CAD/CAM THEORY AND PRACTICE

Ibrahim Zeid

Department of Mechanical Engineering
Northeastern University

McGraw-Hill, Inc.

New York St. Louis San Francisco Auckland Bogotá Caracas
Hamburg Lisbon London Madrid Mexico Milan Montreal New Delhi
Paris San Juan São Paulo Singapore Sydney Tokyo Toronto

Forrest, A. R.: "Curves and Surfaces for Computer-Aided Design," Ph.D. Thesis, University of Cambridge, 1968.
Giloi, W. K.: *Interactive Computer Graphics; Data Structures, Algorithms, Languages*, Prentice-Hall, Englewood Cliffs, N.J., 1978.
Miller, J. R.: "Sculptured Surfaces in Solid Models: Issues and Alternative Approaches," *IEEE CG&A*, pp. 37–48, December 1986.
Mortenson, M. E.: *Geometric Modeling*, John Wiley, New York, 1985.
Newman, W. A., and R. F. Sproull: *Principles of Interactive Computer Graphics*, 2d ed., McGraw-Hill, New York, 1979.
Piegl, L.: "Representation of Quadratic Primitives by Rational Polynomials," *CAGD J.*, vol. 2, pp. 151–155, 1985.
Piegl, L.: "The Sphere as a Rational Bezier Surface," *CAGD J.*, vol. 3, pp. 45–52, 1986.
Pratt, M. J.: "Smooth Parametric Surface Approximation to Discrete Data," *CAGD J.*, vol. 2, pp. 165–171, 1985.
Rogers, D. F., and J. A. Adams: *Mathematical Elements for Computer Graphics*, McGraw-Hill, New York, 1976.
Rogers, D. F., S. G. Satterfield, and F. Rodriguez: "Ship Hulls, B-Spline Surfaces, and CAD/CAM," *IEEE CG&A*, pp. 37–43, December 1983.
Satterfield, S. G., and D. F. Rogers: "A Procedure for Generating Contour Lines from a B-Spline Surface," *IEEE CG&A*, pp. 71–75, April 1985.
Sederberg, T. W.: "Piecewise Algebraic Surface Patches," *CAGD J.*, vol. 2, pp. 53–59, 1985.
Worsey, A. J.: "A Modified C2 Coon's Patch," *CAGD J.*, vol. 1, pp. 357–360, 1984.

CHAPTER 7

TYPES AND MATHEMATICAL REPRESENTATIONS OF SOLIDS

7.1 INTRODUCTION

Wireframe and surface geometric modeling techniques have been presented in Chaps. 5 and 6 respectively. This chapter presents the third modeling technique available to designers on a CAD/CAM system, that is, solid modeling. The use of solid modeling in design and manufacturing is increasing rapidly because of the reduced computing costs, fast computing hardware, improved user interfaces, increased capabilities of solid modeling itself, and software improvements. Some twenty modelers are commercially available in the United States, and a body of theory and technology continues to develop. It is forecast[1] that the total solid modeling market will grow steadily over the next few years reaching $2.9 billion by 1991 in comparison to $0.73 billion in 1985. This constitutes a growth in the solid software revenues from $45 million in 1985 to $892 million in 1991. Hardware running solid modeling software is projected to grow from $284 million in 1985 to $2 billion in 1991.

Solid modeling has been acknowledged as the technological solution to automating and integrating design and manufacturing functions. Indeed, the complete definition of part shape (geometry and topology) through solids models has been called a key to CIM. Programmable or flexible automation could very well be achieved via developing application algorithms that operate directly on

[1] Solid modeling market forecast is based on The Merrit Company's *Solid Modeling Today*, vol. 1, no. 1, May 1986.

solid modeling databases. Most of the solid modeling systems in the design/manufacturing environment were installed to test the feasibility of integrating these two functions. However, recent use of these systems has started shifting toward the design and manufacturing of actual parts and assemblies. It is expected, though, that the original goal of automation and integration set for solid modeling will eventually be achieved.

Solid modeling techniques are based on informationally complete, valid, and unambiguous representations of objects. Simply stated, a complete geometric data representation of an object is one that enables points in space to be classified relative to the object, if it is inside, outside, or on the object. This classification is sometimes called spatial addressability. If completeness, validity, and unambiguity are not achieved formally by the geometric modeling technique, the technique has no other option but to depend on users to verify the creations of models interactively. Therefore, automation and integration of tasks such as interference analysis, mass property calculations, finite element modeling, computer aided process planning (CAPP), machine vision, and NC machining are not possible to achieve.

Solid modelers store more information (geometry and topology) than wireframe or surface modelers (geometry only). Both wireframe and surface models are incapable of handling spatial addressability as well as verifying that the model is well formed, the latter meaning that these models cannot verify whether two objects occupy the same space.

Other disadvantages of wireframe modeling have been discussed in Chap. 5. While surface models provide a precise definition of surfaces and can handle complex geometries, they are slow to render, are computationally intensive, and do not further CAD/CAM automation and integration goals. A shaded surface model is by no means considered a solid model. On the other hand, solid modeling produces accurate designs, provides complete three-dimensional definition, improves the quality of design, improves visualization, and has potential for functional automation and integration. However, solid modeling has some limitations. For example, it cannot automatically create other models from the solid definition and neither can it automatically use data created in other models to create a solid. In addition, solid modeling has not been proven for large-scale production applications. Other limitations such as slow rendering and computations as well as poor user interface are fading away with the rapid enhancement of both hardware and software.

Solid modeling (sometimes called volumetric modeling) techniques began to develop in the late 1960s and early 1970s. Early solid modeling projects appeared at this period of time in Europe, Japan, and the United States. Build-1 system, and later Build-2, was developed by Braid's CAD group in Cambridge, England, in 1973. TIPS-1 from Hakkaido University was publicized in 1973. In the mid 1970s, GLIDE-1 was developed by Eastman's group at Carnegie-Mellon University. Baumgart, from Stanford University, introduced Euler operators and a winged-edge polyhedron structure for boundary representation. The Production Automation Project (PAP) was founded at the University of Rochester in 1972. The PAP group headed by Voelcker and Requicha launched the research that led to the PADL-1 and PADL-2 systems. By the late 1970s, solid modeling had

gained enough credibility to penetrate the commercial market. In 1980, Evans and Sutherland began to market Romulus; in 1981 Applicon and Computervision announced their SynthaVision-based and Solidesign systems respectively.

Until recently, solid modeling has been confined to mainframes and large minicomputers (early runs were in batch mode) primarily because of the extensive computations necessary to produce and render solids. By improving solid modeling algorithms, and with the design of frame buffers especially for solid modeling, microcomputer-based solid modeling systems are now available and expected to grow. These buffers are capable of storing enough information for two complete screen images (for animation) and can support 512×512 resolution with 16 bits per pixel. These bits can be partitioned as four bits for cursor and text and 12 bits of colors, thus allowing $4096(2^{12})$ colors to be displayed concurrently from a palette of 16.8 million, assuming 256 intensities for each primary color (refer to Chap. 2). In addition, the buffers do not have processors for computations which can, instead, be performed by the host microcomputer to keep hardware costs down.

User input required to create solid models on existing CAD/CAM systems depends on both the internal representation scheme used by each system as well as the user interface. It is crucial to distinguish between the user interface and the internal data representation of a given CAD/CAM system. The two are quite separate aspects of the systems and can be linked together by software that is transparent to the user. For example, a system that has a B-rep (boundary representation) internal data representation may use a CSG (constructive solid geometry)-oriented user interface; that is, input a solid model by its primitives. Most systems use the building-block approach (CSG oriented) and sweep operations as the basis for user interface. Some early user interfaces were based on the boundary representation scheme and used commands such as "make edge/face," "kill edge/face," etc. Such interfaces are not efficient. Object-oriented user interfaces (input a solid by its features) are more acceptable by users. For example, a user can create a hole in a block using a command such as "create hole" instead of the "subtract cylinder" command. In order to best visualize solids on a graphics display, a shaded image is usually displayed. Some systems, especially those based on the boundary representation, are also available to display a mesh on the solid as in the case of surface models.

This chapter covers the theoretical and practical aspects of solids. Throughout the chapter, related issues to constructing solids on CAD/CAM systems are covered. Sections 7.2 to 7.5 are directly related to practicing solid modeling theory. Sections 7.6 to 7.14 discuss the mathematical representations of solids and other important topics. Section 7.15 applies the chapter material to design and engineering applications.

7.2 SOLID MODELS

A solid model of an object is a more complete representation than its surface model. It is unique from the latter in the topological information it stores which potentially permits functional automation and integration. For example, the mass property calculations or finite element mesh generation of an object can be performed fully automatically, at least in theory, without any user intervention. Typically, a solids model consists of both the topological and geometrical data of its corresponding object.

Defining an object with a solid model is the easiest of the available three modeling techniques (curves, surfaces, and solids). Solid models can be quickly created without having to define individual locations as with wireframes. In many cases, solid models are easier to build than wireframe or surface models. For example, representing the intersection of two cylinders using wireframe modeling is not possible unless points on the intersection curve are evaluated in order to be input to a CAD/CAM system and connected with a B-spline curve. Another example is shown in Fig. 7-1. Using solid modeling, the object shown can be created as a block with six cylinders subtracted from it. It is a cumbersome and lengthy process to create the surface model of the same object although only ruled surfaces are needed (refer to Prob. 6.16).

The completeness and unambiguity of solid models are attributed to the information that the related databases of these models store. Unlike wireframe and surface models, which contain only geometric data, solid models contain both geometric data and topological information of the corresponding objects. The difference between geometry and topology is illustrated in Fig. 7-2. Geometry (sometimes called metric information) is the actual dimensions that define the entities of the object. The geometry that defines the object shown in Fig. 7-2 is the lengths of lines L_1, L_2, and L_3, the angles between the lines, and the radius R and the center P_1 of the half-circle. Topology (sometimes called combinatorial structure), on the other hand, is the connectivity and associativity of the object entities. It has to do with the notion of neighborhood; that is, it determines the relational information between object entities. The topology of the object shown in Fig. 7-2*b* can be stated as follows: L_1 shares a vertex (point) with L_2 and C_1, L_2 shares a vertex with L_1 and L_3, L_3 shares a vertex with L_2 and C_1, L_1 and L_3 do not overlap, and P_1 lies outside the object. Based on these definitions, neither geometry nor topology alone can completely model objects. Wireframe and surface models deal only with geometrical information of objects, and are therefore considered incomplete and ambiguous. From a user point of view, geometry is visible and topology is considered to be nongraphical relational information that is stored in solid model databases and are not visible to users.

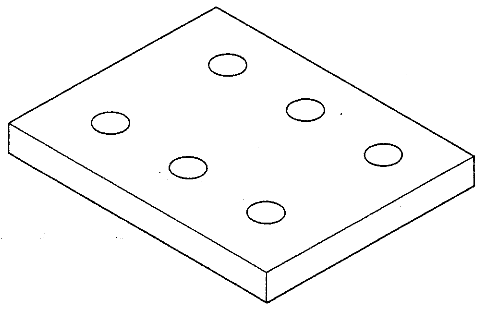


FIGURE 7-1
A typical solid model.

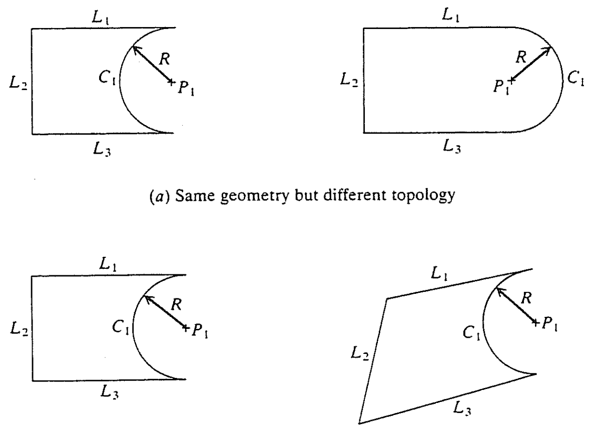

L_1
L_2
L_3
C_1
R
P_1
(*a*) Same geometry but different topology
(*b*) Same topology but different geometry

FIGURE 7-2
Difference between geometry and topology of an object.

For automation and integration purposes, solid models must be accurate. Although accurate models are not a necessity during conceptual design, they are needed for analysis and application algorithms that work off the solid model. Accuracy and speed of creation of a solid model depend directly on the representation scheme, and consequently the data stored in the database of the model. The various available schemes are discussed later in this chapter. Each of those schemes has its own advantages and disadvantages, depending on the application. For example, B-rep modelers can better represent general shapes but usually require more processing time. In contrast, CSG models are easier to build and better suited for display purposes. However, it may be difficult to define a complex shape.

In constructing a solid model on a CAD/CAM system, the user should follow the modeling guidelines discussed in Chap. 3. All the design tools provided by these systems and covered in Part IV of the book, excluding the geometric modifiers, are applicable to solid models. Practically, it might be more convenient to construct solid models in isometric views to enable clear display and visualization of the solid as it is being constructed. It is also recommended that solid entities (primitives) as well as intermediate solids be placed on different layers to allow convenient reference to them during the construction process. A mesh similar to that used with surface models can be added to B-rep-based solid models after they are created. However, solid models are better visualized via shading. Finally, it should be noted that most user interfaces available to input solids have compatibility for CSG input. Such compatibility does not reflect the internal core representation scheme implemented in a particular solid modeling package, and users must consult with the package developers if they wish to know that information.

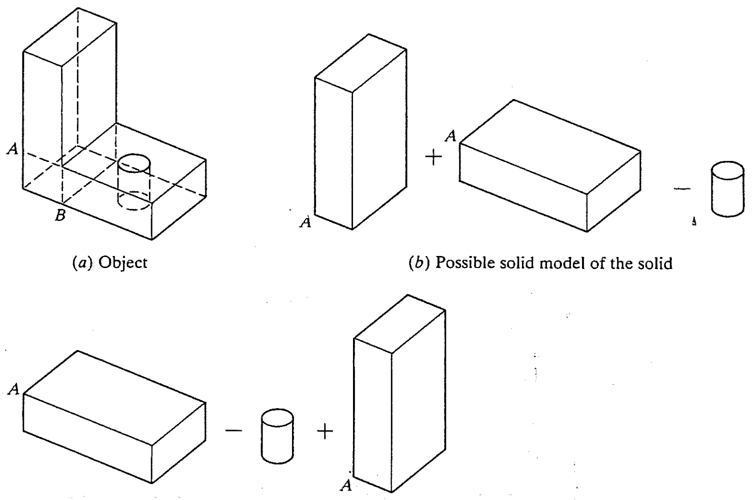

A
B
(a) Object
A
+
A
−
(b) Possible solid model of the solid
A
−
+
A
(c) Alternative solid model of the object

FIGURE 7-3
Nonuniqueness of solid model of an object.

While solid models are complete and unambiguous, they are not unique. An object may be constructed in various ways. Consider the solid shown in Fig. 7-3. One can construct the solid model of the object shown by extending the horizontal block to point A, add two blocks, and subtract a cylinder as shown in Fig. 7-3*b*. Another alternative is shown in Fig. 7-3*c*, where the subtraction is performed first followed by the addition. Other possibilities exist including extending the vertical block to point B instead and repeating the same two alternatives. Regardless of the order and method of construction as well as the representation scheme utilized, the resulting solid model of the object is always complete and unambiguous. However, there will always be a more efficient way than others to construct the solid models as in the case with wireframe and surface models.

Users are now more aware of the potential benefits of solid models. Consequently CAD/CAM vendors are investing more resources into developing solid modeling. However, most existing CAD/CAM systems offer solid modeling as packages that are not linked to wireframe or surface capabilities offered by these systems. It is expected, though, that the next generation of these systems will be based on solid modeling if it matures and proves useful in the production environment.

7.3 SOLID ENTITIES

Most commercially available solid modeling packages have a CSG-compatible user input and therefore provide users with a certain set of building blocks, often called primitives. Primitives are simple basic shapes and are considered the solid modeling entities which can be combined by a mathematical set of boolean operations to create the solid. Primitives themselves are considered valid "off-the-shelf" solids. In addition, some packages, especially those that support sweeping operations, permit users to utilize wireframe entities to create faces that are swept later to create solids. The user usually positions primitives as required before applying boolean operations to construct the final solid.

There is a wide variety of primitives available commercially to users. However, the four most commonly used are the block, cylinder, cone, and sphere. These are based on the four natural quadrics: planes, cylinders, cones, and spheres. For example, the block is formed by intersecting six planes. These quadrics are considered natural because they represent the most commonly occurring surfaces in mechanical design which can be produced by rolling, turning, milling, cutting, drilling, and other machining operations used in industry. Planar surfaces result from rolling, chamfering, and milling; cylindrical surfaces from turning or filleting; spherical surfaces from cutting with a ball-end cutting tool; conical surfaces from turning as well as from drill tips and countersinks. Natural quadrics are distinguished by the fact that they are combinations of linear motion and rotation. Other surfaces, except the torus, require at least dual axis control.

From a user-input point of view and regardless of a specific system syntax, a primitive requires a set of location data, a set of geometric data, and a set of orientation data to define it completely. Location data entails a primitive local coordinate system and an input point defining its origin. Geometrical data differs from one primitive to another and are user-input. Orientation data is typically used to orient primitives properly relative to the MCS or WCS of the solid model under construction. Primitives are usually translated and/or rotated to position and orient them properly before applying boolean operations. Following are descriptions of the most commonly used primitives (refer to Fig. 7-4):

1. *Block.* This is a box whose geometrical data is its width, height, and depth. Its local coordinate system $X_L Y_L Z_L$ is shown in Fig. 7-4. Point P defines the origin of the $X_L Y_L Z_L$ system. The signs of W, H, and D determine the position of the block relative to its coordinate system. For example, a block with a negative value of W is displayed as if the block shown in Fig. 7-4 is mirrored about the $Y_L Z_L$ plane.
2. *Cylinder.* This primitive is a right circular cylinder whose geometry is defined by its radius (or diameter) R and length H. The length H is usually taken along the direction of the Z_L axis. H can be positive or negative.
3. *Cone.* This is a right circular cone or a frustum of a right circular cone whose base radius R, top radius (for truncated cone), and height H are user-defined.
4. *Sphere.* This is defined by its radius or diameter and is centered about the origin of its local coordinate system.
5. *Wedge.* This is a right-angled wedge whose height H, width W, and base depth D form its geometric data.
6. *Torus.* This primitive is generated by the revolution of a circle about an axis lying in its plane (Z_L axis in Fig. 7-4). The torus geometry can be defined by the radius (or diameter) of its body R_1 and the radius (or diameter) of the centerline of the torus body R_2, or the geometry can be defined by the inner radius (or diameter) R_I and outer radius (or diameter) R_O.

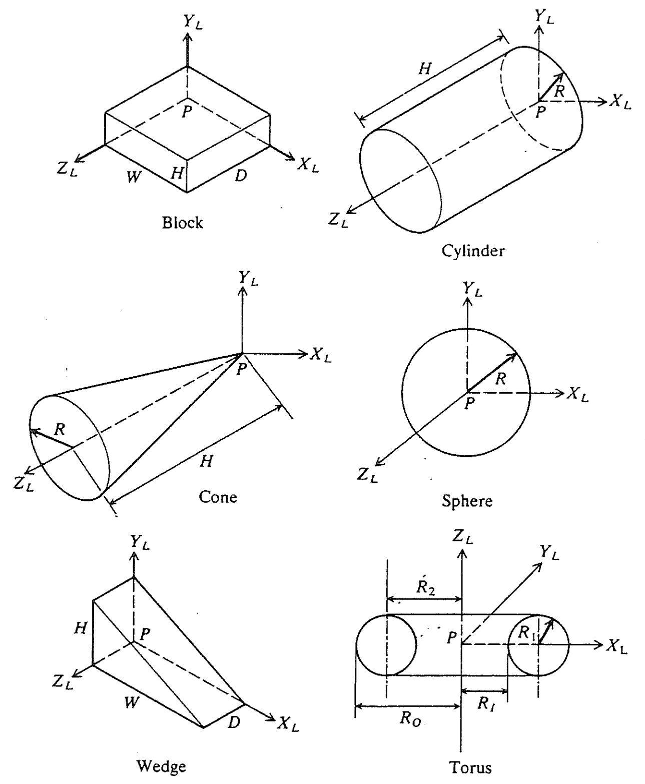


FIGURE 7-4
Most common primitives.

For all the above primitives, there are default values for the data defining their geometries. Most packages use default values of 1. In addition, the local coordinate systems for the various primitives shown in Fig. 7-4 may change from one package to another. Some packages assume that the origin, P, of the local coordinate system is coincident with that of the MCS or WCS and require the user to translate the primitive to the desired location, thus eliminating the input of point P by the user.

Two or more primitives can be combined to form the desired solid. To ensure the validity of the resulting solid, the allowed combinatorial relationships between primitives are achieved via boolean (or set) operations. The available boolean operators are union ($\cup$ or $+$), intersection ($\cap$ or I), and difference ($-$). The union operator is used to combine or add together two objects or primitives. Intersecting two primitives gives a shape equal to their common volume. The difference operator is used to subtract one object from the other and results in a shape equal to the difference in their volumes. Figure 7-5 shows boolean operations of a block A and a cylinder B.

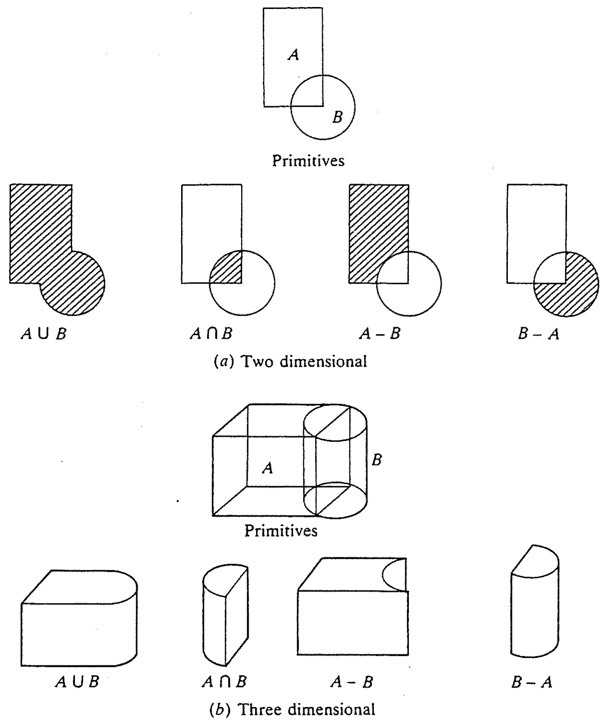
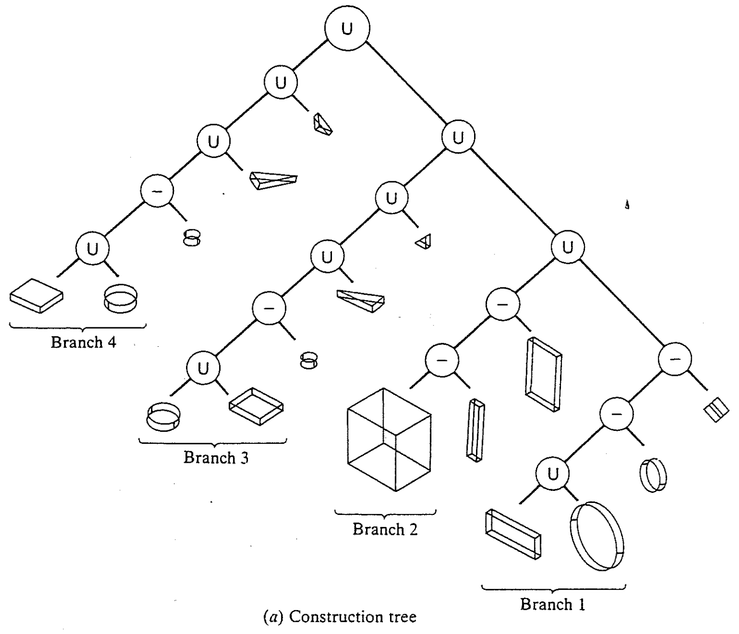


FIGURE 7-5
Boolean operations of a block A and cylinder B.

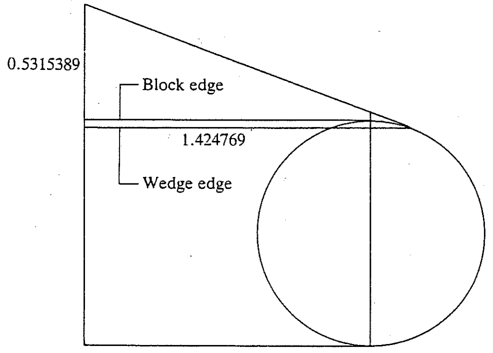
Example 7.1. Create the solid model of the guide bracket shown in Fig. 5-2.

Solution. The creation of the solid model of the guide bracket is much simpler than its wireframe and surface models created in Examples 5.1 and 6.1 respectively. In fact, combinations of blocks and cylinders are all that is needed to create the solid model. While translational sweep can be used to create the solid model, it is not discussed in this example and is left to the reader as an exercise. The following steps may be followed to construct the solid model:

1. Follow the setup procedure discussed in Chap. 3.
2. To create the upper part of the object, create a block of size $2 \times 1 \times 0.25$, and two cylinders of sizes $R = 1.0$, $H = 0.25$ and $R = 0.5$, $H = 0.25$. Create another block of size $0.5 \times 0.5 \times 0.25$ and rotate it 45° about the Z axis (assuming the MCS shown in Fig. 5-2 is used here). These primitives are combined to produce the upper part as shown by branch 1 of the tree shown in Fig. 7-6*a*. The locations and orientations of these primitives can be easily done and are not discussed here.
3. In a similar fashion, branches 2, 3, and 4 of the tree show how to create the lower part of the object, the left flange, and the right flange respectively.
4. The union of all the four branches produces the final solid model.



