Cubic Bézier curves are generated with four control points. The four blending functions for cubic Bézier curves, obtained by substituting $n = 3$ into Eq. 10-41 are

$$\begin{aligned} BEZ_{0,3}(u) &= (1-u)^3 \\ BEZ_{1,3}(u) &= 3u(1-u)^2 \\ BEZ_{2,3}(u) &= 3u^2(1-u) \\ BEZ_{3,3}(u) &= u^3 \end{aligned} \qquad (10\text{-}50)$$

Plots of the four cubic Bézier blending functions are given in Fig. 10-38. The form of the blending functions determine how the control points influence the shape of the curve for values of parameter u over the range from 0 to 1. At $u = 0$,

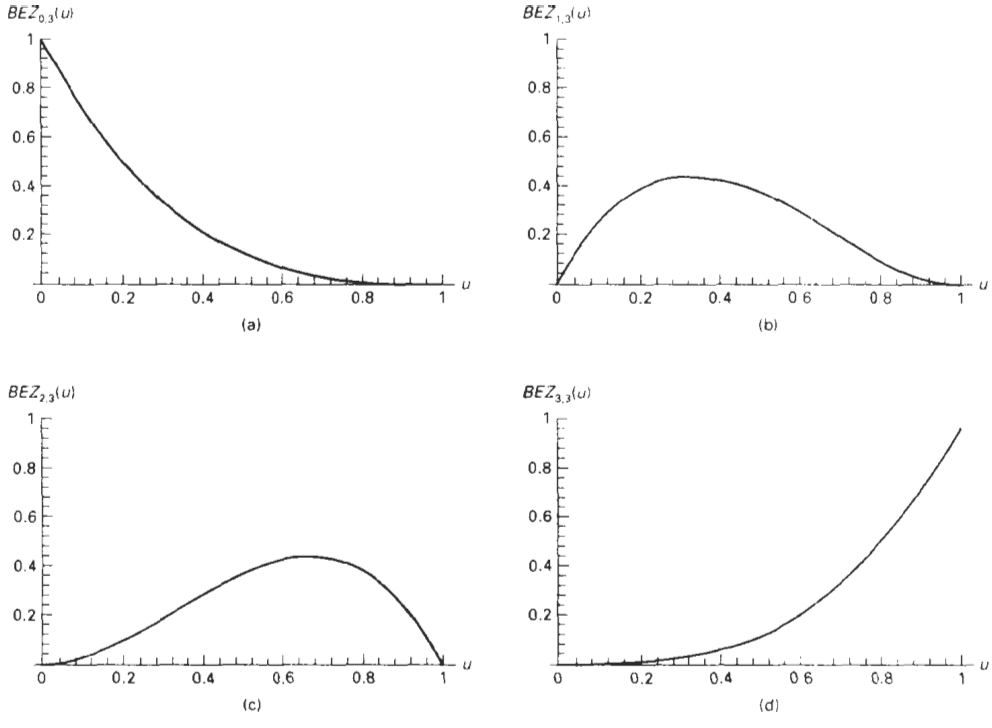


Figure 10-38
The four Bézier blending functions for cubic curves ($n = 3$)

the only nonzero blending function is $BEZ_{0,3}$, which has the value 1. At $u = 1$, the only nonzero function is $BEZ_{3,3}$, with a value of 1 at that point. Thus, the cubic Bézier curve will always pass through control points $\mathbf{p}_0$ and $\mathbf{p}_3$. The other functions, $BEZ_{1,3}$ and $BEZ_{2,3}$, influence the shape of the curve at intermediate values of parameter u, so that the resulting curve tends toward points $\mathbf{p}_1$ and $\mathbf{p}_2$. Blending function $BEZ_{1,3}$ is maximum at $u = 1/3$, and $BEZ_{2,3}$ is maximum at $u = 2/3$.

We note in Fig. 10-38 that each of the four blending functions is nonzero over the entire range of parameter u. Thus, Bézier curves do not allow for *local control* of the curve shape. If we decide to reposition any one of the control points, the entire curve will be affected.

At the end positions of the cubic Bézier curve, the parametric first derivatives (slopes) are

$$\mathbf{P}'(0) = 3(\mathbf{p}_1 - \mathbf{p}_0), \qquad \mathbf{P}'(1) = 3(\mathbf{p}_3 - \mathbf{p}_2)$$

And the parametric second derivatives are

$$\mathbf{P}''(0) = 6(\mathbf{p}_0 - 2\mathbf{p}_1 + \mathbf{p}_2), \qquad \mathbf{P}''(1) = 6(\mathbf{p}_1 - 2\mathbf{p}_2 + \mathbf{p}_3)$$

We can use these expressions for the parametric derivatives to construct piecewise curves with C^1 or C^2 continuity between sections.

By expanding the polynomial expressions for the blending functions, we can write the cubic Bézier point function in the matrix form

$$\mathbf{P}(u) = [u^3\ u^2\ u\ 1] \cdot \boldsymbol{M}_{\text{Bez}} \cdot \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} \tag{10-51}$$

where the **Bézier matrix** is

$$\mathbf{M}_{\text{Bez}} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{10-52}$$

We could also introduce additional parameters to allow adjustment of curve "tension" and "bias", as we did with the interpolating splines. But the more useful B-splines, as well as β-splines, provide this capability.

Bézier Surfaces

Two sets of orthogonal Bézier curves can be used to design an object surface by specifying by an input mesh of control points. The parametric vector function for the Bézier surface is formed as the Cartesian product of Bézier blending functions:

$$\mathbf{P}(u, v) = \sum_{j=0}^{m} \sum_{k=0}^{n} \mathbf{p}_{j,k} BEZ_{j,m}(v) BEZ_{k,n}(u) \tag{10-53}$$

with $\mathbf{p}_{j,k}$ specifying the location of the $(m + 1)$ by $(n + 1)$ control points.

Figure 10-39 illustrates two Bézier surface plots. The control points are connected by dashed lines, and the solid lines show curves of constant u and constant v. Each curve of constant u is plotted by varying v over the interval from 0 to 1, with u fixed at one of the values in this unit interval. Curves of constant v are plotted similarly.

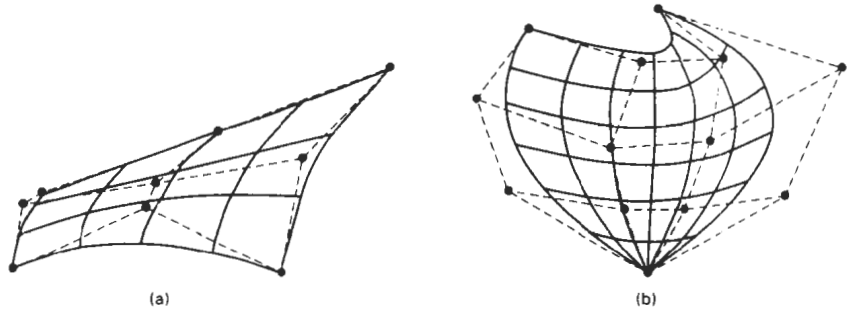


Figure 10-39
Bézier surfaces constructed for (a) $m = 3$, $n = 3$, and (b) $m = 4$, $n = 4$. Dashed lines connect the control points.

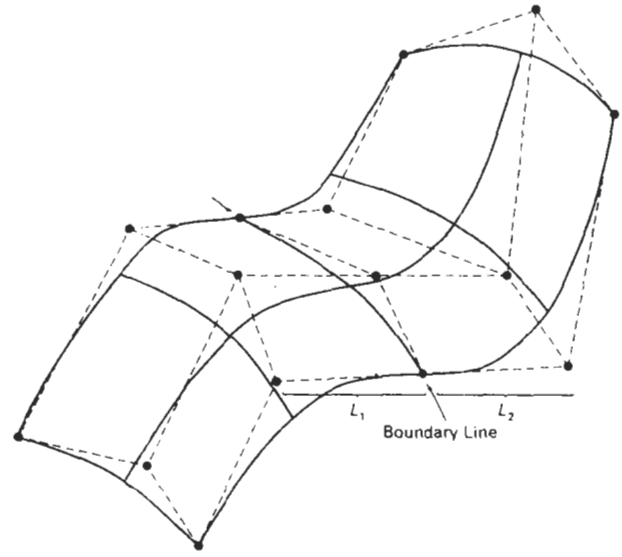


Figure 10-40
A composite Bézier surface constructed with two Bézier sections, joined at the indicated boundary line. The dashed lines connect specified control points. First-order continuity is established by making the ratio of length L_1 to length L_2 constant for each collinear line of control points across the boundary between the surface sections.

Bézier surfaces have the same properties as Bézier curves, and they provide a convenient method for interactive design applications. For each surface patch, we can select a mesh of control points in the xy "ground" plane, then we choose elevations above the ground plane for the z-coordinate values of the control points. Patches can then be pieced together using the boundary constraints.

Figure 10-40 illustrates a surface formed with two Bézier sections. As with curves, a smooth transition from one section to the other is assured by establishing both zero-order and first-order continuity at the boundary line. Zero-order continuity is obtained by matching control points at the boundary. First-order continuity is obtained by choosing control points along a straight line across the boundary and by maintaining a constant ratio of collinear line segments for each set of specified control points across section boundaries.

10-9
B-SPLINE CURVES AND SURFACES

These are the most widely used class of approximating splines. **B-splines** have two advantages over Bézier splines: (1) the degree of a B-spline polynomial can be set independently of the number of control points (with certain limitations), and (2) B-splines allow local control over the shape of a spline curve or surface. The trade-off is that B-splines are more complex than Bézier splines.

B-Spline Curves

We can write a general expression for the calculation of coordinate positions along a B-spline curve in a blending-function formulation as

$$\mathbf{P}(u) = \sum_{k=0}^{n} \mathbf{p}_k B_{k,d}(u), \qquad u_{\min} \le u \le u_{\max}, \qquad 2 \le d \le n+1 \tag{10-54}$$

where the $\mathbf{p}_k$ are an input set of $n + 1$ control points. There are several differences between this B-spline formulation and that for Bézier splines. The range of parameter u now depends on how we choose the B-spline parameters. And the B-spline blending functions $B_{k,d}$ are polynomials of degree $d - 1$, where parameter d can be chosen to be any integer value in the range from 2 up to the number of control points, $n + 1$. (Actually, we can also set the value of d at 1, but then our "curve" is just a point plot of the control points.) Local control for B-splines is achieved by defining the blending functions over subintervals of the total range of u.

Blending functions for B-spline curves are defined by the Cox–deBoor recursion formulas:

$$\begin{aligned} B_{k,1}(u) &= \begin{cases} 1, & \text{if } u_k \le u < u_{k+1} \\ 0, & \text{otherwise} \end{cases} \\ B_{k,d}(u) &= \frac{u - u_k}{u_{k+d-1} - u_k} B_{k,d-1}(u) + \frac{u_{k+d} - u}{u_{k+d} - u_{k+1}} B_{k+1,d-1}(u) \end{aligned} \tag{10-55}$$

where each blending function is defined over d subintervals of the total range of u. The selected set of subinterval endpoints u_j is referred to as a **knot vector**. We can choose any values for the subinterval endpoints satisfying the relation $u_j \le u_{j+1}$. Values for $u_{\min}$ and $u_{\max}$ then depend on the number of control points we select, the value we choose for parameter d, and how we set up the subintervals (knot vector). Since it is possible to choose the elements of the knot vector so that the denominators in the previous calculations can have a value of 0, this formulation assumes that any terms evaluated as 0/0 are to be assigned the value 0.

Figure 10-41 demonstrates the local-control characteristics of B-splines. In addition to local control, B-splines allow us to vary the number of control points used to design a curve without changing the degree of the polynomial. Also, any number of control points can be added or modified to manipulate curve shapes. Similarly, we can increase the number of values in the knot vector to aid in curve design. When we do this, however, we also need to add control points since the size of the knot vector depends on parameter n.

B-spline curves have the following properties:

- The polynomial curve has degree $d - 1$ and C^{d-2} continuity over the range of u.
- For $n + 1$ control points, the curve is described with $n + 1$ blending functions.
- Each blending function $B_{k,d}$ is defined over d subintervals of the total range of u, starting at knot value u_k.
- The range of parameter u is divided into $n + d$ subintervals by the $n + d + 1$ values specified in the knot vector.

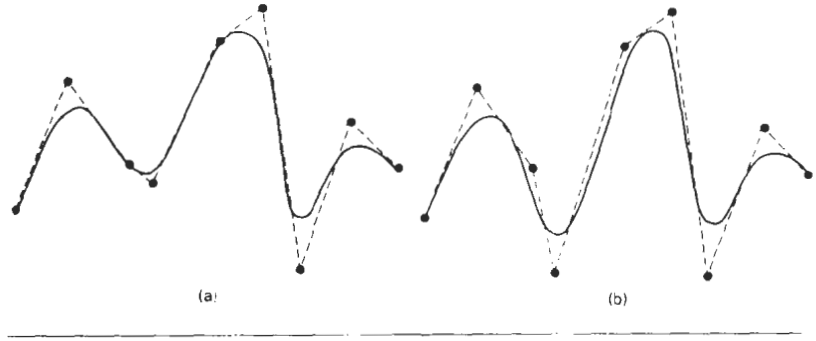


Figure 10-41
Local modification of a B-spline curve. Changing one of the control points in (a) produces curve (b), which is modified only in the neighborhood of the altered control point.

- With knot values labeled as $\{u_0, u_1, \ldots, u_{n+d}\}$, the resulting B-spline curve is defined only in the interval from knot value u_{d-1} up to knot value u_{n+1}.
- Each section of the spline curve (between two successive knot values) is influenced by d control points.
- Any one control point can affect the shape of at most d curve sections.

In addition, a B-spline curve lies within the convex hull of at most $d + 1$ control points, so that B-splines are tightly bound to the input positions. For any value of u in the interval from knot value u_{d-1} to u_{n+1}, the sum over all basis functions is 1:

$$\sum_{k=0}^{n} B_{k,d}(u) = 1 \tag{10-56}$$

Given the control-point positions and the value of parameter d, we then need to specify the knot values to obtain the blending functions using the recurrence relations 10-55. There are three general classifications for knot vectors: uniform, open uniform, and nonuniform. B-splines are commonly described according to the selected knot-vector class.

Uniform, Periodic B-Splines

When the spacing between knot values is constant, the resulting curve is called a **uniform** B-spline. For example, we can set up a uniform knot vector as

$$\{-1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0\}$$

Often knot values are normalized to the range between 0 and 1, as in

$$\{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$$

It is convenient in many applications to set up uniform knot values with a separation of 1 and a starting value of 0. The following knot vector is an example of this specification scheme.

$$\{0, 1, 2, 3, 4, 5, 6, 7\}$$

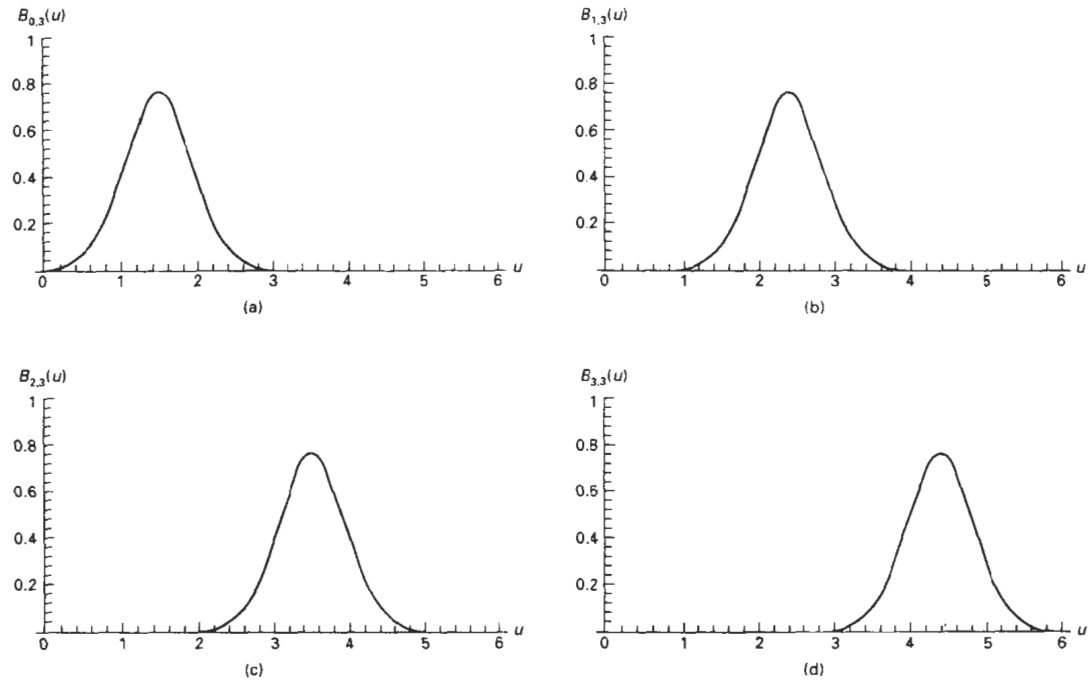


Figure 10-42
Periodic B-spline blending functions for $n = d = 3$ and a uniform, integer knot vector.

Uniform B-splines have **periodic** blending functions. That is, for given values of n and d, all blending functions have the same shape. Each successive blending function is simply a shifted version of the previous function:

$$B_{k,d}(u) = B_{k+1,d}(u+\Delta u) = B_{k-2,d}(u + 2\,\Delta u) \qquad (10\text{-}57)$$

where Δu is the interval between adjacent knot values. Figure 10-42 shows the quadratic, uniform B-spline blending functions generated in the following example for a curve with four control points.

Example 10-1 Uniform, Quadratic B-Splines

To illustrate the calculation of B-spline blending functions for a uniform, integer knot vector, we select parameter values $d = n = 3$. The knot vector must then contain $n + d + 1 = 7$ knot values:

$$\{0, 1, 2, 3, 4, 5, 6\}$$

and the range of parameter u is from 0 to 6, with $n + d = 6$ subintervals.

Each of the four blending functions spans $d = 3$ subintervals of the total range of u. Using the recurrence relations 10-55, we obtain the first blending function as

$$B_{0,3}(u) = \begin{cases} \frac{1}{2}u^2, & \text{for } 0 \le u < 1 \\ \frac{1}{2}u(2-u) + \frac{1}{2}(u-1)(3-u), & \text{for } 1 \le u < 2 \\ \frac{1}{2}(3-u)^2, & \text{for } 2 \le u < 3 \end{cases}$$

We obtain the next periodic blending function using relationship 10-57, substituting $u - 1$ for u in $B_{0,3}$, and shifting the starting positions up by 1:

$$B_{1,3}(u) = \begin{cases} \frac{1}{2}(u-1)^2, & \text{for } 1 \le u < 2 \\ \frac{1}{2}(u-1)(3-u) + \frac{1}{2}(u-2)(4-u), & \text{for } 2 \le u < 3 \\ \frac{1}{2}(4-u)^2, & \text{for } 3 \le u < 4 \end{cases}$$

Similarly, the remaining two periodic functions are obtained by successively shifting $B_{1,3}$ to the right:

$$B_{2,3}(u) = \begin{cases} \frac{1}{2}(u-2)^2, & \text{for } 2 \le u < 3 \\ \frac{1}{2}(u-2)(4-u) + \frac{1}{2}(u-3)(5-u), & \text{for } 3 \le u < 4 \\ \frac{1}{2}(5-u)^2, & \text{for } 4 \le u < 5 \end{cases}$$

$$B_{3,3}(u) = \begin{cases} \frac{1}{2}(u-3)^2, & \text{for } 3 \le u < 4 \\ \frac{1}{2}(u-3)(5-u) + \frac{1}{2}(u-4)(6-u), & \text{for } 4 \le u < 5 \\ \frac{1}{2}(6-u)^2, & \text{for } 5 \le u < 6 \end{cases}$$

A plot of the four periodic, quadratic blending functions is given in Fig. 10-42, which demonstrates the local feature of B-splines. The first control point is multiplied by blending function $B_{0,3}(u)$. Therefore, changing the position of the first control point only affects the shape of the curve up to $u = 3$. Similarly, the last control point influences the shape of the spline curve in the interval where $B_{3,3}$ is defined.

Figure 10-42 also illustrates the limits of the B-spline curve for this example. All blending functions are present in the interval from $u_{d-1} = 2$ to $u_{n+1} = 4$. Below 2 and above 4, not all blending functions are present. This is the range of the poly-



Figure 10-43
Quadratic, periodic B-spline fitted to four control points in the xy plane.

nomial curve, and the interval in which Eq. 10-56 is valid. Thus, the sum of all blending functions is 1 within this interval. Outside this interval, we cannot sum all blending functions, since they are not all defined below 2 and above 4.

Since the range of the resulting polynomial curve is from 2 to 4, we can determine the starting and ending positions of the curve by evaluating the blending functions at these points to obtain

$$\mathbf{P}_{start} = \tfrac{1}{2}(\mathbf{p}_0 + \mathbf{p}_1), \qquad \mathbf{P}_{end} = \tfrac{1}{2}(\mathbf{p}_2 + \mathbf{p}_3)$$

Thus, the curve starts at the midposition between the first two control points and ends at the midposition between the last two control points.

We can also determine the parametric derivatives at the starting and ending positions of the curve. Taking the derivatives of the blending functions and substituting the endpoint values for parameter u, we find that

$$\mathbf{P}'_{start} = \mathbf{p}_1 - \mathbf{p}_0, \qquad \mathbf{P}'_{end} = \mathbf{p}_3 - \mathbf{p}_2$$

The parametric slope of the curve at the start position is parallel to the line joining the first two control points, and the parametric slope at the end of the curve is parallel to the line joining the last two control points.

An example plot of the quadratic periodic B-spline curve is given in Figure 10-43 for four control points selected in the xy plane.

In the preceding example, we noted that the quadratic curve starts between the first two control points and ends at a position between the last two control points. This result is valid for a quadratic, periodic B-spline fitted to any number of distinct control points. In general, for higher-order polynomials, the start and end positions are each weighted averages of $d - 1$ control points. We can pull a spline curve closer to any control-point position by specifying that position multiple times.

General expressions for the boundary conditions for periodic B-splines can be obtained by reparameterizing the blending functions so that parameter u is mapped onto the unit interval from 0 to 1. Beginning and ending conditions are then obtained at $u = 0$ and $u = 1$.

Cubic, Periodic B-Splines

Since cubic, periodic B-splines are commonly used in graphics packages, we consider the formulation for this class of splines. Periodic splines are particularly useful for generating certain closed curves. For example, the closed curve in Fig. 10-44 can be generated in sections by cyclically specifying four of the six control

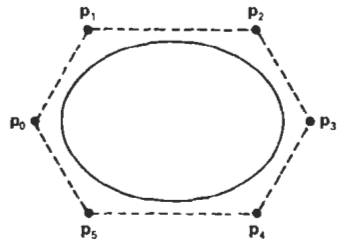


Figure 10-44
A closed, periodic, piecewise, cubic B-spline constructed with cyclic specification of the six control points.

points shown at each step. If any three consecutive control points are identical, the curve passes through that coordinate position.

For cubics, $d = 4$ and each blending function spans four subintervals of the total range of u. If we are to fit the cubic to four control points, then we could use the integer knot vector

$$\{0, 1, 2, 3, 4, 5, 6, 7\}$$

and recurrence relations 10-55 to obtain the periodic blending functions, as we did in the last section for quadratic periodic B-splines.

In this section, we consider an alternate formulation for periodic cubic B-splines. We start with the boundary conditions and obtain the blending functions normalized to the interval $0 \le u \le 1$. Using this formulation, we can also easily obtain the characteristic matrix. The boundary conditions for periodic cubic B-splines with four consecutive control points, labeled $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$, are

$$\begin{aligned} \mathbf{P}(0) &= \tfrac{1}{6}(\mathbf{p}_0 + 4\mathbf{p}_1 + \mathbf{p}_2) \\ \mathbf{P}(1) &= \tfrac{1}{6}(\mathbf{p}_1 + 4\mathbf{p}_2 + \mathbf{p}_3) \\ \mathbf{P}'(0) &= \tfrac{1}{2}(\mathbf{p}_2 - \mathbf{p}_0) \\ \mathbf{P}'(1) &= \tfrac{1}{2}(\mathbf{p}_3 - \mathbf{p}_1) \end{aligned} \qquad (10\text{-}58)$$

These boundary conditions are similar to those for cardinal splines: Curve sections are defined with four control points, and parametric derivatives (slopes) at the beginning and end of each curve section are parallel to the chords joining adjacent control points. The B-spline curve section starts at a position near $\mathbf{p}_1$ and ends at a position near $\mathbf{p}_2$.