(*a*) Construction tree

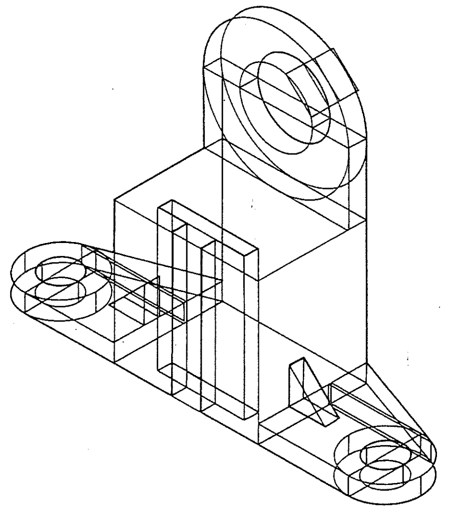


(*b*) Determining wedge dimensions via wireframe modeling

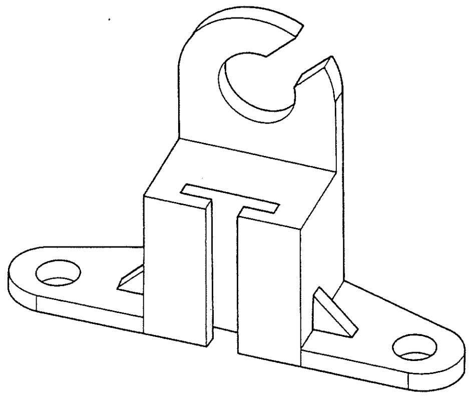
FIGURE 7-6
Solid model of the guide bracket of Example 5.1.

In the above steps, the dimensions of the horizontal wedges used to create the flange are obtained using a wireframe construction, as shown in Fig. 7-6*b*. Figure 7-6*c* and *d* shows the primitives used to create the model and the final solid model respectively.

It is useful in practice to place primitives, intermediate solids, and the final solid in the above example on separate layers. This makes management of creating the solid model easier. For example, if two blocks are added and the result is placed on the same layer, distinguishing between the three solids becomes difficult and requires some of them to be blanked. A need to reuse a primitive after repositioning it might arise, in which case having it on a different layer is very helpful.



(*c*) Primitives in their proper locations and orientations



(*d*) Final solid model

FIGURE 7-6 *(continued)*

The above example illustrates most of the experiences encountered in creating solid models that do not have sculptured surfaces. If the reader were to create the guide bracket on a CAD/CAM system and keep track of the amount of time and effort needed to create the wireframe, surface, and solid models of the object, the latter would clearly require the minimum of both time and effort. This results from the richness of information embedded in defining the primitives and boolean operations. Consequently, solid models appeal strongly to engineering, design, and manufacturing applications.

7.4 SOLID REPRESENTATION

Solid representation of an object can support reliably and automatically, at least in theory, related design and manufacturing applications due to its informational completeness. Such representation is based fundamentally on the notion that a physical object divides an *n*-dimensional space, E^n, into two regions: interior and exterior separated by the object boundaries. A region is defined as a portion of space E^n and the boundary of a region is a closed surface, as in the case of a sphere, or a collection of open surfaces connected at proper edges, as in the case of a box.

In terms of the above notion, a solid model of an object is defined mathematically as a point set S in three-dimensional euclidean space (E^3). If we denote the interior and boundary of the set by iS and bS respectively, we can write

$$S = \mathrm{i}S \cup \mathrm{b}S \tag{7.1}$$

and if we let the exterior be defined by cS (complement of S), then

$$W = \mathrm{i}S \cup \mathrm{b}S \cup \mathrm{c}S \tag{7.2}$$

where W is the universal set, which in the case of E^3 is all possible three-dimensional points.

The solid definition given by Eq. (7.1) introduces the concept of geometric closure which implies that the interior of the solid is geometrically closed by its boundaries. Thus, Eq. (7.1) can be rewritten as

$$S = \mathrm{k}S \tag{7.3}$$

where kS is the closure of the solid or point set S, and is given by the right-hand side of Eq. (7.1); that is, $\mathrm{k}S = \mathrm{i}S \cup \mathrm{b}S$.

Figure 7-7 shows the geometric explanation of Eqs. (7.1) to (7.3). It should be noted here that both wireframe and surface models lack geometric closure which is the main reason for their incompleteness and ambiguity. Based on Eq. (7.1), an object is represented by bS (its boundary) only in both modeling techniques. In the wireframe technique, bS represents E^3 curves that occupy one-dimensional parametric regions while it represents E^3 surfaces that occupy two-dimensional regions in surface modeling.

The foundations of formalizing the solid modeling theory have been well established by the Requicha and Voelcker research group (PADL-1 and PADL-2 authors and developers) and others. The successful representation of solid models in computers and their utilization in engineering applications depend on their properties as well as the properties of the schemes representing them. In the context of the solid modeling theory, the solid model (sometimes called the

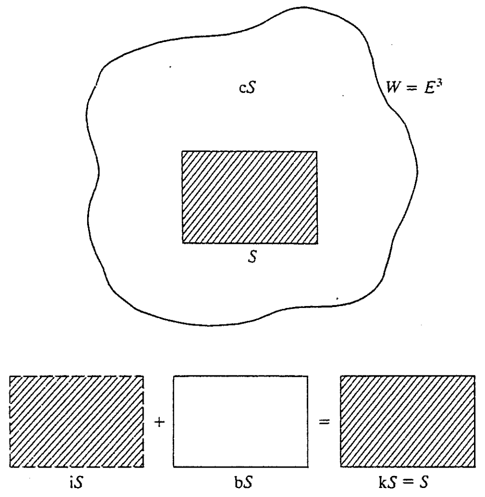


FIGURE 7-7
Solid and geometric closure definitions.

abstract solid) is considered the mathematical model of the real object (sometimes called the physical solid). The properties of this mathematical model determine its behavior when geometric algorithms manipulate its related data structure. The importance of these properties can perhaps be realized if we relate to other classical engineering fields. As a dynamic model of a car dictates the relevance of the associated equilibrium equation and its results, a mathematical model of an object decides the class of algorithms that can be applied to it and the level of their automation.

The properties that a solid model or an abstract solid should capture mathematically can be stated as follows:

1. Rigidity. This implies that the shape of a solid model is invariant and does not depend on the model location or orientation in space.
2. Homogeneous three-dimensionality. Solid boundaries must be in contact with the interior. No isolated or dangling boundaries (see Fig. 7-8) should be permitted.
3. Finiteness and finite describability. The former property means that the size of the solid is not infinite while the latter ensures that a limited amount of information can describe the solid. The latter property is needed in order to be able to store solid models into computers whose storage space is always limited. It should be noted that the former property does not include the latter and vice versa. For example, a cylinder which may have a finite radius and length may be described by an infinite number of planar faces.
4. Closure under rigid motion and regularized boolean operations. This property ensures that manipulation of solids by moving them in space or changing them via boolean operations must produce other valid solids.
5. Boundary determinism. The boundary of a solid must contain the solid and hence must determine distinctively the interior of the solid.

The mathematical implication of the above properties suggests that valid solid models are bounded, closed, regular, and semi-analytic subsets of E^3. These subsets are called r-sets (regularized sets). Intuitively, r-sets are "curved

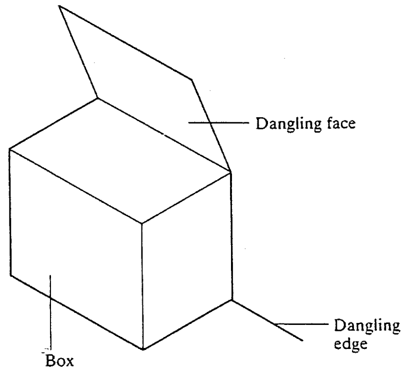


FIGURE 7-8
Example of isolated boundaries.

polyhedra" with "well-behaved" boundaries. The point set S that defines a solid model and is given by Eq. (7.1) is always an r-set. Intuitively, a "closed regular set" means that the set is closed and has no dangling portions, as shown in Fig. 7-8, and a "semi-analytic set" means that the set does not oscillate infinitely fast anywhere within the set. The concept of "semi-analytic set" is important in choosing equations to describe surfaces or primitives of solid models. For example, the point set that satisfies $\sin(x) < 0$ is a semi-analytic set while the set that satisfies $\sin(1/x) < 0$ is not because the function $\sin(1/x)$ oscillates fast when x approaches zero.

Having discussed the desired properties of solid models, let us discuss the properties of representation schemes that usually operate on point sets or r-sets to produce valid solid models. A representation scheme is defined as a relation that maps a valid point set into a valid model. For example, a CSG scheme maps valid primitives into valid solids via boolean operations. Informally, a representation scheme is unambiguous or complete, and unique if one model produced by the scheme represents one and only one object, that is, one-to-one mapping. A scheme is unambiguous or complete, but not unique, if more than one model can represent the object (refer to Fig. 7-3). On the other hand, a scheme is ambiguous or incomplete if one model can represent more than one object, as in the case of wireframe models. Figure 7-9 shows these various schemes.

The formal properties of representation schemes which determine their usefulness in geometric modeling can be stated as follows:

1. Domain. The domain of a representation scheme is the class of objects that the scheme can represent or it is the geometric coverage of the scheme.
2. Validity. The validity of a representation scheme is determined by its range, that is, the set of valid representations or models it can produce. If a scheme produces an invalid model, the CAD/CAM system in use may crash or the model database may be lost or corrupted if an algorithm is invoked on the model database. Validity checks can be achieved in three ways: test the resulting databases via a given algorithm, build checks into the scheme generator itself, or design scheme elements (such as primitives) that can be manipulated via a given syntax.
3. Completeness or unambiguousness. This property determines the ability of the scheme to support analysis and other engineering applications. A complete scheme must provide models with sufficient data for any geometric calculation to be performed on them.
4. Uniqueness. This property is useful to determine object equality. It is a custom in algebra to check for uniqueness but it is rare to do so in geometry. This is because it is difficult to develop algorithms to detect the equivalence of two objects and it is computationally expensive to implement these algorithms if they exist. Positional and permutational nonuniqueness are two simple cases shown in Fig. 7-10. Figure 7-10*a* shows a two-dimensional rectangular solid (of side lengths a and b) in two different positions and orientations. The two-dimensional solid S shown in Fig. 7-10*b* is divided into three blocks A, B, and C that can be unioned in a different order.

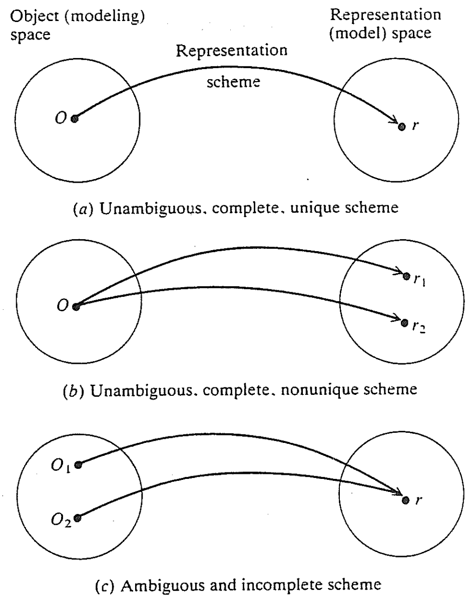

Object (modeling) space
Representation (model) space
Representation scheme
O
r
(a) Unambiguous, complete, unique scheme
O
r₁
r₂
(b) Unambiguous, complete, nonunique scheme
O₁
O₂
r
(c) Ambiguous and incomplete scheme

FIGURE 7-9
Classification of representation schemes.

There are other properties of representation schemes such as conciseness, ease of creation, and efficacy in the context of applications. These properties cannot be formalized and therefore are considered informal. Conciseness is a measure of the size of data a scheme requires to describe an object. Concise representation schemes generate compact databases that contain few redundant data, are convenient to store, and are efficient to transmit over data links (networks) from one system to another. Selectively imposed redundancy may save computational time and may increase the number of application algorithms that can utilize the stored data. Ease of creation of a representation is important to users and determines the user-friendliness of a scheme to a great extent. This is why most existing solid modelers have a CSG-compatible user input because it is

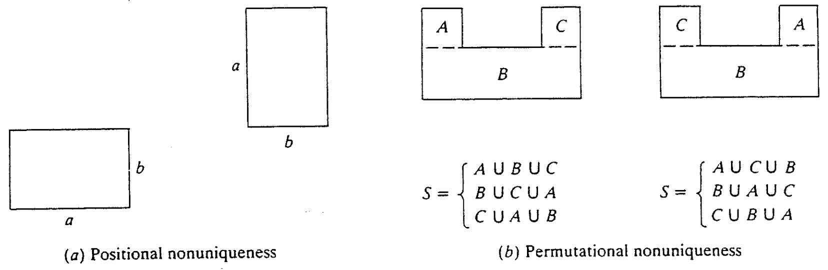

a
b
b
a
(a) Positional nonuniqueness
A
C
B
S = {A ∪ B ∪ C, B ∪ C ∪ A, C ∪ A ∪ B}
C
A
B
S = {A ∪ C ∪ B, B ∪ A ∪ C, C ∪ B ∪ A}
(b) Permutational nonuniqueness

FIGURE 7-10
Positional and permutational nonuniqueness.

concise and easy to create. Efficacy in the context of applications measures how accessible a representation is by downstream applications. Representations of objects in themselves are useless and should be viewed as sources of data for algorithms. Good representation schemes should permit the use of a wide variety of application algorithms for evaluating various functions.

Various representation schemes have been designed and developed, with the above properties in mind, to create solid models of real objects. Nine schemes can be identified. Some of them are more popular than the others. These are half-spaces, boundary representation (B-rep), constructive solid geometry (CSG), sweeping, analytic solid modeling, cell decomposition, spatial enumeration, octree encoding, and primitive instancing. Each of these schemes has its properties, advantages, and disadvantages that are discussed later in the chapter. The three most popular schemes are B-rep, CSG, and sweeping. Most existing solid modeling packages or systems use one or more of the known schemes. Table 7.1 lists some of the existing solid modelers with their core representation scheme. In most packages or systems, one scheme is considered the primary representation

TABLE 7.1
Some available solid modelers

		Primary representation scheme		**User modeling input based on**	
Modeler	**Vendor**	**B-rep**	**CSG**	**B-rep**	**CSG**
BMOD	Auto-trol	×		×	
CATIA	IBM	×		×	×
CMOD	Auto-trol		×		×
DDM SOLIDS	GE Calma	×		×	×
EUCLID	Matra Datavision	×		×	×
GEMSMITH	Vulcan	×		×	×
GEOMED	SDRC	×		×	×
GEOMETRIC MODELING SYSTEM	Graftek	×		×	
ICEM	CDC		×		×
ICM GMS	ICM	×		×	×
INSIGHT	Phoenix Data Systems	×		×	×
MEDUSA	Prime Computer	×		×	×
PADL-2	Cornell University		×		×
PATRAN-G	PDA Engineering	ASM†		Hyper-patches	×
ROMULUS	Evans and Sutherland	×		×	
SOLIDESIGN	Computervision	×		×	×
SOLIDS MODELING II	Applicon		×	×	×
SOLID MODELING SYSTEM	Intergraph	×		×	×
SYNTHVISION	MAGI		×	×	×
TIPS-1	CAM-I		×		×
UNIS-CAD	Sperry Univac		×		×
UNISOLIDS	McDonell Douglas		×	×	×

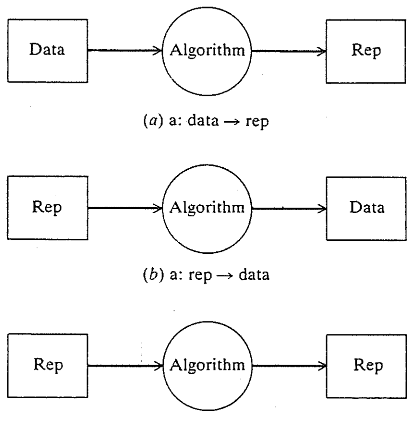
† Analytic solid modeling (see Sec. 7.10 for details).

scheme that can be converted to others or other schemes can be converted to it. For example, a CSG-based system utilizes the CSG scheme as the core of its geometric modeling engine which, in turn, can be converted into a B-rep for application purposes, or sweeping can be converted into a B-rep or CSG by a package whose core representation is one of these two. Conversion between various schemes is not always possible and depends primarily on how data is stored. Conversion from CSG to B-rep, octree, or spatial enumeration is possible. However, converting B-rep to CSG is not well known (conversion in two dimensions is known). Simple sweep can be converted to B-rep, CSG, or cell decomposition.

The major geometric procedures needed for solid modeling, regardless of any representation scheme, are curve/curve, curve/surface, and surface/surface intersection calculations. Conceptually, any geometric entity (a primitive or a surface) could be added to any representation scheme to increase its modeling domain. However, unless such a scheme can support these intersections of the entity, its use in modeling and applications becomes useless. Support of sculptured surface geometries by solid modeling depends on developing efficient methods to perform the intersection calculations for these geometries. Once these methods are available, solid modeling systems would cater to both solid and surface modeling within the same conceptual and algorithmic framework.

Representations of solids are built and invoked via algorithms (sometimes called processors). Informally, an algorithm is a procedure that takes certain input and produces a desired output. Algorithms should be developed carefully and tested for a wide variety of input to ensure their generality, reliability, and consistency. Algorithms can be classified into three types according to their input and output. Some algorithms take data and produce representations; that is, $a: \text{data} \rightarrow \text{rep}$ (reads as algorithm a is defined as taking data and producing representation). These algorithms build, maintain, and manage representations. Representation schemes mentioned above fall into this type. The other type of algorithms compute property values by taking a representation and producing data; that is, $a: \text{rep} \rightarrow \text{data}$. All application algorithms belong to this type. For example, a mass property algorithm takes a solid model representation and produces volume, mass, and inertial properties. Algorithms of the third type take representations and produce representations; that is, $a: \text{rep} \rightarrow \text{rep}$. For example, an algorithm that converts CSG to B-rep or one that simulates (models) processes (such as motion or machining) on objects belongs to this type. An algorithm might take a piece of stock and end up with a machined part. Figure 7-11 illustrates the three types of algorithms.

Conversion of solid models into wireframe models or an edge representation is well understood and is used to generate orthographic views for display and drafting purposes. While the views are generated automatically, they are not dimensioned automatically, and a manual or semi-automatic dimensioning is required. However, the opposite problem of creating solid models from wireframe models or from existing orthographic views or drawings is largely unsolved. Mathematically, this is a problem of converting an edge representation into a solid representation. This problem is not complete or well defined due to two reasons. First, edges of curved solids (curved polyhedra) may not be easily found



(*a*) a: data → rep

(*b*) a: rep → data

(*c*) a: rep → rep

FIGURE 7-11
Types of solid modeling algorithms.

from a finite number of projections. Second, the edge representation itself is ambiguous and can correspond to more than one object. Algorithms for disambiguating wireframe models exist. These algorithms find all possible objects that correspond to one drawing. The main thrust to convert drawings to solid models stems from the large existing industrial base of wireframe models.

7.5 FUNDAMENTALS OF SOLID MODELING

Before covering the details of the various representation schemes, it is appropriate to discuss the details of some of the underlying fundamentals of solid modeling theory. These are geometry, topology, geometric closure, set theory, regularization of set operations, set membership classification, and neighborhood. Geometry and topology have been covered in Sec. 7.2 and geometric closure is introduced in Sec. 7.4. This section covers set theory, regularization, classification, and neighborhood. The significance of these topics to solid modeling stems from the definition of a solid model as a point set in E^3 as given in Eq. (7.1). They provide good rigorous mathematical foundations for developing and analyzing solids.

7.5.1 Set Theory

We begin the review of set theory by introducing some definitions followed by set algebra (operations on sets) and laws (properties) of the algebra of sets. At the end, the concept of ordered pairs and cartesian product is introduced. A set is defined as a collection or aggregate of objects. The objects that belong to the set are called the elements or members of the set. For example, the digits 0, 1, ..., 9 form a set (set of digits) D whose elements are 0, 1, ..., 9. While the concept is relatively simple, the elements of a set must satisfy certain requirements. First, the elements must be well defined to determine unequivocally whether or not any

object belongs to the set; that is, fuzzy sets are excluded. Second, the elements of a set must be distinct and no element may appear twice. Third, the order of the elements within the set must be immaterial. To realize the importance of these requirements in geometric modeling, the reader can apply them to a point set of eight elements which are the corner points of a block.

The elements of a set can be designated by one of two methods: the roster method or the descriptive method. The former involves listing within braces all the elements of the set and the latter involves describing the condition(s) that every element in the set must meet. The set of digits D can be written using the roster and the descriptive methods respectively as

$$D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \tag{7.4}$$

and

$$D = \{x: x = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \tag{7.5}$$

Equation (7.4) reads as "D is equal to the set of elements 0, 1, 2, 3, 4, 5, 6, 7, 8, 9." Equation (7.5) reads as "D is equal to the set of elements x such that x equals 0, 1, 2, 3, 4, 5, 6, 7, 8, 9." The colon in Eq. (7.5) is sometimes replaced by a vertical bar, that is, $D = \{x \mid x = 0, 1, \ldots, 9\}$. Regardless of set designation, set membership and nonmembership is customarily indicated by $\in$ and $\notin$ respectively. If we write $9 \in D$, we mean 9 is an element (or member) of the set of digits D or 9 belongs to D. Similarly, $-2 \notin D$ means that -2 is not an element of D.

Two sets P and Q are equal, written $P = Q$, if the two sets contain exactly the same elements. For example, the two sets $P = \{1, 3, 5, 7\}$ and $Q = \{1, 5, 7, 3\}$ are equal, since every element in P is in Q and every element in Q is in P. The inequality is denoted by $\neq$ ($P \neq Q$ reads "P does not equal Q").

A set R is a subset of another set S if every element in R is in S. The notation for subset is $\subseteq$ and $R \subseteq S$ reads "R is a subset of S." Analogous to $\in$ and $\notin$, the notation for not subset is $\nsubseteq$. If it happens that all elements in R are in S but all elements in S are not in R, then R is called a proper subset of S and is written $R \subset S$. This means that for R to be a proper subset of S, S must have all elements of R plus at least one element that is not in R. For example, given $S = \{1, 3, 5, 7\}$, then $R = \{1, 3, 5, 7\}$ is a subset of S and $R = \{5, 7\}$ is a proper subset of S. Formally, $R \subset S \Leftrightarrow R \cap S = R$ and $R \neq S$ ($\Leftrightarrow$ reads "if and only if") or $R \subset S \Leftrightarrow R \cup S = S$ and $R \neq S$.

There are two sets that usually come to mind when discussing sets and subsets. The universal set W is a set that contains all the elements that the analyst wishes to consider. It is problem-dependent. In solid modeling, W contains E^3 and all points in E^3 are the elements of W. In contrast the null (sometimes referred to as the empty) set is defined as a set that has no elements or members. It is designated by the null set symbol $\varnothing$. The null set is analogous to zero in ordinary algebra.

Having introduced the required definitions, we now discuss set algebra. Set algebra consists of certain operations that can be performed on sets to produce other sets. These operations are simple in themselves but are powerful when combined with the laws of set algebra to solve geometric modeling problems. The operations are most easily illustrated through use of the Venn diagram named after the English logician John Venn. It consists of a rectangle that conceptually represents the universal set. Subsets of the universal set are represented by circles drawn within the rectangle or the universal set.

The three essential set operations are complement, union, and intersection. The complement of P, denoted by cP (reads "P complement"), is the subset of elements of W that are not members of P, that is,

$$cP = \{x: x \notin P\} \tag{7.6}$$

The shaded portion of the Venn diagram in Fig. 7-12*a* shows the complement of P.

The union of two sets $P \cup Q$ (read "P union Q") is the subset of elements of W that are members of either P or Q, that is,

$$P \cup Q = \{x: x \in P \text{ or } x \in Q\} \tag{7.7}$$

The union is shown in Fig. 7-12*b* as the shaded area.

The intersection of two sets $P \cap Q$ (read "P intersect Q") is the subset of elements of W that are simultaneously elements of both P and Q, that is,

$$P \cap Q = \{x: x \in P \text{ and } x \in Q\} \tag{7.8}$$

The shaded portion in Fig. 7-12*c* shows the intersection of P and Q. It is easy to realize that $P \cap W = P$ and $P \cap cP = \varnothing$. Sets that have no common elements are termed disjoint or mutually exclusive.

Two additional set operators that can be derived from the above set operations are difference and exclusive union. The difference of two sets $P - Q$ (read "P minus Q") is the subset of elements of W that belong to P and not Q, that is,

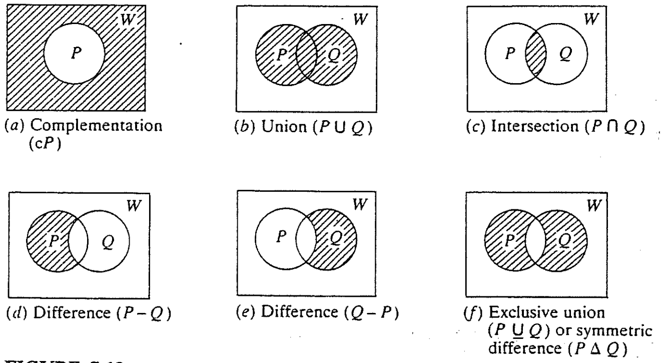
$$P - Q = \{x: x \in P \text{ and } x \notin Q\} \tag{7.9}$$

or

$$Q - P = \{x: x \in Q \text{ and } x \notin P\} \tag{7.10}$$

Figure 7-12*d* and *e* shows the difference operator. The difference can also be expressed as

$$P - Q = P \cap cQ \tag{7.11}$$



(*a*) Complementation (cP)
(*b*) Union ($P \cup Q$)
(*c*) Intersection ($P \cap Q$)
(*d*) Difference ($P - Q$)
(*e*) Difference ($Q - P$)
(*f*) Exclusive union ($P \underline{\cup} Q$) or symmetric difference ($P \, \Delta \, Q$)

FIGURE 7-12
Venn diagram of set algebra.

The exclusive union (also known as symmetric difference) of two sets $P \underline{\cup} Q$ (also written as $P \, \Delta \, Q$) is the subset of elements of W that are members of P or Q but not of both, that is,

$$P \underline{\cup} Q = \{x: x \notin P \cap Q\} \tag{7.12}$$

Figure 7-12*f* shows the exclusive union. Using the Venn diagram it can be shown that $P \cup Q$ can also be expressed as $c(P \cap Q) \cap (P \cup Q)$, $(P \cap cQ) \cup (cP \cap Q)$, $(P - Q) \cup (Q - P)$, or $(P \cup Q) - (P \cap Q)$.

The laws of set algebra are in some cases similar to the laws of ordinary algebra. Just as the latter can be used to simplify algebraic equations and expressions, the former can be used to simplify sets. The laws of set algebra are stated here without any mathematical proofs. Interested readers can prove most of them using the Venn diagram. These laws are:

the commutative law (similar to ordinary algebra $p + q = q + p$ and $pq = qp$):

$$P \cup Q = Q \cup P \tag{7.13}$$

$$P \cap Q = Q \cap P \tag{7.14}$$

the associative law [similar to ordinary algebra $p + (q + r) = (p + q) + r$ and $p(qr) = (pq)r$]:

$$P \cup (Q \cup R) = (P \cup Q) \cup R \tag{7.15}$$

$$P \cap (Q \cap R) = (P \cap Q) \cap R \tag{7.16}$$

the distributive law [similar to $p(q + r) = pq + pr$]:

$$P \cup (Q \cap R) = (P \cup Q) \cap (P \cup R) \tag{7.17}$$

$$P \cap (Q \cup R) = (P \cap Q) \cup (P \cap R) \tag{7.18}$$

the idemoptence law:

$$P \cap P = P \tag{7.19}$$

$$P \cup P = P \tag{7.20}$$

the involution law:

$$c(cP) = P \tag{7.21}$$

and

$$P \cup \varnothing = P \tag{7.22}$$

$$P \cap W = P \tag{7.23}$$

$$P \cup cP = W \tag{7.24}$$

$$P \cap cP = \varnothing \tag{7.25}$$

$$c(P \cup Q) = cP \cap cQ \tag{7.26}$$

$$c(P \cap Q) = cP \cup cQ \tag{7.27}$$

where Eqs. (7.26) and (7.27) are DeMorgan's laws and Eqs. (7.13) to (7.26) provide the tools necessary to manipulate and simplify sets. For example, using Eqs. (7.13), (7.17), and (7.19) one can prove that the set $(P \cup Q) \cup (P \cap Q)$ is equal to the set $P \cup Q$. The Venn diagram can also be used as an informal method to reach the same conclusion. From a geometric modeling point of view, these equations, or the set theory in general, can operate on point sets that represent solids in E^3 or they can be used to classify other point sets in space against solids to determine which points in space are inside, on, or outside a given solid.

The concept of the cartesian product of two sets is useful to geometric modeling because it can be related to coordinates of points in space. The concept of an ordered pair must be introduced first. Let us assume that a and b are two elements. An ordered pair of a and b is denoted by (a, b); a is the first coordinate of the pair (a, b) and b is the second coordinate. This guarantees that $(a, b) \neq (b, a)$ if $a \neq b$. The ordered pair of a and b is a set and can be defined as

$$(a, b) = \{\{a\}, \{a, b\}\} \tag{7.28}$$

Equation (7.28) implies that the first coordinate of the ordered pair is the first element $\{a\}$ and the second coordinate is the second element $\{a, b\}$; both elements form the set of the ordered pair (a, b). If $a = b$, then $(a, a) = \{\{a\}, \{a, a\}\} = \{\{a\}, \{a\}\} = \{\{a\}\}$. Based on this definition, there is a theorem which states that two ordered pairs are equal if and only if their corresponding coordinates are equal, that is, $(a, b) = (c, d) \Leftrightarrow a = c$ and $b = d$.

The cartesian product is the concept that can be used to form ordered pairs. If A and B are two sets, the cartesian product of the sets, designated by $A \times B$, is the set containing all possible ordered pairs (a, b) such that $a \in A$ and $b \in B$, that is,

$$A \times B = \{(a, b): a \in A \text{ and } b \in B\} \tag{7.29}$$

If, for example, $A = \{1, 2, 3\}$ and $B = \{1, 4\}$, then $A \times B = \{(1, 1), (1, 4), (2, 1), (2, 4), (3, 1), (3, 4)\}$. Note that $A \times B \neq B \times A$. We denote $A \times A$ by A^2. The cartesian product of three sets can now be introduced as

$$A \times B \times C = (A \times B) \times C = \{(a, b, c): a \in A, b \in B, c \in C\} \tag{7.30}$$

where (a, b, c) is an ordered triple defined by $(a, b, c) = ((a, b), c)$. $A \times A \times A$ is usually denoted by A^3. In general, an n-tuple can be defined as the cartesian product of n sets and takes the form $(a_1, a_2, \ldots, a_n)$. Ordered pairs and triples are considered 2-tuples and 3-tuples respectively.

Equations (7.29) and (7.30) can be used to define points and their coordinates in the context of set theory. If we consider a set of points (set of real numbers) R^1 in one-dimensional euclidean space E^1, then R^2 defines a set of points in E^2; each is defined by two numbers or an ordered pair. Similarly, R^3 defines a set of points in E^3; each is defined by three numbers or an ordered triple.

Example 7.2. A point set S that defines a solid in E^3 is a set of ordered triples. Find the three sets whose cartesian product produces S.

Solution. The point set can be written as

$$S = \{P_1, P_2, \ldots, P_n\} \tag{7.31}$$

where $P_1, P_2, \ldots, P_n$ are points inside or on the solid. This set can also be written as

$$S = \{(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_n, y_n, z_n)\} = \{(x_i, y_i, z_i): 1 \leq i \leq n\} \tag{7.32}$$

We can define three sets A, B, and C such that

$$A = \{x_1, x_2, \ldots, x_n\} \tag{7.33}$$

$$B = \{y_1, y_2, \ldots, y_n\} \tag{7.34}$$

$$C = \{z_1, z_2, \ldots, z_n\} \tag{7.35}$$

Let us define the set P as the cartesian product $A \times B \times C$, that is,

$$P = A \times B \times C = \{(x_i, y_j, z_k): 1 \leq i \leq n, 1 \leq j \leq n, i \leq k \leq n\} \tag{7.36}$$

The point set S of the solid given by Eq. (7.32) is clearly a (proper) subset of the set P, that is, $S \subset P$. The elements of S are equal to the elements of P only when $i = j = k$.

Let us introduce a new notion called the *ordered* cartesian product. It is a more restricted special case of the cartesian product concept. It is applied only to sets that have the same number of elements. We denote it by "$\otimes$" to differentiate it from "$\times$" which is used for the cartesian product (not ordered). If we have two sets defined as $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_n\}$, then

$$A \otimes B = \{(a_i, b_i): a_i \in A, b_i \in B, \text{ and } 1 \leq i \leq n\} \tag{7.37}$$

The ordered cartesian product of three sets is similarly given by

$$A \otimes B \otimes C = (A \otimes B) \otimes C = \{(a_i, b_i, c_i): a_i \in A, b_i \in B, c_i \in C, \text{ and } 1 \leq i \leq n)\} \tag{7.38}$$

Comparing Eqs. (7.32) and (7.38) shows that the *ordered* cartesian product of the three sets A, B, and C given by Eqs. (7.33) to (7.35) gives the point set S of a solid. This observation that S can be related to A, B, and C might be useful in classification problems.

7.5.2 Regularized Set Operations

The set operations (c, $\cup$, $\cap$, and $-$) covered in the previous section are also known as the set-theoretic operations. When we use these operations in geometric modeling to build complex objects from primitive ones, the complement operation is usually dropped because it might create unacceptable geometry. Furthermore, if we use the other operations ($\cup$, $\cap$, $-$) without regularization in solid modeling, they may cause user inconvenience (say, user must not have overlapping faces of objects or primitives). In addition, objects resulting from these operations may lack geometric closure, may be difficult to validate, or may be inadequate for application (e.g., interference analysis).

To avoid the above problems, the point sets that represent objects and the set operations that operate on them must be regularized. Regular sets and regularized set operations (boolean operations) are considered as boolean algebra.

A regular set is defined as a set that is geometrically closed [refer to Eq. (7.3)]. The notion of a regular set is introduced in geometric modeling to ensure the validity of objects they represent and therefore eliminate nonsense objects. Under geometric closure, a regular set has interior and boundary subsets. More importantly, the boundary contains the interior and any point on the boundary is in contact with a point in the interior. In other words, the boundary acts as a skin wrapped around the interior. The set S shown in Fig. 7-7 is an example of a regular set while Fig. 7-8 shows a nonregular set because the dangling edge and face are not in contact with the interior of the set (in this case the box).

Mathematically, a set S is regular if and only if

$$S = \mathrm{ki}S \tag{7.39}$$

This equation states that if the closure of the interior of a given set yields that same given set, then the set is regular. Figure 7-13*a* shows that set S is not regular because $S' = \mathrm{ki}S$ is not equal to S. Some modeling systems use regular sets that are open or do not have boundaries. A set S is regular open if and only if

$$S = \mathrm{ik}S \tag{7.40}$$

This equation states that a set is regular open if the interior of its closure is equal to the original set. Figure 7-13*b* shows that S is not regular open because $S' = \mathrm{ik}S$ is not equal to S.

Set operations (known also as boolean operators) must be regularized to ensure that their outcomes are always regular sets. For geometric modeling, this means that solid models built from well-defined primitives are always valid and represent valid (no-nonsense) objects. Regularized set operators preserve homogeneity and spatial dimensionality. The former means that no dangling parts should result from using these operators and the latter means that if two three-dimensional objects are combined by one of the operators, the resulting object should not be of lower dimension (two or one dimension). Regularization of set operators is particularly useful when users deal with overlapping faces of different objects, or in other words when dealing with tangent objects, as will be seen shortly in an example.

Based on the above description, regularized set operators can be defined as follows:

$$P \cup^* Q = \mathrm{ki}(P \cup Q) \tag{7.41}$$

$$P \cap^* Q = \mathrm{ki}(P \cap Q) \tag{7.42}$$

$$P -^* Q = \mathrm{ki}(P - Q) \tag{7.43}$$

$$\mathrm{c}^* P = \mathrm{ki}(\mathrm{c}P) \tag{7.44}$$

where the superscript * to the right of each operator denotes regularization. The sets P and Q used in Eqs. (7.41) to (7.44) are assumed to be any arbitrary sets. However, if two sets X and Y are r-sets (regular sets), which is always the case for geometric modeling, then Eqs. (7.41) to (7.44) become

$$X \cup^* Y = X \cup Y \tag{7.45}$$

$$X \cap^* Y = X \cap Y \Leftrightarrow \mathrm{b}X \text{ and } \mathrm{b}Y \text{ do not overlap} \tag{7.46}$$

$$X -^* Y = \mathrm{k}(X - Y) \tag{7.47}$$

$$\mathrm{c}^* X = \mathrm{k}(\mathrm{c}X) \tag{7.48}$$

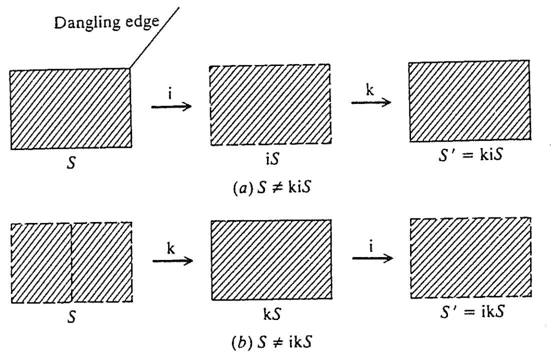


FIGURE 7-13
Set regularity.

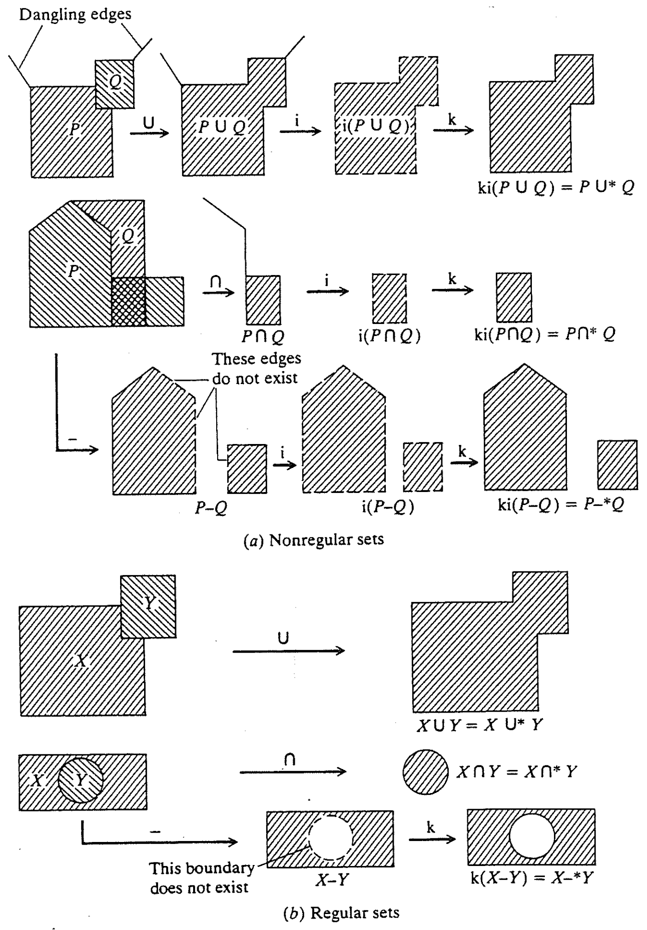


FIGURE 7-14
Regularized set operators.

If bX and bY overlap in Eq. (7.46), Eq. (7.42) is used and the result is a null object. Figure 7-14 illustrates Eqs. (7.41) to (7.48) geometrically. The figure does not include the complement operation.

Example 7.3. What are the results of applying the regularized set operations to objects A and B shown in Fig. 7-15?

Solution. The positions of objects A and B shown in Fig. 7-15 are chosen to illustrate some tangency cases of objects. A and B are r-sets. The results of applying Eqs. (7.45) to (7.47) are shown in Table 7.2 for each case. For all the cases, the results of the regularized union operations are obvious. However, the results of the intersection operations may be less obvious. For case 1, $A \cap B$ is the common face which is eliminated by the regularization process. For case 2, the intersection does not exist; therefore the result is an empty set or a null object. For case 3, $A \cap B$ is the common edge which is eliminated by the regularization process. For case 4, $A \cap B$ is the common block and the common face. The common face is eliminated after regularization. The results of the regularized difference operations are obvious. In cases 1, 2, and 3, $A -^* B$ is the object A itself. For case 4, the difference is a disjoint object. Such an object should not be viewed as two objects. Any further set operation or rigid-body motion treats it as one object.

The reader is advised to carry the details of these results following the steps illustrated in Fig. 7-14. The reader should also try to use these cases to test any available solid modeling package.

7.5.3 Set Membership Classification

In various geometric problems involving solid models, we are often faced with the following question: given a particular solid, which point, line segment, or a portion of another solid intersects with such a solid? These are all geometric intersection problems. For a point/solid, line (curve)/solid, or solid/solid intersec-

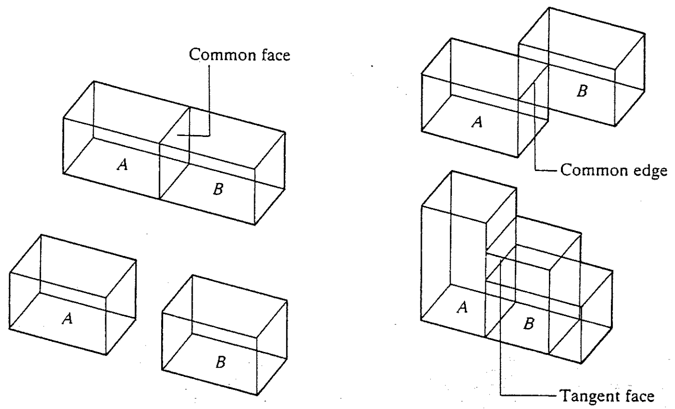
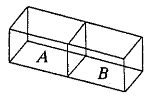
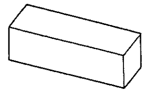
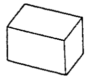
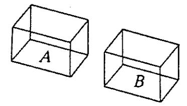
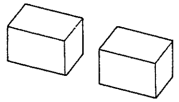
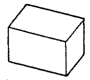
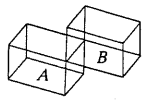
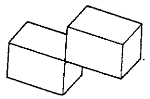
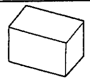
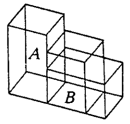
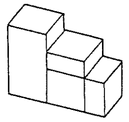
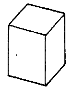
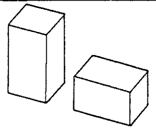
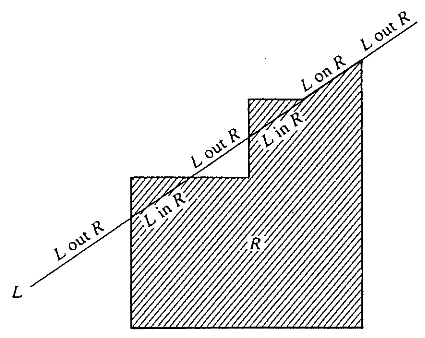


FIGURE 7-15
Sample objects.

TABLE 7.2
Results of Example 7.3

Case	Objects	Set operation	Result
1		$A \cup^* B$	
		$A \cap^* B$	$\varnothing$ (null object)
		$A -^* B$	
2		$A \cup^* B$	
		$A \cap^* B$	$\varnothing$ (null object)
		$A -^* B$	
3		$A \cup^* B$	
		$A \cap^* B$	$\varnothing$ (null object)
		$A -^* B$	
4		$A \cup^* B$	
		$A \cap^* B$	
		$A -^* B$	

tion, we need to know respectively which points, line segments, or solid portions are inside, outside, or on the boundary of a given solid. These geometric intersection problems have useful practical engineering applications. For example, line/solid intersection can be used to shade or calculate mass properties of given



$M[L,R] = (L \text{ in } R, L \text{ on } R, L \text{ out } R)$

FIGURE 7-16
Line/polygon set membership classification.

solids via ray-tracing algorithms, while solid/solid intersection can be used for interference checking between two solids.

In each of the above problems, we are given two point sets: a reference set S and a candidate set X. The reference set is usually the given solid whose inside (interior) and boundary are iS and bS respectively. The outside of S is its complement cS. The candidate set is the geometric entity that must be classified against S. The process by which various parts of X (points, line segments, or solid portions) are assigned to iS, bS, and/or cS is called set membership classification.

A function called a set membership classification function exists which provides a unifying approach to study the behavior of the candidate set X relative to the reference set S. The function is denoted by $M[\,.\,]$ and is defined as

$$M[X, S] = (X \text{ in } S, X \text{ on } S, X \text{ out } S) \tag{7.49}$$

Equation (7.49) implies that the input to $M[\,.\,]$ is the two sets X and S and the output is the classification of X relative to S as in, on, or out S. Figure 7-16 shows an example of classifying a portion of a line L against the polygon R.

The implementation of the classification function given by Eq. (7.49) depends to a great extent on the representations of both X and S and their data structures. Let us consider the line/polygon classification problem when the polygon (reference solid) is stored as a B-rep or a CSG. Figure 7-17 shows the B-rep case. The line L is chosen such that no "on" segments result for simplicity.

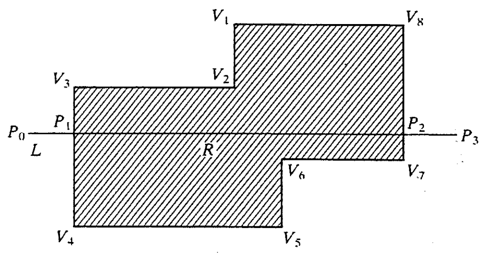


FIGURE 7-17
Line/polygon classification for B-rep.

The algorithm for this case can be described as follows:

1. Utilizing a line/edge intersection routine, find the boundary crossings P_1 and P_2.
2. Sort the boundary crossings according to any agreed direction for L. Let the sorted boundary crossing list be given by (P_0, P_1, P_2, P_3).
3. Classify L with respect to R. For this simple case, we know that the odd boundary crossings (such as P_1) flags "in" segments and the even boundary crossings (such as P_2) flags "out" segments. Therefore, the classification of L with respect to R becomes

$$[P_0, P_1] \subset L \text{ out } R$$
$$[P_1, P_2] \subset L \text{ in } R$$
$$[P_2, P_3] \subset L \text{ out } R$$

If the line L contains an edge of the polygon, the above classification criterion of odd and even crossings would not work and another criterion should be found. In this case, a direction (clockwise or counterclockwise) to traverse the polygon boundaries is needed. Let us apply this idea to the problem at hand to see how it would work. If we choose the counterclockwise direction, polygon vertices would be numbered as shown in Fig. 7-17. Now we know that iR is always to the left of any edge. The new classification criterion can be stated as follows. Let us assume that an edge is defined by the two vertices V_i and V_{i+1}. Whenever there is a boundary crossing on an edge whose V_i is above L and V_{i+1} is below L, this crossing is flagged as "in" and whenever V_i is below L and V_{i+1} is above, it is flagged "out." This criterion obviously gives the same result as the previous criterion for this example.

Let us consider the same line/polygon classification problem when the polygon is stored as a CSG representation. The classification for this case is done at the primitive level and the algorithm becomes as follows:

1. Utilize a line/primitive intersection routine to find the intersection points of the line with each primitive of R.
2. Use these intersection points to classify the line against each primitive of R.
3. Combine the "in" and "on" line segments obtained in step 2 using the same boolean operators that combine the primitives. For example, if two primitives A and B are unioned, then the "in" and "on" line segments are added.
4. Find the "out" segments by taking the difference between the line (candidate set) and the "in" and "on" segments. Figure 7-18 shows the "classify" and "combine" strategy for the three boolean operations of two blocks A and B. Notice that the polygon that results from the union operation is the same as the polygon R used in the classification of the B-rep case. The classification of L relative to A and B is straightforward. To combine these classifications, we first combine L in A and L in B to obtain L in R, using the proper boolean operator. The L on R can result from combining three possibilities: L in A and

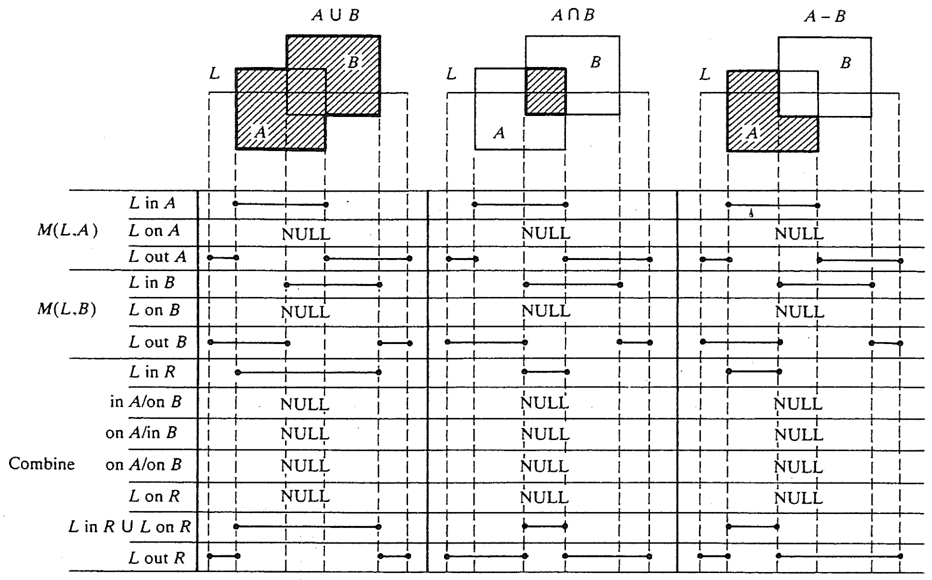


FIGURE 7-18
Line/polygon classification for CSG rep.

L on B, L on A and L in B, and L on A and L on B. All these possibilities are obtained and then combined to give L on R. The remaining classification L out R is obtained by adding L in R and L on R, and subtracting the result from L itself.