A matrix formulation for a cubic periodic B-splines with four control points can then be written as

$$\mathbf{P}(u) = [u^3\ u^2\ u\ 1] \cdot \mathbf{M}_B \cdot \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} \qquad (10\text{-}59)$$

where the B-spline matrix for periodic cubic polynomials is

$$\mathbf{M}_B = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \tag{10-60}$$

This matrix can be obtained by solving for the coefficients in a general cubic polynomial expression using the specified four boundary conditions.

We can also modify the B-spline equations to include a tension parameter t (as in cardinal splines). The periodic, cubic B-spline with tension matrix then has the form

$$\mathbf{M}_{Bt} = \frac{1}{6}\begin{bmatrix} -t & 12-9t & 9t-12 & t \\ 3t & 12t-18 & 18-15t & 0 \\ -3t & 0 & 3t & 0 \\ t & 6-2t & t & 0 \end{bmatrix} \tag{10-61}$$

which reduces to M_B when $t = 1$.

We obtain the periodic, cubic B-spline blending functions over the parameter range from 0 to 1 by expanding the matrix representation into polynomial form. For example, for the tension value $t = 1$, we have

$$\begin{aligned} B_{0,3}(u) &= \tfrac{1}{6}(1-u)^3, \qquad 0 \le u \le 1 \\ B_{1,3}(u) &= \tfrac{1}{6}(3u^3 - 6u^2 + 4) \\ B_{2,3}(u) &= \tfrac{1}{6}(-3u^3 + 3u^2 + 3u + 1) \\ B_{3,3}(u) &= \tfrac{1}{6}u^3 \end{aligned} \tag{10-62}$$

Open Uniform B-Splines

This class of B-splines is a cross between uniform B-splines and nonuniform B-splines. Sometimes it is treated as a special type of uniform B-spline, and sometimes it is considered to be in the nonuniform B-spline classification. For the **open uniform** B-splines, or simply **open** B-splines, the knot spacing is uniform except at the ends where knot values are repeated d times.

Following are two examples of open uniform, integer knot vectors, each with a starting value of 0:

$$\begin{aligned} &\{0, 0, 1, 2, 3, 3,\}, && \text{for } d = 2 \text{ and } n = 3 \\ &\{0, 0, 0, 0, 1, 2, 2, 2, 2,\}, && \text{for } d = 4 \text{ and } n = 4 \end{aligned}$$

We can normalize these knot vectors to the unit interval from 0 to 1:

$$\begin{aligned} &\{0, 0, 0.33, 0.67, 1, 1,\}, && \text{for } d = 2 \text{ and } n = 3 \\ &\{0, 0, 0, 0, 0.5, 1, 1, 1, 1\}, && \text{for } d = 4 \text{ and } n = 4 \end{aligned}$$

For any values of parameters d and n, we can generate an open uniform knot vector with integer values using the calculations

$$u_j = \begin{cases} 0, & \text{for } 0 \le j < d \\ j - d + 1, & \text{for } d \le j \le n \\ n - d + 2, & \text{for } j > n \end{cases} \tag{10-63}$$

for values of j ranging from 0 to $n + d$. With this assignment, the first d knots are assigned the value 0, and the last d knots have the value $n - d + 2$.

Open uniform B-splines have characteristics that are very similar to Bézier splines. In fact, when $d = n + 1$ (degree of the polynomial is n) open B-splines reduce to Bézier splines, and all knot values are either 0 or 1. For example, with a cubic, open B-spline ($d = 4$) and four control points, the knot vector is

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

The polynomial curve for an open B-spline passes through the first and last control points. Also, the slope of the parametric curves at the first control point is parallel to the line connecting the first two control points. And the parametric slope at the last control point is parallel to the line connecting the last two control points. So geometric constraints for matching curve sections are the same as for Bézier curves.

As with Bézier curves, specifying multiple control points at the same coordinate position pulls any B-spline curve closer to that position. Since open B-splines start at the first control point and end at the last specified control point, closed curves are generated by specifying the first and last control points at the same position.

Example 10-2 Open Uniform, Quadratic B-Splines

From conditions 10-63 with $d = 3$ and $n = 4$ (five control points), we obtain the following eight values for the knot vector:

$$\{u_0, u_1, u_2, u_3, u_4, u_5, u_6, u_7\} = \{0, 0, 0, 1, 2, 3, 3, 3\}$$

The total range of u is divided into seven subintervals, and each of the five blending functions $B_{k,3}$ is defined over three subintervals, starting at knot position u_k. Thus, $B_{0,3}$ is defined from $u_0 = 0$ to $u_3 = 1$, $B_{1,3}$ is defined from $u_1 = 0$ to $u_4 = 2$, and $B_{4,3}$ is defined from $u_4 = 2$ to $u_7 = 3$. Explicit polynomial expressions are obtained for the blending functions from recurrence relations 10-55 as

$$B_{0,3}(u) = (1 - u)^2, \qquad 0 \le u < 1$$

$$B_{1,3}(u) = \begin{cases} \frac{1}{2}u(4 - 3u), & 0 \le u < 1 \\ \frac{1}{2}(2 - u)^2, & 1 \le u < 2 \end{cases}$$

$$B_{2,3}(u) = \begin{cases} \frac{1}{2}u^2, & 0 \le u < 1 \\ \frac{1}{2}u(2-u) + \frac{1}{2}(u-1)(3-u), & 1 \le u < 2 \\ \frac{1}{2}(3-u)^2, & 2 \le u < 3 \end{cases}$$

$$B_{3,3}(u) = \begin{cases} \frac{1}{2}(u-1)^2, & 1 \le u < 2 \\ \frac{1}{2}(3-u)(3u-5), & 2 \le u < 3 \end{cases}$$

$$B_{4,3}(u) = (u-2)^2, \qquad 2 \le u < 3$$

Figure 10-45 shows the shape of the these five blending functions. The local features of B-splines are again demonstrated. Blending function $B_{0,3}$ is nonzero only in the subinterval from 0 to 1, so the first control point influences the curve only in this interval. Similarly, function $B_{4,3}$ is zero outside the interval from 2 to 3, and the position of the last control point does not affect the shape of the beginning and middle parts of the curve.

Matrix formulations for open B-splines are not as conveniently generated as they are for periodic, uniform B-splines. This is due to the multiplicity of knot values at the beginning and end of the knot vector.

Nonuniform B-Splines

For this class of splines, we can specify any values and intervals for the knot vector. With **nonuniform** B-splines, we can choose multiple internal knot values and unequal spacing between the knot values. Some examples are

$$\{0, 1, 2, 3, 3, 4\}$$
$$\{0, 2, 2, 3, 3, 6\}$$
$$\{0, 0, 0, 1, 1, 3, 3, 3\}$$
$$\{0, 0.2, 0.6, 0.9, 1.0\}$$

Nonuniform B-splines provide increased flexibility in controlling a curve shape. With unequally spaced intervals in the knot vector, we obtain different shapes for the blending functions in different intervals, which can be used to adjust spline shapes. By increasing knot multiplicity, we produce subtle variations in curve shape and even introduce discontinuities. Multiple knot values also reduce the continuity by 1 for each repeat of a particular value.

We obtain the blending functions for a nonuniform B-spline using methods similar to those discussed for uniform and open B-splines. Given a set of $n + 1$ control points, we set the degree of the polynomial and select the knot values. Then, using the recurrence relations, we could either obtain the set of blending functions or evaluate curve positions directly for the display of the curve. Graphics packages often restrict the knot intervals to be either 0 or 1 to reduce computations. A set of characteristic matrices then can be stored and used to compute

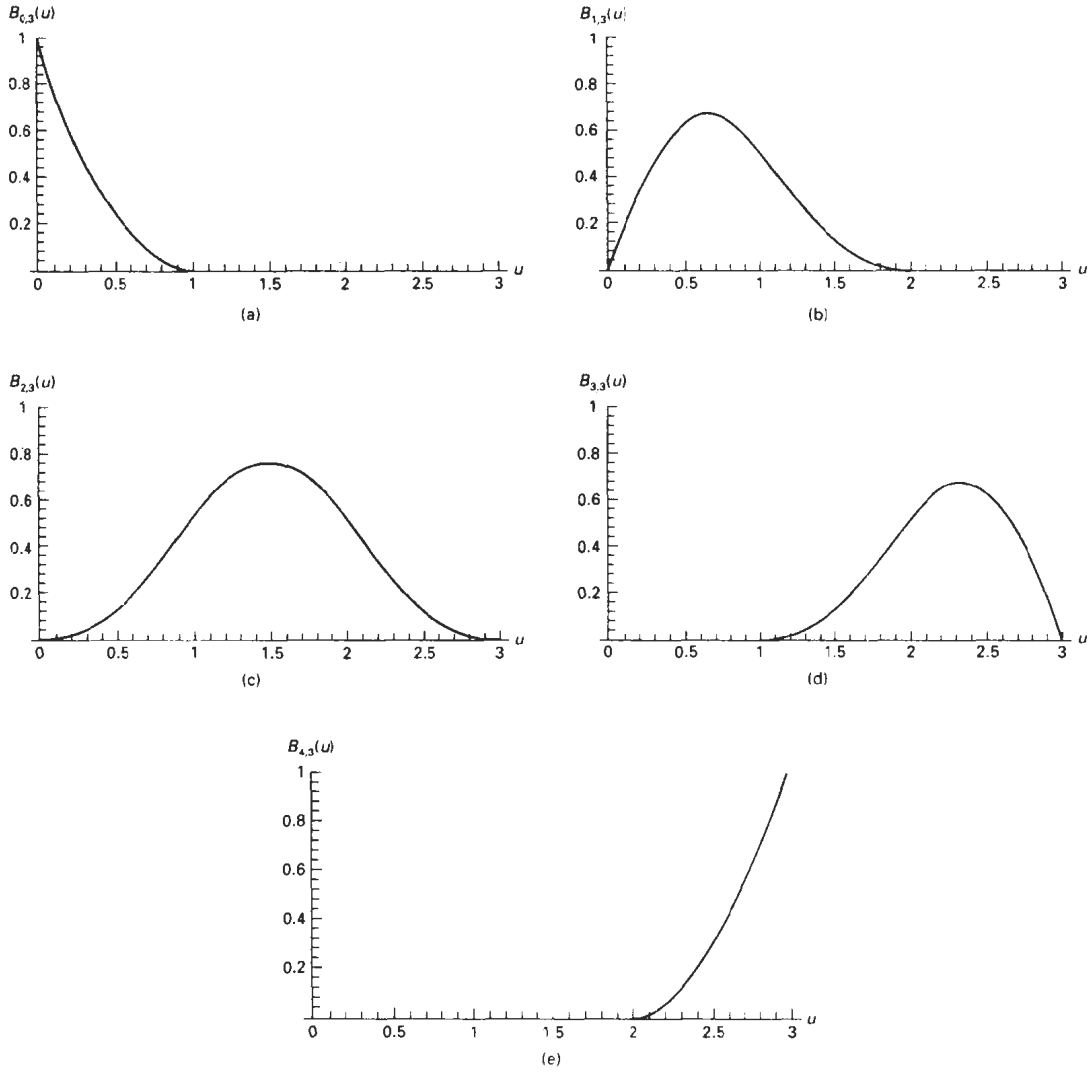


Figure 10-45
Open, uniform B-spline blending functions for $n = 4$ and $d = 3$.

values along the spline curve without evaluating the recurrence relations for each curve point to be plotted.

B-Spline Surfaces

Formulation of a B-spline surface is similar to that for Bézier splines. We can obtain a vector point function over a B-spline surface using the Cartesian product of B-spline blending functions in the form

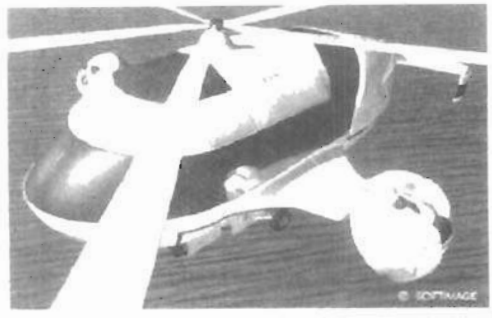


Figure 10-46
A prototype helicopter, designed and modeled by Daniel Langlois of SOFTIMAGE, Inc., Montreal, using 180,000 B-spline surface patches. The scene was then rendered using ray tracing, bump mapping, and reflection mapping. (*Courtesy of Silicon Graphics, Inc.*)

$$\mathbf{P}(u, v) = \sum_{k_1=0}^{n1} \sum_{k_2=0}^{n2} \mathbf{p}_{k_1,k_2} B_{k_1,d_1}(u) B_{k_2,d_2}(v) \tag{10-64}$$

where the vector values for $\mathbf{p}_{k_1,k_2}$ specify positions of the $(n_1 + 1)$ by $(n_2 + 1)$ control points.

B-spline surfaces exhibit the same properties as those of their component B-spline curves. A surface can be constructed from selected values for parameters d_1 and d_2 (which determine the polynomial degrees to be used) and from the specified knot vector. Figure 10-46 shows an object modeled with B-spline surfaces.

10-10
BETA-SPLINES

A generalization of B-splines are the **beta-splines**, also referred to as **β-splines**, that are formulated by imposing geometric continuity conditions on the first and second parametric derivatives. The continuity parameters for beta-splines are called *β parameters*.

Beta-Spline Continuity Conditions

For a specified knot vector, we can designate the spline sections to the left and right of a particular knot u_j with the position vectors $\mathbf{P}_{j-1}(u)$ and $\mathbf{P}_j(u)$ (Fig. 10-47). Zero-order continuity (*positional continuity*), G^0, at u_j is obtained by requiring

$$\mathbf{P}_{j-1}(u_j) = \mathbf{P}_j(u_j) \tag{10-65}$$

First-order continuity (*unit tangent continuity*), G^1, is obtained by requiring tangent vectors to be proportional:

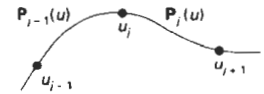


Figure 10-47
Position vectors along curve sections to the left and right of knot u_j.

$$\beta_1 \mathbf{P}'_{j-1}(u_j) = \mathbf{P}'_j(u_j), \qquad \beta_1 > 0 \tag{10-66}$$

Here, parametric first derivatives are proportional, and the unit tangent vectors are continuous across the knot.

Second-order continuity (*curvature vector continuity*), G^2, is imposed with the condition

$$\beta_1^2 \mathbf{P}''_{j-1}(u_j) + \beta_2 \mathbf{P}'_{j-1}(u_j) = \mathbf{P}''_j(u_j) \tag{10-67}$$

where β_2 can be assigned any real number, and $\beta_1 > 0$. The curvature vector provides a measure of the amount of bending of the curve at position u_j. When $\beta_1 = 1$ and $\beta_2 = 0$, beta-splines reduce to B-splines.

Parameter β_1 is called the *bias parameter* since it controls the skewness of the curve. For $\beta_1 > 1$, the curve tends to flatten to the right in the direction of the unit tangent vector at the knots. For $0 < \beta_1 < 1$, the curve tends to flatten to the left. The effect of β_1 on the shape of the spline curve is shown in Fig. 10-48.

Parameter β_2 is called the *tension parameter* since it controls how tightly or loosely the spline fits the control graph. As β_2 increases, the curve approaches the shape of the control graph, as shown in Fig. 10-49.

Cubic, Periodic Beta-Spline Matrix Representation

Applying the beta-spline boundary conditions to a cubic polynomial with a uniform knot vector, we obtain the following matrix representation for a periodic beta-spline:

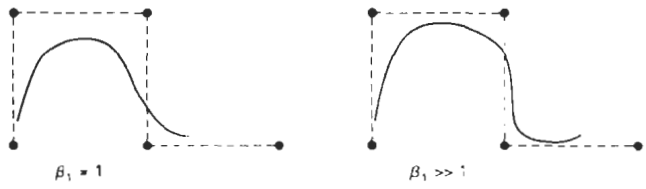


Figure 10-48
Effect of parameter β_1 on the shape of a beta-spline curve.

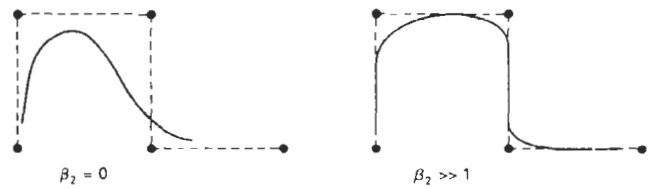


Figure 10-49
Effect of parameter β_2 on the shape of a beta-spline curve.

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

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

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

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

10-11
RATIONAL SPLINES

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

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

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

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

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

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

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

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

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

$$\omega_0 = \omega_2 = 1$$
$$\omega_1 = \frac{r}{1 - r}, \qquad 0 \le r < 1 \tag{10-70}$$

and the rational B-spline representation is

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

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

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

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

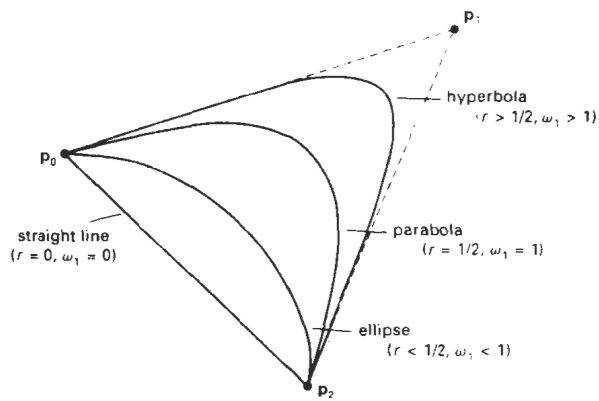


Figure 10-50
Conic sections generated with various values of the rational-spline weighting factor ω_1.

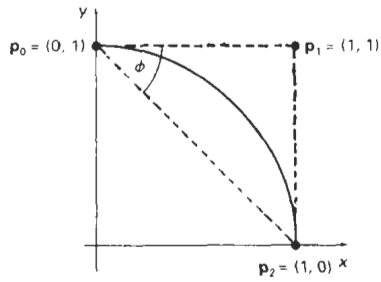


Figure 10-51
A circular arc in the first quadrant of the xy plane.

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

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

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

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

10-12
CONVERSION BETWEEN SPLINE REPRESENTATIONS

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

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

$$\mathbf{P}(u) = \mathbf{U} \cdot \mathbf{M}_{\text{spline1}} \cdot \mathbf{M}_{\text{geom1}} \tag{10-72}$$

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

$$\mathbf{P}(u) = \mathbf{U} \cdot \mathbf{M}_{spline2} \cdot \mathbf{M}_{geom2} \tag{10-73}$$

or

$$\mathbf{U} \cdot \mathbf{M}_{spline2} \cdot \mathbf{M}_{geom2} = \mathbf{U} \cdot \mathbf{M}_{spline1} \cdot \mathbf{M}_{geom1}$$

Solving for $\mathbf{M}_{geom2}$, we have

$$\begin{aligned} \mathbf{M}_{geom2} &= \mathbf{M}_{spline2}^{-1} \cdot \mathbf{M}_{spline1} \cdot \mathbf{M}_{geom1} \\ &= \mathbf{M}_{s1,s2} \cdot \mathbf{M}_{geom1} \end{aligned} \tag{10-74}$$

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

$$\mathbf{M}_{s1,s2} = \mathbf{M}_{spline2}^{-1} \cdot \mathbf{M}_{spline1} \tag{10-75}$$

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

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

$$\begin{aligned} \mathbf{M}_{B,Bez} &= \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}^{-1} \cdot \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \\ &= 1/6 \begin{bmatrix} 1 & 4 & 1 & 0 \\ 0 & 4 & 2 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix} \end{aligned} \tag{10-76}$$

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

$$\begin{aligned} \mathbf{M}_{Bez,B} &= \begin{bmatrix} -1/6 & 1/2 & -1/2 & 1/6 \\ 1/2 & -1 & 1/2 & 0 \\ -1/2 & 0 & 1/2 & 0 \\ 1/6 & 2/3 & 1/6 & 0 \end{bmatrix}^{-1} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 6 & -7 & 2 & 0 \\ 0 & 2 & -1 & 0 \\ 0 & -1 & 2 & 0 \\ 0 & 2 & -7 & 6 \end{bmatrix} \end{aligned} \tag{10-77}$$

10-13 DISPLAYING SPLINE CURVES AND SURFACES

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

Horner's Rule

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

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

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

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

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

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

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

Forward-Difference Calculations

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

$$x_{k+1} = x_k + \Delta x_k \tag{10-80}$$

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

$$u_{k+1} = u_k + \delta, \qquad k = 0, 1, 2, \ldots \tag{10-81}$$

and $u_0 = 0$.

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

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

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

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

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

The forward difference now evaluates to

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

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

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

where the second forward difference is the linear function

$$\Delta^2 x_k = 6a_x \delta^2 u_k + 6a_x \delta^3 + 2b_x \delta^2 \tag{10-86}$$

Repeating this process once more, we can write

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

with the third forward difference as the constant

$$\Delta^3 x_k = 6a_x \delta^3 \tag{10-88}$$

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

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

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

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

Subdivision Methods

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

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

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

Figure 10-52 shows the first step in a recursive subdivision of a cubic Bézier curve section. Positions along the Bézier curve are described with the parametric point function $\mathbf{P}(u)$ for $0 \le u \le 1$. At the first subdivision step, we use the halfway point $\mathbf{P}(0.5)$ to divide the original curve into two sections. The first section is then described with the point function $\mathbf{P}_1(s)$, and the section is described with $\mathbf{P}_2(t)$, where

$$\begin{aligned} s &= 2u, \qquad \text{for } 0 \le u \le 0.5 \\ t &= 2u-1, \qquad \text{for } 0.5 \le u \le 1 \end{aligned} \tag{10-90}$$

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

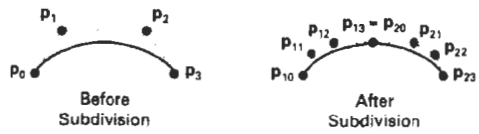


Figure 10-52
Subdividing a cubic Bézier curve section into two sections, each with four control points.

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

$$\begin{aligned}
\mathbf{p}_{1,0} &= \mathbf{p}_0 \\
\mathbf{p}_{1,1} &= \tfrac{1}{2}(\mathbf{p}_0 + \mathbf{p}_1) \\
\mathbf{p}_{1,2} &= \tfrac{1}{4}(\mathbf{p}_0 + 2\mathbf{p}_1 + \mathbf{p}_2) \\
\mathbf{p}_{1,3} &= \tfrac{1}{8}(\mathbf{p}_0 + 3\mathbf{p}_1 + 3\mathbf{p}_2 + \mathbf{p}_3)
\end{aligned} \tag{10-91}$$

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

$$\begin{aligned}
\mathbf{p}_{2,0} &= \tfrac{1}{8}(\mathbf{p}_0 + 3\mathbf{p}_1 + 3\mathbf{p}_2 + \mathbf{p}_3) \\
\mathbf{p}_{2,1} &= \tfrac{1}{4}(\mathbf{p}_1 + 2\mathbf{p}_2 + \mathbf{p}_3) \\
\mathbf{p}_{2,2} &= \tfrac{1}{2}(\mathbf{p}_2 + \mathbf{p}_3) \\
\mathbf{p}_{2,3} &= \mathbf{p}_3
\end{aligned} \tag{10-92}$$

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

$$\begin{aligned}
\mathbf{p}_{1,0} &= \mathbf{p}_0 \\
\mathbf{p}_{1,1} &= \tfrac{1}{2}(\mathbf{p}_0 + \mathbf{p}_1) \\
\mathbf{T} &= \tfrac{1}{2}(\mathbf{p}_1 + \mathbf{p}_2) \\
\mathbf{p}_{1,2} &= \tfrac{1}{2}(\mathbf{p}_{1,1} + \mathbf{T}) \\
\mathbf{p}_{2,3} &= \mathbf{p}_3 \\
\mathbf{p}_{2,2} &= \tfrac{1}{2}(\mathbf{p}_2 + \mathbf{p}_3) \\
\mathbf{p}_{2,1} &= \tfrac{1}{2}(\mathbf{T} + \mathbf{p}_{2,2}) \\
\mathbf{p}_{2,0} &= \tfrac{1}{2}(\mathbf{p}_{1,2} + \mathbf{p}_{2,1}) \\
\mathbf{p}_{1,3} &= \mathbf{p}_{2,0}
\end{aligned} \tag{10-93}$$

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

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

$$\mathbf{p}_{1,k} = \frac{1}{2^k} \sum_{i=0}^{k} C(k, i)\mathbf{p}_i, \qquad k = 0, 1, 2, \ldots, n$$
$$\mathbf{p}_{2,k} = \frac{1}{2^{n-k}} \sum_{i=k}^{n} C(n-k, n-i)\,\mathbf{p}_i \tag{10-94}$$

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

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

10-14 SWEEP REPRESENTATIONS

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

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



Figure 10-53
Constructing a solid with a translational sweep. Translating the control points of the periodic spline curve in (a) generates the solid shown in (b), whose surface can be described with point function $\mathbf{P}(u,v)$.

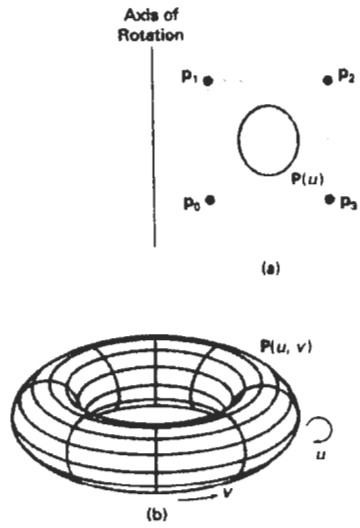


Figure 10-54
Constructing a solid with a rotational sweep. Rotating the control points of the periodic spline curve in (a) about the given rotation axis generates the solid shown in (b), whose surface can be described with point function $P(u,v)$.

sweep by moving the control points $\mathbf{p}_0$ through $\mathbf{p}_3$ a set distance along a straight-line path perpendicular to the plane of the cross section. At intervals along this path, we replicate the cross-sectional shape and draw a set of connecting lines in the direction of the sweep to obtain the wireframe representation shown in Fig. 10-53(b).

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

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

10-15
CONSTRUCTIVE SOLID-GEOMETRY METHODS

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

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

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

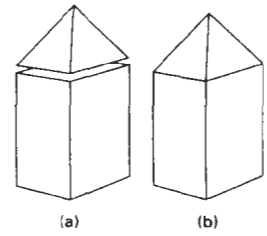


Figure 10-55
Combining two objects (a) with a union operation produces a single, composite solid object (b).

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

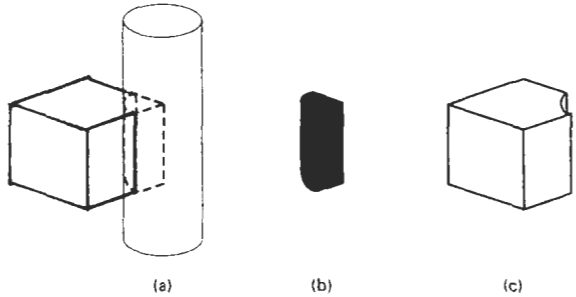


Figure 10-56
(a) Two overlapping objects. (b) A wedge-shaped CSG object formed with the intersection operation. (c) A CSG object formed with a difference operation by subtracting the overlapping volume of the cylinder from the block volume.

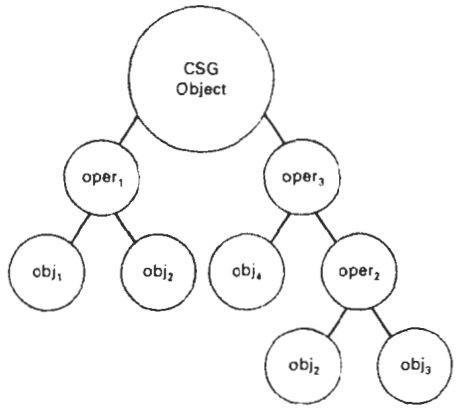


Figure 10-57
A CSG tree representation for an object.

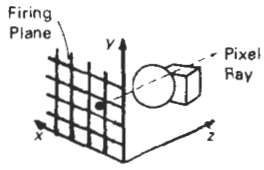
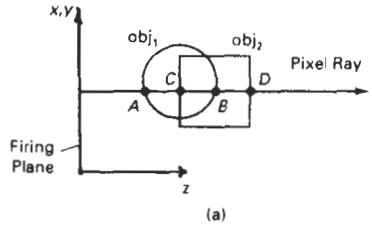


Figure 10-58
Implementing CSG operations using ray casting.



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

(b)

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

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

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

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

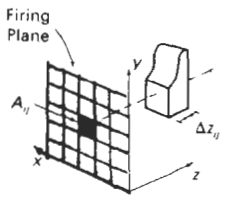


Figure 10-60
Determining object volume along a ray path for a small area A_{ij} on the firing plane.

$$V_{ij} \approx A_{ij} \Delta z_{ij} \tag{10-95}$$

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

$$V \approx \sum_{i,j} V_{ij} \tag{10-96}$$

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

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

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

$$m \approx \sum_{m,i,j} M_{ij} \tag{10-98}$$

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

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

10-16 OCTREES

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

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

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

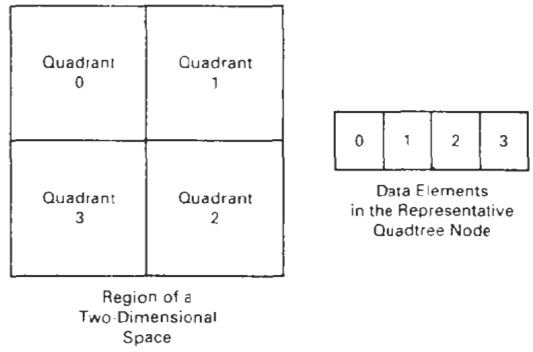


Figure 10-61
Region of a two-dimensional space divided into numbered quadrants and the associated quadtree node with four data elements.

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

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

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

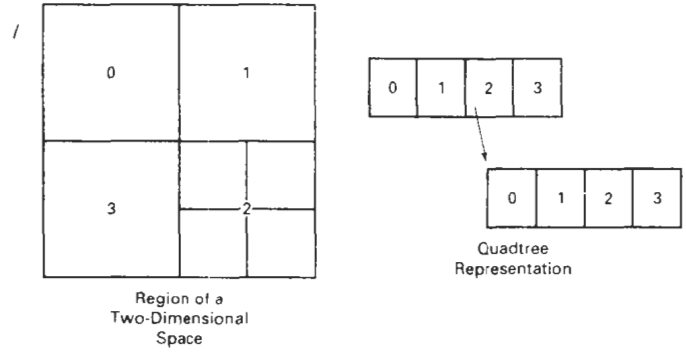


Figure 10-62
Region of a two-dimensional space with two levels of quadrant divisions and the associated quadtree representation

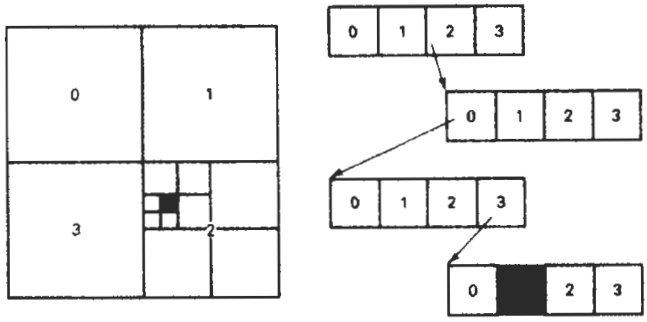


Figure 10-63
Quadtree representation for a region containing one foreground-color pixel on a solid background.

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

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

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

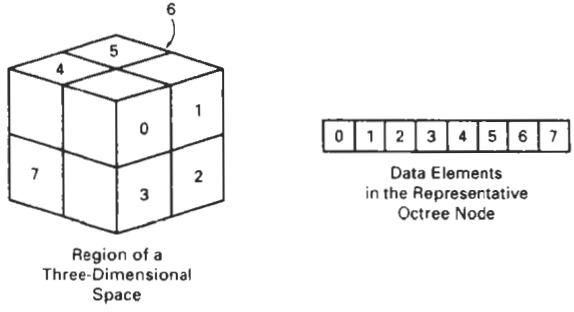


Figure 10-64
Region of a three-dimensional space divided into numbered octants and the associated octree node with eight data elements.

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

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

10-17 BSP TREES

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

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

10-18 FRACTAL-GEOMETRY METHODS

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

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

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

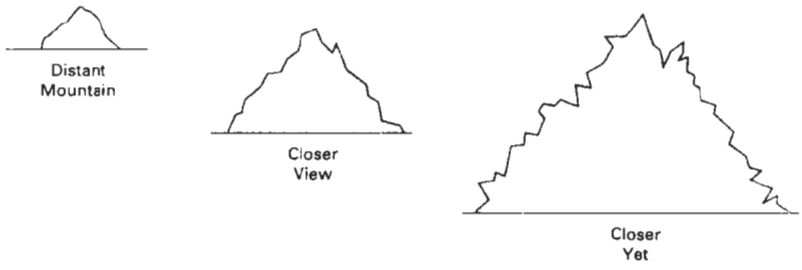


Figure 10-65
The ragged appearance of a mountain outline at different levels of magnification.

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

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

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

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

Fractal-Generation Procedures

A fractal object is generated by repeatedly applying a specified transformation function to points within a region of space. If $\mathbf{P}_0 = (x_0, y_0, z_0)$ is a selected initial point, each iteration of a transformation function F generates successive levels of detail with the calculations

$$\mathbf{P}_1 = F(\mathbf{P}_0), \qquad \mathbf{P}_2 = F(\mathbf{P}_1), \qquad \mathbf{P}_3 = F(\mathbf{P}_2), \quad \ldots \tag{10-99}$$

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

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

Classification of Fractals

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

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

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

Fractal Dimension

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

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

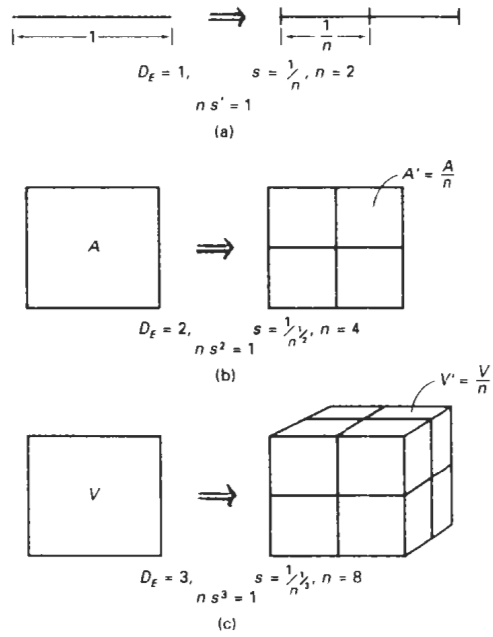


Figure 10-66
Subdividing objects with Euclidean dimensions (a) $D_E = 1$, (b) $D_E = 2$, and (c) $D_E = 3$ using scaling factor $s = 1/2$.

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

$$ns^D = 1 \tag{10-100}$$

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

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

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

$$\sum_{k=1}^{n} s_k^D = 1 \tag{10-102}$$

where s_k is the scaling factor for subpart number k.

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

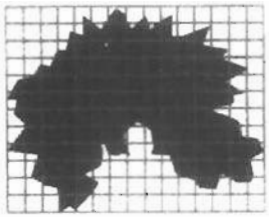


Figure 10-67
Box covering of an irregularly shaped object.

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

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

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

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

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

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

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

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

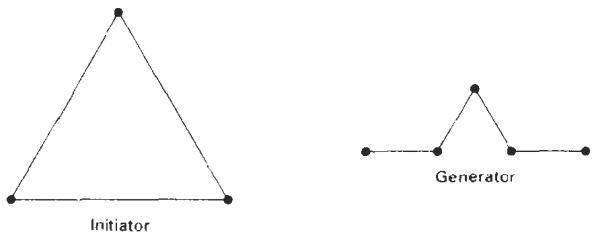


Figure 10-68
Initiator and generator for the Koch curve.

Geometric Construction of Deterministic Self-Similar Fractals

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

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

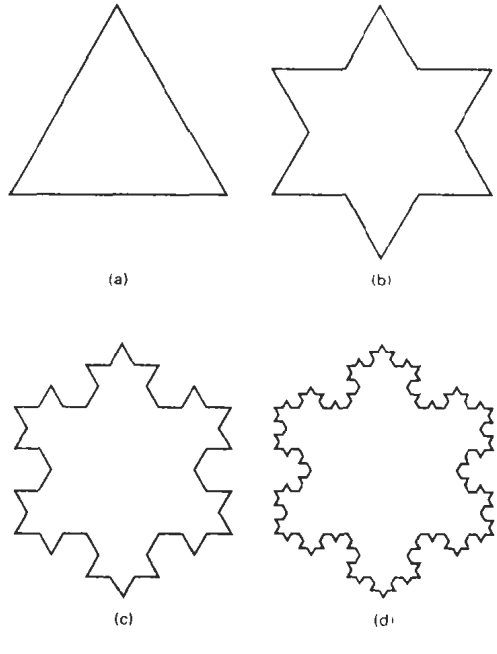


Figure 10-69
First three iterations in the generation of the Koch curve.

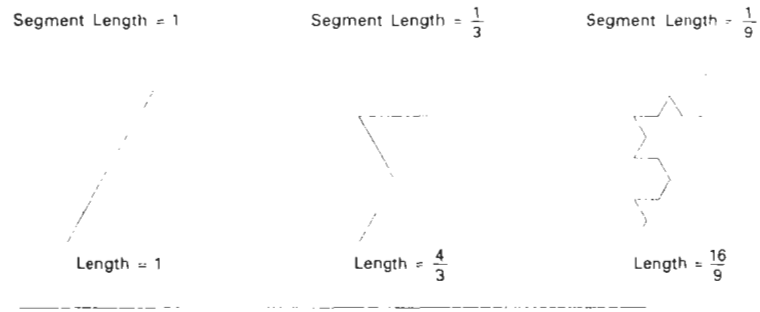


Figure 10-70
Length of each side of the Koch curve increases by a factor of 4/3 at each step, while the line-segment lengths are reduced by a factor of 1/3.

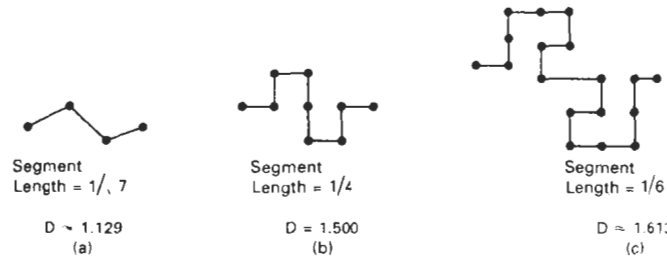


Figure 10-71
Self-similar curve constructions and associated fractal dimensions.

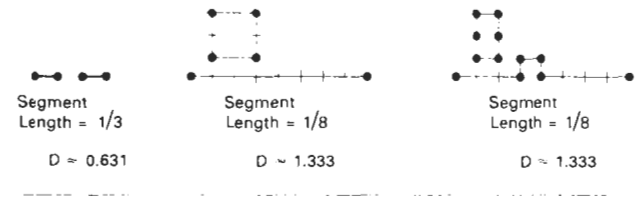


Figure 10-72
Generators with multiple, disjoint parts.

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

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

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

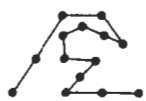


Figure 10-73
A snowflake-filling Peano curve.

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

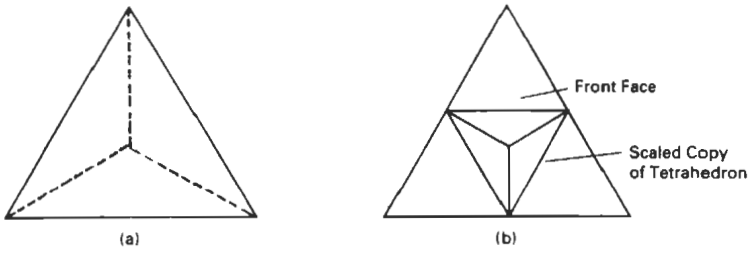


Figure 10-74
Scaling the tetrahedron in (a) by a factor of 1/2 and positioning the scaled version on one face of the original tetrahedron produces the fractal surface (b).

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

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

which indicates a fairly fragmented surface.

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

Geometric Construction of Statistically Self-Similar Fractals

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

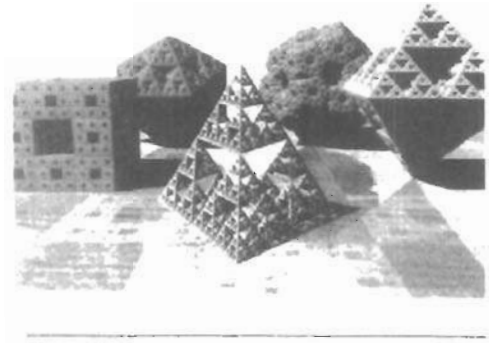


Figure 10-75
Self-similar, three-dimensional fractals formed with generators that subtract subparts from an initiator.
(Courtesy of John C. Hart, Washington State University)

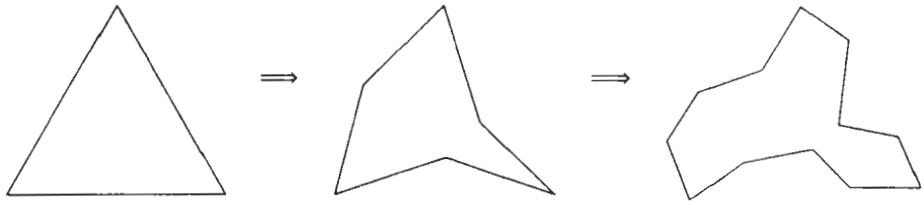
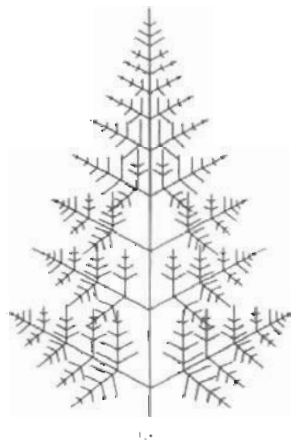


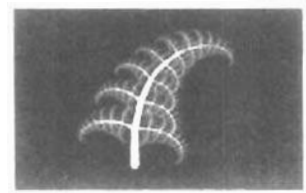
Figure 10-76
A modified "snowflake" pattern using random midpoint displacement.

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

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

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





(b)

Figure 10-77
Self-similar constructions for a fern. (*Courtesy of Peter Oppenheimer, Computer Graphics Lab, New York Institute of Technology.*)

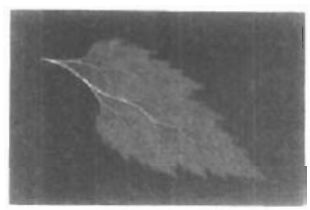


Figure 10-78
Random, self-similar construction of vein branching in a fall leaf. Boundary of the leaf is the limit of the vein growth. (*Courtesy of Peter Oppenheimer, Computer Graphics Lab, New York Institute of Technology.*)

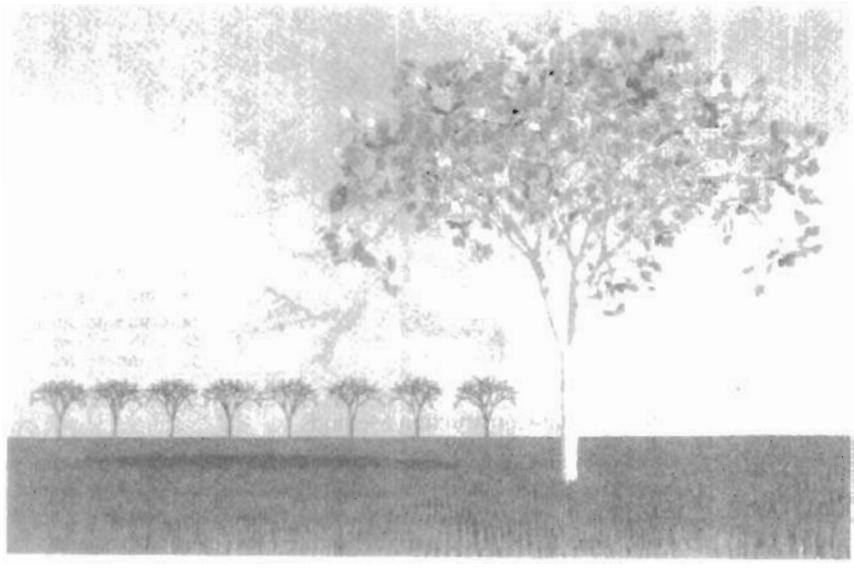


Figure 10-79
Modeling a scene using multiple object instancing. Fractal leaves are attached to a tree, and several instances of the tree are used to form a grove. The grass is modeled with multiple instances of green cones. (*Courtesy of John C. Hart, Washington State University.*)

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



Figure 10-80
A fractal forest created with multiple instances of leaves, pine needles, grass, and tree bark. (*Courtesy of John C. Hart, Washington State University.*)

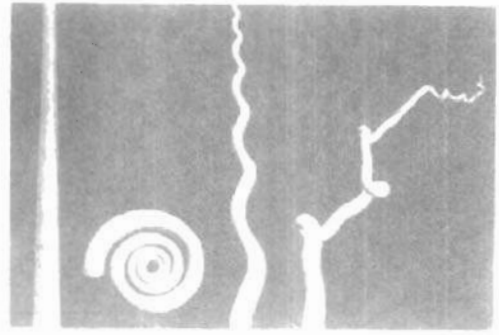


Figure 10-81
Modeling tree branches with spiral, helical, and random twists. (*Courtesy of Peter Oppenheimer, Computer Graphics Lab, New York Institute of Technology.*)



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

Affine Fractal-Construction Methods

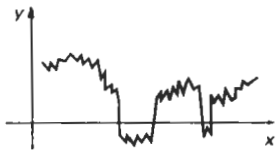


Figure 10-83
An example of Brownian motion (random walk) in the *xy* plane.

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

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



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

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

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

Random Midpoint-Displacement Methods

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

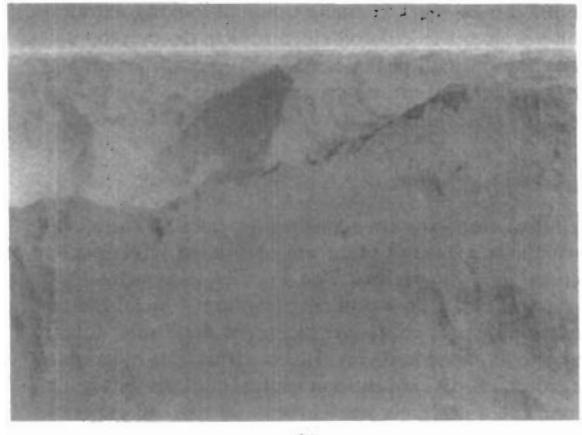
Although random midpoint-displacement methods are faster than fractional Brownian-motion calculations, they produce less realistic-looking terrain features. Figure 10-87 illustrates the midpoint-displacement method for generating a random-walk path in the xy plane. Starting with a straight-line segment,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

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

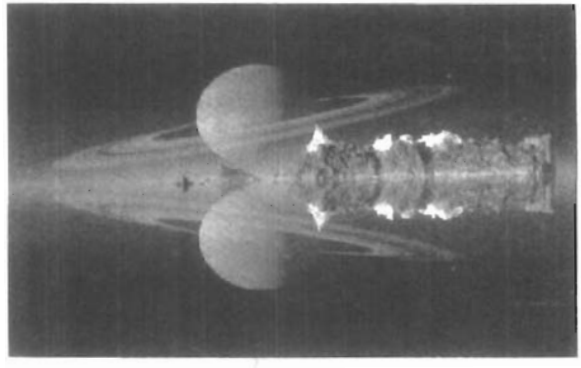
To approximate fractional Brownian motion, we choose a value for r from a Gaussian distribution with a mean of 0 and a variance proportional to $|b - a|^{2H}$, where $H = 2 - D$ and $D > 1$ is the fractal dimension. Another way to obtain a random offset is to take $r = sr_g|b - a|$, with parameter s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



(a)



(b)



(c)

Figure 10-85
Variations in terrain features modeled with fractional Brownian motion. (*Courtesy of (a) R. V. Voss and B. B. Mandelbrot, adapted from* The Fractal Geometry of Nature *by Benoit B. Mandelbrot (New York: W. H. Freeman and Co., 1983); and (b) and (c) Ken Musgrave and Benoit B. Mandelbrot, Mathematics and Computer Science, Yale University.*)



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

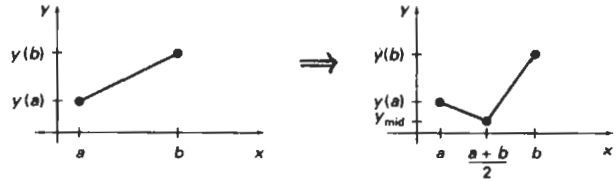


Figure 10-87
Random midpoint-displacement of a straight-line segment.