The above example has considered the polygon case. The example does not purposely include "on" segments because they are ambiguous and need more information (neighborhoods) to resolve their ambiguities for both B-rep and CSG (refer to Sec. 7.8 for details). Algorithms to classify candidate sets against three-dimensional solids can follow similar steps to those described in the above example but with more elaborate details.

7.6 HALF-SPACES

Half-spaces form a basic representation scheme for bounded solids. By combining half-spaces (using set operations) in a building block fashion, various solids can be constructed. Half-spaces are usually unbounded geometric entities; each one of them divides the representation space into two infinite portions, one filled with material and the other empty. Surfaces can be considered half-space boundaries and half-spaces can be considered directed surfaces.

A half-space is defined as a regular point set in E^3 as follows:

$$H = \{P: P \in E^3 \text{ and } f(P) < 0\} \tag{7.50}$$

where P is a point in E^3 and $f(P) = 0$ defines the surface equation of the half-space boundaries. Half-spaces can be combined together using set operations to create complex objects.

7.6.1 Basic Elements

Various half-spaces can be described and created using Eq. (7.50). However, to make them useful for design and manufacturing applications, supporting algorithms and utility routines must be provided. For example, if one were to add a cylindrical half-space to a modeling package, intersecting routines that enable this half-space to intersect itself as well as other existing half-spaces must be developed and added as well.

The most widely used half-spaces (unbounded) are planar, cylindrical, spherical, conical, and toroidal half-spaces. They form the natural quadrics discussed earlier in Sec. 7.3 (with the exception of the torus which can be formed from the other half-spaces). The regular point set of each half-space is a set of ordered triplets (x, y, z) given by

Planar half-space:	$H = \{(x, y, z): z < 0\}$	(7.51)
Cylindrical half-space:	$H = \{(x, y, z): x^2 + y^2 < R^2\}$	(7.52)
Spherical half-space:	$H = \{(x, y, z): x^2 + y^2 + z^2 < R^2\}$	(7.53)
Conical half-space:	$H = \{(x, y, z): x^2 + y^2 < [(\tan \alpha/2)z]^2\}$	(7.54)
Toroidal half-space:	$H = \{(x, y, z): (x^2 + y^2 + z^2 - R_2^2 - R_1^2)^2 < 4R_2^2(R_1^2 - z^2)\}$	(7.55)

Equations (7.51) to (7.55) are implicit equations and are expressed in terms of each half-space local coordinate system whose axes are X_H, Y_H, and Z_H. The implicit form is efficient to find surface intersections (refer to Chap. 6). The corresponding surface of each half-space is given by its equation when the right and left sides are equal. For the planar half-space, Eq. (7.51) is based on the vertical plane $z = 0$. Other definitions can be easily written. Figure 7-19 shows the various half-spaces with their local coordinate systems and the limits on their configuration parameters.

7.6.2 Building Operations

Complex objects can be modeled as half-spaces combined by the set operations. As a matter of fact, half-spaces are treated as lower level primitives and all the related construction techniques to CSG can be used here. As will be seen later, one form of CSG can be based on unbounded half-spaces. Regularized set operations can be used to combine half-spaces to form complex solids. Most often, half-spaces may have to undergo rigid motion via homogeneous transformations to be positioned properly before intersection.

Let us represent the solid S shown in Fig. 7-20a using half-spaces. Parameters of the solid are shown. The hole is centered in the top face. The MCS of the

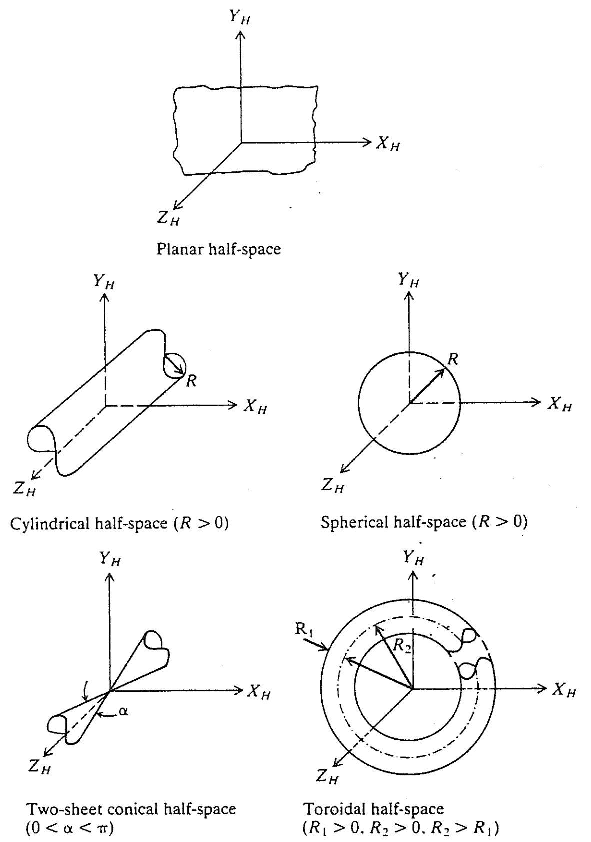
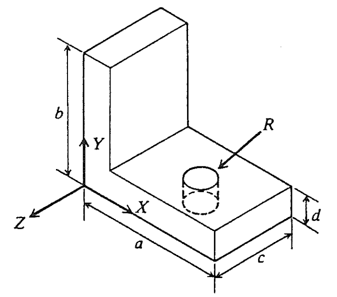
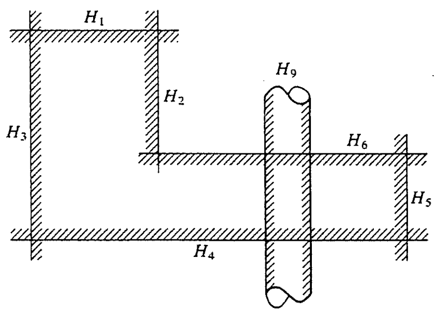

Planar half-space
Cylindrical half-space ($R > 0$)
Spherical half-space ($R > 0$)
Two-sheet conical half-space ($0 < \alpha < \pi$)
Toroidal half-space ($R_1 > 0, R_2 > 0, R_2 > R_1$)
Y_H
X_H
Z_H
R
R_1
R_2
α

FIGURE 7-19
Unbounded half-spaces.



(*a*) Solid S



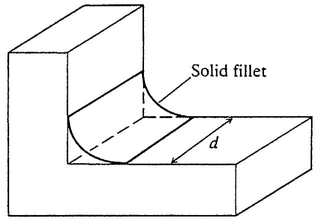
(*b*) Half-space representation

FIGURE 7-20
Half-space representation of solid S.

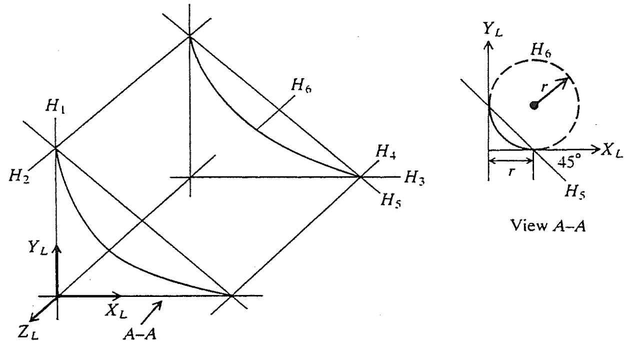
solid model is chosen as shown in the figure. Figure 7-20*b* shows that nine half-spaces (eight planes and one cylinder) H_1 to H_9 are needed to represent S. Half-spaces H_7 and H_8 that model the front and back faces of the model are not shown in the figure. Utilizing local coordinate systems shown in Fig. 7-19, some half-spaces have to be positioned first. For example, rotate H given by Eq. (7.51) an angle $-90°$ about the X axis and translate it up in the Y direction a distance b to obtain H_1. In a similar fashion, the other half-spaces can be positioned using the proper rigid motion. Only H_7 needs no positioning. The positioned half-spaces can be intersected and then boolean operations are used to combine them. H_1 to H_8 are unioned and H_9 is subtracted from the result.

Example 7.4. How can you create a solid fillet using unbounded half-spaces?

Solution. Surface fillet has been defined in Chap. 6 as a B-spline surface blending two given surfaces. Similarly, a solid fillet can be used to blend sharp edges of a solid as shown in Fig. 7-21*a*. The solid fillet is defined by its radius r and length d. Six half-spaces H_1 to H_6 (see Fig. 7-21*b*) are needed to construct the fillet. H_1, H_2, H_3, and H_4 represent the front, left, back, and bottom faces respectively. H_6 is the cylindrical face. H_5 is an auxiliary half-space positioned at distance r from the origin of the $X_L Y_L Z_L$ local coordinate system of the fillet and oriented at 45° as shown in



(*a*) Solid fillet



(*b*) Half-space representation

FIGURE 7-21
Solid fillet and its half-space representation.

view *A-A* in Fig. 7-21*b*. H_5 is used to intersect H_2, H_4, and H_6 so that the boundaries of the fillet can be evaluated. This is because the cylindrical half-space is tangent to H_2 and H_4.

Except for H_1, the other half-spaces must be positioned before intersection and set operations are performed to create the fillet. Let us look at positioning H_5 and H_6 as an example. Using Eq. (7.51), *H* has to rotate an angle of 90° about the *Y* axis, followed by a 45° rotation about the *Z* axis, and finally translated a distance *r* in the positive *X* direction to produce H_5. H_6 is obtained by translating the cylindrical half-space *H* given by Eq. (7.52) an equal distance *r* in both the positive *X* and *Y* directions. At this position, the complement of the cylindrical half-space, c*H*, is taken to obtain H_6. Theoretically, c*H* is equal to E^3 minus the cylindrical half-space. For practical and implementation purposes, E^3 can be limited to a bounded volume, such as a box, enclosing the cylindrical half-space, or the complement process can be replaced by choosing a surface normal to be positive on one side of the half-space and negative on the other side.

The intersections of H_1 to H_6 with each other can now be performed and the results can be unioned to obtain the solid fillet. Notice that the solid fillet could have been created without the complement operation by subtracting the cylindrical half-space itself after its positioning from the intersection results of H_1 to H_5. However, the complement of a half-space is generally used to minimize the number of half-spaces used in modeling objects.

It should be noted from this example that using half-spaces and/or their complements or directed surface normals, any complex object can be modeled as the union of the intersection of half-spaces, that is,

$$S = \cup \left(\bigcap_{i=1}^{n} H_i \right) \tag{7.56}$$

where *S* is the solid and *n* is the number of half-spaces and/or their complements. As an example, a box is the union of six intersected half-spaces.

7.6.3 Remarks

The half-space representation scheme is the lowest level available to represent a complex object as a solid model. The main advantage of half-spaces is its conciseness in representing objects compared to other schemes such as CSG. However, it has a few disadvantages. This representation can lead to unbounded solid models if the user is not careful. Such unboundedness can result in missing faces and abnormal shaded images. It can also lead to system crash or producing wrong results if application algorithms attempt to access databases of unbounded models. Another major disadvantage is that modeling with half-spaces is cumbersome for casual users and designers to use and may be difficult to understand. Therefore, half-space representation is probably useful only for research purposes. Modelers, such as SHAPES, TIPS, and PADL, attempt to shield users from dealing directly with the unbounded half-spaces.

7.7 BOUNDARY REPRESENTATION (B-rep)

Boundary representation is one of the two most popular and widely used schemes (the other is CSG discussed in Sec. 7.8) to create solid models of physical

objects. A B-rep model or boundary model is based on the topological notion that a physical object is bounded by a set of faces. These faces are regions or subsets of closed and orientable surfaces. A closed surface is one that is continuous without breaks. An orientable surface is one in which it is possible to distinguish two sides by using the direction of the surface normal to point to the inside or outside of the solid model under construction. Each face is bounded by edges and each edge is bounded by vertices. Thus, topologically, a boundary model of an object is comprised of faces, edges, and vertices of the object linked together in such a way as to ensure the topological consistency of the model.

The database of a boundary model contains both its topology and geometry. Topology is created by performing Euler operations and geometry is created by performing euclidean calculations. Euler operations are used to create, manipulate, and edit the faces, edges, and vertices of a boundary model as the set (boolean) operations create, manipulate, and edit primitives of CSG models. Euler operators, as boolean operators, ensure the integrity (closeness, no dangling faces or edges, etc.) of boundary models. They offer a mechanism to check the validity of these models. Other validity checks may be used as well. Geometry includes coordinates of vertices, rigid motion and transformation (translation, rotation, etc.), and metric information such as distances, angles, areas, volumes, and inertia tensors. It should be noted that topology and geometry are interrelated and cannot be separated entirely. Both must be compatible otherwise nonsense objects may result. Figure 7-22 shows a square which, after dividing its top edges by introducing a new vertex, is still valid topologically but produces a nonsense object depending on the geometry of the new vertex.

In addition to ensuring the validity of B-rep models, Euler operators provide designers with drafting functionality. These allow solid models to be built up graphically by incrementally adding individual vertices, edges, and faces to the model in such a way as to always obey Euler's laws, as will be seen in Sec. 7.7.2. Euler operators are considered to be lower level operators than boolean operators in the sense that they combine faces, edges, and vertices to form B-rep models.

Boolean operations are not considered a part of the representation of a B-rep model, but they are often employed as one of the means of creating, manipulating, and editing the model as mentioned in Sec. 7.1 and shown in Table 7.1.

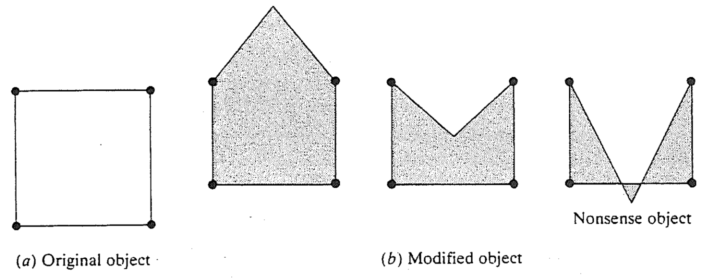


FIGURE 7-22
Effect of topology and geometry on boundary models.

The effect of a Boolean operation on a CSG model (see Sec. 7.8) is simply an addition to the CSG tree. However, since B-rep systems require an explicit representation of the boundary of the solid, they must evaluate the new boundary that is the result of the operation.

While B-rep systems store only the bounding surfaces of the solid, it is still possible to compute volumetric properties such as mass properties (assuming uniform density) by virtue of the Gauss divergence theorem which relates volume integrals to surface ones. The speed and accuracy of these calculations depend on the types of surfaces used by the models. More details are covered in Chap. 17.

The modeling domain (or the range of objects that can be modeled) of a B-rep scheme is potentially large and depends mainly on the primitive surfaces (planar, curved, or sculptured) that are admissible by the scheme to form the faces of various models. For example, given the modeling domain of a scheme based on half-spaces, a B-rep scheme with the same domain can be designed by using the boundary surfaces of the half-spaces as its primitive surfaces.

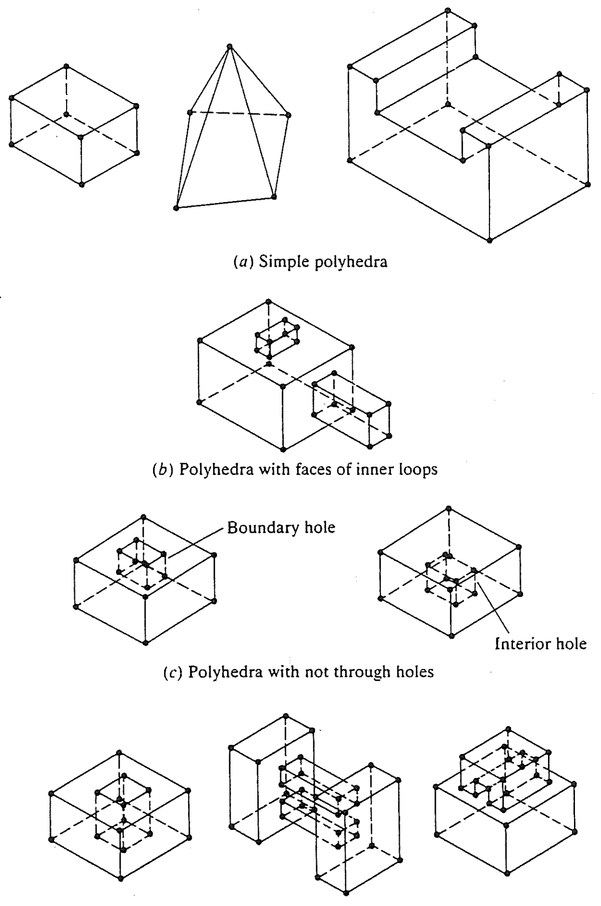
The desired properties of a representation scheme discussed in Sec. 7.4 apply to B-rep schemes. These schemes are unambiguous if faces are represented unambiguously, that is, as regions of closed orientable surfaces. This claim (unambiguous faces result in unambiguous B-rep) is based on the fact that an r-set is defined unambiguously by its boundary and that non-r-sets are not defined unambiguously by their boundaries. The validity of B-rep models is ensured via Euler operations which can be built into the syntax of a CAD/CAM system. However, these models are not unique because the boundary of any object can be divided into faces, edges, and vertices in many ways. Verification of uniqueness of boundary models is computationally expensive and is not performed in practice.

7.7.1 Basic Elements

If a solid modeling system is to be designed, the domain of its representation scheme (objects that can be modeled) must be defined, the basic elements (primitives) needed to cover such modeling domain must be identified, the proper operators that enable the system users to build complex objects by combining the primitives must be developed, and finally a suitable data structure must be designed to store all relevant data and information of the solid model. Other system and geometric utilities (such as intersection algorithms) may also need to be designed. Let us apply these ingredients to a B-rep system.

Objects that are often encountered in engineering applications can be classified as either polyhedral or curved objects. A polyhedral object (plane-faced polyhedron) consists of planar faces (or sides) connected at straight (linear) edges which, in turn, are connected at vertices. A cube or a tetrahedron is an obvious example. A curved object (curved polyhedron) is similar to a polyhedral object but with curved faces and edges instead. The identification of faces, edges, and vertices for curved closed objects such as a sphere or a cylinder needs careful attention, as will be seen later in this section. Polyhedral objects are simpler to deal with and are covered first.

The reader might have jumped intuitively to the conclusion that the primitives of a B-rep scheme are faces, edges, and vertices. This is true if we can answer the following two questions. First, what is a face, edge, or a vertex? Second, knowing the answer to the first question, how can we know that when we combine these primitives we would create valid objects? Answers to these questions can help users to create B-rep solid models of objects successfully. To show that these answers are not always simple, consider the polyhedral objects shown in Fig. 7-23. Polyhedral objects can be classified into four classes. The first class (Fig. 7-23*a*) is the simple polyhedra. These do not have holes (through or not through) and each face is bounded by a single set of connected edges, that is, bounded by one loop of edges. The second class (Fig. 7-23*b*) is similar to the first



(*a*) Simple polyhedra

(*b*) Polyhedra with faces of inner loops

(*c*) Polyhedra with not through holes

(*d*) Polyhedra with handles (through holes)

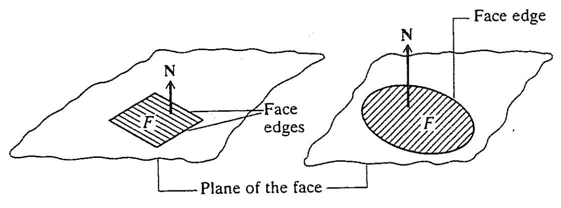
FIGURE 7-23
Types of polyhedral objects.

with the exception that a face may be bounded by more than one loop of edges (inner loops are sometimes called rings). The third class (Fig. 7-23*c*) includes objects with holes that do not go through the entire object. For this class, a hole may have a face coincident with the object boundary; in this case we call it a boundary hole. On the other hand, if it is an interior hole (as a void or crack inside the object), it has no faces on the boundary. The fourth and the last class (Fig. 7-23*d*) includes objects that have holes that go through the entire objects. Topologically, these through holes are called handles.

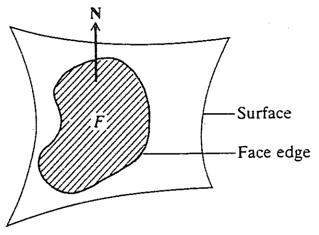
With the above physical insight, let us define the primitives of a B-rep scheme and other related topological items that enable a user to create the boundary model of an object. They apply to both polyhedral and curved objects. A vertex is a unique point (an ordered triplet) in space. An edge is a finite, non-self-intersecting, directed space curve bounded by two vertices that are not necessarily distinct. A face is defined as a finite connected, non-self-intersecting, region of a closed oriented surface bounded by one or more loops. A loop is an ordered alternating sequence of vertices and edges. A loop defines a non-self-intersecting, piecewise, closed space curve which, in turn, may be a boundary of a face. In Fig. 7-23*a*, each face has one loop while the top and the right side faces of the object shown in Fig. 7-23*b* have two loops each (one inner and one outer). A "not" through hole is defined as a depression in a face of an object. A handle (or through hole) is defined as a passageway that pierces the object completely. The topological name for the number of handles in an object is genus. The last item to be defined is a body (sometimes called a shell). It is a set of faces that bound a single connected closed volume. Thus a body is an entity that has faces, edges, and vertices. Such an entity may be a useful solid or an intermediate polyhedron. A minimum body is a point. Topologically this body has one face, one vertex, and no edges. It is called a seminal or singular body. It is initially attached as part of the world. The object on the right of Fig. 7-23*c* has two bodies (the exterior and interior cubes) and any other object in Fig. 7-23 has only one body.

Faces of boundary models possess certain essential properties and characteristics that ensure the regularity of the model; that is, the model has an interior and a boundary. The face of a solid is a subset of the solid boundary and the union of all faces of a solid defines such a boundary. Faces are two-dimensional homogeneous regions so they have areas and no dangling edges. In addition, a face is a subset of some underlying closed oriented surface. Figure 7-24 shows the relationship between a face and its surface. At each point on the face, there is a surface normal **N** that has a sign associated with it to indicate whether it points into or away from the solid interior. One convention is to assume **N** positive if it points away from the solid. It is desirable, but not required, that a face has a constant surface normal.

The representation of a face must ensure that both the face and solid interiors can be deduced from the representation. The direction of the face's surface normal can be used to indicate the inside or outside of the model. The surface equation must be consistent with the normal chosen convention. For example, if the face belongs to a Bezier or B-spline surface, the normal vector could be defined as $\partial\mathbf{P}/\partial v \times \partial\mathbf{P}/\partial u$ or $\partial\mathbf{P}/\partial u \times \partial\mathbf{P}/\partial v$ depending on the chosen normal convention and the directions of parametrizing the surface. Practically, some



(*a*) Underlying surface is a plane



(*b*) A general underlying surface

FIGURE 7-24
Underlying surface of a face.

CAD/CAM systems store the surface normal and its sign as part of the face data (although it could be computed from the surface equation) since it is a useful parameter in many applications such as generating graphics displays or NC machining data. The face interior can be determined by traversing the face loops in a certain direction or assigning flags to them. In traversing loops, the edges of the face outer loop is traversed, say, in a counterclockwise direction and the edges of the inner loops are traversed in the opposite direction, say the clockwise direction. If one of the loops is a continuous or piecewise continuous curve, the parametrization direction is chosen to reflect the traversal direction. Figure 7-25 shows some traversal examples. The other alternative assigns one flag to outer loops and another one to inner loops.

Having defined the boundary model primitives, we now return to the question of how they can be combined to generate topologically valid models. The development of volume measure (valid models) based on faces, edges, and vertices is rigorous and not easy. Euler (in 1752) proved that polyhedra that are homomorphic to a sphere (i.e., their faces are non-self-intersecting and belong to closed orientable surfaces) are topologically valid if they satisfy the following equation:

$$F - E + V - L = 2(B - G) \tag{7.57}$$

where F, E, V, L, B, and G are the number of faces, edges, vertices, faces' inner loop, bodies, and genus (handles or through holes) respectively. Equation (7.57) is known as the Euler or Euler-Poincare law. The simplest version of this equation is $F - E + V = 2$ which applies to polyhedra shown in Fig. 7-23*a*. With Eq. (7.57) in hand, it has been easier to take it as the more primitive definition of a polyhedron on which to base its construction and data structure. From a user

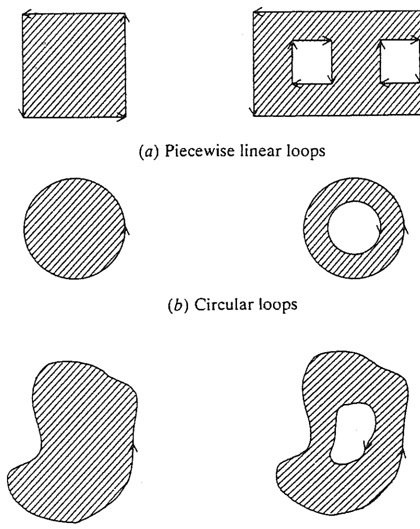


FIGURE 7-25
Traversal of face's loops.

point of view, to create the boundary model of a given object, the user identifies the proper number for all the variables of Eq. (7.57) and substitutes them into the equation to ensure validity. Then system commands (Euler operations) are used to create the model and ensures the validity simultaneously. This is similar to identifying primitives and boolean operators in the case of a CSG-based user interface. Table 7.3 shows the counts of the various variables of Eq. (7.57) for polyhedra shown in Fig. 7-23. The numbering of these polyhedra in the table is taken from left to right and top to bottom with the top left cube being polyhedron number 1 and the bottom right object being number 9.