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

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

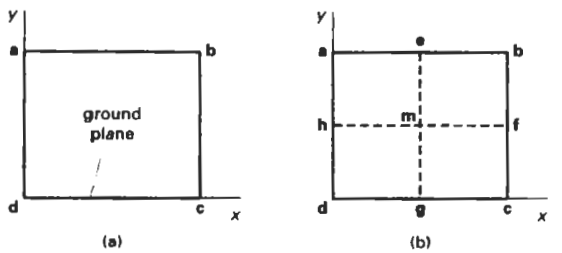


Figure 10-89
A rectangular ground plane (a) is subdivided into four equal grid sections (b) for the first step in a random midpoint-displacement procedure to calculate terrain elevations.

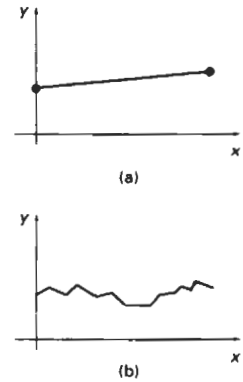


Figure 10-88
A random-walk path generated from a straight-line segment with four iterations of the random midpoint-displacement procedure.

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

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

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

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

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

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

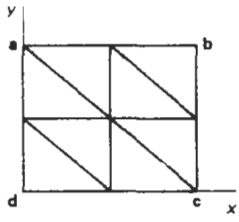


Figure 10-90
Eight surface patches formed over a ground plane at the first step of a random midpoint-displacement procedure for generating terrain features.

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

Controlling Terrain Topography

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

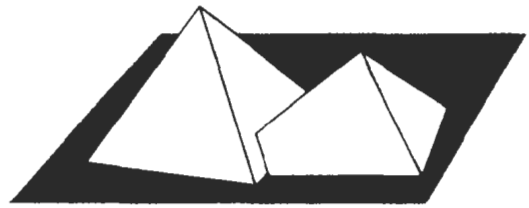


Figure 10-91
Control surfaces over a ground plane.

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

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

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

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

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

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

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

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

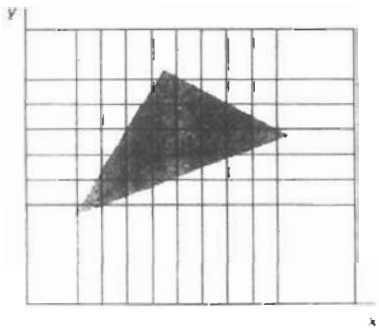


Figure 10-92
Projection of a triangular control surface onto the ground-plane grid.

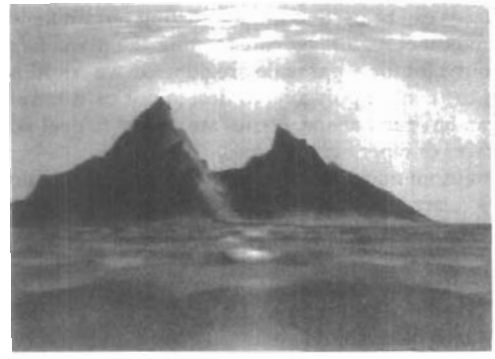


Figure 10-93
A composite scene modeled with a random midpoint-displacement method and planar control surfaces over a ground plane. Surface features for the terrain, water, and clouds were modeled and rendered separately, then combined to form the composite picture. (*Courtesy of Eng-Kiat Koh, Information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

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

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

Self-Squaring Fractals

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

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

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

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

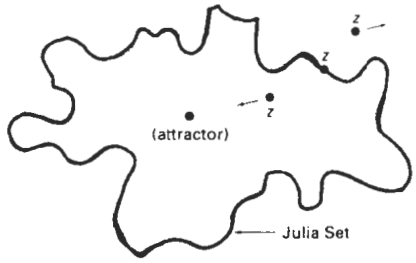


Figure 10-94
Possible outcomes of a self-squaring transformation $f(z)$ in the complex plane, depending on the position of the selected initial position.

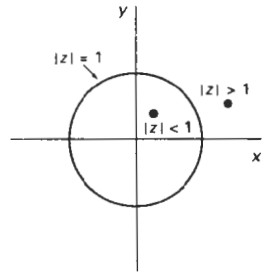


Figure 10-95
A unit circle in the complex plane. The nonfractal, complex squaring function $f(z) = z^2$ moves points that are inside the circle toward the origin, while points outside the circle are moved farther away from the circle. Any initial point on the circle remains on the circle.

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

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

A function that is rich in fractals is the squaring transformation

$$z' = f(z) = \lambda z(1 - z) \tag{10-105}$$

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

$$z^2 - z + z'/\lambda = 0 \tag{10-106}$$

The inverse transformation is then the quadratic formula:

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

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

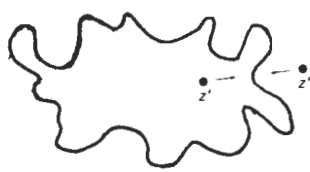


Figure 10-96
Locating the fractal boundary with the inverse, self-squaring function $z' = f^{-1}(z)$.

$$x = \mathrm{Re}(z) = \frac{1}{2}\left(1 \pm \sqrt{\frac{|\mathrm{discr}| + \mathrm{Re}(\mathrm{discr})}{2}}\right)$$

$$y = \mathrm{Im}(z) = \pm\frac{1}{2}\sqrt{\frac{|\mathrm{discr}| - \mathrm{Re}(\mathrm{discr})}{2}} \tag{10-108}$$

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

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

typedef struct {
  float x, y;
} Complex;

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

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

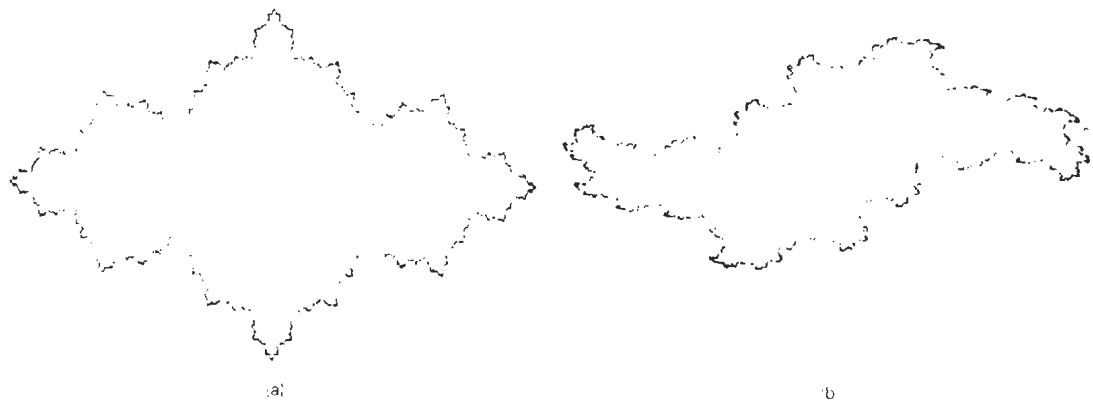


Figure 10-97
Two fractal curves generated with the inverse of the function $f(z) = \lambda z(1-z)$ by procedure `selfSquare`: (a) $\lambda = 3$ and (b) $\lambda = 2 + i$. Each curve is plotted with 10,000 points.

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

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

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

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

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

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

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

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

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

$$z_0 = z$$
$$z_k = z_{k-1}^2 + z_0, \qquad k = 1, 2, 3, \ldots \tag{10-109}$$

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

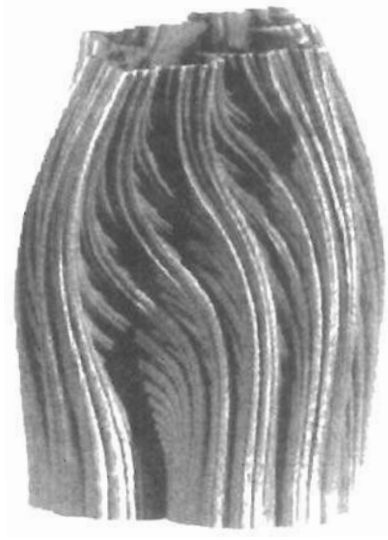


Figure 10-98
The function $f(z) = \lambda z(1-z)$ plotted in three dimensions with normalized λ values plotted as the vertical axis. (*Courtesy of Alan Norton, IBM Research.*)

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

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

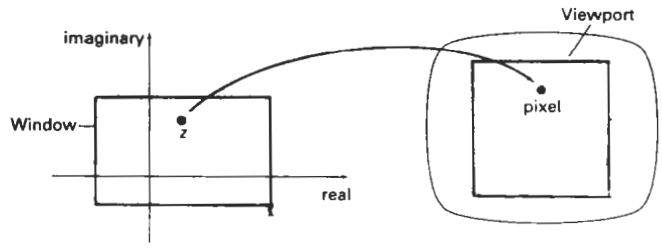


Figure 10-99
Mapping positions in the complex plane to color-coded pixel positions on a video monitor.

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

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

$$-2.25 \leq \mathrm{Re}(z) \leq 0.75$$

$$-1.25 \leq \mathrm{Im}(z) \leq 1.25$$

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

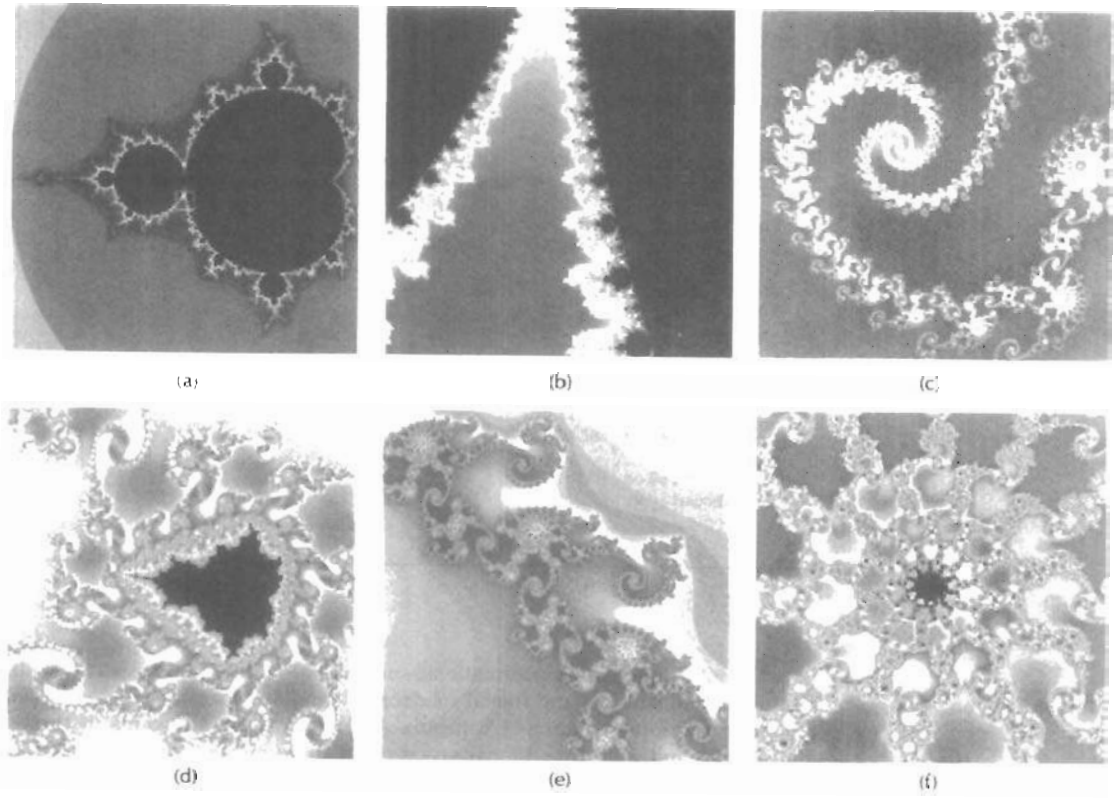


Figure 10-100
Zooming in on the Mandelbrot set. Starting with a display of the Mandelbrot set (a), we zoom in on selected regions (b) through (f). The white box outline shows the window area selected for each successive zoom. (*Courtesy of Brian Evans, Vanderbilt University.*)

```
#include "graphics.h"

typedef struct { float x, y; } Complex;

Complex complexSquare (Complex c)
{
  Complex cSq;

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

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

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

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

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

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

$$q = s + ia + jb + kc \tag{10-110}$$

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

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

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

Self-Inverse Fractals

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

As an example, we consider a two-dimensional inversion transformation with respect to a circle with radius r and center at position $\mathbf{P}_0 = (x_0, y_0)$. Any point $\mathbf{P}$ outside the circle will be inverted to a position $\mathbf{P}'$ inside the circle (Fig. 10-103) with the transformation

$$(\overline{\mathbf{P}_0\mathbf{P}})(\overline{\mathbf{P}_0\mathbf{P}'}) = r^2 \qquad (10\text{-}111)$$

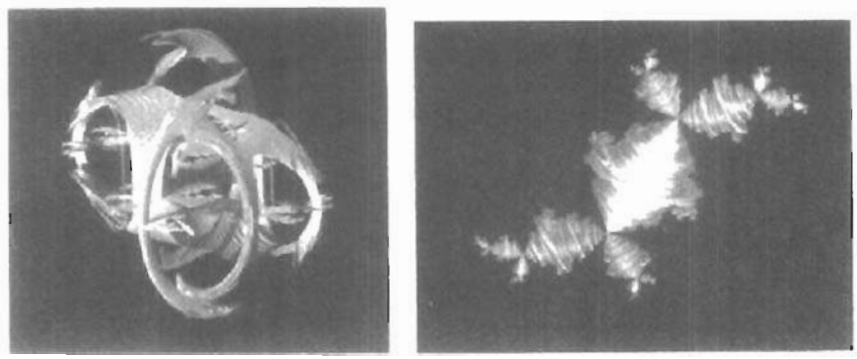


Figure 10-101
Three-dimensional projections of four-dimensional fractals generated with the self-squaring, quaternion function $f(q) = \lambda q(1-q)$: (a) $\lambda = 1.475 + 0.9061i$, and (b) $\lambda = -0.57 + i$. *(Courtesy of Alan Norton, IBM Research.)*

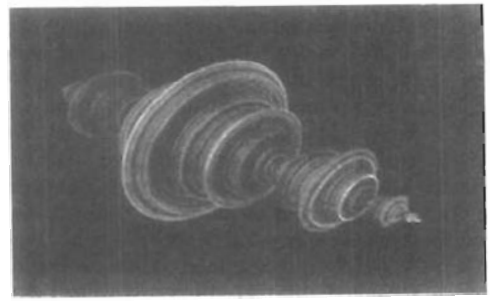


Figure 10-102
A three-dimensional surface projection of a four-dimensional object generated with the self-squaring, quaternion function $f(q) = q^2 - 1$.
(Courtesy of Alan Norton, IBM Research.)

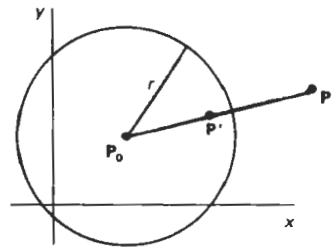


Figure 10-103
Inverting point **P** to a position **P'** inside a circle with radius *r*.

Reciprocally, this transformation inverts any point inside the circle to a point outside the circle. Both **P** and **P'** lie on a straight line passing through the circle center $\mathbf{P}_0$.

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

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

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

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

Figure 10-104 illustrates the inversion of points along another circle boundary. As long as the circle to be inverted does not pass through $\mathbf{P}_0$, it will transform to another circle. But if the circle circumference passes through $\mathbf{P}_0$, the circle

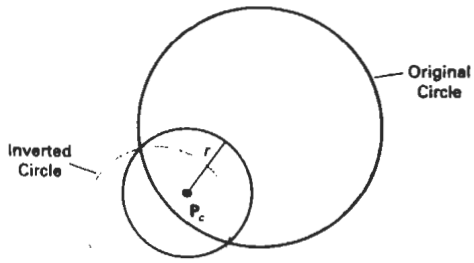


Figure 10-104
Inversion of a circle with respect to another circle.

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

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

10-19
SHAPE GRAMMARS AND OTHER PROCEDURAL METHODS

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

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

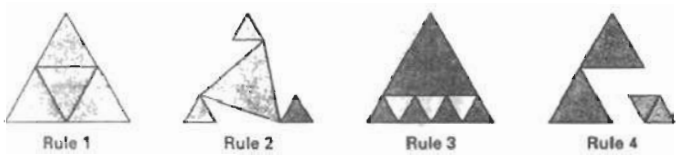
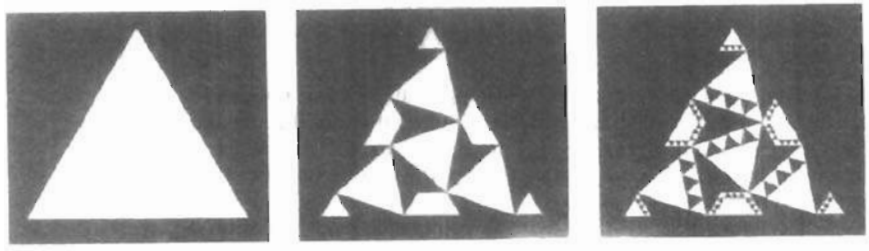


Figure 10-105
Four geometric substitution rules for subdividing and altering the shape of an equilateral triangle.

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

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



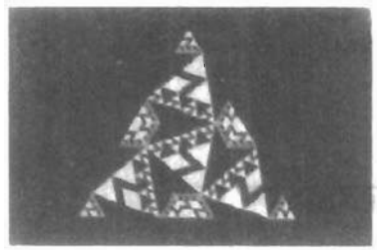


Figure 10-106
An equilateral triangle (a) is converted to shape (b) using substitution rules 1 and 2 in Fig. 10-105. Rule 3 is then used to convert (b) into shape (c), which in turn is transformed to (d) using rule 4. (*Copyright © 1992 Andrew Glassner, Xerox PARC (Palo Alto Research Center).*)

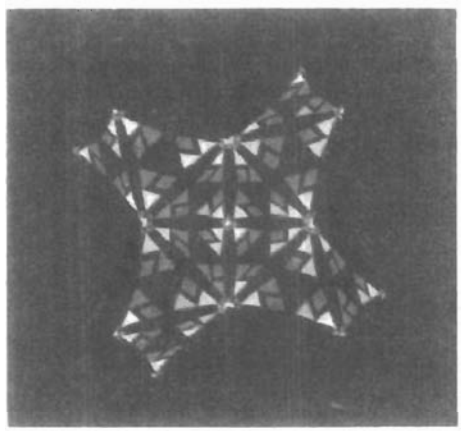


Figure 10-107
A design created with geometric substitution rules for altering triangle shapes. (*Copyright © 1992 Andrew Glassner, Xerox PARC (Palo Alto Research Center).*)

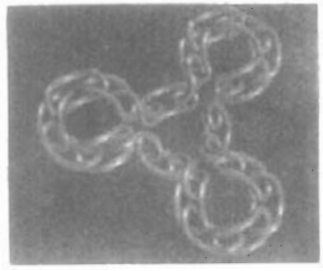


Figure 10-108
A design created with geometric substitution rules for altering prism shapes. The initial shape for this design was a representation of Rubik's Snake. (*Copyright © 1992 Andrew Glassner, Xerox PARC (Palo Alto Research Center).*)

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

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

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

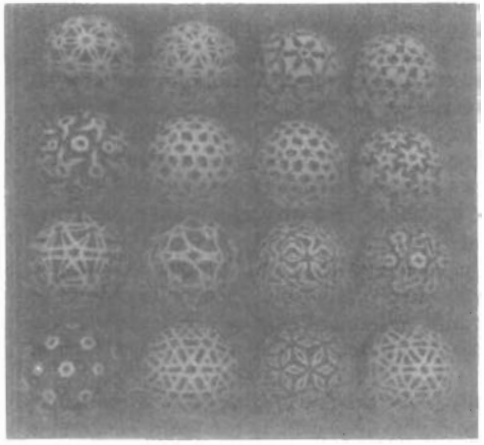


Figure 10-109
Designs created on the surface of a sphere using triangle substitution rules applied to the plane faces of an icosahedron, followed by projections to the sphere surface. (*Copyright © 1992 Andrew Glassner, Xerox PARC (Palo Alto Research Center).*)



Figure 10-110
Realistic scenery generated with the TDI-AMAP software package, which can generate over 100 varieties of plants and trees using procedures based on botanical laws. (*Courtesy of Thomson Digital Image.*)

10-20
PARTICLE SYSTEMS

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

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

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

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

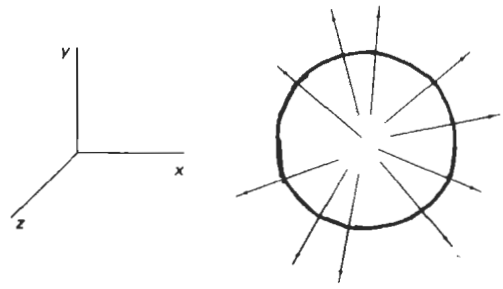


Figure 10-111
Modeling fireworks as a particle system with particles traveling radially outward from the center of the sphere.

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

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

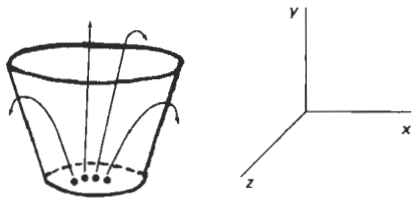


Figure 10-112
Modeling a clump of grass by firing particles upward within a tapered cylinder. The particle paths are parabolas due to the downward force of gravity.

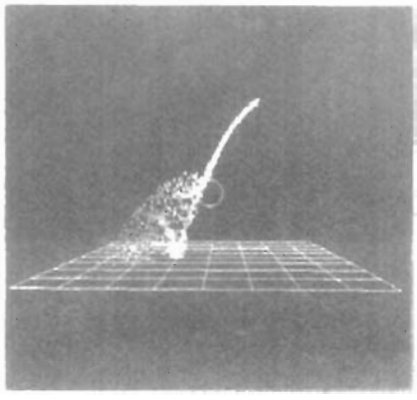


Figure 10-113
Simulation of the behavior of a waterfall hitting a stone (circle). The water particles are deflected by the stone and then splash up from the ground. (*Courtesy of M. Brooks and T. L. J. Howard, Department of Computer Science, University of Manchester.*)

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



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



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

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

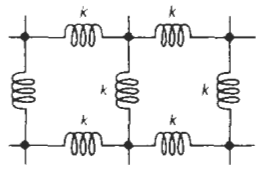


Figure 10-116
A two-dimensional spring network, constructed with identical spring constants k.

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

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

$$F_s = -F_x = -kx \tag{10-113}$$

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

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

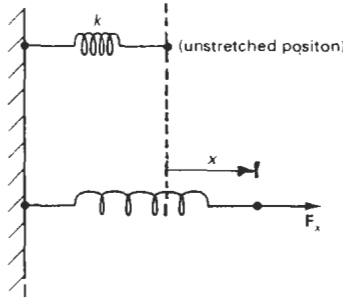


Figure 10-117
An external force F_x pulling on one end of a spring, with the other end rigidly fixed.

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

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

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

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

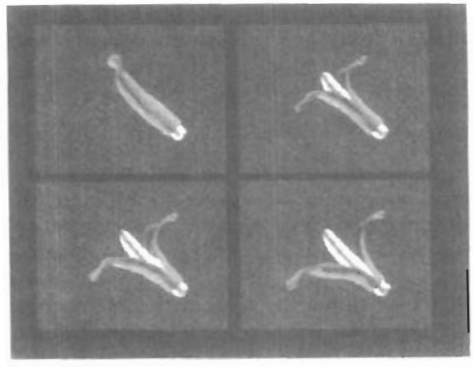


Figure 10-118
Modeling the flexible behavior of a banana peel with a spring network.
(Copyright © 1992 David Laidlaw, John Snyder, Adam Woodbury, and Alan Barr, Computer Graphics Lab, California Institute of Technology.)



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

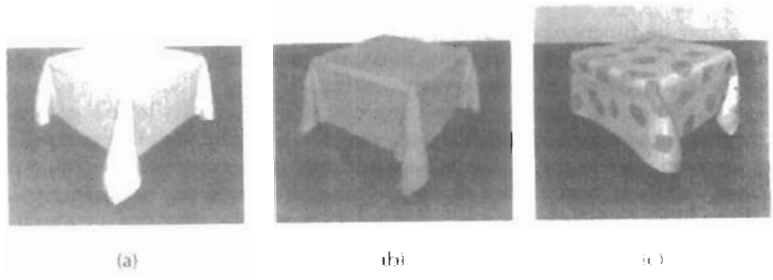


Figure 10-120
Modeling the characteristics of (a) cotton, (b) wool, and (c) polyester cotton using energy-function minimization. (*Copyright © 1992 David E. Breen and Donald H. House, Design Research Center, Rensselaer Polytechnic Institute.*)

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

10-22
VISUALIZATION OF DATA SETS

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

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

Visual Representations for Scalar Fields

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

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

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

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

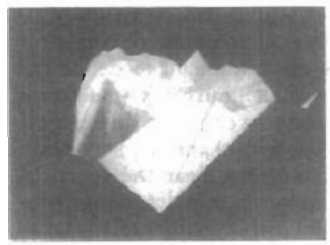


Figure 10-121
A financial surface plot showing stock-growth potential during the October 1987 stock-market crash. Red indicates high returns, and the plot shows that low-growth stocks performed better in the crash. *(Courtesy of Eng-Kiat Koh, Information Technology Institute, Republic of Singapore.)*

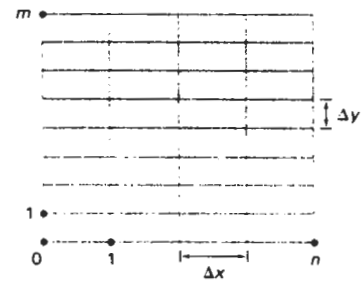


Figure 10-122
A regular, two-dimensional grid with data values at the intersection of the grid lines. The x grid lines have a constant Δx spacing, and the y grid lines have a constant Δy spacing, where the spacing in the x and y directions may not be the same.

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

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

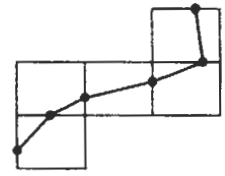


Figure 10-123
The path of an isoline across five grid cells.

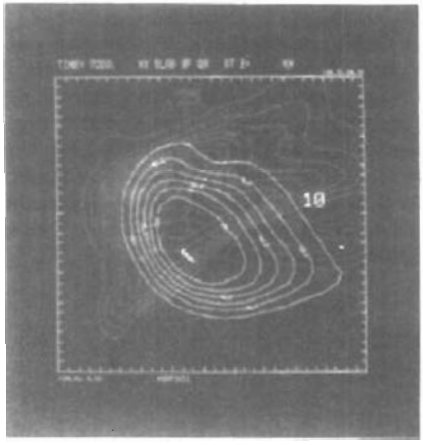


Figure 10-124
Color-coded contour plots for three data sets within the same region of the xy plane. (*Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.*)

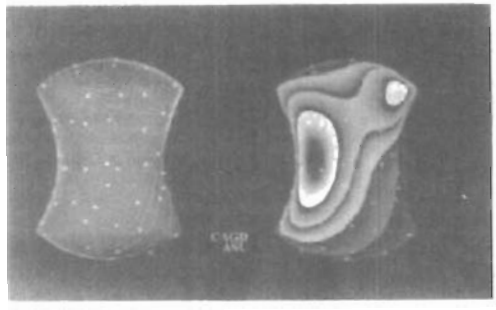


Figure 10-125
Color-coded contour plots over the surface of an apple-core-shaped region of space. (*Courtesy of Greg Nielson, Department of Computer Science and Engineering, Arizona State University.*)

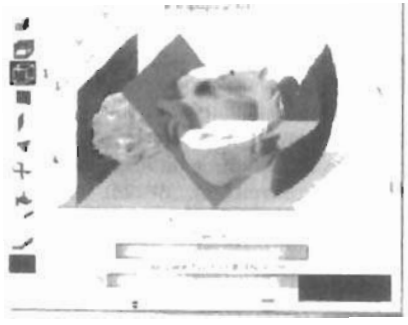


Figure 10-126
Cross-sectional slices of a three-dimensional data set. (*Courtesy of Spyglass, Inc.*)

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

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

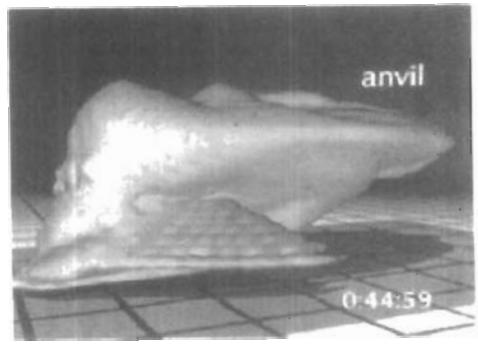


Figure 10-127
An isosurface generated from a set of water-content values obtained from a numerical model of a thunderstorm. (*Courtesy of Bob Wilhelmson, Department of Atmospheric Sciences and National Center for Supercomputing Applications, University of Illinois at Urbana Champaign.*)

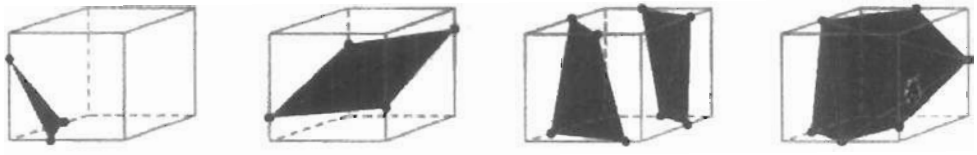


Figure 10-128
Isosurface intersections with grid cells, modeled with triangle patches.

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

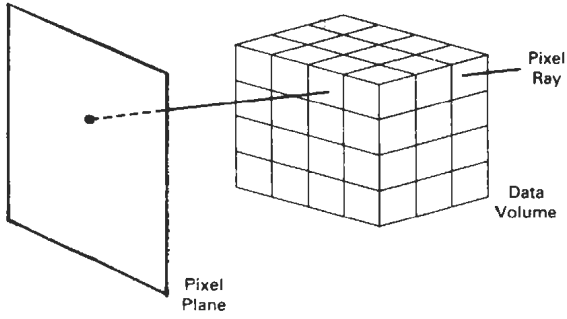


Figure 10-129
Volume visualization of a regular, Cartesian data grid using ray casting to examine interior data values.

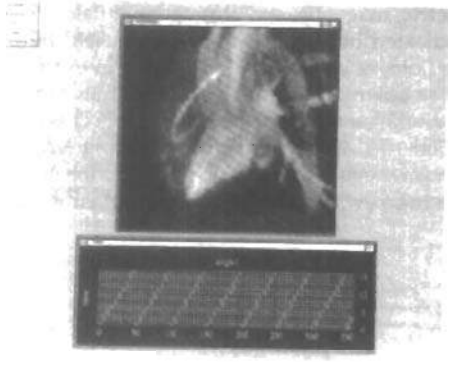


Figure 10-130
Volume visualization of a data set for a dog heart, obtained by plotting the color-coded distance to the maximum voxel value for each pixel. (*Courtesy of Patrick Moran and Clinton Potter, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.*)

Visual Representations for Vector Fields

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

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

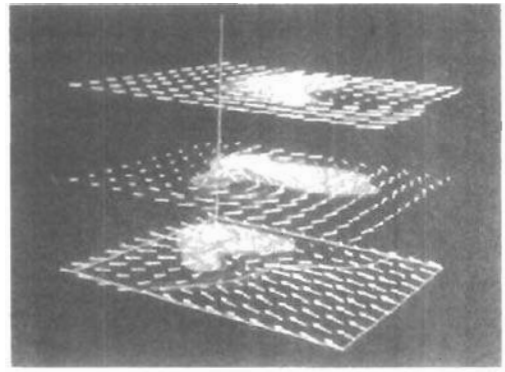


Figure 10-131
Arrow representation for a vector field over cross-sectional slices. (*Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.*)

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

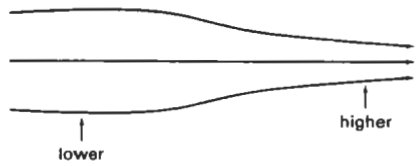


Figure 10-132
Field-line representation for a vector data set.

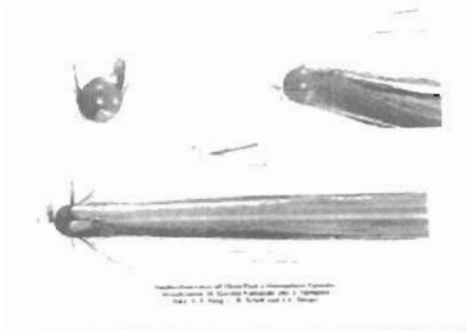


Figure 10-133
Visualizing airflow around a cylinder with a hemispherical cap that is tilted slightly relative to the incoming direction of the airflow. *(Courtesy of M. Gerald-Yamasaki, J. Hultquist, and Sam Uselton, NASA Ames Research Center.)*

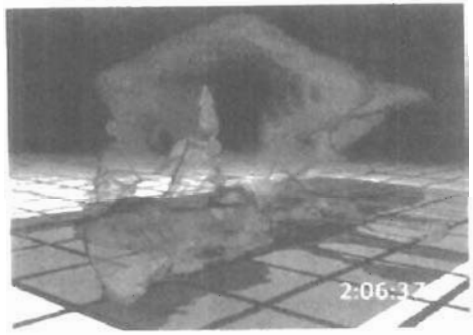


Figure 10-134
Twisting airflow patterns, visualized with wide streamlines inside a transparent isosurface plot of a thunderstorm. *(Courtesy of Bob Wilhelmson, Department of Atmospheric Sciences and National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)*

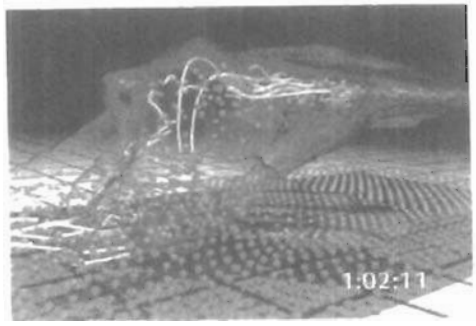


Figure 10-135
Airflow patterns, visualized with both streamlines and particle motion inside a transparent isosurface plot of a thunderstorm. Rising sphere particles are colored orange, and falling sphere particles are blue. *(Courtesy of Bob Wilhelmson, Department of Atmospheric Sciences and National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)*

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

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

Visual Representations for Tensor Fields

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

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

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

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

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

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

Visual Representations for Multivariate Data Fields

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

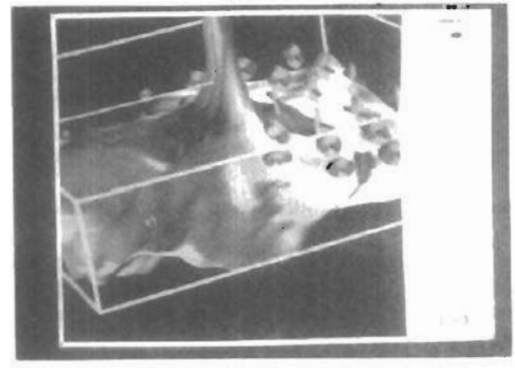


Figure 10-136
Representing stress and strain tensors with an elliptical disk and a rod over the surface of a stressed material. (*Courtesy of Bob Haber, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.*)

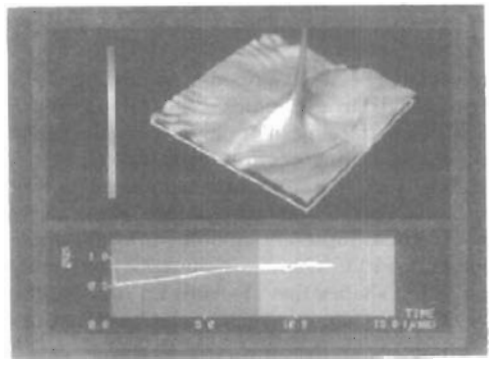


Figure 10-137
Representing stress and strain tensors with a strain-energy density plot in a visualization of crack propagation on the surface of a stressed material. (*Courtesy of Bob Haber, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.*)

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

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

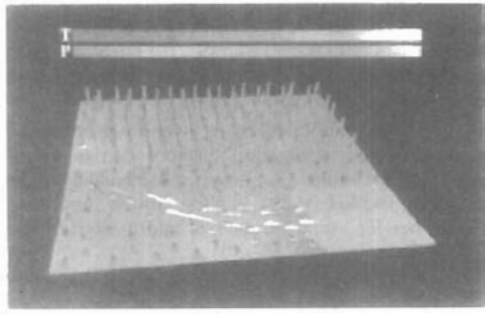


Figure 10-138
One frame from an animated visualization of a multivariate data field using glyphs. The wedge-shaped part of the glyph indicates the direction of a vector quantity at each point. (*Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.*)

SUMMARY

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

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

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

REFERENCES

A detailed discussion of superquadrics is contained in Barr (1981). For more information on blobby object modeling, see Blinn (1982). The metaball model is discussed in Nishimura (1985); and the soft-object model is discussed in Wyvill, Wyvill, and McPheeters (1987).

Sources of information on parametric curve and surface representations include Bezier (1972), Burt and Adelson (1983), Barsky (1983, 1984), Kochanek and Bartels (1984), Farouki and Hinds (1985), Huitric and Nahas (1985), Mortenson (1985), Farin (1988), and Rogers and Adams (1990).

Octrees and quadtrees are discussed by Doctor (1981), Yamaguchi, Kunii, and Fujimura (1984), and by Carlbom, Chakravarty, and Vanderschel (1985). Solic-modeling references include Casale and Stanton (1985) and Requicha and Rossignac (1992).

For further information on fractal representations see Mandelbrot (1977, 1982), Fournier, Fussel, and Carpenter (1982), Norton (1982), Peitgen and Richter (1986), Peitgen and Saupe (1988). Koh and Hearn (1992), and Barnsley (1993). Shape grammars are discussed in Glassner (1992), and particle systems are discussed in Reeves (1983). A discussion of physically based modeling is given in Barzel (1992).

A general introduction to visualization methods is given in Hearn and Baker (1991). Additional information on specific visualization methods can be found in Sabin (1985), Lorensen and Cline (1987), Drebin, Carpenter, and Hanrahan (1988), Sabella (1988), Upson and Keeler (1988), Frenkel (1989), Nielson, Shriver, and Rosenblum (1990), and Nielson (1993). Guidelines for visual displays of information are given in Tufte (1983, 1990).

EXERCISES

10-1. Set up geometric data tables as in Fig. 10-2 for a unit cube.

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

10-3. Define an efficient polygon representation for a cylinder. Justify your choice of representation.

10-4. Set up a procedure for establishing polygon tables for any input set of data points defining an object.

10-5. Devise routines for checking the data tables in Fig. 10-2 for consistency and completeness.

10-6. Write a program that calculates parameters A, B, C, and D for any set of three-dimensional plane surfaces defining an object.

10-7. Given the plane parameters A, B, C, and D for all surfaces of an object, devise an algorithm to determine whether any specified point is inside or outside the object.

10-8. How would the values for parameters A, B, C, and D in the equation of a plane surface have to be altered if the coordinate reference is changed from a right-handed system to a left-handed system?

10-9. Set up an algorithm for converting any specified sphere, ellipsoid, or cylinder to a polygon-mesh representation.

10-10. Set up an algorithm for converting a specified superellipsoid to a polygon-mesh representation.

10-11. Set up an algorithm for converting a metaball representation to a polygon-mesh representation.

10-12. Write a routine to display a two-dimensional, cardinal-spline curve, given an input set of control points in the xy plane.

10-13. Write a routine to display a two-dimensional, Kochanek-Bartels curve, given an input set of control points in the xy plane.

10-14. Determine the quadratic Bézier blending functions for three control points. Plot each function and label the maximum and minimum values.

10-15. Determine the Bézier blending functions for five control points. Plot each function and label the maximum and minimum values.

10-16. Write an efficient routine to display two-dimensional, cubic Bézier curves, given a set of four control points in the xy plane.

10-17. Write a routine to design two-dimensional, cubic Bézier curve shapes that have first-order piecewise continuity. Use an interactive technique for selecting control-point positions in the xy plane for each section of the curve.

10-18. Write a routine to design two-dimensional, cubic Bézier curve shapes that have second-order piecewise continuity. Use an interactive technique for selecting control-point positions in the xy plane for each section of the curve.

10-19. Write a routine to display a cubic Bézier curve using a subdivision method.

10-20. Determine the blending functions for uniform, periodic B-spline curves for $d = 5$.

10-21. Determine the blending functions for uniform, periodic B-spline curves for $d = 6$.

10-22. Write a program using forward differences to calculate points along a two-dimensional, uniform, periodic, cubic B-spline curve, given an input set of control points.

10-23. Write a routine to display any specified conic in the xy plane using a rational Bézier spline representation.

10-24. Write a routine to display any specified conic in the xy plane using a rational B-spline representation.

10-25. Develop an algorithm for calculating the normal vector to a Bezier surface at the point $\mathbf{P}(u, v)$.

10-26. Write a program to display any specified quadratic curve using forward differences to calculate points along the curve path.

10-27. Write a program to display any specified cubic curve using forward differences to calculate points along the curve path.

10-28. Derive expressions for calculating the forward differences for any specified quadratic curve.

10-29. Derive expressions for calculating the forward differences for any specified cubic curve.

10-30. Set up procedures for generating the description of a three-dimensional object from input parameters that define the object in terms of a translational sweep.

10-31. Develop procedures for generating the description of a three-dimensional object using input parameters that define the object in terms of a rotational sweep.

10-32. Devise an algorithm for generating solid objects as combinations of three-dimensional primitive shapes, each defined as a set of surfaces, using constructive solid-geometry methods.

10-33. Develop an algorithm for performing constructive solid-geometry modeling using a primitive set of solids defined in octree structures.

10-34. Develop an algorithm for encoding a two-dimensional scene as a quadtree representation.

10-35. Set up an algorithm for loading a quadtree representation of a scene into a frame buffer for display of the scene.

10-36. Write a routine to convert the polygon definition of a three-dimensional object into an octree representation.

10-37. Using the random, midpoint-displacement method, write a routine to create a mountain outline, starting with a horizontal line in the xy plane.

10-38. Write a routine to calculate elevations above a ground plane using the random, midpoint-displacement method.

10-39. Write a program for generating a fractal snowflake (Koch curve) for any given number of iterations.

10-40. Write a program to generate a fractal curve for a specified number of iterations using one of the generators in Fig. 10-71 or 10-72. What is the fractal dimension of your curve?

10-41. Write a program to generate fractal curves using the self-squaring function $f(z) = z^2 + \lambda$, where λ is any selected complex constant.

10-42. Write a program to generate fractal curves using the self-squaring function $f(x) = i(z^2 + 1)$, where $i = \sqrt{-1}$.

10-43. Write a routine to interactively select different color combinations for displaying the Mandelbrot set.

10-44. Write a program to interactively select any rectangular region of the Mandelbrot set and to zoom in on the selected region.

10-45. Write a routine to implement point inversion, Eq. 10-112, for any specified circle and any given point position.

10-46. Devise a set of geometric-substitution rules for altering the shape of an equilateral triangle.

10-47. Write a program to display the stages in the conversion of an equilateral triangle into another shape, given a set of geometric-substitution rules.

10-48. Write a program to model an exploding firecracker in the xy plane using a particle system.

10-49. Devise an algorithm for modeling a rectangle as a nonrigid body, using identical springs for the four sides of the rectangle.

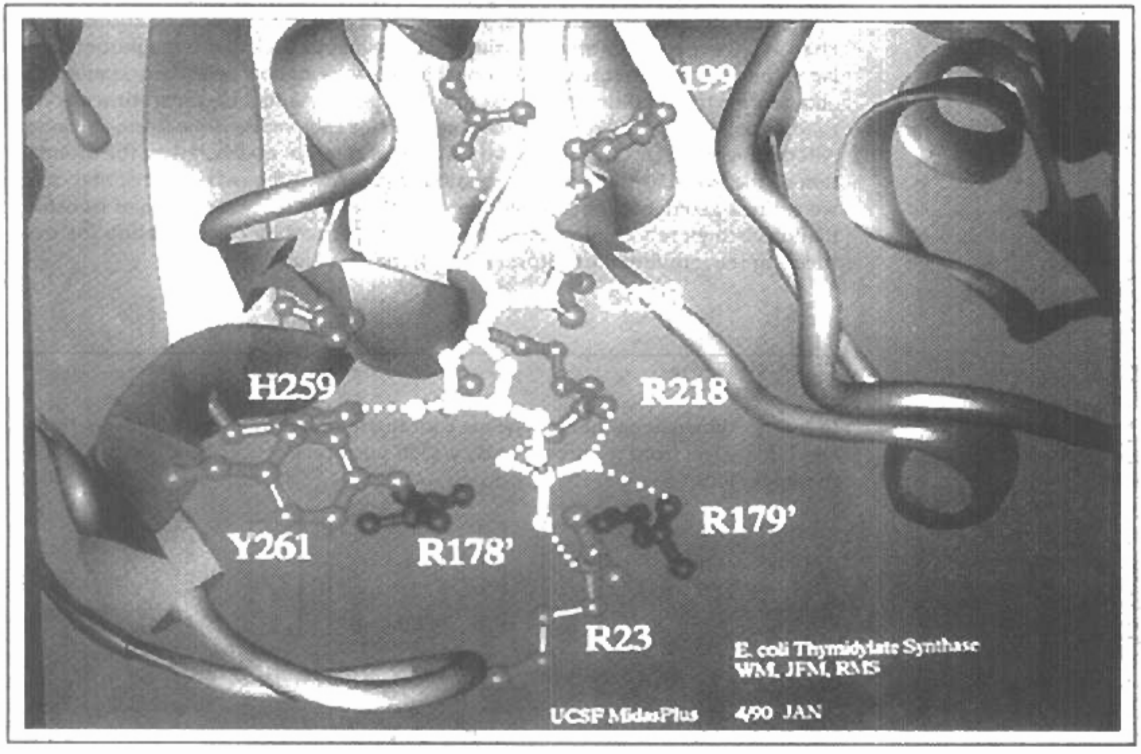
10-50. Write a routine to visualize a two-dimensional, scalar data set using pseudo-color methods.

10-51. Write a routine to visualize a two-dimensional, scalar data set using contour lines.

10-52. Write a routine to visualize a two-dimensional, vector data set using an arrow representation for the vector values. Make all arrows the same length, but display the arrows with different colors to represent the different vector magnitudes.

CHAPTER

11 Three-Dimensional Geometric and Modeling Transformations



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

11-1 TRANSLATION

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

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

or

$$\mathbf{P}' = \mathbf{T} \cdot \mathbf{P} \tag{11-2}$$

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

$$x' = x + t_x, \qquad y' = y + t_y, \qquad z' = z + t_z \tag{11-3}$$

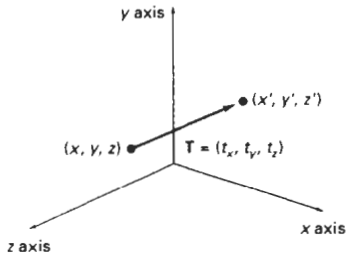


Figure 11-1
Translating a point with translation vector $\mathbf{T} = (t_x, t_y, t_z)$.

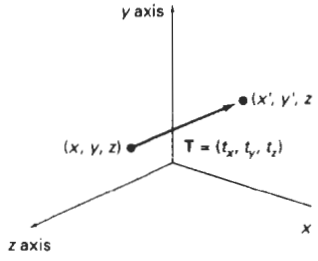


Figure 11-2
Translating an object with translation vector **T**.

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

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

11-2 ROTATION

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

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

Coordinate-Axes Rotations

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

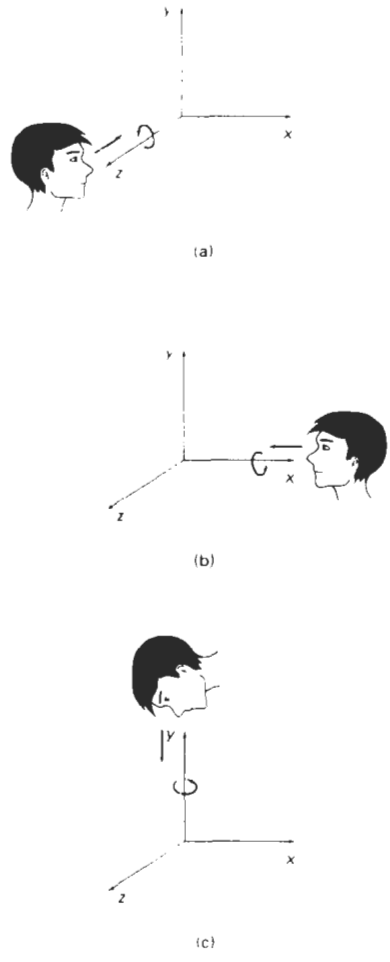


Figure 11-3
Positive rotation directions about the coordinate axes are counterclockwise, when looking toward the origin from a positive coordinate position on each axis.

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

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

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

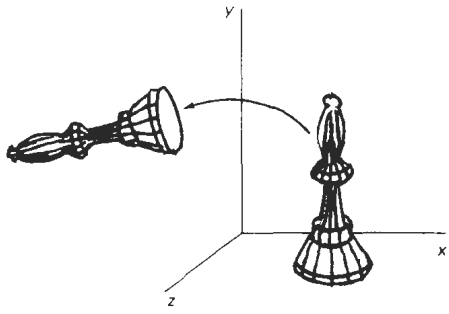


Figure 11-4
Rotation of an object about the z axis.

which we can write more compactly as

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

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

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

$$x \rightarrow y \rightarrow z \rightarrow x \tag{11-7}$$

as illustrated in Fig. 11-5.

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

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

which can be written in the homogeneous coordinate form

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

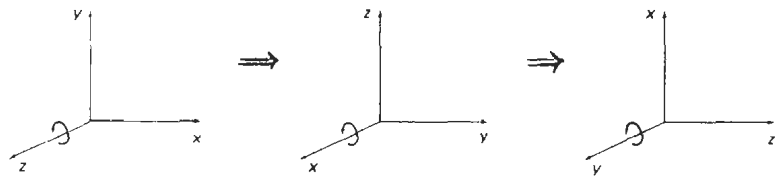


Figure 11-5
Cyclic permutation of the Cartesian-coordinate axes to produce the three sets of coordinate-axis rotation equations.

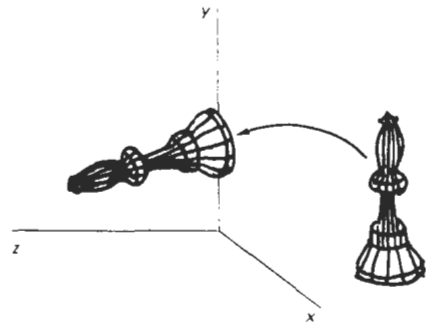


Figure 11-6
Rotation of an object about the x axis.

or

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

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

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

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

The matrix representation for y-axis rotation is

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

or

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

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

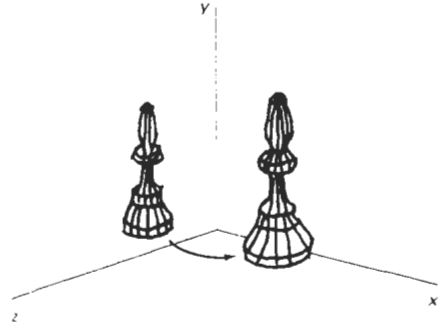


Figure 11-7
Rotation of an object about the y axis.

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

General Three-Dimensional Rotations

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

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

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

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

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

where the composite matrix for the transformation is

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

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

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

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

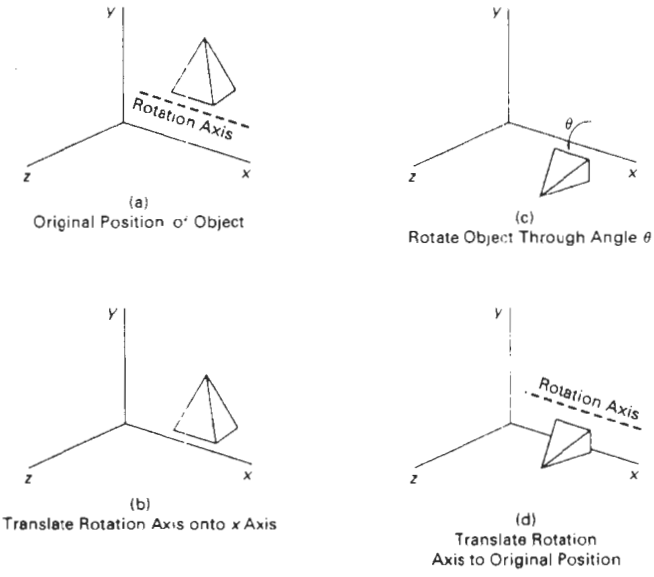


Figure 11-8
Sequence of transformations for rotating an object about an axis that is parallel to the x axis

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

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

A rotation axis can be defined with two coordinate positions, as in Fig. 11-10, or with one coordinate point and direction angles (or direction cosines) between the rotation axis and two of the coordinate axes. We will assume that the rotation axis is defined by two points, as illustrated, and that the direction of rotation is to be counterclockwise when looking along the axis from $\mathbf{P}_2$ to $\mathbf{P}_1$. An axis vector is then defined by the two points as

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

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

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

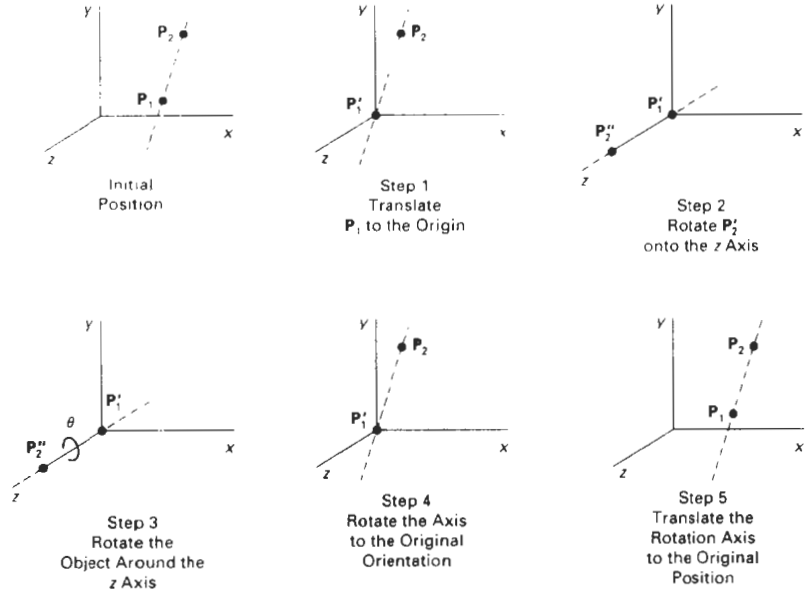


Figure 11-9
Five transformation steps for obtaining a composite matrix for rotation about an arbitrary axis, with the rotation axis projected onto the z axis.

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

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

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

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

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

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

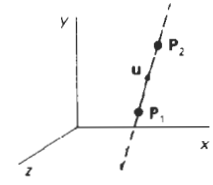


Figure 11-10
An axis of rotation (dashed line) defined with points P_1 and P_2. The direction for the unit axis vector $\mathbf{u}$ is determined by the specified rotation direction.

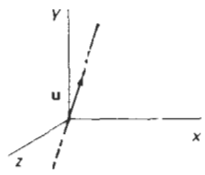


Figure 11-11
Translation of the rotation axis to the coordinate origin.

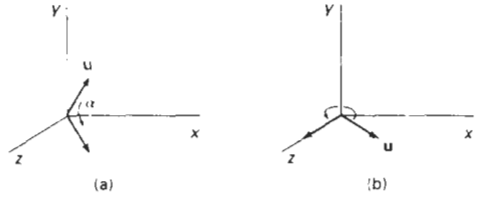


Figure 11-12
Unit vector **u** is rotated about the x axis to bring it into the xz plane (a), then it is rotated around the y axis to align it with the z axis (b).

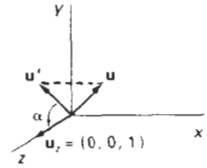


Figure 11-13
Rotation of **u** around the x axis into the xz plane is accomplished by rotating **u**′ (the projection of **u** in the yz plane) through angle α onto the z axis.

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

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

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

$$\cos\alpha = \frac{\mathbf{u}' \cdot \mathbf{u}_z}{|\mathbf{u}'||\mathbf{u}_z|} = \frac{c}{d} \tag{11-18}$$

where d is the magnitude of **u**′:

$$d = \sqrt{b^2 + c^2} \tag{11-19}$$

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

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

and the Cartesian form for the cross product gives us

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

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

$$d \sin\alpha = b$$

or

$$\sin\alpha = \frac{b}{d} \tag{11-22}$$

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

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

This matrix rotates unit vector **u** about the x axis into the xz plane.

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

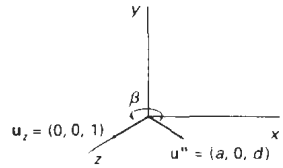


Figure 11-14
Rotation of unit vector **u**″ (vector **u** after rotation into the xz plane) about the y axis. Positive rotation angle β aligns **u**″ with vector $\mathbf{u}_z$.

$$\cos\beta = \frac{\mathbf{u}'' \cdot \mathbf{u}_z}{|\mathbf{u}''|\,|\mathbf{u}_z|} = d \tag{11-24}$$

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

$$\mathbf{u}'' \times \mathbf{u}_z = \mathbf{u}_y |\mathbf{u}''|\,|\mathbf{u}_z| \sin\beta \tag{11-25}$$

with the Cartesian form

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

we find that

$$\sin\beta = -a \tag{11-27}$$

Thus, the transformation matrix for rotation of **u**″ about the y axis is

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

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

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

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

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

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

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

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

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

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

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

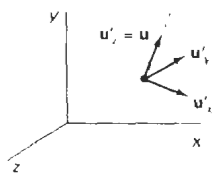


Figure 11-15
Local coordinate system for a rotation axis defined by unit vector **u**.

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

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

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

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

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

Rotations with Quaternions

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

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

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

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

$$s = \cos\frac{\theta}{2}, \qquad \mathbf{v} = \mathbf{u}\sin\frac{\theta}{2} \tag{11-36}$$

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

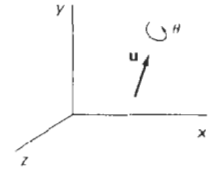


Figure 11-16
Unit quaternion parameters θ and $\mathbf{u}$ for rotation about a specified axis.

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

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

$$\mathbf{P}' = q\mathbf{P}q^{-1} \tag{11-37}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

11-3 SCALING

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

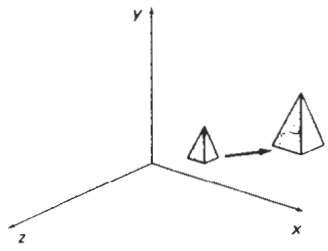


Figure 11-17
Doubling the size of an object with transformation 11-42 also moves the object farther from the origin.

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

or

$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P} \tag{11-43}$$

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

$$x' = x \cdot s_x, \qquad y' = y \cdot s_y, \qquad z' = z \cdot s_z \tag{11-44}$$

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

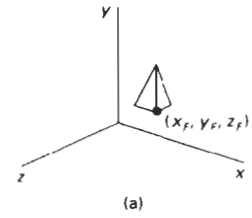
Scaling with respect to a selected fixed position (x_f, y_f, z_f) can be represented with the following transformation sequence:

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

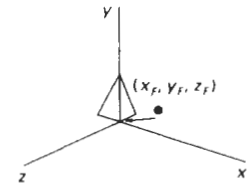
This sequence of transformations is demonstrated in Fig. 11-18. The matrix representation for an arbitrary fixed-point scaling can then be expressed as the concatenation of these translate-scale-translate transformations as

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

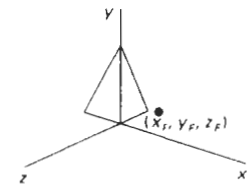
We form the inverse scaling matrix for either Eq. 11-42 or Eq. 11-45 by replacing the scaling parameters s_x, s_y, and s_z with their reciprocals. The inverse ma-



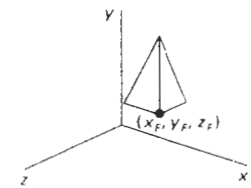
(a)



(b)



(c)



(d)

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

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

11-4 OTHER TRANSFORMATIONS

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

Reflections

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

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

$$RF_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{11-46}$$

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

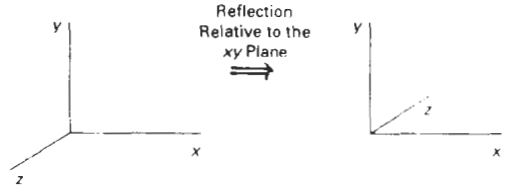


Figure 11-19
Conversion of coordinate specifications from a right-handed to a left-handed system can be carried out with the reflection transformation 11-46.

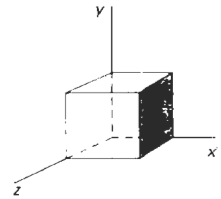
Shears

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

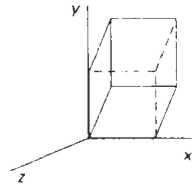
As an example of three-dimensional shearing, the following transformation produces a z-axis shear:

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

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



(a)



(b)

Figure 11-20
A unit cube (a) is sheared (b) by transformation matrix 11-47, with $a = b = 1$.

11-5
COMPOSITE TRANSFORMATIONS

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

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

#define PI 3.14159

typedef float Matrix4x4[4][4];

Matrix4x4 theMatrix;

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

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

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

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

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

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

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

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

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

void rotate3 (wcPt3 p1, wcPt3 p2, float radianAngle)
{
  float length = sqrt ((p2.x - p1.x) * (p2.x - p1.x) +
                       (p2.y - p1.y) * (p2.y - p1.y) +
                       (p2.z - p1.z) * (p2.z - p1.z));
  float cosA2 = cosf (radianAngle / 2.0);
  float sinA2 = sinf (radianAngle / 2.0);
  float a = sinA2 * (p2.x - p1.x) / length;
  float b = sinA2 * (p2.y - p1.y) / length;
  float c = sinA2 * (p2.z - p1.z) / length;
  Matrix4x4 m;

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

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

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

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

void main (int argc, char ** argv)
{
  wcPt3 pts[5] = { 10,10,0, 100,10,0, 125,50,0, 35,50,0, 10,10,0 };
  wcPt3 p1 = { 10,10,0 }, p2 = { 10,10,10 };
  wcPt3 refPt = { 68.0,30.0,0.0 };

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

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

  sleep (10);
  closeGraphics (windowID);
}
```