Euler's law given by Eq. (7.57) applies to closed polyhedral objects only. These are the valid solid models we like to deal with. However, open polyhedral objects do not satisfy Eq. (7.57). This class of objects includes open polyhedra

TABLE 7.3
Counts of polyhedral values for objects of Fig. 7-23

Object number	F	E	V	L	B	G
1	6	12	8	0	1	0
2	5	8	5	0	1	0
3	10	24	16	0	1	0
4	16	36	24	2	1	0
5	11	24	16	1	1	0
6	12	24	16	0	2	0
7	10	24	16	2	1	1
8	20	48	32	4	1	1
9	14	36	24	2	1	1

that may result during constructing boundary models of closed objects as well as all two-dimensional polygonal objects. Open objects satisfy the following Euler's law:

$$F - E + V - L = B - G \tag{7.58}$$

Figure 7-26 shows some examples of open objects. The reader can easily verify that they satisfy the above equation. In the above equation, B refers to an open body which can be a wire, an area, or a volume. All the objects in Fig. 7-26 have one body and only bodies of Fig. 7-26*c* have one genus each. It might be interesting to mention that Eq. (7.58) can form the basis of creating a boundary model based on wireframe modeling. There are some systems such as MEDUSA that do that.

We now turn from polyhedral objects to curved objects such as cylinders and spheres. The same rules and guidelines for boundary modeling discussed thus far for the former objects apply to the latter. The major difference between the two types of objects results if closed curved edges or faces exist. Consider, for example, the closed cylinder and sphere shown in Fig. 7-27. As shown in Fig. 7-27, a closed cylindrical face (and alike) has one edge and two vertices and a spherical face (and alike) has one vertex and no edges. The boundary model of a cylinder has three faces (top, bottom, and cylindrical face itself), two vertices, and three edges connecting the two vertices. The other "edges" are for visualization purposes. They are called limbs, virtual edges, or silhouette edges. The problem of computing the silhouette curve of a solid object is covered in Chap. 10. The boundary model of a sphere, on the other hand, consists of one face, one vertex, and no edges. Notice that both models satisfy Euler laws $F - E + V = 2$ for simple polyhedra.

The representation of curved edges is more complex than representing piecewise linear edges. There are direct and indirect schemes. In direct schemes, an edge is represented by a curve equation and ordered endpoints. In indirect schemes, the edge is represented by the intersection of two surfaces. In practice,

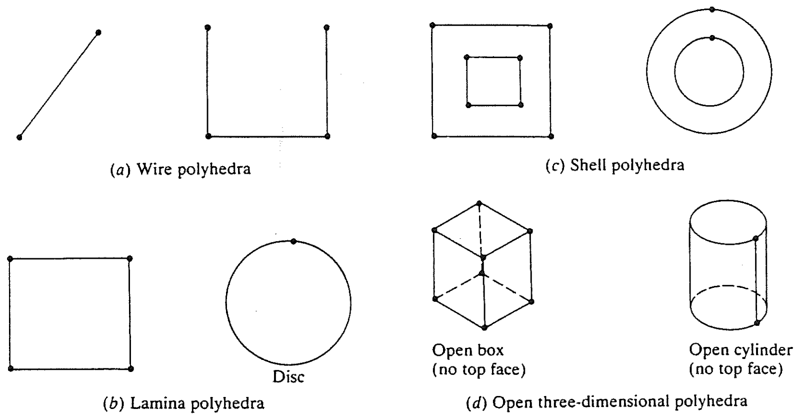


FIGURE 7-26
Open polyhedral objects.

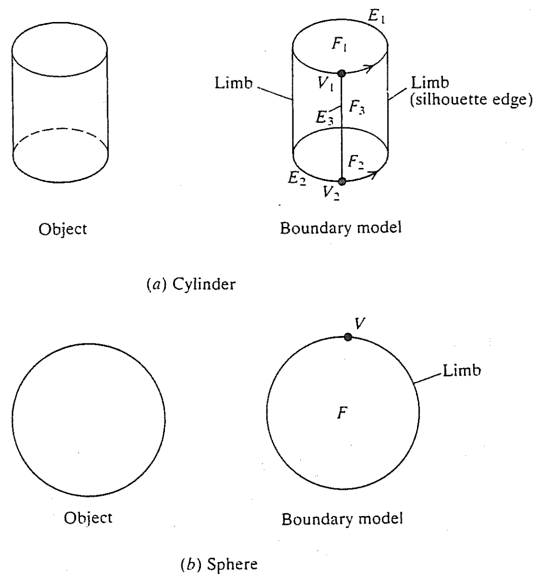


FIGURE 7-27
Exact B-rep of a cylinder and a sphere.

indirect schemes are probably preferred because the intersection of two underlying surfaces of two faces produces the curved edge of the two faces.

If the curved objects are represented by storing the equations of the underlying curves and surfaces of the object edges and faces respectively, the resulting boundary scheme is known as an exact B-rep scheme. Another alternative is the approximate or faceted B-rep (sometimes called tessellation rep). In this scheme, any curved face is divided into planar facets—hence the name faceted B-rep. Figure 7-28 shows a faceted B-rep of a cylinder and sphere. The faceted cylinder is generated by rotating a line incrementally about the cylinder axis the desired total number of facets. This is accomplished via a rotational sweep operator. A faceted sphere is formed in a similar way by rotating m connected line segments (edges) about the sphere axis for a total of n sides. MEDUSA, for example, is a faceted B-rep package. The numbers n and m are user inputs. This representation, although continuous, will no longer be smooth and as the number of facets increases to give a more accurate representation, the computing time involved increases dramatically.

A general data structure for a boundary model should have both topological and geometrical information. The structure shown in Fig. 7-29 is based on Eq. (7.57). A relational database model is very effective to implement such a data structure. Lists for bodies, faces, loops, edges, and vertices are generated and stored in tables. Each line in Fig. 7-29 represents a pointer in the database.

The winged edge data structure is a particularly useful data structure which has been adopted by several modeling systems such as GLIDE and BUILD. In this structure, all the adjacency relations of each edge are described explicitly. Since an edge is adjacent to exactly two faces, it is a component in two loops, one

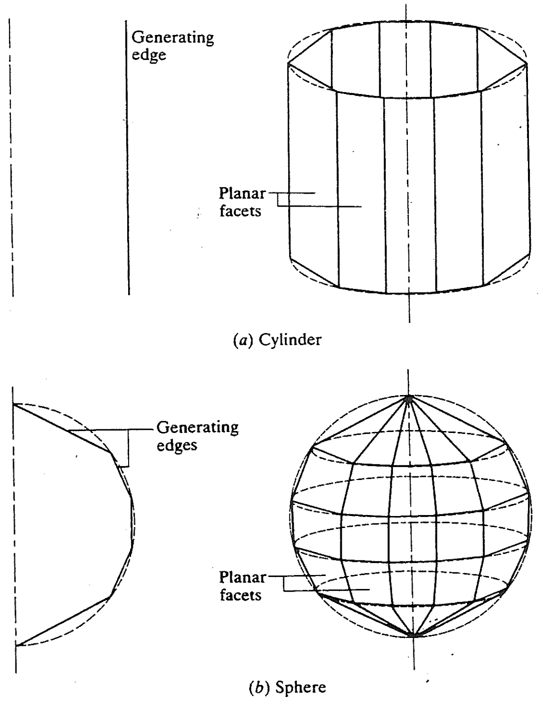


FIGURE 7-28
Faceted B-rep of a cylinder and a sphere.

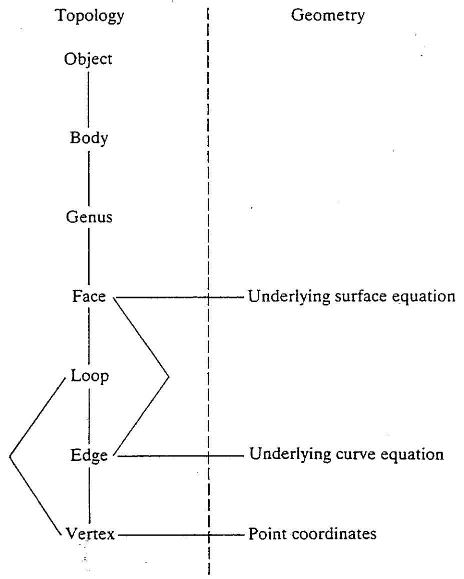


FIGURE 7-29
General data structure for boundary modeling.

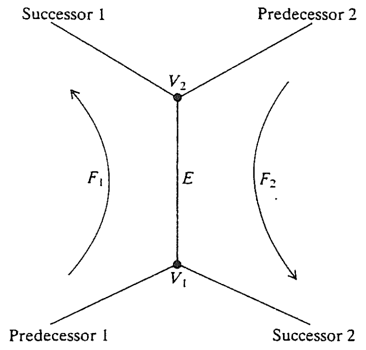


FIGURE 7-30
The winged edge data structure.

for each face. If these loops are oriented, that is, edges of a loop are traversed in a given direction, say counterclockwise, the edge has a predecessor and a successor in addition to the two bounding vertices (see Fig. 7-30). This edge structure together with its implication on the loop and face elements is extremely efficient for manipulation purposes (adding or deleting vertices, edges, or faces) using Euler's law. For example, the insertion of a new edge in the structure changes the predecessor/successor relationship of its adjacent edges, possibly splitting a face into two and therefore adding a new loop.

7.7.2 Building Operations

Equation (7.57) forms the basis to develop building operations to create boundary models of complex objects. Euler operators (or Euler primitives) are based on this equation. There are many variations on how these operators can be implemented. Sample operators are MBFV, MEV, MEF, and GLUE. In these operators, M and K stand for Make and Kill respectively and the other letters mean the same as in Eq. (7.57). Other operators are available to add convenience and flexibility to the construction process. Each operator usually has a complement that has the exact opposite effect on the construction process. Table 7.4 shows some Euler operators. The table shows that the user is not free to construct faces, edges, or vertices as with wireframe and surface modeling. There is no such operator as ME, MV, or MF only because they all violate Euler's law. To create an edge, for example, a new vertex or a new face must be created to preserve the topology. Thus, the two operators MEV and MEF are legitimate. The operator MBFV is usually used to begin constructing the boundary model and it returns a seminal or singular body. It could be thought of as creating the first vertex of the model. The gluing operator is used to glue bodies together at certain faces. The gluing can result in forming a genus or killing one body, as indicated by the operators KFEVMG and KFEVB respectively. Both operators can be called the GLUE operator whose complement would be UNGLUE. The composite commands are available for efficiency of construction and can always be replaced by a sequence of other basic operators. For example, ESPLIT can be replaced by KEV, MEV, and MEV. Similarly, the KVE operator which kills a vertex and all

TABLE 7.4
Some Euler operations

Operation	Operator	Complement	Description of operator
Initialize database and begin creation	MBFV	KBFV	Make Body, Face, Vertex
Create edges and vertices	MEV	KEV	Make Edge, Vertex
Create edges and faces	MEKL	KEML	Make Edge, Kill Loop
	MEF	KEF	Make Edge, Face
	MEKBFL	KEMBFL	Make Edge, Kill Body, Face, Loop
	MFKLG	KFMLG	Make Face, Kill Loop, Genus
Glue	KFEVMG	MFEVKG	Kill Face, Edge, Vertex, Make Genus
	KFEVB	MFEVB	Kill Face, Edge, Vertex, Body
Composite operations	MME	KME	Make Multiple Edges
	ESPLIT	ESQUEEZE	Edge-Split
	KVE		Kill Vertex, Edge

attached edges to it can be replaced by an $(n - 1)$ KEF followed by a single KEV if n edges are attached to the vertex. Notice that some Euler operators do not tell directly their end result (see Table 7.5). For example, one would think the glue operator KFEVMG kills only one face. It is less confusing to write KFFEVMG, especially to new users of a system. The actual implementation and syntax of the above operators into modeling systems are not discussed here. Instead, Fig. 7-31 shows how they can be used conceptually in constructing boundary models.

Euler operators create changes in the number of the components in the topology under construction. The operators can be characterized by the transition status they make in the six-space defined by the parameters of Euler's law. Table 7.5 shows these changes for the operators listed in Table 7.4. Observe that the transition state of each operator satisfies Euler's law. This observation provides a general rule to design any new Euler operator. Moreover, if the operator

TABLE 7.5
Transition states of some Euler operators

Operator	F	E	V	L	B	G
MBFV	1	0	1	0	1	0
MEV	0	1	1	0	0	0
MEKL	0	1	0	−1	0	0
MEF	1	1	0	0	0	0
MEKBFL	−1	1	0	−1	−1	0
MFKLG	1	0	0	−1	0	−1
KFEVMG	−2	$-n$	$-n$	0	0	1
KFEVB	−2	$-n$	$-n$	0	−1	0
MME	0	n	n	0	0	0
ESPLIT	0	1	1	0	0	0
KVE	$-(n-1)$	$-n$	−1	0	0	0

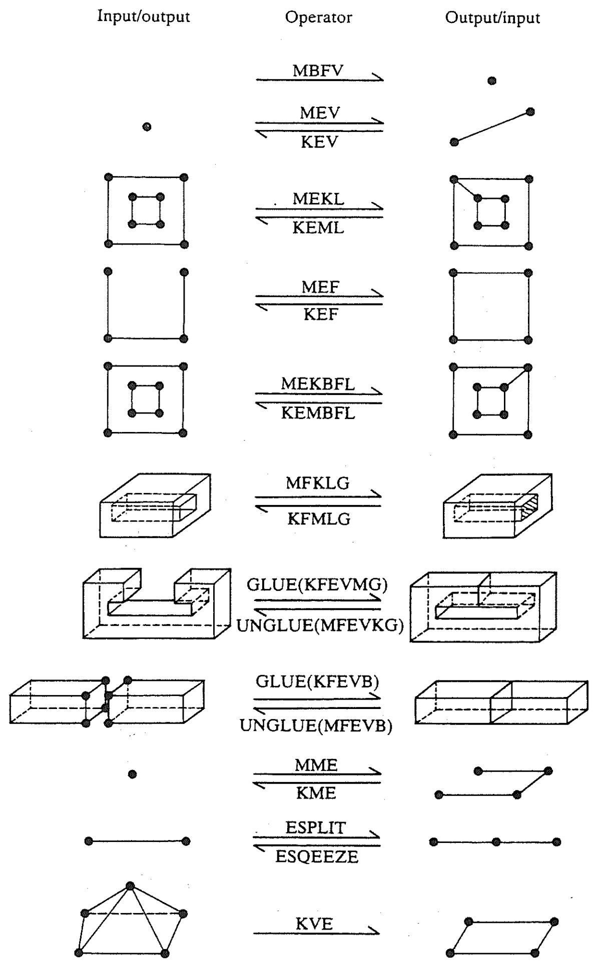


FIGURE 7-31
Topology creation via Euler operators.

acts on valid topology and the state transition it generates is valid, then the resulting topology is a valid solid. Therefore, Euler's law is never verified explicitly by the modeling system and its software. It should also be noticed that intermediate topology during construction may not make geometrical sense or may not represent an acceptable, though valid, solid (see Fig. 7-31).

Higher-level Euler operators are possible to develop. Examples include MCUBE, MCYL, MSPH, SWEEPR, and SWEEPT to respectively create a cube, a cylinder, a sphere, axisymmetric (rotational sweep) objects, and uniform (translational) objects. Figure 7-32 shows a wire and a face that are rotated to

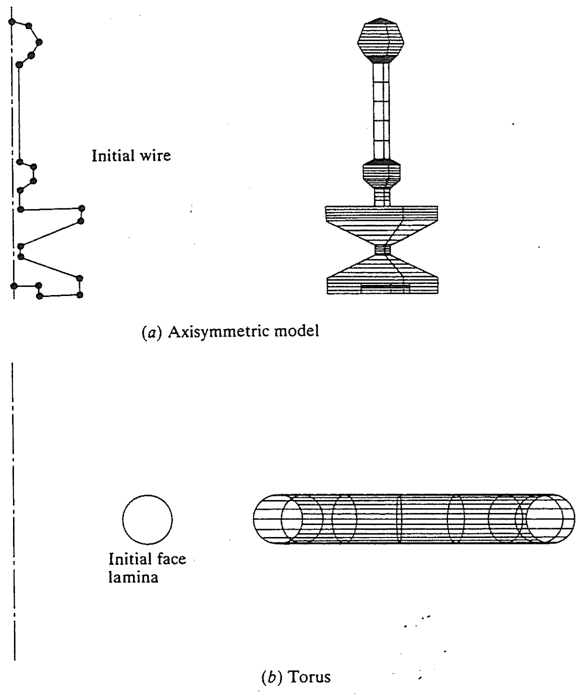


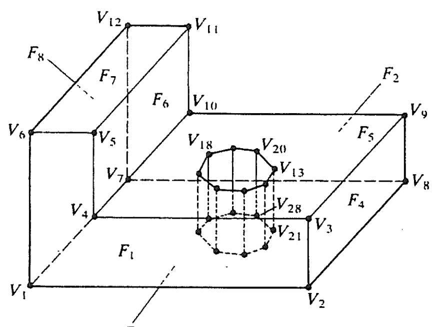
FIGURE 7-32
Rotational sweep (approximate) boundary models.

create a symmetric object and a torus respectively. In addition, union, difference, and intersection operators can be developed.

The advantages of Euler operators are that they ensure creating valid topology, they provide full generality and reasonable simplicity, and they achieve a higher semantic level than that of manipulating faces, edges, and vertices directly. However, Euler operators do not provide any geometrical information to define a solid polyhedron. They do not impose any restriction on surface orientation, face planarity, or surface self-intersection. Nevertheless, in practice, Euler operators perform a useful role as a topological foundation for developing routines that embody more algebra and geometry.

Example 7.5. Create the boundary model of solid S shown in Fig. 7-20*a*.

Solution. First let us develop the boundary model of the solid S. For simplicity, we assume that an approximate B-rep scheme is used. Figure 7-33 shows the boundary model of the solid. Based on the figure, the model has 16 faces, 28 vertices, 42 edges, 2 loops, 1 body, and 1 genus. They all together satisfy Euler's law. A suggested sequence to create the model is shown in Fig. 7-34. The sequence matches the planning strategy reflected in Fig. 7-33 in which the cylindrical face of the hole has been approximated by eight facets. Figure 7-34 is shown in an isometric view although it



Faces F_9 to F_{16} for hole are not shown

FIGURE 7-33
Boundary model of solid S.

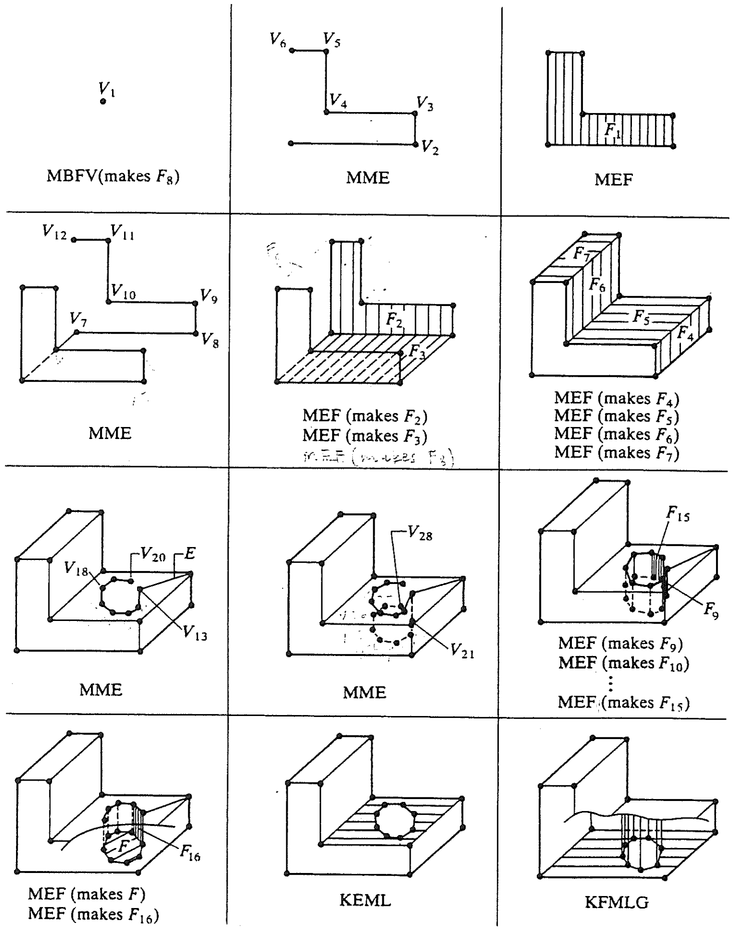


FIGURE 7-34
Creation of boundary model of solid S.

only reflects topology construction and not geometry. It is presented in this way for clarity and learning purposes. However, this topology could have been shown in one plane or one surface; that is, model S is homomorphic to a sphere. Each step in the figure shows the operator(s) used and its result(s) and whatever is needed from the previous steps. It is observed that at the end of construction the final number of primitives (faces, edges, vertices, loops, bodies, and genus) created is equal to the number calculated from Fig. 7-33. Any intermediate topology (edge E and face F) that is created for construction purposes has to be killed. Notice that the face created by the MBFV operator is chosen arbitrarily as F_8 for convenience. It could have been equally chosen to be F_1 or F_3. At this point, the face has no edges. However, its edges are formed automatically later on after creating the faces (F_1, F_6, and F_7) surrounding it. It would have been impossible to create F_8 otherwise and the topology would have been invalid if we had ignored the face created by MBFV.

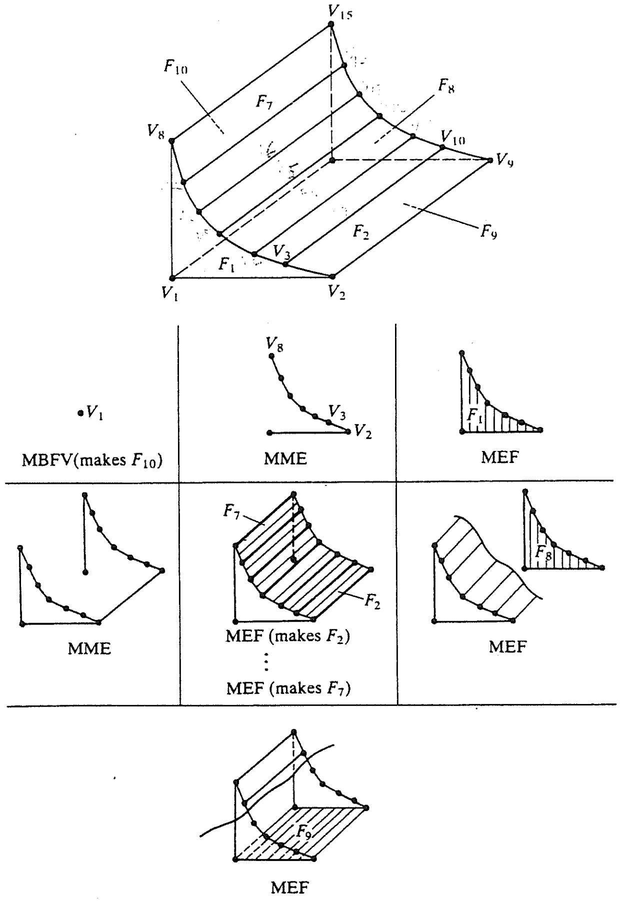


FIGURE 7-35
Creation of boundary model of solid fillet.

Similarly, the body created by this command is part of the topology and in fact is the body of solid *S*.

The reader can perhaps find a totally different set of steps than those shown in Fig. 7-34 to construct the boundary model, or these steps can change significantly depending on the available set of Euler operators. For example, if composite Euler operators for linear sweep and making cylinders are available, the model can easily be constructed in a smaller number of steps. The reader is encouraged to investigate this route.

Example 7.6. Create the boundary model of the solid fillet shown in Fig. 7-21.

Solution. Figure 7-35 shows the boundary model of the solid fillet and its creation. The curved face has been approximated by six facets. The construction steps follow the same general outline as in Example 7.5. It is obvious that the larger the number of facets, the more the CPU time and storage needed to create the model.

Example 7.7. Develop an algorithm that can enable the user to create and manipulate boundary models by using set operations.

Solution. We have mentioned in Sec. 7.2 and at the beginning of Sec. 7.7 that B-rep-based packages use set operations to create and manipulate boundary models. They seem to be more efficient than Euler operators. We have also mentioned in Sec. 7.7 that the effect of set operations on a B-rep model is different from that on a CSG model. In the former, the new boundary that results from the operation must be evaluated.

This example illustrates how to develop an algorithm to provide the user with the set operations union, difference, and intersection. The problem at hand can be stated as follows. Given two solids or primitives as boundary models, find their union, difference, and intersection. This problem is also known as boundary merging for B-reps and boundary evaluation for CSG. Set-operation algorithms are in general very complex programs. For simplicity purposes, let us assume that the two solids are not tangent to or touch each other. Thus, if two solids intersect

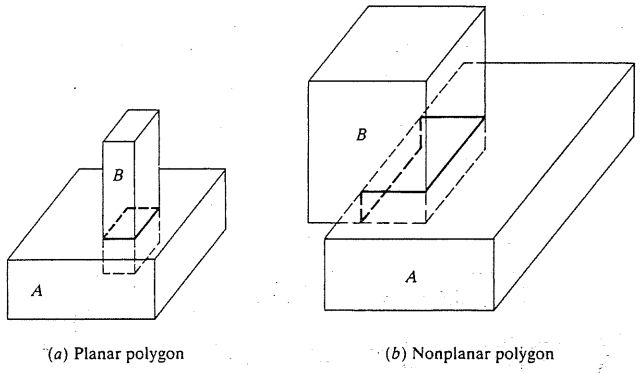


FIGURE 7-36
Intersection polygons of two solids.

at all, they do so along one or more closed (maybe nonplanar) intersection polygons. Figure 7-36 shows a planar and a nonplanar intersection polygon. Such polygons result from the edges of each solid intersecting the faces of the other solid.