11-6
THREE-DIMENSIONAL TRANSFORMATION FUNCTIONS

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

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

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

And we can construct composite transformations with the functions

```
composeMatrix3
buildTransformationMatrix3
composeTransformationMatrix3
```

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

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

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

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

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

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

11-7 MODELING AND COORDINATE TRANSFORMATIONS

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

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

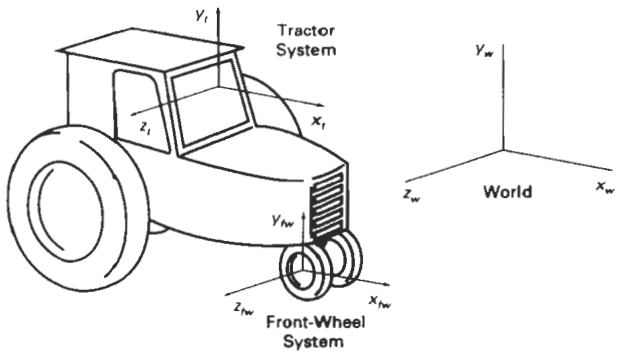


Figure 11-21
Possible coordinate systems used in simulating tractor movement. Wheel rotations are described in the front-wheel system. Turning of the tractor is described by a rotation of the front-wheel system in the tractor system. Both the wheel and tractor reference frames move in the world-coordinate system.

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

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

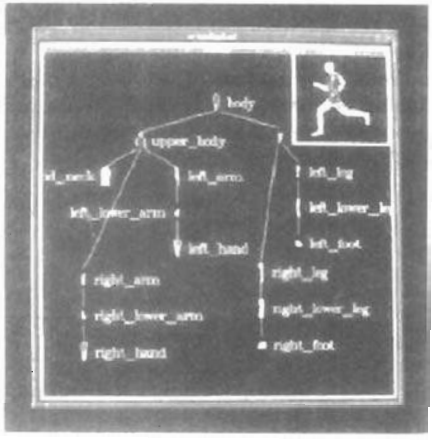


Figure 11-22
Displaying an object hierarchy using the PHIGS Toolkit package developed at the University of Manchester. The displayed object tree is itself a PHIGS structure.
(Courtesy of T. L. J. Howard, J. G. Williams, and W. T. Hewitt, Department of Computer Science, University of Manchester, United Kingdom.)

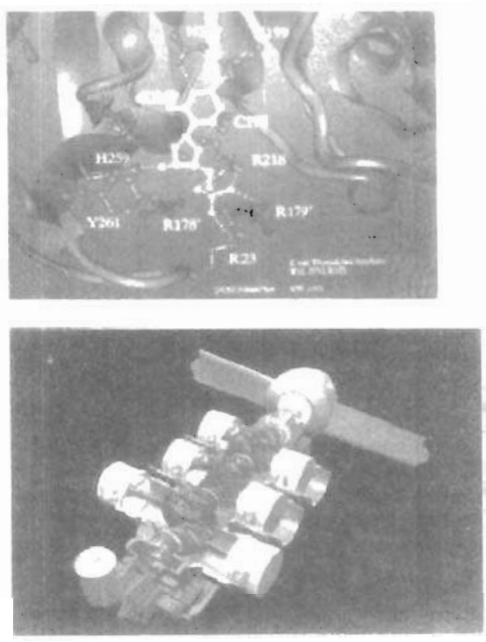


Figure 11-23
Three-dimensional modeling: (a) A ball-and-stick representation for key amino acid residues interacting with the natural substrate of Thymidylate Synthase, modeled and rendered by Julie Newdoll, *UCSF Computer Graphics Lab.* (b) A CAD model showing individual engine components, rendered by Ted Malone, FTI/3D-Magic. (*Courtesy of Silicon Graphics, Inc.*)

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

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

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

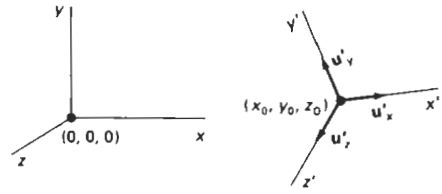


Figure 11-24
Transformation of an object description from one coordinate system to another.

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

Summary

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

SUMMARY

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

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

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

REFERENCES

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

EXERCISES

11-1. Prove that the multiplication of three-dimensional transformation matrices for each of the following sequence of operations is commutative:
(a) Any two successive translations.
(b) Any two successive scaling operations.
(c) Any two successive rotations about any one of the coordinate axes.

11-2. Using either Eq. 11-30 or Eq. 11-41, prove that any two successive rotations about a given rotation axis is commutative.

11-3. By evaluating the terms in Eq. 11-39, derive the elements for general rotation matrix given in Eq. 11-40.

11-4. Show that rotation matrix 11-35 is equal to the composite matrix $\mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha)$.

11-5. Prove that the quaternion rotation matrix Eq. 11-40 reduces to the matrix representation in Eq. 11-5 when the rotation axis is the coordinate z axis.

11-6. Prove that Eq. 11-41 is equivalent to the general rotation transformation given in Eq. 11-30.

11-7. Write a procedure to implement general rotation transformations using the rotation matrix 11-35.

11-8. Write a routine to implement quaternion rotations, Eq. 11-41, for any specified axis.

11-9. Derive the transformation matrix for scaling an object by a scaling factor s in a direction defined by the direction angles α, β, and γ.

11-10. Develop an algorithm for scaling an object defined in an octree representation.

11-11. Develop a procedure for animating an object by incrementally rotating it about any specified axis. Use appropriate approximations to the trigonometric equations to speed up the calculations, and reset the object to its initial position after each complete revolution about the axis.

11-12. Devise a procedure for rotating an object that is represented in an octree structure.

11-13. Develop a routine to reflect an object about an arbitrarily selected plane.

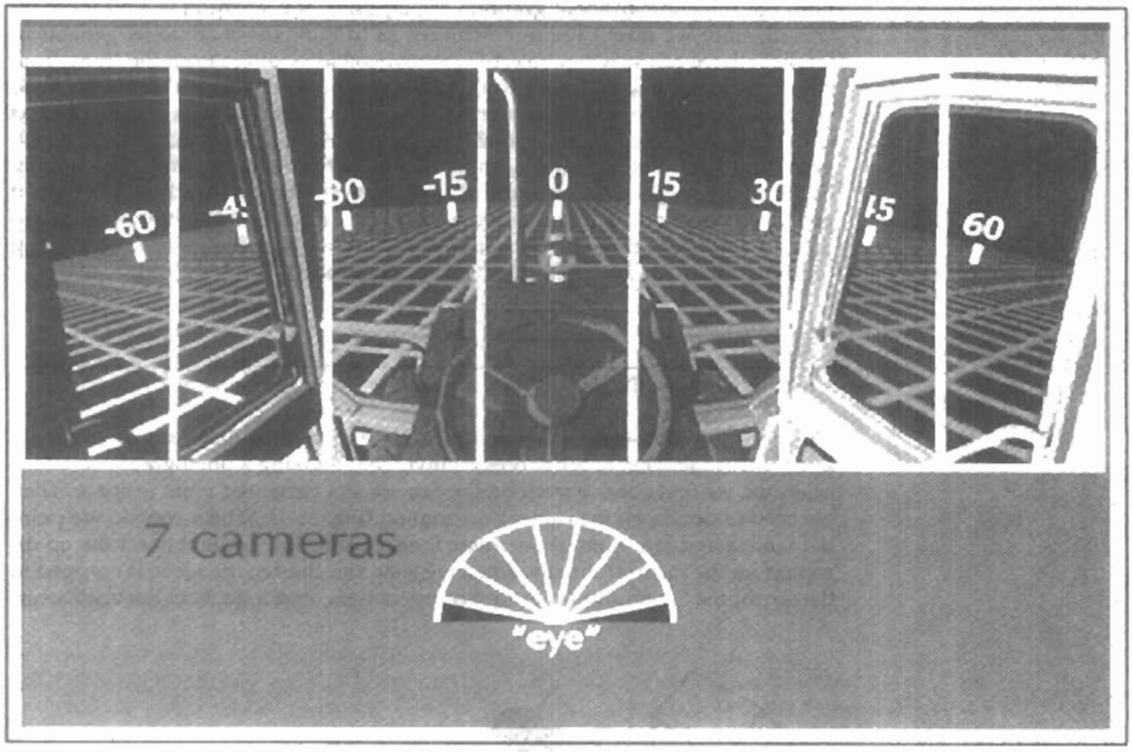
11-14. Write a program to shear an object with respect to any of the three coordinate axes, using input values for the shearing parameters.

11-15. Develop a procedure for converting an object definition in one coordinate reference to any other coordinate system defined relative to the first system.

11-16. Develop a complete algorithm for implementing the procecures for constructive solid modeling by combining three-dimensional primitives to generate new shapes. Initially, the primitives can be combined to form subassemblies, then the subassemblies can be combined with each other and with primitive shapes to form the final assembly. Interactive input of translation and rotation parameters can be used to position the objects. Output of the algorithm is to be the sequence of operations needed to produce the final CSG object.

CHAPTER

12 Three-Dimensional Viewing



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

12-1 VIEWING PIPELINE

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

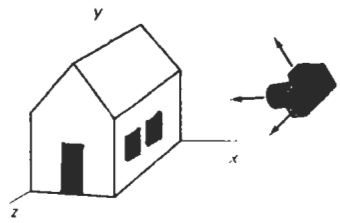


Figure 12-1
Photographing a scene involves selection of a camera position and orientation.

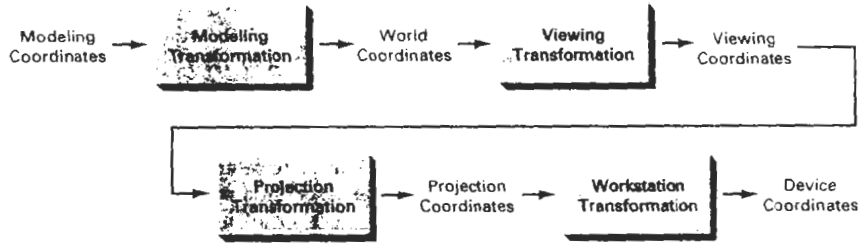


Figure 12-2
General three-dimensional transformation pipeline, from modeling coordinates to final device coordinates.

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

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

12-2
VIEWING COORDINATES

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

Specifying the View Plane

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

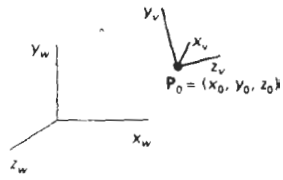


Figure 12-3
A right-handed viewing-coordinate system, with axes x_v, y_v, and z_v, relative to a world-coordinate scene.

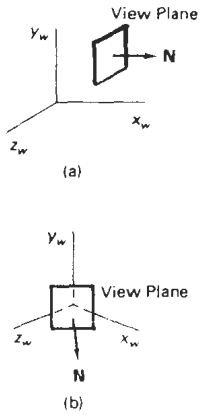


Figure 12-4
Orientations of the view plane for specified normal vector coordinates relative to the world origin. Position (1, 0, 0) orients the view plane as in (a), while (1, 0, 1) gives the orientation in (b).

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

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

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

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

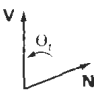


Figure 12-6
Specifying the view-up vector with a twist angle θ_t.