While set-operation algorithms could be written to manipulate the low-level data structures directly, basing them on Euler operators offers important advantages. They guarantee the topological validity of the result of the set operation and they hide the actual data structure during writing of the algorithms. The algorithm we develop here is therefore based on Euler operators. First, it finds the intersection polygon between the two solids and checks for its planarity. Then it splits each solid's boundary along this polygon. The two solids are then classified against each other. Finally, the proper parts of the two solids are glued together to give the desired result.

The detailed steps of the algorithm for the case of a planar intersection polygon (see Fig. 7-36*a*) are described as follows:

1. Compare each of the faces of *A* with the edges of *B*. If an edge intersects a face in a point, we create a null edge as a loop into the face and split the edge by two coincident vertices to form a null edge. Figure 7-37*a* shows the result of this step. To create the null edges in the top face of *A*, the MME operator is used four times to create two edges (the dashed and the null edges) each time. This is followed by KEML four times to eliminate the undesired edges (dashed). To create the null edges in solid *B*, each of its intersecting edges is split twice by the ESPLIT operator. Each null edge in *A* or *B* has two coincident vertices, that is, geometrically identical vertices.
2. Repeat step 1 for faces of *B* and edges of *A*. The result of this step is null for this example.
3. Connect neighbor null edges in each solid to create the intersection polygons. A "combine" algorithm is needed to achieve this step and is assumed to be available. Such an algorithm could be based on connecting the vertices of a null edge to the vertices of the nearest null edge on the same face by the MEF operator. To accomplish connecting the null edges, they are sorted in a given direction by the coordinates of their vertices. An intersection polygon is constructed for each solid and since these polygons are identical, the "combine" algorithm can construct both of them simultaneously.

   In order to construct the intersection polygons, null faces have to be created. Some of these faces will be killed with the null edges created in steps 1 and 2 to complete the construction process and some will remain to enable the two solids *A* and *B* to be split. To avoid any confusion that may result from this process, the following rule can be followed. Use the MEKL operator to create as many edges as there are loops created in steps 1 and 2 and then use the MEF operator to create the remaining edges. For solid *A* in Fig. 7-37*b*, MEKL is applied four times to create four edges, say the top polygon, and kill the four loops created in step 1. Then MEF is used four times to create, say, the bottom polygon. The double-intersection polygon is needed to create the intersection faces. For solid *B*, MEF is applied eight times.

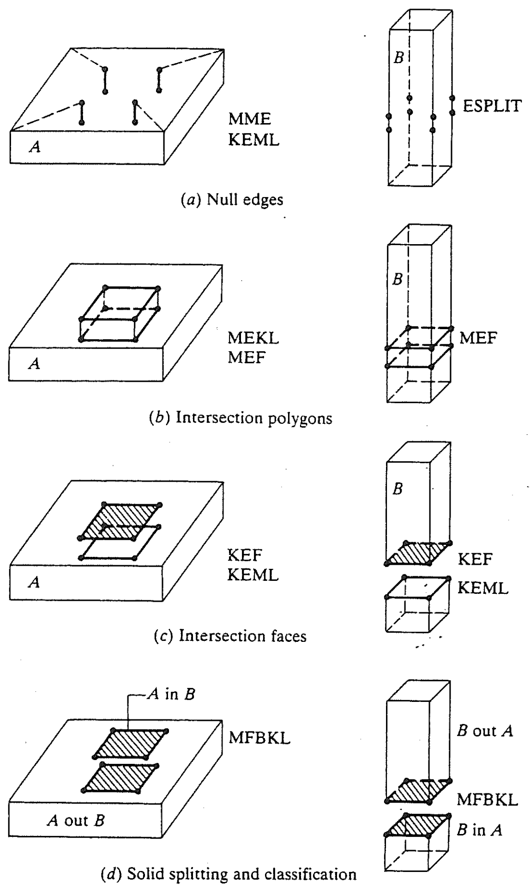


FIGURE 7-37
Splitting two solids by a set-operation algorithm.

4. Construct the intersection faces by deleting all the null edges in each solid. This is accomplished by using the KEF operator for all of the edges except the last null edge which is killed by KEML. In Fig. 7-37*c*, KEF is applied three times followed by one KEML.
5. Split each solid into two by using the MFBKL operator as shown in Fig. 7-37*d*.

   By following steps 1 to 5, the reader can easily find out that all the null edges are killed for each solid; eight edges and two faces are added to solid A, eight edges and six faces are added to solid B, and one body is added to each solid. While it is easy to interpret all these topological changes, faces need some clarification. For solid A, the two faces are the intersection faces. In this case, the top face of A is split into two faces; one of them is an intersection face. For solid B, two of the six faces are the intersection faces and the other four result from splitting each side face by the intersection polygon.

6. Each subsolid resulting from step 5 is classified against each original solid, that is, A is classified as A in B and A out B and B is classified as B in A and B out A. A in B, for example, means the parts of A's boundary inside B whereas A out B is outside B (see Fig. 7-37*d*). Notice that any part of the original solid is by itself a valid solid.
7. Combine the proper parts of each solid to obtain the desired set operation as follows:

$$A \cup B = \text{Glue } (A \text{ out } B,\ B \text{ out } A)$$
$$A \cap B = \text{Glue } (A \text{ in } B,\ B \text{ in } A)$$
$$A - B = \text{Glue } (A \text{ out } B,\ B \text{ in } A)$$
or
$$B - A = \text{Glue } (B \text{ out } A,\ A \text{ in } B)$$

   The gluing operator for $A \cup B$ and $A - B$ is simple because all the subsolids involved are closed objects. The KFEVB operator described in Tables 7.4 and 7.5 can be used to glue the subsolids to give the proper results, that is, closed solids. However, if the same gluing operator is used in $A \cap B$ and $B - A$, open (unregularized open sets) objects would result. This is because one of its operands (A in B) is an open object—in this example the two-dimensional intersection face. In such a case, the same previous gluing operator, KFEVB, can be used with the difference that it kills only one face instead of two. Therefore, the operator satisfies Eq. (7.58) and is used to kill the open object (intersection face).

The above described algorithm can be applied to any two boundary models whose classifications with each other do not yield A on B and/or B on A cases. This is why we mentioned in the beginning that A and B should not touch each other. The reader can extend solid B to pierce through A and apply the above steps.

The reader is also encouraged to apply this algorithm to the two solids shown in Fig. 7-36*b*. It can be assumed that an algorithm that sorts vertices by their planes is available. In this case, six null edges on each of A and B, four loops for A, eight faces for A, two loops for B, and twelve faces for B are created as intermediate results. After killing the null edges and splitting A and B, four and eight faces are created to split A and B respectively to give the final result. The gluing process is exactly as above except that the number of faces the gluing operator has to kill is four instead of two.

7.7.3 Remarks

The B-rep scheme is very popular and has a strong history in computer graphics because it is closely related to traditional drafting. Its main advantage is that it is very appropriate to construct solid models of unusual shapes that are difficult to build using primitives. Examples are aircraft fuselage and automobile body styling. Another major advantage is that it is relatively simple to convert a B-rep model into a wireframe model because the model's boundary definition is similar to the wireframe definition. For engineering applications studied to date, algorithms based on B-rep are reliable and competitive with those based on CSG.

One of the major disadvantages of the boundary model is that it requires large amounts of storage because it stores the explicit definition of the model boundaries. It is also a verbose scheme—more verbose than CSG. The model is defined by its faces, edges, and vertices which tend to grow fairly fast for complex models. If B-rep systems do not have a CSG-compatible user interface, then it becomes slow and inconvenient to use Euler operators in a design and production environment. In addition, faceted B-rep is not suitable for many applications such as tool path generations.

7.8 CONSTRUCTIVE SOLID GEOMETRY (CSG)

CSG and B-rep schemes are the most popular schemes to create solid models of physical objects. This is apparent from the existing research and technological activities. They are the most popular because they are the best understood representations thus far. CSG offers representations that are succinct, easy to create and store, and easy to check for validity. Moreover, difference and intersection operations can respectively provide means for material removal processes and interference checking between objects. Interference checking is useful in many applications such as vision and robot path planning.

A CSG model is based on the topological notion that a physical object can be divided into a set of primitives (basic elements or shapes) that can be combined in a certain order following a set of rules (boolean operations) to form the object. Primitives themselves are considered valid CSG models. Each primitive is bounded by a set of surfaces; usually closed and orientable. The primitives' surfaces are combined via a boundary evaluation process to form the boundary of the object, that is, to find its faces, edges, and vertices. In addition to degenerating an object to a collection of primitives, a CSG model is fundamentally and topologically different from a B-rep model in that the former does not store explicitly the faces, edges, and vertices. Instead, it evaluates them whenever they are needed by applications' algorithms, e.g., generation of line drawings. The reader might then ask the question: if a CSG scheme has to evaluate faces, edges, and vertices, why not use a B-rep scheme from the beginning? The answer to this question entails close comparison between all aspects of both schemes including efficiency and performance. Such comparison is difficult to make due to all implementation and algorithmic details involved. However, one answer can be given. The concept of primitives offers a different conceptual way of thinking that may be extended to model engineering processes such as design and manufacturing. It also appears that CSG representations might be of considerable importance for manufacturing automation as in the study of process planning and rough machining operations.

There are two main types of CSG schemes. The most popular one, and the one we always mean when we talk about CSG models, is based on bounded solid primitives, that is, r-sets. The other one, less popular, is based on generally unbounded half-spaces, that is, non-r-sets. The latter scheme belongs more to half-space representation covered in Sec. 7.6. As a matter of fact, bounded solid

primitives are considered composite half-spaces and the boundaries of these primitives are the surfaces of the corresponding half-spaces. CSG systems based on bounded primitives (e.g., PADL-2 and GMSOLID) allow their sophisticated users to use both their bounded primitives and/or half-spaces to create new primitives, typically called metaprimitives. It is also possible to extend the modeling domain of a system by implementing new half-spaces, and eventually new primitives, into its software. This implementation does not only require the trivial inclusion of the half-space equation into the software, but more importantly it requires developing supporting utilities such as intersecting the half-space with itself as well as other already existing half-spaces.

The modeling domain of a CSG scheme depends on the half-spaces that underlie its bounded solid primitives, on the available rigid motion and on the available set operators. For example, if two schemes have the same rigid motion and set operations but one has just a block and a cylinder primitive and the other has these two plus a tetrahedron, the two schemes are considered to have the same domain. Each has only planar and cylindrical half-spaces, and the tetrahedron primitive the other system offers is just a convenience to the user and does not extend its modeling domain. Similarly, the surfaces that a CSG scheme can represent directly depend on the bounding surfaces of its underlying half-spaces. The most widely represented surfaces are the quadric surfaces that bound most existing primitives. Extending the solid modeling domain to cover sculptured surfaces requires representing a "sculptured" half-space and its supporting utilities.

CSG schemes based on bounded primitives are usually more concise than those based on half-spaces because half-spaces are lower-level primitives. As an example, consider the solid shown in Fig. 7-38*a*. The model is represented by three bounded primitives (Fig. 7-38*b*) and seven half-spaces (Fig. 7-38*c*). Considering the half-spaces composing the three bounded primitives, it is obvious that 15 half-spaces (six for each block and three for the cylinder) have been used. Some of these half-spaces, such as the two at the bottom of blocks *A* and *B*, are redundant. This redundancy is perfectly accepted by users in trade of the conveniences they gain from using bounded primitives. However, it raises the question of the minimal CSG representation of a solid.

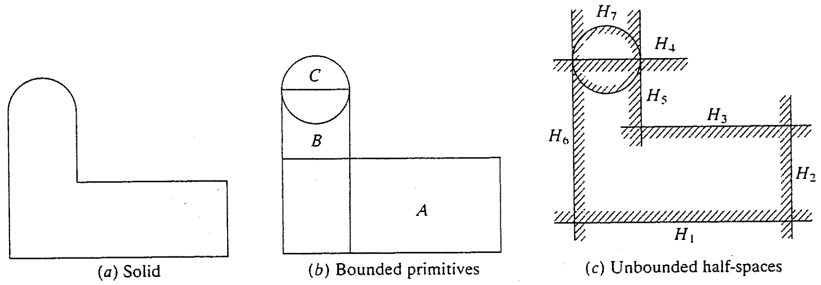


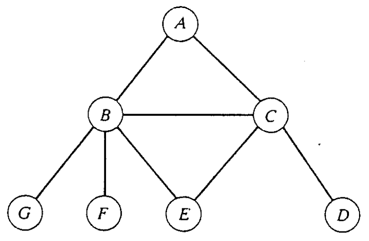
FIGURE 7-38
Bounded and unbounded primitives.

The database of a CSG model, similar to B-rep, stores its topology and geometry. Topology is created via the regularized set (boolean) operations that combine primitives. Therefore, the validity of the resulting model is reduced to the validity checks of the used primitives. For bounded primitives, these checks are usually simple (in the form of greater than zero) and the validity of the CSG model may be ensured essentially at the syntactical level. This means that in a CSG language a model is valid if it can be described syntactically correct using this language (user interface). The geometry stored in the database of a CSG model includes configuration parameters of its primitives and rigid motion and transformation. Geometry of faces, edges, and vertices are not stored but can be calculated via the boundary evaluation process.

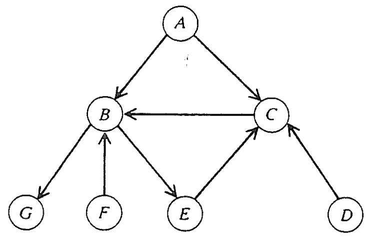
While data structures of most boundary representations are based on the winged-edge structure developed by Baumgart in 1972, data structures of most CSG representations are based on the concept of graphs and trees. This concept is introduced here in enough depth to enable understanding of CSG data structures. The interested reader is referred to any standard textbook on Pascal or data structures for more details.

A graph is defined as a set of nodes connected by a set of branches or lines. Each branch in a graph is specified by a pair of nodes. Figure 7-39*a* illustrates a graph. The set of nodes is $\{A, B, C, D, E, F, G\}$ and the set of branches, or the set of pairs, is $\{\{A, B\}, \{A, C\}, \{B, C\}, \{B, E\}, \{B, F\}, \{B, G\}, \{C, D\}, \{C, E\}\}$. Notice that these pairs are unordered, that is, no relations exist between the elements of each pair. For example, the pair $\{A, B\}$ can also be $\{B, A\}$. If the pairs of nodes that make up the branches are ordered pairs, the graph is said to be a directed graph or digraph. This means that branches have directions in a digraph and become in a sense arrows going from one node to another, as shown in Fig. 7-39*b*. The tail of each arrow represents the first node in the pair and its head represents the second node. The set of ordered pairs for Fig. 7-39*b* is $\{(A, B), (A, C), (C, B), (B, E), (F, B), (B, G), (D, C), (E, C)\}$.

Each node in a digraph has an indegree and outdegree and has a path it belongs to. The indegree of a node is the number of arrow heads entering the node and its outdegree is the number of arrow tails leaving the node. For example, node B in Fig. 7-39*b* has an indegree of 3 and an outdegree of 2 while node D has a zero indegree and an outdegree of 1. Each node in a digraph belongs to a path. A path from node n to node m is defined as a sequence of nodes $n_1, n_2, \ldots, n_k$ such that $n_1 = n$ and $n_k = m$ and any two subsequent nodes (n_i, n_{i+1}) in the sequence are adjacent to each other. For example, the path from node A to node G in Fig. 7-39*b* is A, B, G or A, C, B, G. If the start and end nodes of a path are the same, the path is a cycle. If a graph contains a cycle, it is cyclic; otherwise it is acyclic.



(*a*) Graph

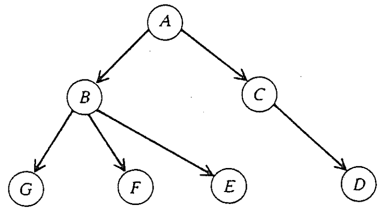


(*d*) Digraph

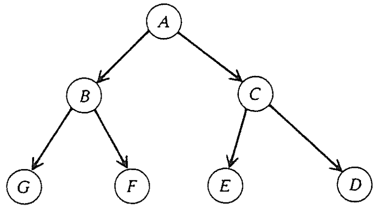
FIGURE 7-39
Graphs and digraphs.

We now turn our attention to trees. A tree is defined as an acyclic digraph in which only a single node, called the root, has a zero indegree and every other node has an indegree of 1. This implies that any node in the tree except the root has predecessors or ancestors. Based on this definition, a graph need not be a tree but a tree must be a graph. The digraph shown in Fig. 7-39*b* is not a tree. However, its modification shown in Fig. 7-40*a* is a tree. Node A is the root of the tree and nodes E, F, and G, for example, have node B as their ancestor or node B has nodes E, F, and G as its descendants. If the descendants of each node are in order, say, from left to right, then the tree is an ordered one. Moreover, when each node of an ordered tree has two descendants (left and right), the tree is called a binary tree (see Fig. 7-40*b*). Finally, if the arrow directions in a binary tree are reversed such that every node, except the root, in the tree has an outdegree of 1 and the root has a zero outdegree, the tree is called an inverted binary tree (see Fig. 7-40*c*). An inverted binary tree is very useful to understand the data structure of CSG models (sometimes called boolean models).

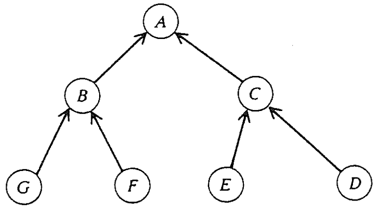
Any node in a tree that does not have descendants, that is, with an outdegree equal to zero, is called a leaf node and any node that does have descendants (outdegree greater than zero) is an interior node. In Fig. 7-40*b*, nodes D, E, F, and G are leaf nodes and nodes B and C are interior nodes. Nodes G and D are called the leftmost leaf and the rightmost leaf of the tree respectively. Nodes



(*a*) Tree



(*b*) Binary tree



(*c*) Inverted binary tree

FIGURE 7-40
Types of trees.

in a tree can also be viewed from a different perspective as follows. Every node of a tree T is a root of another tree, called a subtree of T, contained in the original tree T. A subtree is itself a binary tree. Any tree can be divided into two subtrees: left and right subtrees of the original tree. Considering Fig. 7-40b, the original tree consists of seven nodes with A as its root. Its left subtree is rooted at B and its right subtree is rooted at C. This is indicated by the two branches emanating from A to B on the left and to C on the right. The absence of a branch indicates an empty subtree. The binary trees rooted at the leaves D, E, F, and G have empty (nil) left and right subtrees.

Let us return back to the data structures of CSG representations and relate them to graphs and trees. Consider the solid shown in Fig. 7-41a with its MCS. A block and a cylinder primitive are enough to create the CSG model of the solid. Figure 7-41b shows one of the possible ways to decompose the solid into its primitives. Using the local coordinate systems of the primitives as shown in Fig. 7-4, and regardless of the user interface or command syntax offered by a particular CAD/CAM system, a user can construct the CSG model using the following steps:

$$\left.\begin{aligned}
&B_1 = \text{block positioned properly}\\
&B_2 = \text{block positioned properly}\\
&B_3 = \text{block}\\
&B_4 = B_3 \text{ moved properly in the } X \text{ direction}\\
&C_1 = \text{cylinder positioned properly}\\
&C_2 = C_1 \text{ moved properly in the } X \text{ direction}\\
&C_3 = \text{cylinder positioned properly}\\
&C_4 = C_3 \text{ moved properly in the } X \text{ direction}
\end{aligned}\right\}\ \text{Primitives' definitions}$$

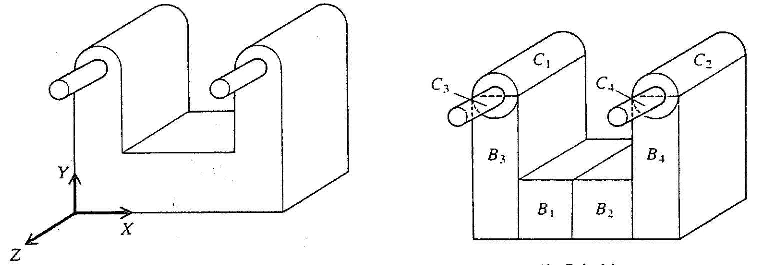
$$\left.\begin{aligned}
S_1 &= B_1 \cup^* B_3\\
S_2 &= S_1 \cup^* C_1\\
S_3 &= S_2 \cup^* C_3
\end{aligned}\right\}\ \text{Construct left half}$$

$$\left.\begin{aligned}
S_4 &= B_2 \cup^* B_4\\
S_5 &= C_2 \cup^* S_4\\
S_6 &= C_4 \cup^* S_5
\end{aligned}\right\}\ \text{Construct right half}$$

$$S = S_3 \cup^* S_6\ \}\ \text{Model}$$

To save the above steps in a data structure, such a structure must preserve the sequential order of the steps as well as the order of the boolean operations in any step; that is, the left and right operands of a given operator. The ideal solution is a digraph; call it a CSG graph. A CSG graph is a symbolic (unevaluated) representation and is intimately related to the modeling steps used by the user. This makes the CSG graph a very efficient data structure to define and edit a solid. The CSG graph representing the above steps is shown in Fig. 7-42. Each of the intermediate solids S_1 to S_6 is shown as the same node of its corresponding set operation node. Notice that the steps starting from S_1 and ending at S can be replaced by

$$S = B_1 \cup^* B_3 \cup^* C_1 \cup^* C_3 \cup^* B_2 \cup^* B_4 \cup^* C_2 \cup^* C_4$$



(a) Typical solid (b) Primitives

FIGURE 7-41
A typical solid and its building primitives.

where set operations are evaluated from left to right unless otherwise indicated by parenthesis. In this case the intermediate solids S_1 to S_6 do not exist and should be removed from the CSG graph.

While a CSG graph has a succinct data structure to represent a solid model and is suitable for convenient and efficient editing of the model, it is not suitable to use in geometric computation. This is mainly because of the cycles that the graph may have which, in turn, means graph nodes may be shared to reflect congruence relationships in the solid. This sharing means that useful information about the solid such as the locations of shared nodes is not explicitly stored by the graph structure. Another reason the CSG graph is not efficient in computations is its storage of real expressions that may be used in defining a solid (e.g., $c = b^2$, then use c as a primitive parameter) as strings, that is, unevaluated.

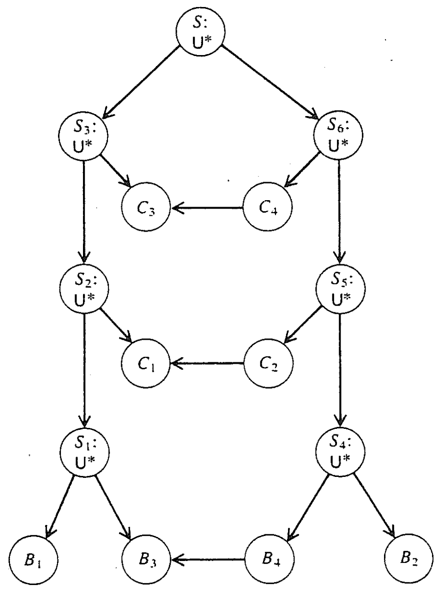


FIGURE 7-42
CSG graph of a typical solid.

Therefore, a less symbolic and more evaluated data structure is needed before involving computation and application algorithms such as boundary evaluation and mass properties of a solid. A CSG tree data structure is an ideal solution. It is a natural extension of the CSG graph and results from copying shared nodes and evaluating all strings (real expressions). Some solid modelers such as PADL-2 has both data structures. In these modelers, the CSG graph is the primary data structure and the CSG tree structure is derived from it whenever needed. Other modelers may have the CSG tree as their only data structure.

A CSG tree is defined as an inverted ordered binary tree whose leaf nodes are primitives and interior nodes are regularized set operations. Figure 7-43 shows the CSG tree derived from the CSG graph shown in Fig. 7-42. Notice that this CSG tree can be derived directly from the modeling steps without having to create the CSG graph. As a matter of fact, the tree can be created from the planning strategy shown in Fig. 7-41*b*. In Fig. 7-43, blocks B_1 to B_4, cylinders C_1 to C_4, and union operators are renamed as P_1 to P_4, P_5 to P_8, and OP$_1$ to OP$_7$ respectively to emphasize the fact that they are evaluated and stored explicitly compared to their counterparts used in the CSG graph (Fig. 7-42). The CSG tree is shown with its full details including arrows. In practice, the arrows are usually not shown, the leaf nodes are just shown as primitives' names without circles surrounding them, and a line extends from the tree root up to indicate the result of the final solid. Other styles of showing a CSG tree may replace primitive names by their sketches as well as showing each intermediate solid that results from an operator in the stream of the tree branches.