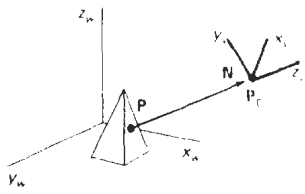


Figure 12-5
Orientation of the view plane for a specified look-at point P, relative to the viewing-coordinate origin P_0.

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

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

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

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

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

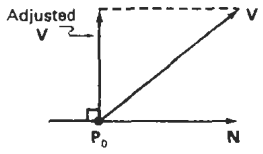


Figure 12-7
Adjusting the input position of the view-up vector **V** to a position perpendicular to the normal vector **N**.

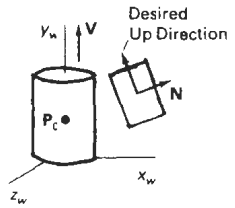


Figure 12-8
Choosing **V** along the y_w axis sets the up orientation for the view plane in the desired direction.

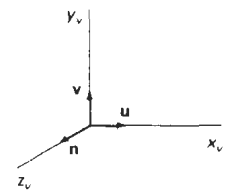


Figure 12-9
A right-handed viewing system defined with unit vectors **u**, **v**, and **n**.

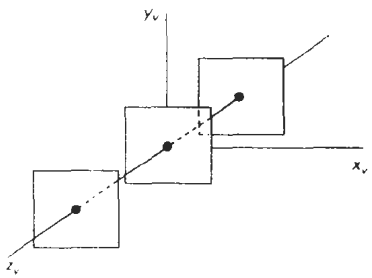


Figure 12-10
View-plane positioning along the z_v axis.

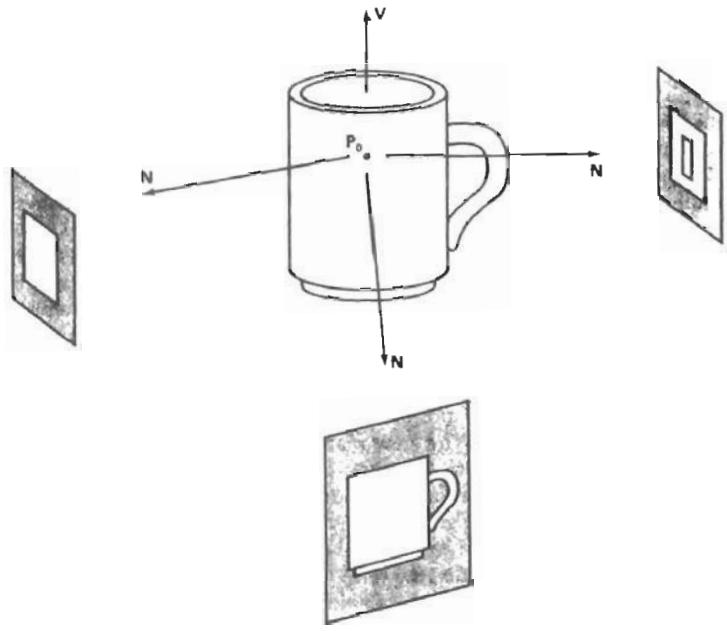


Figure 12-11
Viewing a scene from different directions with a fixed view-reference point.

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

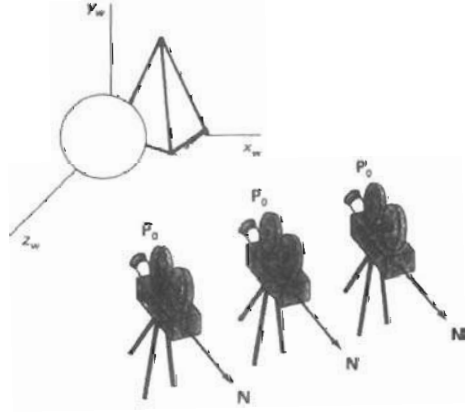


Figure 12-12
Moving around in a scene by changing the position of the view reference point.

Transformation from World to Viewing Coordinates

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

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

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

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{12-1}$$

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

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

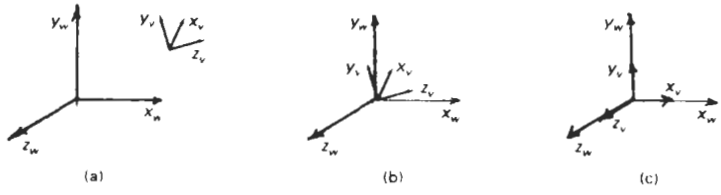


Figure 12-13
Aligning a viewing system with the world-coordinate axes using a sequence of translate-rotate transformations.

cussed in Section 11-7. Given vectors **N** and **V**, these unit vectors are calculated as

$$\mathbf{n} = \frac{\mathbf{N}}{|\mathbf{N}|} = (n_1, n_2, n_3)$$

$$\mathbf{u} = \frac{\mathbf{V} \times \mathbf{N}}{|\mathbf{V} \times \mathbf{N}|} = (u_1, u_2, u_3) \tag{12-2}$$

$$\mathbf{v} = \mathbf{n} \times \mathbf{u} = (v_1, v_2, v_3)$$

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

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

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

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

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

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

12-3 PROJECTIONS

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

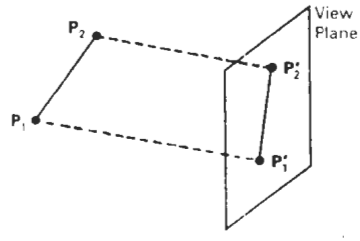


Figure 12-14
Parallel projection of an object to the view plane

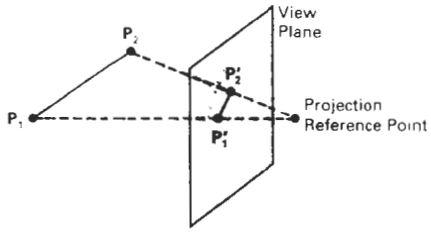


Figure 12-15
Perspective projection of an object to the view plane.

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

Parallel Projections

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

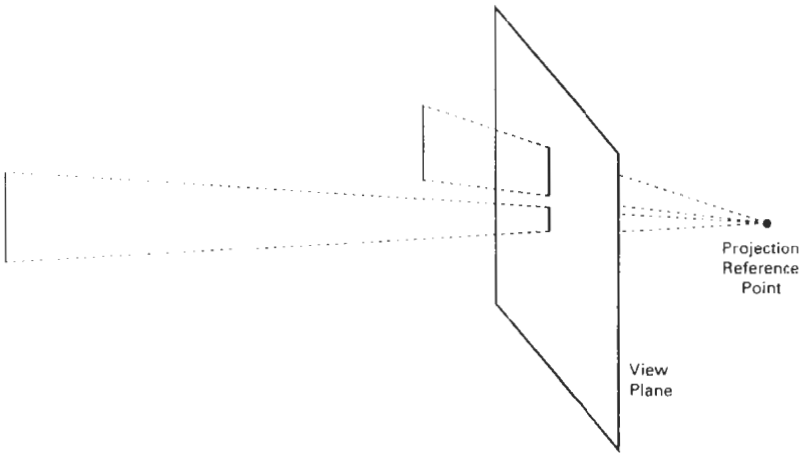


Figure 12-16
Perspective projection of equal-sized objects at different distances from the view plane.

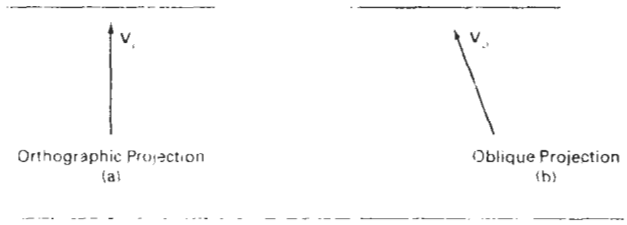


Figure 12-17
Orientation of the projection vector $\mathbf{V}_p$ to produce an orthographic projection (a) and an oblique projection (b).

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

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

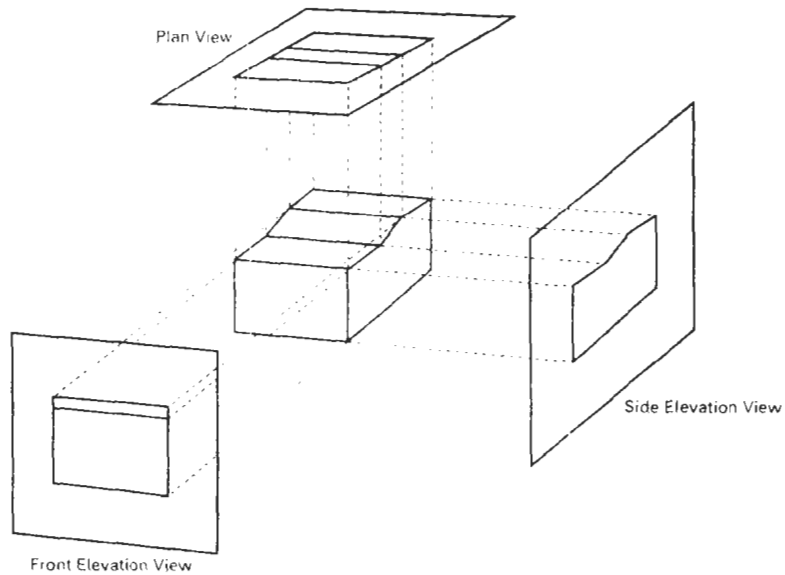


Figure 12-18
Orthographic projections of an object, displaying plan and elevation views.

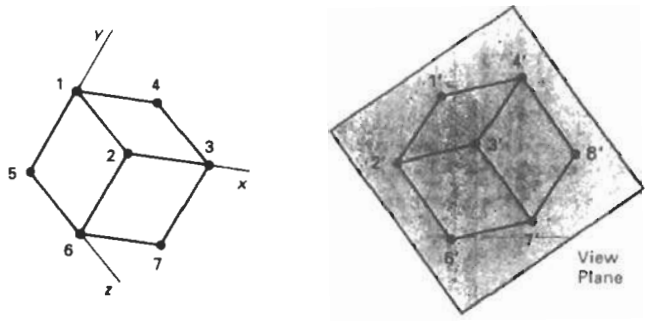


Figure 12-19
Isometric projection for a cube.

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

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

$$x_p = x, \qquad y_p = y \tag{12-5}$$

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

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

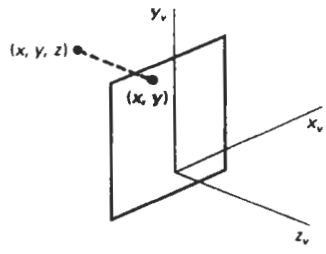


Figure 12-20
Orthographic projection of a point onto a viewing plane.

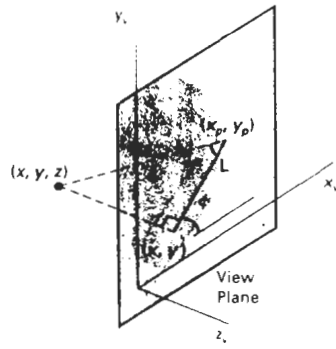


Figure 12-21
Oblique projection of coordinate position (x, y, z) to position (x_p, y_p) on the view plane.

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

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

$$\tan\alpha = \frac{z}{L} \tag{12-7}$$

Thus,

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

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

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

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

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

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

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

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

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

Perspective Projections

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

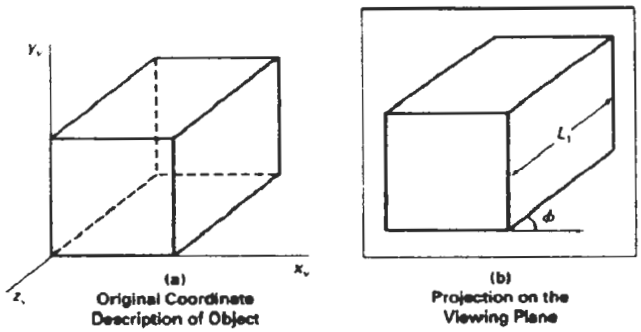


Figure 12-22
Oblique projection of a box onto the $z_v = 0$ plane.

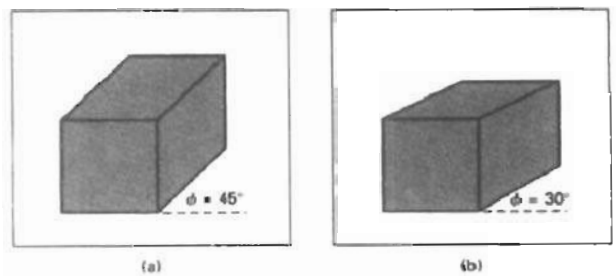


Figure 12-23
Cavalier projections of a cube onto a view plane for two values of angle ϕ.
Note: Depth of the cube is projected equal to the width and height.

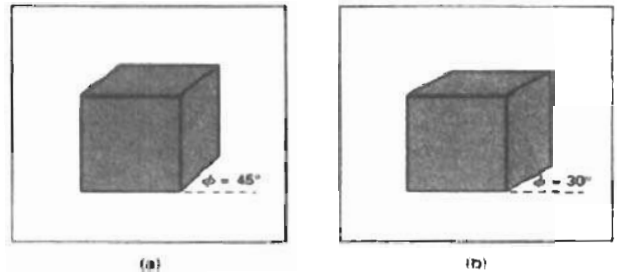


Figure 12-24
Cabinet projections of a cube onto a view plane for two values of angle ϕ. Depth is projected as one-half that of the width and height.

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

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

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

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

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

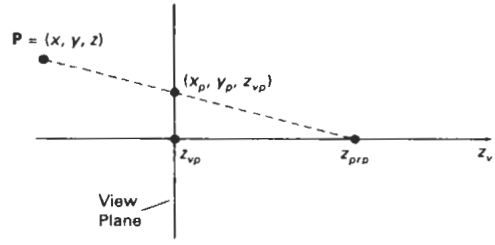


Figure 12-25
Perspective projection of a point **P** with coordinates (x, y, z) to position (x_p, y_p, z_{vp}) on the view plane.

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) = x\left(\frac{d_p}{z_{prp} - z}\right)$$
$$y_p = y\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) = y\left(\frac{d_p}{z_{prp} - z}\right) \tag{12-13}$$

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

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

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

In this representation, the homogeneous factor is

$$h = \frac{z_{prp} - z}{d_p} \tag{12-15}$$

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

$$x_p = x_h/h, \qquad y_p = y_h/h \tag{12-16}$$

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

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

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

$$x_p = x\left(\frac{z_{prp}}{z_{prp} - z}\right) = x\left(\frac{1}{1 - z/z_{prp}}\right)$$
$$y_p = y\left(\frac{z_{prp}}{z_{prp} - z}\right) = y\left(\frac{1}{1 - z/z_{prp}}\right) \tag{12-17}$$

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

$$x_p = x\left(\frac{z_{vp}}{z}\right) = x\left(\frac{1}{z/z_{vp}}\right)$$
$$y_p = y\left(\frac{z_{vp}}{z}\right) = y\left(\frac{1}{z/z_{vp}}\right) \tag{12-18}$$

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

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

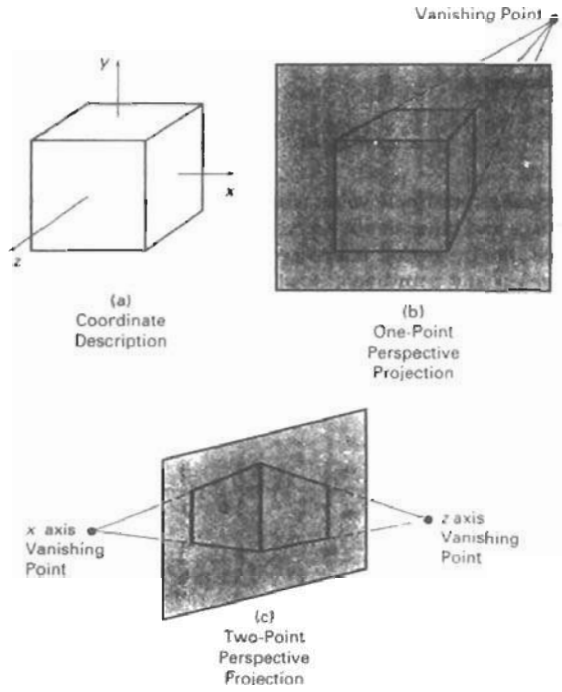


Figure 12-26
Perspective views and principal vanishing points of a cube for various orientations of the view plane relative to the principal axes of the object.

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

12-4 VIEW VOLUMES AND GENERAL PROJECTION TRANSFORMATIONS

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

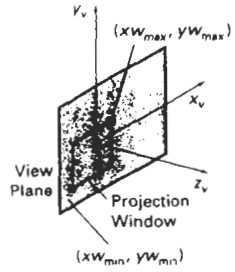


Figure 12-27
Window specification on the view plane, with minimum and maximum coordinates given in the viewing reference system.

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

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

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

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

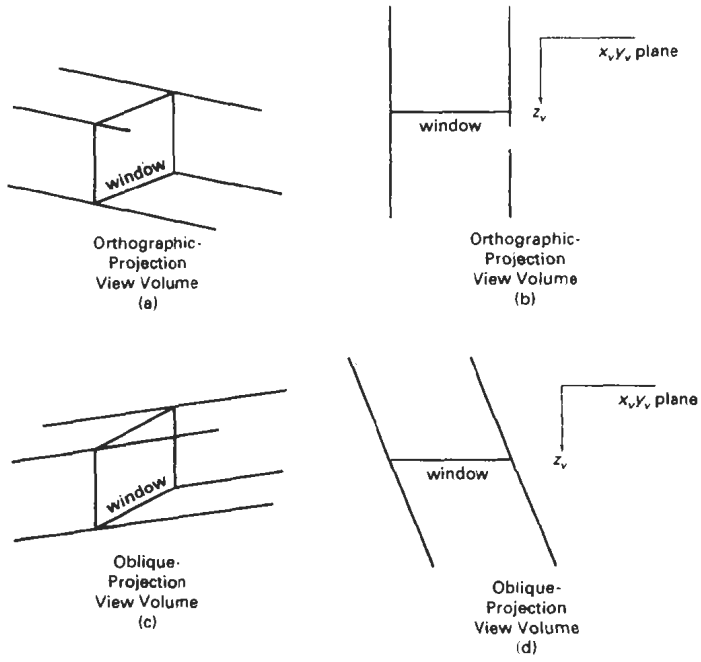


Figure 12-28
View volume for a parallel projection. In (a) and (b), the side and top views of the view volume for an orthographic projection are shown; and in (c) and (d), the side and top views of an oblique view volume are shown.

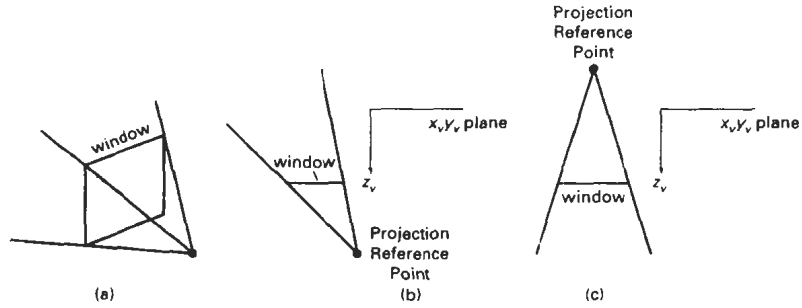


Figure 12-29
Examples of a perspective-projection view volume for various positions of the projection reference point.

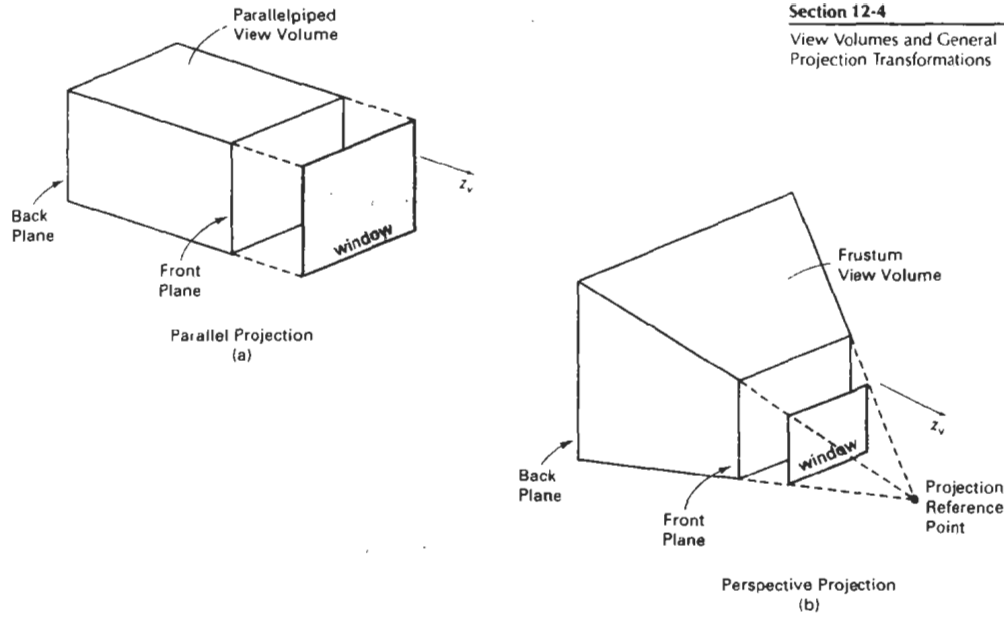


Figure 12-30
View volumes bounded by front and back planes, and by top, bottom, and side planes. Front and back planes are parallel to the view plane at positions z_{front} and z_{back} along the z_v axis.

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

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

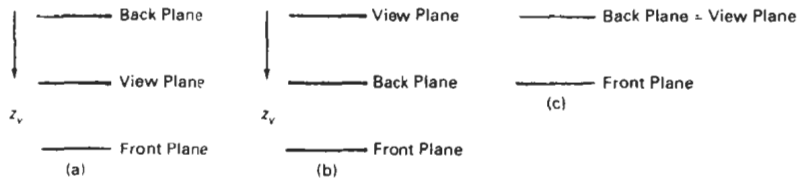


Figure 12-31
Possible arrangements of the front and back clipping planes relative to the view plane.

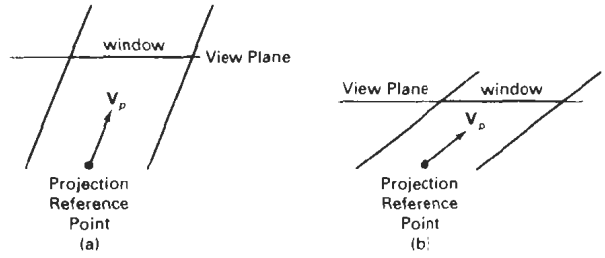


Figure 12-32
Changing the shape of the oblique-projection view volume by moving the window position, when the projection vector $\mathbf{V}_p$ is determined by the projection reference point and the window position.

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

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

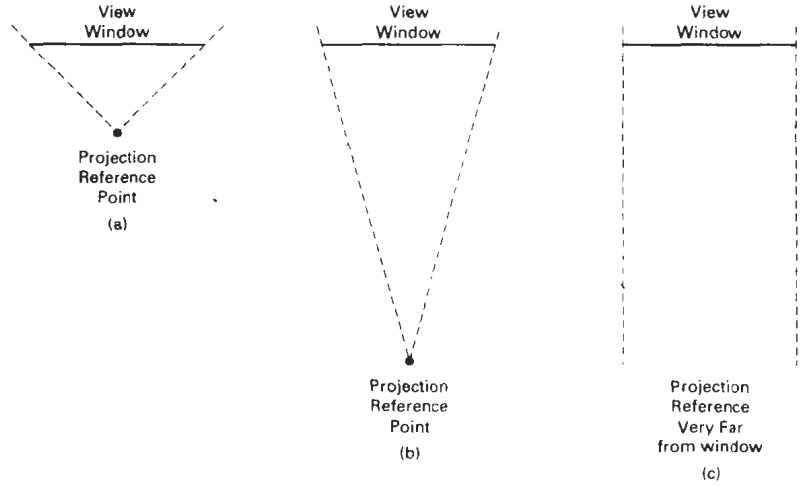


Figure 12-33
Changing perspective effects by moving the projection reference point away from the view plane

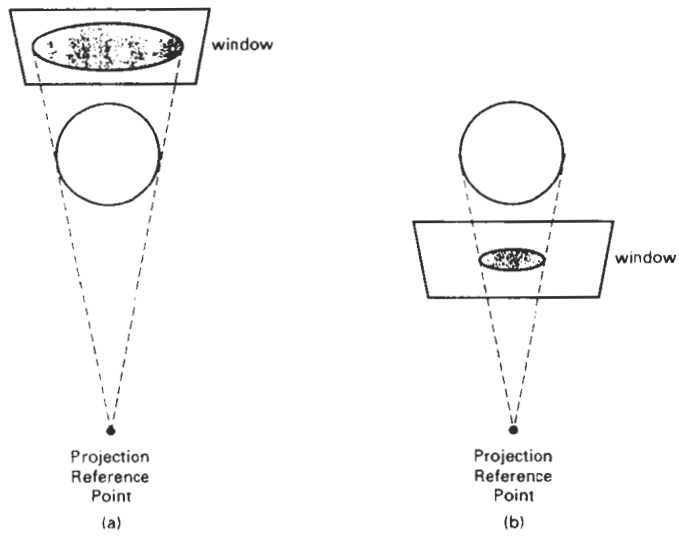


Figure 12-34
Projected object size depends on whether the view plane is positioned in front of the object or behind it, relative to the position of the projection reference point.