The total number of nodes in a CSG tree of a given solid is directly related to the number of primitives the solid is decomposed to. The number of primitives decides automatically the number of boolean operations required to construct the solid. If a solid has n primitives, then there are $(n - 1)$ boolean operations for a total of $(2n - 1)$ nodes in its CSG tree. The balanced distribution of these nodes in the tree is a desired characteristic for various applications, especially those that use ray casting such as shading and mass properties. A balanced tree is defined as

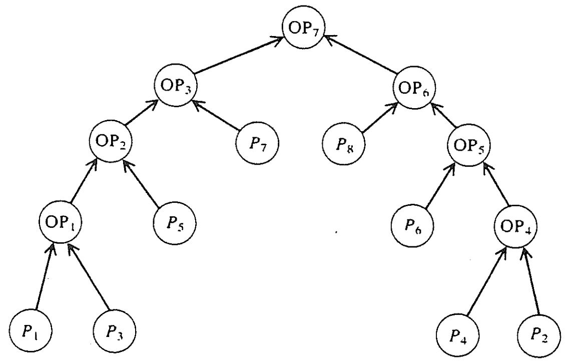


FIGURE 7-43
CSG tree of a typical solid.

a tree whose left and right subtrees have almost an equal number of nodes; that is, the absolute value of the difference $(n_L - n_R)$ is as minimal as possible where

$$n_L + n_R = 2n - 2 \tag{7.59}$$

The root node is not included in this equation. n_L and n_R are the number of nodes of the left and right subtrees respectively. A perfect tree is one whose $|n_L - n_R|$ is equal to zero. A perfect tree results only if the number of primitives is even. For a perfect tree, the following equation applies:

$$n_L = n_R = n - 1 \tag{7.60}$$

Each subtree has $n/2$ leaf nodes (primitives) and $(n - 2)/2$ interior nodes (boolean operations). Figure 7-43 shows a perfect tree.

The creation of a balanced, unbalanced, or a perfect CSG tree depends solely on the user and how he/she decomposes a solid into its primitives. The general rule to create balanced trees is to start to build the model from an almost central position and branch out in two opposite directions or vice versa; that is, start from two opposite positions and meet in a central one. The tree shown in Fig. 7-43 begins at the central blocks B_1 and B_2 and branches out. Another useful rule is that symmetric objects can lead to perfect trees if they are decomposed properly (see Figs. 7-41*b* and 7-42) starting from the plane(s) of symmetry. Figure 7-44 shows an unbalanced tree of the same solid shown in Fig. 7-41. This tree results if the user starts building the model from the left or right side. In this figure, primitives P_1 to P_7 correspond to primitives C_1, C_3, B_3, $B_1 + B_2$, B_4, C_4, and C_2 respectively, shown in Fig. 7-41*b*. In this tree $n_L = 11$ and $n_R = 1$. Reorganizing an unbalanced tree internally by a solid modeler is possible but is not practical to do.

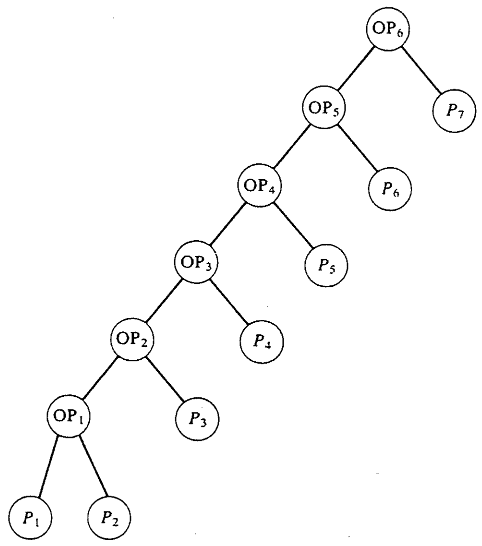


FIGURE 7-44
An unbalanced CSG tree.

Application algorithms must traverse a CSG tree, that is, pass through the tree and visit each of its nodes. Also traversing a tree in a certain order provides a way of storing a data structure. The order in which the nodes are visited in a traversal is clearly from the first node to the last one. However, there is no such natural linear order for the nodes of a tree. Thus different orderings are possible for different cases. There exist three main traversal methods. The methods are all defined recursively so that traversing a binary tree involves visiting the root and traversing its left and right subtrees. The only difference among the methods is the order in which these three operations are performed. The three methods are preorder, inorder, and postorder traversals. Sometimes, these methods are referred to as prefix, infix, and postfix traversals. Three other methods can be derived from these three main ones by reversing the order of the traversal to give reverse preorder, reverse inorder, and reverse postorder traversals.

To traverse a tree in preorder, we perform the following three operations in the order they are listed:

1. Visit the root.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

In the reverse preorder method, the three operations are reversed to give the sequence of visiting the right subtree, then the left subtree, and then the root. Figure 7-45 shows the preorder, and its reverse, traversal of the tree shown in Fig. 7-43.

To traverse a tree in inorder (or symmetric order):

1. Traverse the left subtree in inorder.
2. Visit the root.
3. Traverse the right subtree in inorder.

In the reverse inorder method, the tree is traversed by visiting the right subtree, then the root, and then the left subtree (see Fig. 7-46).

To traverse a tree in postorder:

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root.

In the reverse postorder method, the tree is traversed by visiting the root, then the right subtree, and then the left subtree, as shown in Fig. 7-47.

By comparing Figs. 7-45 to 7-47, the reader can easily observe that the reverse preorder is a mirror image of the postorder, the reverse postorder is a mirror image of the preorder, and the reverse inorder is a mirror image of the inorder.

Which of the traversal methods shown in Figs. 7-45 to 7-47 is more suitable to store a tree in a solid modeler? In arithmetic expressions, e.g., $A + (B + C)D$,

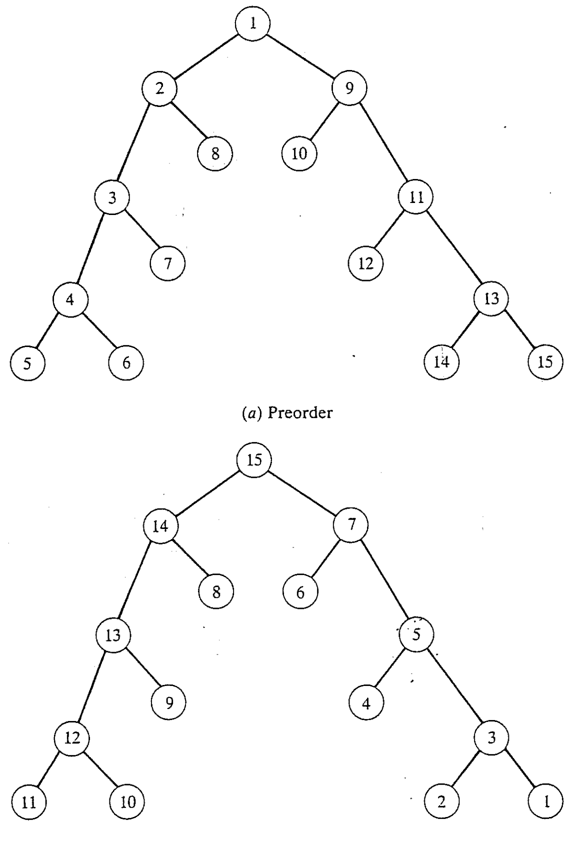
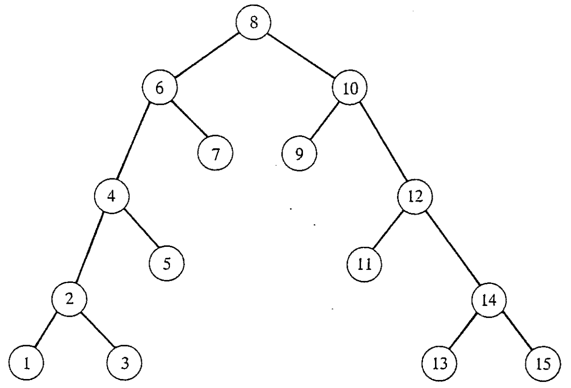
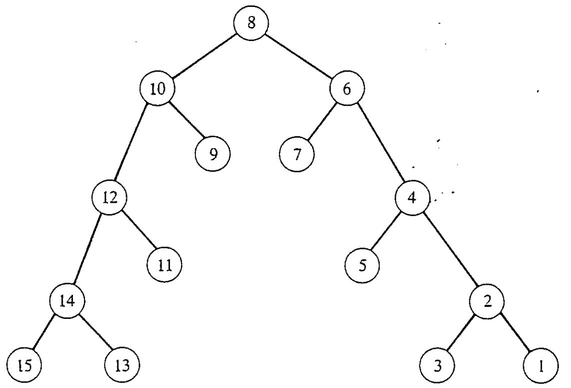


FIGURE 7-45
Preorder and reverse preorder traversals of a tree.

the order of operations in an infix expression might require cumbersome parentheses while a prefix form requires scanning the expression from right to left. Since most algebraic expressions are read from left to right, postfix is a more natural choice. In addition, if the concept of a stack (refer to Pascal textbooks) (last-in, first-out behavior) is used in an algorithm to evaluate an expression, the postfix becomes the most efficient form. These same rationales can be extended to binary trees. Trees are derived from steps that are commands input by a user to create a solid. These commands are scanned from left to right by the software and they might contain parentheses. In addition, if stacks are used in algorithms that evaluate trees (PADL-2 does that), then the postorder is the ideal choice to traverse a tree. However, the problem with the postorder traversal, as shown in Fig. 7-47*a*, is that the root of the tree has the highest node number. It is more



(*a*) Inorder



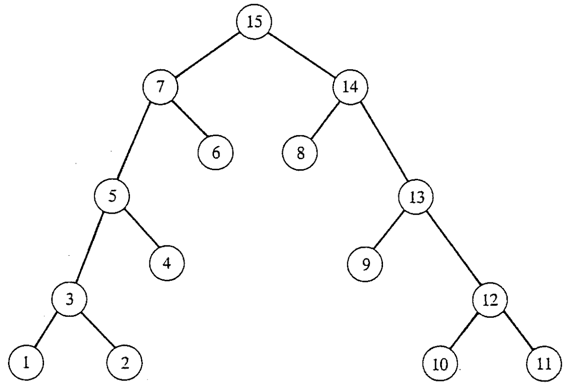
(*b*) Reverse inorder

FIGURE 7-46
Inorder and reverse inorder traversals of a tree.

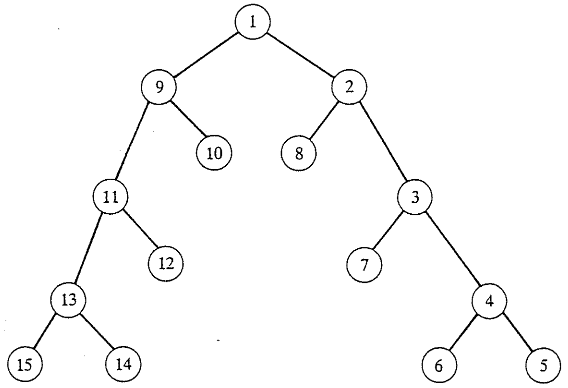
natural to assign the root the number 1. Therefore, the reverse postorder seems the ideal traversal method of a CSG tree. PADL-2 solid modeler uses such a method. In this method also the leftmost leaf node of the tree has the highest node number in the tree.

7.8.1 Basic Elements

Bounded solid primitives, or primitives for short, are the basic elements or building blocks a CSG scheme utilizes to build a model. Primitives can be viewed as parametric solids which are defined by two sets of geometric data. The first set is called configuration parameters and the second is the rigid motion parameters. The most common primitives are shown in Fig. 7-4. Each one of these primitives



(*a*) Postorder



(*b*) Reverse postorder

FIGURE 7-47
Postorder and reverse postorder traversals of a tree.

is defined by its configuration and rigid motion parameters. For example, the configuration parameters of a block primitive is the triplet (ordered 3-tuple) (W, H, D) and its rigid motion is given by the location of its origin P relative to a reference coordinate system, say MCS or WCS, or by explicit rigid motion values (translation and/or rotation). The configuration parameters of the other primitives are shown in Fig. 7-4.

Each primitive, viewed as a parametric object, corresponds to a family of parts. Each given part of the family is called a primitive instance and corresponds to one and only one value set of the primitive configuration parameters. Each primitive has a valid configuration domain which is maintained by its solid modeler. User input values of any primitive parameters are usually checked against its valid domain. For example, a block primitive instance of the triplet

(0, 0, 0) is not a valid instance because the corresponding parameters are not within the valid domain of a block.

The choice of the two sets of the geometric data to define the size (via configuration parameters) and the orientation (via rigid motion parameters) of a primitive are based on the fact that any primitive can be described generically by an equation in its local coordinate system. The configuration parameters define such an equation completely. Utilizing the rigid motion parameters, the equation and, therefore, the primitive can be transformed properly into another coordinate system. Therefore, primitives' information such as equations, intersections, boundaries, and others are usually expressed in terms of the primitive local coordinate system $X_L Y_L Z_L$.

Mathematically, each primitive is defined as a regular point set of ordered triplets (x, y, z). For the primitives shown in Fig. 7-4, these point sets are given by:

Block: $\{(x, y, z)\colon 0 < x < W,\ 0 < y < H, \text{ and } 0 < z < D\}$ (7.61)

Cylinder: $\{(x, y, z)\colon x^2 + y^2 < R^2, \text{ and } 0 < z < H\}$ (7.62)

Cone: $\{(x, y, z)\colon x^2 + y^2 < [(R/H)z]^2, \text{ and } 0 < z < H\}$ (7.63)

Sphere: $\{(x, y, z)\colon x^2 + y^2 + z^2 < R^2\}$ (7.64)

Wedge: $\{(x, y, z)\colon 0 < x < W,\ 0 < y < H,\ 0 < z < D,$
$\text{and } yW + xH < HW\}$ (7.65)

Torus: $\{(x, y, z)\colon (x^2 + y^2 + z^2 - R_2^2 - R_1^2)^2 < 4R_2^2(R_1^2 - z^2)\}$ (7.66)

Comparing Eqs. (7.61) to (7.66) with the half-space equations (7.51) to (7.55), it is obvious that each of the above bounded primitives is a combination of a finite number of half-spaces. A block is the regularized union of six intersecting half-spaces. Each of these half-spaces is given by one limit of the three inequalities of Eq. (7.61). Similarly, a cylinder, cone, and a wedge are the union of three, three, and five half-spaces respectively. Figure 7-48 shows two-dimensional illustrations of the half-spaces of each primitive shown in Fig. 7-4. Some half-spaces for the block and wedge primitives are not shown in the figure for clarity purposes.

There are many representational alternatives for primitives. Some representations are terse and contain little or no redundant data. These are called input representations and are convenient for user input. Other representations are verbose, contain lots of redundant data, and are therefore convenient and efficient for computational purposes. They are called internal representations. Most CSG-based modelers use both and usually derive the internal representation from the input one. While one of these modelers may provide alternative input representation, mainly for user convenience, it usually has only one internal representation and all input alternatives are converted to it before storage. Consider, for example, the torus primitive shown in Fig. 7-4. A user can specify its R_I and R_O or R_1 and R_2 as input representations to create it.

What are the redundant data of a primitive that a solid modeler calculates based on user input representation and stores as its internal representation for computational purposes? An internal representation of a primitive that does not

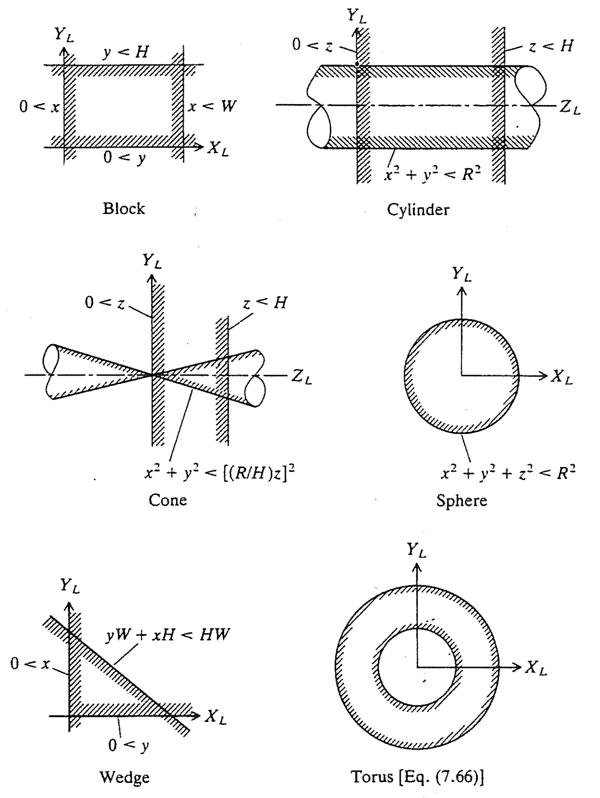


FIGURE 7-48
Half-spaces of bounded primitives.

have redundant data would only store the primitive's underlying half-spaces positioned and oriented properly in space, based on the user's configuration and rigid motion input parameters. Any other data such as primitive faces and edges that might be needed to evaluate the result of, say, a boolean operation must be derived by explicitly calculating the proper intersections of the underlying half-spaces. Such an approach would make application and computational algorithms totally inefficient. Therefore, underlying surfaces, faces, and edges, surface normals, and other data that are considered redundant are stored internally for each primitive in addition to its half-spaces. In essence the internal representation of each primitive is a CSG-rep plus a B-rep plus other information that is computationally useful. This "other information" could be engineering and design related in the case of implementing a new application into solid modeling.

Let us now look closely into how faces, edges, and other redundant data of a primitive are represented. Analogous to decomposing a solid into a combination of primitives, each primitive can be decomposed into a collection of faces and edges. Each face is a finite region of a closed orientable surface and each edge is a finite segment of an underlying curve. Therefore, a CSG scheme would have a set of primitives for its users to use and internally would have a set of half-spaces, a set of closed orientable surfaces (boundaries of these half-spaces), a set of primitive faces, and a set of primitive edges. Figure 7-49 shows such a data structure (internal representation) of a typical primitive. The PADL-2 solid modeler uses the structure shown in this figure.

The underlying surfaces, primitive faces, and primitive edges that a solid modeler can provide are directly related to the half-spaces the modeler CSG scheme utilizes. If a scheme utilizes the natural quadrics given by Eqs. (7.51) to (7.55), then planar, cylindrical, spherical, conical, and toroidal surfaces (called quadric surfaces) become the underlying surfaces of the scheme or the modeler. These surfaces are the boundaries of their corresponding half-spaces and their point sets are given by:

Planar surface: $P = \{(x, y, z): z = 0\}$ (7.67)

Cylinder surface: $P = \{(x, y, z): x^2 + y^2 = R^2\}$ (7.68)

Spherical surface: $P = \{(x, y, z): x^2 + y^2 + z^2 = R^2\}$ (7.69)

Conical surface: $P = \{(x, y, z): x^2 + y^2 = [(R/H)z]^2\}$ (7.70)

Toroidal surface: $P = \{(x, y, z): (x^2 + y^2 + z^2 - R_2^2 - R_1^2)^2 = 4R_2^2(R_1^2 - z^2)\}$ (7.71)

These are infinite surfaces whose intersections yield infinite curves. These curves are usually classified against given primitives using set membership classification to determine which curve segments lie within these primitives and consequently within the solid.

Primitive faces are faces of primitives selected such that the boundary of any primitive may be represented as the union of a finite number of these faces after being positioned properly in space. The sufficient set of primitive faces to represent the boundary of any of the primitives shown in Fig. 7-4 consists of plate, triplate, disc, cylindrical, spherical, conical, and toroidal primitive faces. The equations of these primitive faces (Pfaces for short) are given by:

Plate Pface:: $F = \{(x, y, z): 0 < x < W, 0 < y < H, \text{ and } z = 0\}$ (7.72)

Triplate Pface: $F = \{(x, y, z): 0 < x < W, 0 < y < H, \text{ and } yW + xH < HW\}$ (7.73)

Disc Pface: $F = \{(x, y, z): x^2 + y^2 < R^2, \text{ and } z = 0\}$ (7.74)

Cylindrical Pface: $F = \{(x, y, z): x^2 + y^2 = R^2, \text{ and } 0 < z < H\}$ (7.75)

Spherical Pface: $F = \{(x, y, z): x^2 + y^2 + z^2 = R^2\}$ (7.76)

Conical Pface: $F = \{(x, y, z): x^2 + y^2 = [(R/H)z]^2, \text{ and } 0 < z < H\}$ (7.77)

Toroidal Pface: $F = \{(x, y, z): (x^2 + y^2 + z^2 - R_2^2 - R_1^2)^2 = 4R_2^2(R_1^2 - z^2)\}$ (7.78)

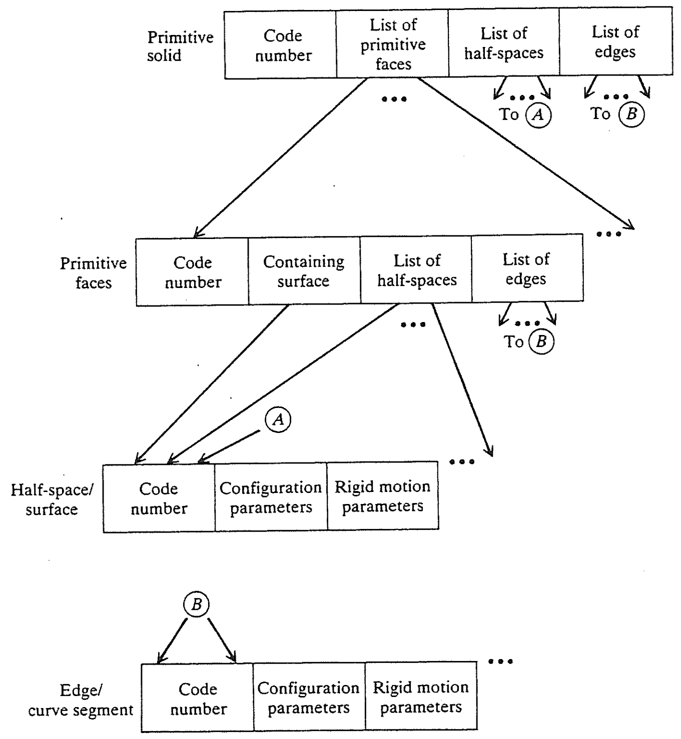


FIGURE 7-49
Data structure of a typical primitive solid.

Any of the above primitive faces is a subset of its underlying surface, that is, $F \subset P$. For example, the plate primitive face, Eq. (7.72), is a subset of the planar surface given by Eq. (7.67). The subset is bounded in the x and y directions. Similarly, a cylindrical primitive face is a finite (but not bounded) region (between $z = 0$ and H) of the cylindrical surface given in Eq. (7.68). In addition, the boundary of any primitive is a combination of these primitive faces. The boundary of a block primitive consists of six plate primitive faces positioned properly while that of a sphere or a torus consists of one spherical or toroidal primitive face respectively. The boundary of a cylinder consists of one cylindrical primitive face closed from each end by a disc primitive face. Lastly, a cone has a conical face closed by a disc and a wedge has three plates and two triplates.

Similar to primitive faces, primitive edges (sometimes called face bounding edges) are edges selected such that the boundary of any primitive face may be represented as the union of a finite number of these edges after being positioned properly in space. Each edge is a finite or bounded region of a corresponding underlying curve that may be possibly unbounded and disjointed (in which case curve segments make up the total curve). The underlying curves are usually obtained by finding all possible intersections of the underlying surfaces represented by a given CSG scheme. Curves, and therefore edges, can be true curves or virtual (profile or silhouette) curves, as in the case of a cylinder, sphere, cone, and a torus. For quadratic surfaces, the virtual edges can be generated by intersecting the surface with a plane. This offers the advantage that true and virtual curves can be obtained via surface/surface intersection. It is useful to realize that edges and curves, like faces and surfaces, have a single representation with a dual purpose. This implies that a curve equation is also an edge equation after imposing the proper parameter limits. Because surface/surface intersections are computationally intensive and because a particular CSG scheme represents a given set of surfaces, surface/surface intersection problems are solved algebraically *a priori* and stored in the corresponding solid modeler. A solid modeler that supports the primitives shown in Fig. 7-4 must contain the intersections of the surfaces given by Eqs. (7.67) to (7.71). For example, a cylinder/plane intersection may give a circle, an ellipse (could be very thin), two infinite parallel lines, or no intersection at all. What usually complicates the surface/surface intersection is the position and orientation of the two intersecting surfaces in space. One solution to this problem is to intersect the two surfaces in a given standard position and orientation and then transform the result to the actual position and orientation. Intersection curves are usually represented in parametric form because quadric surfaces can be parametrized conveniently as shown in Chap. 6. Equations of edges and curves are not given here. The reader is referred to Chap. 5 for some of these equations.

Surface/surface intersection is very crucial in geometric modeling in general and in solid modeling in particular. It is a decisive factor in determining and/or limiting the modeling domain of a solid modeler. As a matter of fact, it is the only factor in slowing down the implementations of sculptured surfaces into solid modeling. One might ask the following basic question. Why is surface/surface intersection so important while algebraic and numerical methods exist to solve virtually any two equations? The answer is not so much in the solution as it is in characterizing the solution. In order to perform boolean operations automatically, efficiently, and unambiguously, we need a precise description of the intersection curve. We need to know which surface pair results in which intersection curve, and we need to know when two different surface pairs give rise to the same curve. For example, if a boolean operation requires the intersection of a plane and a cylinder, the advance knowledge of their intersection curve provides a precise equation of the curve and consequently saves computation time that would otherwise be spent solving the two surface equations. In addition, two curves of a pair of surface/surface intersections can be checked out if they are identical, which can help eliminate many problems that may occur trying to differentiate between them. One may then conclude that if there was a universal description, say an equation, of any intersection curve in terms of the parameters of the two intersecting surfaces, the surface/surface intersection would have been solved once and for all.