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

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

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

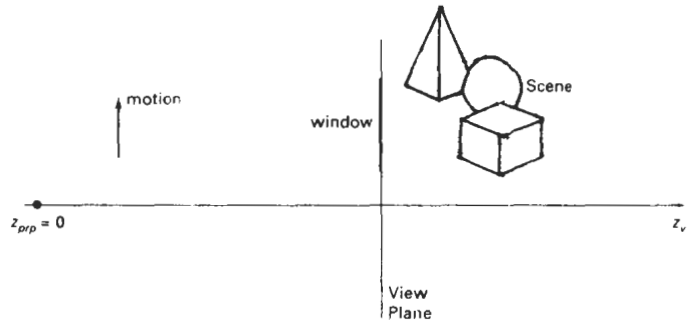


Figure 12-35
View-plane positioning to simulate a camera reference frame for an animation sequence.

General Parallel-Projection Transformations

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

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

Suppose the elements of the projection vector in viewing coordinates are

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

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

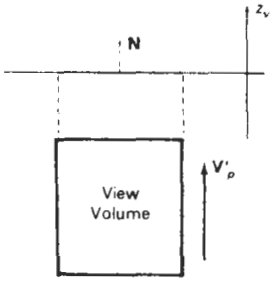


Figure 12-37
Regular parallelepiped view volume obtained by shearing the view volume in Fig. 12-36.

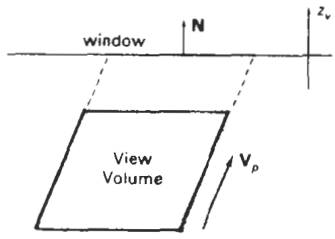


Figure 12-36
Oblique projection vector and associated view volume.

$$\mathbf{V}'_p = \mathbf{M}_{\text{parallel}} \cdot \mathbf{V}_p$$

$$= \begin{bmatrix} 0 \\ 0 \\ p_z \\ 0 \end{bmatrix} \tag{12-20}$$

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

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

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

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

so that the values for the shear parameters are

$$a = -\frac{p_x}{p_z}, \qquad b = -\frac{p_y}{p_z} \tag{12-23}$$

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

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

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

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

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

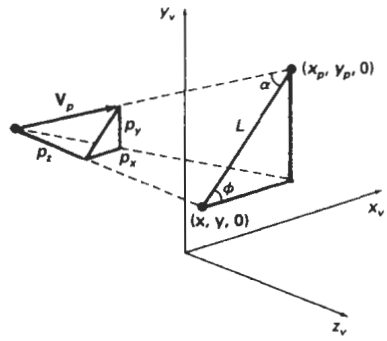


Figure 12-38
Relationship between the parallel-projection vector $\mathbf{V}_p$ and parameters L, α, and ϕ.

General Perspective-Projection Transformations

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

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

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

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

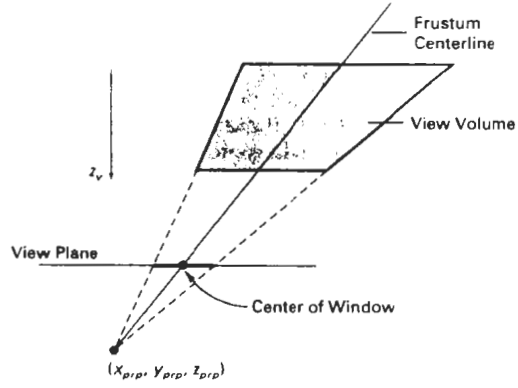


Figure 12-39
General shape for the perspective view volume with a projection reference point that is not on the z_v axis

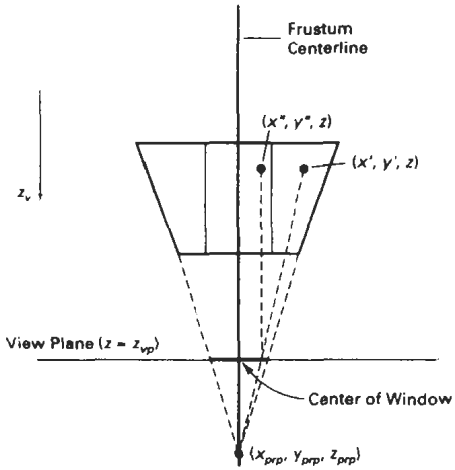


Figure 12-40
Shearing a general perspective view volume to center it on the projection window.

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

$$\mathbf{M}_{\text{shear}} = \begin{bmatrix} 1 & 0 & a & -az_{prp} \\ 0 & 1 & b & -bz_{prp} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{12-26}$$

where the shear parameters are

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

Points within the view volume are transformed by this operation as

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

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

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

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

and the homogeneous matrix representation is

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

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

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

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

12-5 CLIPPING

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

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

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

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

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

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

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

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

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

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

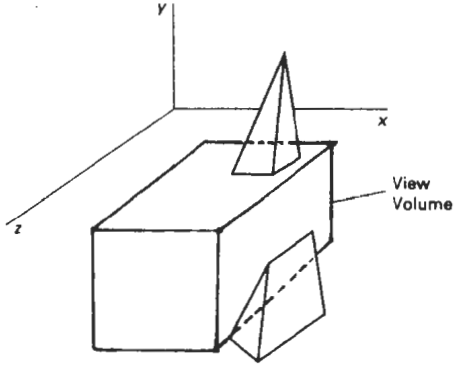


Figure 12-41
An object intersecting a rectangular parallelepiped view volume.

Normalized View Volumes

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

The normalized view volume is a region defined by the planes

$$x = 0, \quad x = 1, \quad y = 0, \quad y = 1, \quad z = 0, \quad z = 1 \qquad (12\text{-}32)$$

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

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

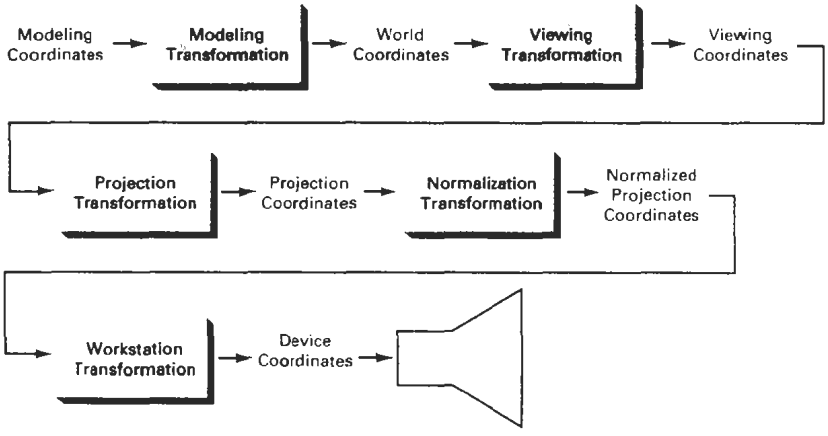


Figure 12-42
Expanded PHIGS transformation pipeline

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

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

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

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

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

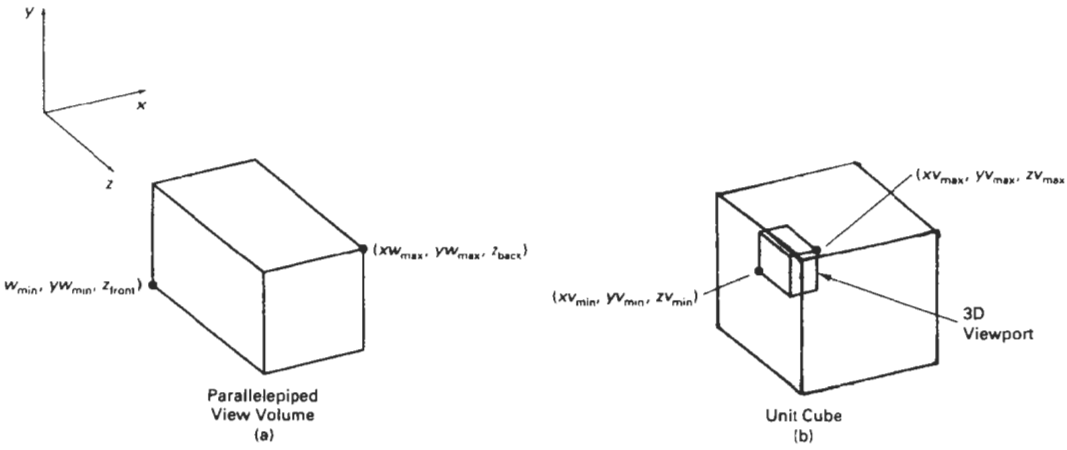


Figure 12-43
Dimensions of the view volume and three-dimensional viewport.

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

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

Viewport Clipping

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

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

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

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

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

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

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

Equations for three-dimensional line segments are conveniently expressed in parametric form. The two-dimensional parametric clipping methods of Cyrus–Beck or Liang–Barsky can be extended to three-dimensional scenes. For a line segment with endpoints $\mathbf{P}_1 = (x_1, y_1, z_1)$ and $\mathbf{P}_2 = (x_2, y_2, z_2)$, we can write the parametric line equations as

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

Coordinates (x, y, z) represent any point on the line between the two endpoints. At $u = 0$, we have the point $\mathbf{P}_1$, and $u = 1$ puts us at $\mathbf{P}_2$.

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

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

When the calculated value for u is not in the range from 0 to 1, the line segment does not intersect the plane under consideration at any point between endpoints $\mathbf{P}_1$ and $\mathbf{P}_2$ (line A in Fig. 12-44). If the calculated value for u in Eq. 12-37 is in the interval from 0 to 1, we calculate the intersection's x and y coordinates as

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

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

Clipping in Homogeneous Coordinates

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

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

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

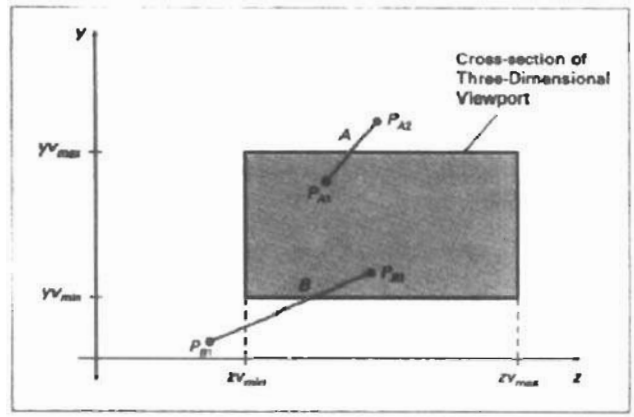


Figure 12-44
Side view of two line segments that are to be clipped against the zv_{min} plane of the viewport. For line A, Eq. 12-37 produces a value of u that is outside the range from 0 to 1. For line B, Eqs. 12-38 produce intersection coordinates that are outside the range from yv_{min} to yv_{max}.

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

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

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

$$x' = \frac{x_h}{h}, \qquad y' = \frac{y_h}{h}, \qquad z' = \frac{z_h}{h} \tag{12-40}$$

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

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

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

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

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

Thus, the homogeneous clipping limits are

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

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

12-6 HARDWARE IMPLEMENTATIONS

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

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

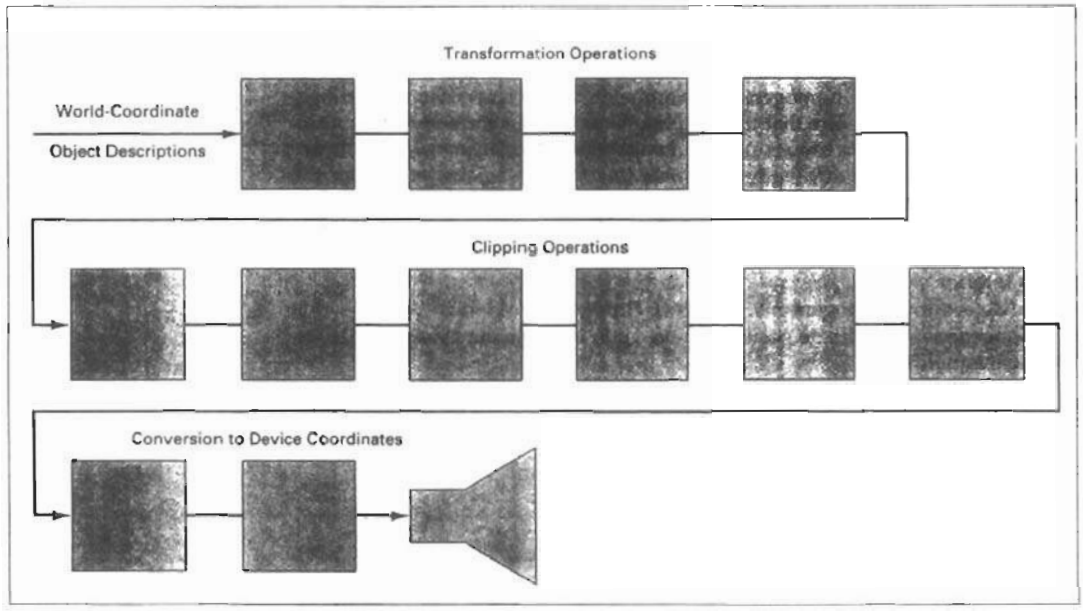


Figure 12-45
A hardware implementation of three-dimensional viewing operations using 12 chips for the coordinate transformations and clipping operations.

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

12-7 THREE-DIMENSIONAL VIEWING FUNCTIONS

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

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

```
evaluateViewOrientationMatrix3 (x0, y0, z0, xN, yN, zN,
                                xV, yV, zV, error, viewMatrix)
```

This function creates the `viewMatrix` from input coordinates defining the viewing system, as discussed in Section 12-2. Parameters `x0`, `y0`, and `z0` specify the

origin (view reference point) of the viewing system. World-coordinate vector (xN, yN, zN) defines the normal to the view plane and the direction of the positive z_v viewing axis. And world-coordinate vector (xV, yV, zV) gives the elements of the view-up vector. The projection of this vector perpendicular to (xN, yN, zN) establishes the direction for the positive y_v axis of the viewing system. An integer error code is generated in parameter `error` if input values are not specified correctly. For example, an error will be generated if we set (xV, yV, zV) parallel to (xN, yN, zN).

To specify a second viewing-coordinate system, we can redefine some or all of the coordinate parameters and invoke `evaluateViewOrientationMatrix3` with a new matrix designation. In this way, we can set up any number of world-to-viewing-coordinate matrix transformations.

The matrix `projMatrix` for transforming viewing coordinates to normalized projection coordinates is created with the function

```
evaluateViewMappingMatrix3 (xwmin, xwmax, ywmin, ywmax,
            xvmin, xvmax, yvmin, yvmax, zvmin, zvmax,
              projType, xprojRef, yprojRef, zprojRef, zview,
                    zback, zfront, error, projMatrix)
```

Window limits on the view plane are given in viewing coordinates with parameters `xwmin`, `xwmax`, `ywmin`, and `ywmax`. Limits of the three-dimensional viewport within the unit cube are set with normalized coordinates `xvmin`, `xvmax`, `yvmin`, `yvmax`, `zvmin`, and `zvmax`. Parameter `projType` is used to choose the projection type as either *parallel* or *perspective*. Coordinate position (`xprojRef`, `yprojRef`, `zprojRef`) sets the projection reference point. This point is used as the center of projection if `projType` is set to *perspective;* otherwise, this point and the center of the view-plane window define the parallel-projection vector. The position of the view plane along the viewing z_v axis is set with parameter `zview`. Positions along the viewing z_v axis for the front and back planes of the view volume are given with parameters `zfront` and `zback`. And the `error` parameter returns an integer error code indicating erroneous input data. Any number of projection matrix transformations can be created with this function to obtain various three-dimensional views and projections.

A particular combination of viewing and projection matrices is selected on a specified workstation with

```
setViewRepresentation3 (ws, viewIndex, viewMatrix, projMatrix,
          xclipmin, xclipmax, yclipmin, yclipmax, zclipmin,
          zclipmax, clipxy, clipback, clipfront)
```

Parameter `ws` is used to select the workstation, and parameters `viewMatrix` and `projMatrix` select the combination of viewing and projection matrices to be used. The concatenation of these matrices is then placed in the workstation **view table** and referenced with an integer value assigned to parameter `viewIndex`. Limits, given in normalized projection coordinates, for clipping a scene are set with parameters `xclipmin`, `xclipmax`, `yclipmin`, `yclipmax`, `zclipmin`, and `zclipmax`. These limits can be set to any values, but they are usually set to the limits of the viewport. Values of *clip* or *noclip* are assigned to parameters `clipxy`, `clipfront`, and `clipback` to turn the clipping routines on or off for the *xy* planes or for the front or back planes of the view volume (or the defined clipping limits).

There are several times when it is convenient to bypass the clipping routines. For initial constructions of a scene, we can disable clipping so that trial placements of objects can be displayed quickly. Also, we can eliminate one or more of the clipping planes if we know that all objects are inside those planes.

Once the view tables have been set up, we select a particular view representation on each workstation with the function

```
setViewIndex (viewIndex)
```

The view index number identifies the set of viewing-transformation parameters that are to be applied to subsequently specified output primitives, for each of the active workstations.

Finally, we can use the **workstation transformation** functions to select sections of the projection window for display on different workstations. These operations are similar to those discussed for two-dimensional viewing, except now our window and viewport regions are three-dimensional regions. The window function selects a region of the unit cube, and the viewport function selects a display region for the output device. Limits, in normalized projection coordinates, for the window are set with

```
setWorkstationWindow3 (ws, xwsWindmin, xwsWindmax,
     ywsWindmin, ywsWindmax, zwsWindmin, zwsWindmax)
```

and limits, in device coordinates, for the viewport are set with

```
setWorkstationViewport3 (ws, xwsVPortmin, xwsVPortmax,
          ywsVPortmin, ywsVPortmax, zwsVPortmin, zwsVPortmax)
```

Figure 12-46 shows an example of interactive selection of viewing parameters in the PHIGS viewing pipeline, using the PHIGS Toolkit software. This software was developed at the University of Manchester to provide an interface to PHIGS with a viewing editor, windows, menus, and other interface tools.

For some applications, composite methods are used to create a display consisting of multiple views using different camera orientations. Figure 12-47 shows

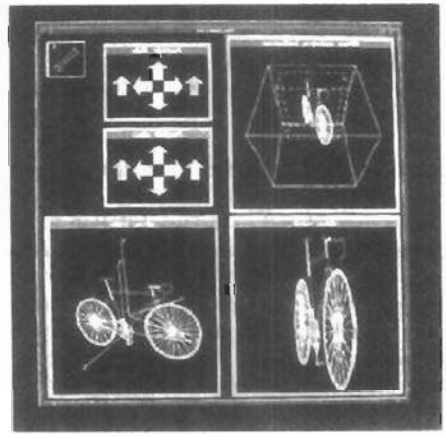


Figure 12-46
Using the PHIGS Toolkit, developed at the University of Manchester, to interactively control parameters in the viewing pipeline. *(Courtesy of T. L. J. Howard, J. G. Williams, and W. T. Hewitt, Department of Computer Science, University of Manchester, United Kingdom.)*

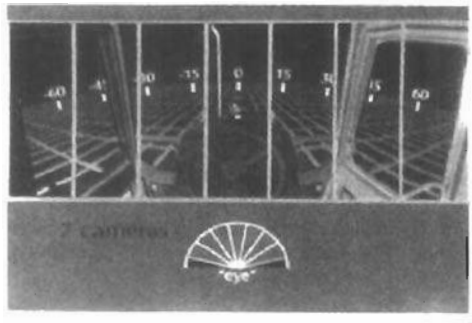


Figure 12-47
A wide-angle view for a virtual-reality display generated with seven sections, each from a slightly different viewing direction. (*Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.*)

a wide-angle perspective display produced for a virtual-reality environment. The wide viewing angle is attained by generating seven views of the scene from the same viewing position, but with slight shifts in the viewing direction.

SUMMARY

Viewing procedures for three-dimensional scenes follow the general approach used in two-dimensional viewing. That is, we first create a world-coordinate scene from the definitions of objects in modeling coordinates. Then we set up a viewing-coordinate reference frame and transfer object descriptions from world coordinates to viewing coordinates. Finally, viewing-coordinate descriptions are transformed to device coordinates.

Unlike two-dimensional viewing, however, three-dimensional viewing requires projection routines to transform object descriptions to a viewing plane before the transformation to device coordinates. Also, three-dimensional viewing operations involve more spatial parameters. We can use the camera analogy to describe three-dimensional viewing parameters, which include camera position and orientation. A viewing-coordinate reference frame is established with a view reference point, a view-plane normal vector **N**, and a view-up vector **V**. View-plane position is then established along the viewing z axis, and object descriptions are projected to this plane. Either perspective-projection or parallel-projection methods can be used to transfer object descriptions to the view plane.

Parallel projections are either orthographic or oblique and can be specified with a projection vector. Orthographic parallel projections that display more than one face of an object are called axonometric projections. An isometric view of an object is obtained with an axonometric projection that foreshortens each principal axis by the same amount. Commonly used oblique projections are the cavalier projection and the cabinet projection. Perspective projections of objects are obtained with projection lines that meet at the projection reference point.

Objects in three-dimensional scenes are clipped against a view volume. The top, bottom, and sides of the view volume are formed with planes that are parallel to the projection lines and that pass through the view-plane window edges. Front and back planes are used to create a closed view volume. For a parallel projection, the view volume is a parallelepiped, and for a perspective projection, the view volume is a frustum. Objects are clipped in three-dimensional viewing by testing object coordinates against the bounding planes of the view volume. Clipping is generally carried out in graphics packages in homogeneous coordinates

after all viewing and other transformations are complete. Then, homogeneous coordinates are converted to three-dimensional Cartesian coordinates.

REFERENCES

For additional information on three-dimensional viewing and clipping operations in PHIGS and PHIGS+, see Howard et al. (1991), Gaskins (1992), and Blake (1993). Discussions of three-dimensional clipping and viewing algorithms can be found in Blinn and Newell (1978), Cyrus and Beck (1978), Riesenfeld (1981), Liang and Barsky (1984), Arvo (1991), and Blinn (1993).

EXERCISES

12-1. Write a procedure to implement the `evaluateViewOrientationMatrix3` function using Eqs. 12-2 through 12-4.

12-2. Write routines to implement the `setViewRepresentation3` and `setViewIndex` functions.

12-3. Write a procedure to transform the vertices of a polyhedron to projection coordinates using a parallel projection with a specified projection vector.

12-4. Write a procedure to obtain different parallel-projection views of a polyhedron by first applying a specified rotation.

12-5. Write a procedure to perform a one-point perspective projection of an object.

12-6. Write a procedure to perform a two-point perspective projection of an object.

12-7. Develop a routine to perform a three-point perspective projection of an object.

12-8. Write a routine to convert a perspective projection frustum to a regular parallelepiped.

12-9. Extend the Sutherland-Hodgman polygon clipping algorithm to clip three-dimensional planes against a regular parallelepiped.

12-10. Devise an algorithm to clip objects in a scene against a defined frustum. Compare the operations needed in this algorithm to those needed in an algorithm that clips against a regular parallelepiped.

12-11. Modify the two-dimensional Liang-Barsky line-clipping algorithm to clip three-dimensional lines against a specified regular parallelepiped.

12-12. Modify the two-dimensional Liang-Barsky line-clipping algorithm to clip a given polyhedron against a specified regular parallelepiped.

12-13. Set up an algorithm for clipping a polyhedron against a parallelepiped.

12-14. Write a routine to perform clipping in homogeneous coordinates.

12-15. Using any clipping procedure and orthographic parallel projections, write a program to perform a complete viewing transformation from world coordinates to device coordinates.

12-16. Using any clipping procedure, write a program to perform a complete viewing transformation from world coordinates to device coordinates for any specified parallel-projection vector.

12-17. Write a program to perform all steps in the viewing pipeline for a perspective transformation.