While the full details of surface/surface intersections of quadric surfaces are beyond the scope of this book, the essence of the problem can be described as follows. Any of the surfaces described by Eqs. (7.67) to (7.71) can be rewritten in

the following polynomial form:

$$Ax^2 + By^2 + Cz^2 + 2Dxy + 2Eyz + 2Fxz + 2Gx + 2Hy + 2Jz + K = 0 \tag{7.79}$$

where $A, B, \ldots, K$ are arbitrary real constants. This equation can be expressed in a quadratic form as

$$F(x, y, z) = \mathbf{V}^T[Q]\mathbf{V} = 0 \tag{7.80}$$

where $\mathbf{V}$ is a vector of homogeneous coordinates of a point on the surface and is given by $[x \quad y \quad z \quad 1]^T$. $[Q]$ is the coefficient matrix. It is symmetric and is given by

$$[Q] = \begin{bmatrix} A & D & F & G \\ D & B & E & H \\ F & E & C & J \\ G & H & J & K \end{bmatrix} \tag{7.81}$$

The coefficient matrix $[Q]$ can be formed for any of the quadric surfaces [Eqs. (7.67) to (7.70)] by comparing the surface equation with Eq. (7.80). This gives

$$\text{Planar surface:} \qquad [Q] = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & 0 \end{bmatrix} \tag{7.82}$$

$$\text{Cylindrical surface:} \qquad [Q] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -R^2 \end{bmatrix} \tag{7.83}$$

$$\text{Spherical surface:} \qquad [Q] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -R^2 \end{bmatrix} \tag{7.84}$$

$$\text{Conical surface:} \qquad [Q] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -(R/H) & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{7.85}$$

The coefficient matrix $[Q]$ depends directly on the surface orientation and position. The above matrices are valid only if the local coordinate system of the primitive is identical to the MCS of the solid model to whom the primitive belongs. Otherwise, each matrix has to be transformed by the transformation matrix that results from the rigid motion (rotation and/or translation) of the primitive. This gives

$$[Q'] = [T]^T[Q][T] \tag{7.86}$$

where $[Q']$ is the transformed coefficient matrix and $[T]$ is the transformation matrix given by Eq. (3.3). For example, if the origin of a sphere is located at point (a, b, c) measured in the MCS, Eq. (7.84) is transformed by a translation vector to give

$$[Q'] = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ a & b & c & a^2 + b^2 + c^2 - R^2 \end{bmatrix} \tag{7.87}$$

This idea of transformation suggests that one can solve the intersection problem of two primitives in any convenient coordinate system and then transform the results as needed. Actually, there exists a set of algebraic constructs derived from $[Q]$ that remain invariant under rigid motions and completely specify the shape of the quadratic surface.

The intersection of two quadric surfaces can be solved as follows. If one surface has a coefficient matrix $[Q_1]$ and the other has $[Q_2]$, then the equation of the intersection curve is

$$\mathbf{V}^T([Q_1] - [Q_2])\mathbf{V} = 0 \tag{7.88}$$

This equation describes an infinite intersection curve. In order to determine finite segments of this curve (edges) which belong to the intersecting primitives, we must find the appropriate bounding points. One way of finding these points is if one of the intersecting surfaces can be parametrized in terms of two parameters u and v, and Eq. (7.88) can be solved for one of the parameters in terms of the other. In this case, the equation of the intersecting curve can be written in a parametric form in terms of one parameter. To understand this approach, consider the cylinder/quadric intersection case. In this problem, we wish to find the intersection curve between a cylinder and any quadric surface. Assuming the cylinder is in a standard position, its parametric equation is known. In this case the vector $\mathbf{V}$ in Eq. (7.88) is $[R \cos u \quad R \sin u \quad v]^T$. Substituting Eqs. (7.81) and (7.83) for $[Q_1]$ and $[Q_2]$ respectively and simplifying the result we get

$$a(u)v^2 + b(u)v + c(u) = 0 \tag{7.89}$$

Equation (7.89) could also be obtained by substituting the parametric equation of a cylinder directly into Eq. (7.79). Equation (7.89) is a quadratic equation that can be solved for v in terms of u. Moreover, the proper range of u can be obtained by investigating the characteristics of the discriminant $b^2(u) - 4a(u)c(u)$. The same analysis can be done for the cone/quadric intersection. In this case the vector $\mathbf{V}$ (the cone parametric equation) is given by $[(R/H)v \cos u \quad (R/H)v \sin u \quad v]^T$.

What made it possible to reduce Eq. (7.88) to Eq. (7.89) in the case of a cylinder or a cone is the fact that either surface is a ruled quadric with v being the parameter along the surface rulings. A sphere/quadric or torus/quadric intersection problem cannot be solved directly by following the above approach. A special transformation must be done first, which is not discussed here.

We have mentioned that surface normals are useful redundant data that is usually a part of the internal representations of solid primitives. A surface normal is usually useful in representing the direction of the unit normal vector of a face

with respect to its solid so that the interior and exterior of the solid can be identified unambiguously. One convention is to choose the surface normal to be positive if it points away from its corresponding half-space. A positive surface normal of a cylindrical surface is one which points away from the cylindrical half-space given by Eq. (7.52) or simply points away from the cylinder axis. For a spherical surface, the positive surface normal points away from the center of the sphere. The surface normal can be calculated following the methods discussed in Chap. 6. Consider the cases of a plane, cylinder, and sphere (Fig. 7-19). The positive surface normal of a plane is simply the unit vector, $\hat{\mathbf{k}}_L$, in the Z_L direction. For a cylindrical surface, assuming that the position vector of a point on the surface is $\mathbf{P}$ (with respect to the local coordinate system), then the positive surface normal is given by $[\mathbf{P} - (\mathbf{P} \cdot \hat{\mathbf{k}}_L)\hat{\mathbf{k}}_L]/R$ where $\hat{\mathbf{k}}_L$ is the unit vector in the Z_L direction. For a sphere, it is $\mathbf{P}/R$.

The way the above convention of a surface normal is used to represent the direction of the unit normal of a face of a solid can be explained as follows. In some cases, the positive surface normal may point away from the solid interior or point into the interior, depending on the face position relative to its solid. In other cases, the negative surface normal may exhibit similar behavior. One can assign a variable to the face that may be 1 or -1. This variable always defines the outward pointing (away from the solid interiors) normal of the face of the solid. If the positive surface normal of the face underlying surface happens to point outward from the solid, then the variable is assigned the value 1. Otherwise, it is assigned the value -1 which indicates that the positive surface normal points inward into the solid. The value of the variable is predefined for each primitive face of all the primitive solids a solid modeler supports.

7.8.2 Building Operations

The main building operations in CSG schemes are achieved via the set operators or more specifically the regularized operators: union ($\cup^*$), intersection ($\cap^*$), and difference ($-^*$). Set operators are also known as boolean operators due to the close correspondence between the two. Union, intersection, and difference are equivalent to OR, AND, and NOT AND respectively. Due to the deep roots of CSG schemes in set theory or boolean algebra, CSG models are usually referred to as set-theoretic, boolean, or combinatorial models. Set-operation algorithms are amongst the most fundamental and delicate software components of solid modelers.

Unlike Euler operators, regularized set operators are not based on a given law or equation, but they derive their properties from the set theory and the concept of closure. They are considered higher-level operators than Euler operators. The validity checks for set operators are usually simple in the case of bounded primitives. They take the form of checking the user input of each primitive parameter. This is due to the fact that if primitives are valid and set operators are regularized, then the topology of the resulting solid is always valid.

Some solid modelers provide their users with other building operators that are less formal than set operators. ASSEMBLE and GLUE are two popular ones. Both of them operate on full solids and usually do not combine the solids.

They are merely assembled or glued. They only allow the user to refer to two solids as a single (usually) named entity. The two solids must be positioned properly first.

Almost all contemporary solid modelers provide their users with boolean operations. Boolean operators are mainly used to define solid models through proper combinations of solid primitives. Other important uses include modeling and simulation of manufacturing processes such as drilling and milling as well as detecting spatial interferences and collisions of positioned solid objects in space. In general, engineering processes that involve volumetric and spatial relationships are amenable to using boolean operations.

While boolean operators seem the same to users on all solid modelers, a set-operation algorithm depends primarily and solely on the solid representation scheme supported by each modeler. Example 7.7 in the previous section shows how set operations are performed for the B-rep scheme. In this section we see how they are performed for a CSG scheme. Regardless of the scheme the set operations depend on, an algorithm implementing them must evaluate the boundary of the resulting solid from a desired operation. For discussion purposes, let us write a typical set operation as $A\langle \mathrm{OP}\rangle B$ where A and B are operands (primitives) and $\langle \mathrm{OP}\rangle$ is any regularized set operator. The central question to implementing set operations can be asked as follows. Given the representations of two operands A and B of a given operator, evaluate the boundary of the resulting solid. The representations of A and B must contain representations of their boundaries and this is where the difference between schemes comes into play. In B-rep, boundaries are faces, edges, and vertices that are all stored explicitly. In CSG, boundaries are primitive faces and edges (no vertices) of primitives that are not stored explicitly and must be computed from their underlying surfaces/half-spaces.

Set operations are performed by so-called boundary merging in B-rep while in CSG they are performed by so-called boundary evaluation. In CSG, nonincremental and incremental boundary evaluations are available. In the nonincremental evaluator, only the boundary of the final solid S is evaluated and not for subsolids of S. In the incremental evaluator, the boundaries of the intermediate subsolids are evaluated as the CSG tree is traversed to produce the boundary for the final solid. This latter evaluator is actually a boundary-merging procedure similar to what is used by B-rep schemes. Regardless of which evaluator is used in a modeler, performing set operations in CSG is equivalent to converting the CSG representation of a solid into its B-rep. The incremental boundary evaluator is more widely used than the nonincremental evaluator due to its computational efficiency and its speed in editing, displaying, and/or sectioning solids. GMSOLID and PADL-2 use incremental evaluators to perform set operations while PAD-1 uses a nonincremental boundary evaluator.

Set-operation algorithms based on a CSG scheme are quite similar in philosophy to those based on a B-rep scheme and discussed in Example 7.7. Both are based on edge/face intersection and classification. The differences arise mainly due to the amount and form of data stored by each scheme in its related data structure which, in turn, influences the classification process and how computations are organized. In Example 7.7, we have assumed that a classification algorithm exists to classify edges against a solid. A line of thinking for such an algorithm can follow the one described for the line/polygon classification problem covered in Sec. 7.5.3. While the line of thinking for the CSG representation of the same problem can be utilized in this section, we choose to investigate classification algorithms in more detail for better understanding.

To fully implement CSG-based set-operation algorithms, the required tools are an edge/solid intersection algorithm (not covered here), a classification algorithm to compute the set membership classification function $M[X, S]$ discussed in Sec. 7.5.3, and an algorithm to combine the resulting classifications. A classification algorithm can be based on the divide-and-conquer paradigm and combining classifications need the introduction of the concept of neighborhoods.

The divide-and-conquer paradigm is very similar in concept to the ray-casting algorithm (see Chap. 10). The main difference between the two is that the divide-and-conquer paradigm replaces the rays that are used by the ray-casting algorithm by the edges of a given solid. The line/polygon problem covered in Sec. 7.5.3 can be considered as a simple application of the paradigm. The paradigm is based on the fact that edge/solid classification is identically equivalent to classifying the edge against the left and right subtrees of the solid and then combining the two classifications using the same set operation that operates on the solid subtrees. If one of the subtrees is all primitives, then the edge is classified against all the primitives of the subtree. The paradigm, of course, requires procedures to classify edges with respect to primitives and to combine classifications (refer to Sec. 7.5.3 on how these procedures can be developed). The paradigm can also be extended to classify points and faces against a given solid.

The concept of neighborhoods is introduced to resolve the on/on ambiguities that result when combining "on" segments in a given classification. They are also used in converting a CSG representation into a B-rep. They are mainly useful with the divide-and-conquer paradigm to classify candidate sets and to combine classifications. Figure 7-18 does not show any "on" segments and also shows that combining "in" segments usually results in "in" segments. Figure 7-50 shows a case where a solid $S = A \cup B$. After classifying the edges of A and B against each other, combining the "on" segments of each primitive may result in "in" or "on" segments of S. The on/on ambiguities usually result when the two subsolids or primitives to be combined are tangent to each other along one or more faces. These ambiguities can be resolved using neighborhood information of any point on the "on" segments. The neighborhood of a point P with respect to a solid S, denoted by $N(P, S)$, is the regularized intersection of a sphere with radius R centered at P with the solid. The value of the radius R is arbitrary and should be chosen sufficiently small. We can generalize the set membership classification function $M[X, S]$ given by Eq. (7.49) to include neighborhood information by assuming that the candidate set X has such information and therefore the resulting segment "X on S" has it.

The representation of the neighborhood of a point is related to its position relative to the solid under investigation. Neighborhoods for points that are in the interior or outside of the solid are full or empty and can be represented easily. One alternative is to assign a variable, say N, to the neighborhood of the point such that it can take the values -1, 0, and 1 if the point is outside, on, or inside

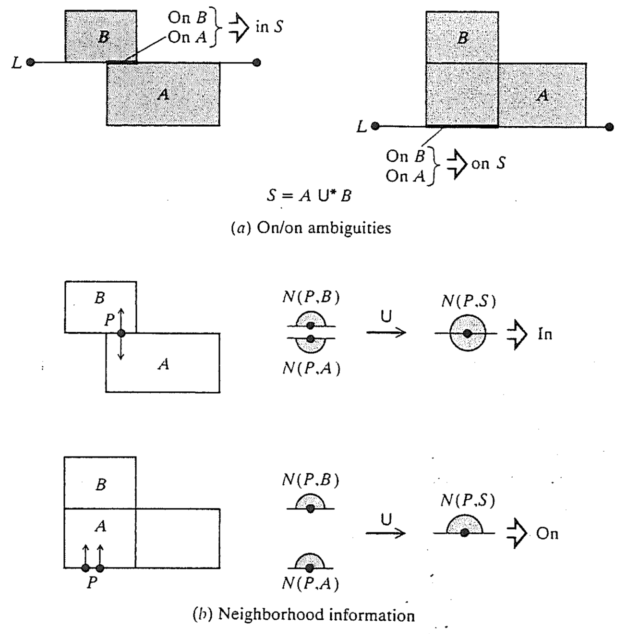


FIGURE 7-50
On/on ambiguities and their resolutions using neighborhood information.

the solid respectively. If $N = -1$ or 1, no further information is needed. For points on the boundaries of the solid ($N = 0$), three cases can arise. A point may be in the interior of a solid's face. This becomes a case of the face neighborhoods, which can be represented using the face and surface normal signs described in Sec. 7.8.1. The second case is edge neighborhoods which result if the point lies on a solid's edge. Assuming that the edge is shared by two faces, the normal and tangent signs of the faces and their underlying surfaces serve to represent the neighborhood. The third case arises when the point is a vertex, thus resulting in vertex neighborhoods. A vertex is typically shared by three faces (or surfaces) and its neighborhood is complex and difficult to manipulate and is not needed in most algorithms. Figure 7-51 shows face and edge neighborhoods.

With all the tools in hand, we can now develop a CSG-based set-operation algorithm. The essence of an algorithm that can perform $S = A\langle \text{OP}\rangle B$ is to classify faces with respect to S by the divide-and-conquer paradigm using face/solid classification, and then combine the classifications using $\langle \text{OP}\rangle$ to obtain the solid S. The resulting classifications produce portions of the faces of A and B that are on S only; that is, that yields the boundary of S. The faces of A and B form a sufficient set of faces which include the boundary of S and which can be used in the paradigm. This is based on the fact $\text{b}(A\langle \text{OP}\rangle B) \subset (\text{b}A \cup \text{b}B)$ where b means boundary. This fact is intuitively acceptable and can be proven mathematically. The faces of A and B are called tentative faces or t-faces.

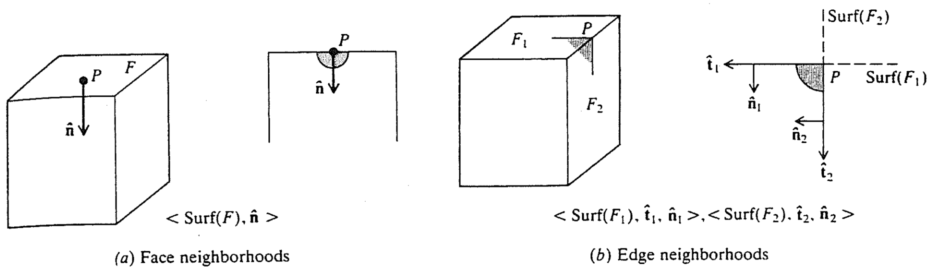


FIGURE 7-51
Face and edge neighborhood representations.

While face/solid classification is possible in theory using the divide-and-conquer paradigm, it is complex and not attractive because it must perform boolean operations on face subsets. Therefore, it is replaced by edge/solid classification which is much simpler. Thus a face classification is done indirectly by classifying its edges with respect to S. This, in turn, requires a set of tentative edges (t-edges) which if classified and combined result in the edges, and consequently the faces, of S. The two types of t-edges that may exist are self-edges, which are the edges of A and B themselves, and cross-edges, which result from intersections between faces of A and B.

One can now devise a set-operation algorithm for a CSG scheme by using the divide-and-conquer paradigm which is based on edge/solid classification. The detailed steps of the algorithm can be envisaged as follows:

1. Generate a sufficient set of t-faces. If A and B are primitives, their primitive faces form such a set.
2. Classify self-edges of A with respect to A including neighborhoods. This is a trivial step because such classifications are already known. This step merely prepares information needed in step 4 below.
3. Classify self-edges of A with respect to B using the divide-and-conquer paradigm. The classification includes neighborhoods as well. If A or B is not a primitive, the paradigm becomes recursive. It is usually based on edge/primitive classifiers which are easy to write for primitives such as blocks, cylinders, and the like.
4. Combine classification results (via a "combine" algorithm) of steps 2 and 3 according to the desired boolean operation. Refer to Sec. 7.5.3 and Fig. 7-18 for an example on how to combine classifications. In following Fig. 7-18, the line L is replaced by a given self(or cross)-edge. The result that is of interest to the set-operation algorithm is the segments that are on the boundary of S (call them "on" segments). Thus, for any edge E that is already classified via the divide-and-conquer paradigm, the combining classification is:

$\langle \text{OP}\rangle = \cup^*$: E on $S = (E$ out A INT* E on $B)$ UN* $(E$ on A INT* E out $B)$

$\langle \text{OP}\rangle = \cap^*$: E on $S = (E$ in A INT* E on $B)$ UN* $(E$ on A INT* E in $B)$

$\langle \text{OP}\rangle = -^*$: E on $S = (E$ in A INT* E on $B)$ UN* $(E$ on A INT* E out $B)$

The UN* and INT* operators are regularized union and intersection operators in one-dimensional space. They are not the ∪* and ∩* we are developing. They combine the classification results by simply comparing and merging the endpoints of different segments. Edges are usually expressed in parametric form and thus UN* and INT* find the appropriate parametric ranges of the result. Thus UN* and INT* are based on scanning parametric intervals and are easy to write.

The above combining rules are valid only if there are no on/on ambiguities. If there are, they should be resolved using neighborhoods and added accordingly to the results obtained from the above rules. This step gives the classification of self-edges of A with respect to solid S.

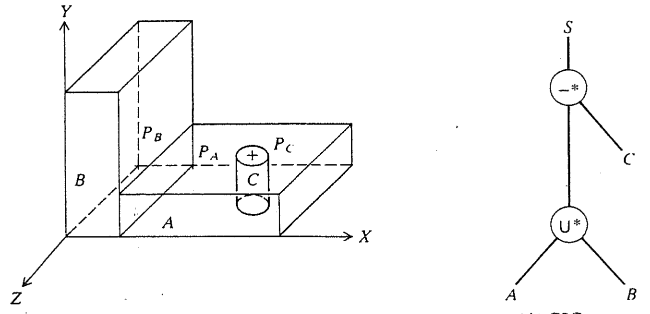
5. Regularize the "on" segments that result from step 4 by discarding the segments that belong to only one face of S. This is done through testing neighborhoods of the segments.
6. Store the final "on" segments that result from step 5 as part of the boundary of S. Steps 2 to 6 are performed for each t-edge of a given t-face of A.
7. Utilize surface/surface intersection to find cross-edges that result from intersecting faces of B (one face at a time) with the same t-face mentioned in step 6. This step results in "oversized" cross-edges which are reduced to "minimal" cross-edges by using step 8 below. Refer to Example 7.9 for more details.
8. Classify each cross-edge with respect to S by repeating steps 2 to 4 with the replacement of self-edges of A used in these steps by each cross-edge. Here, cross-edges are classified with respect to the faces of A and B they belong to. This is a two-dimensional classification and can be combined using the rule E on $S = E$ in A INT* E in B (see Example 7.9).
9. Repeat steps 5 and 6 for each cross-edge.
10. Repeat steps 2 to 9 for each t-face of A.
11. Repeat steps 2 to 6 for each t-face of B.

The above set-operation algorithm is not very efficient. For example, each self-edge is classified at least twice (each edge belongs to two t-faces). This can be easily avoided. Other shortcuts (such as spatial locality in geometric computations) can be used to avoid unnecessary calculations.

Example 7.8. Create the CSG model of the solid S shown in Fig. 7-20*a*.

Solution. The creation of a CSG model of S is by and large much simpler to create using solid primitives and boolean operators than using faces, edges, and vertices and Euler operators utilized in Example 7.5. That is not to say that creating a B-rep model of the same complexity is inefficient because the steps covered in this example can be used in B-rep if it supports boolean operations.

The first step in the creation procedure is the planning strategy. This model is simple and can be created by adding two blocks and subtracting one cylinder. The primitives and the CSG tree are shown in Fig. 7-52. Utilizing the local coordinate systems shown in Fig. 7-4, blocks A and B must be translated and cylinder C translated and rotated relative to the MCS shown to be positioned and oriented properly. Points P_A, P_B, and P_C show the origins of the local coordinate systems of these



(*a*) Decomposing S into primitives (*b*) CSG tree

FIGURE 7-52
CSG model of solid S.

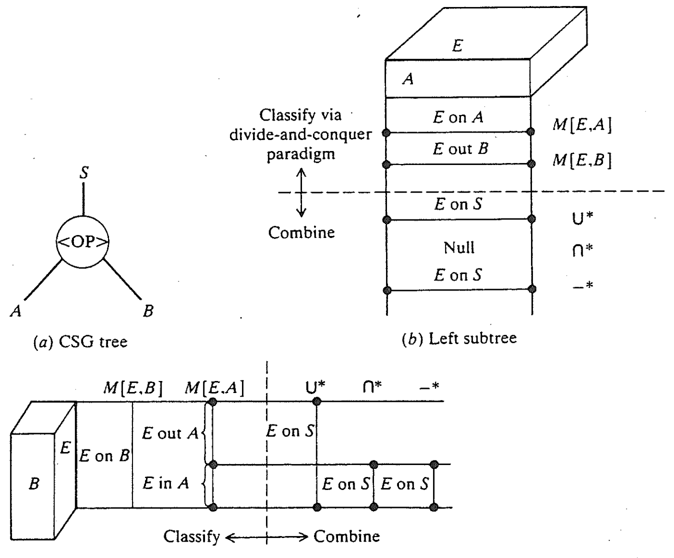
primitives. The geometrical information of each of these primitives is:

Block A: $x_L = a - d$, $y_L = d$, $z_L = c$, $P_A(x, y, z) = P_A(d, 0, -c)$

Block B: $x_L = d$, $y_L = b$, $z_L = c$, $P_B(x, y, z) = P_B(0, 0, -c)$

Cylinder C: $R = R, H = d, P_C(x, y, z) = P_C[(a + d)/2, d, -c/2]$, Rot about $X = 90°$

Assuming the user has created the primitives A, B, and C, the command $S = A ∪^* B -^* C$ creates the CSG tree shown in Fig. 7-52*b*.



(*a*) CSG tree (*b*) Left subtree (*c*) Right subtree

FIGURE 7-53
Self-edge classification and combination.

Example 7.9. Apply the CSG-based set-operation algorithm discussed above to perform set operations on the two primitive blocks shown in Fig. 7-36*a*.

Solution. This example clarifies to a great extent the eleven-step set algorithm described above. The example is intended to particularly show how classifications and their combinations are performed for both self-edges and cross-edges. The CSG tree of the operation $S = A\langle\text{OP}\rangle B$ is shown in Fig. 7-53*a*. The left and right subtrees are block primitives in this example to enhance understanding of the algorithm.

Step 1 of the algorithm is clear here. The sufficient set of t-faces has the 12 faces of A and B. The set of t-edges has 24 edges. To perform steps 2 to 6 and 10 and 11, let us deal with one edge on each A and B. The other edges can be handled in a similar way. Figure 7-53*b* and *c* shows the results. Classifying each edge with respect to its primitives (step 2) is trivial and produces "on" segments. Classifying the same edge with respect to the other primitive (step 3) requires an edge/primitive classifier. The classification results are combined according to the rules of step 4. This procedure is repeated for the other 23 edges.

Unlike self-edges, cross-edges must be found first by using step 7 as shown in Fig. 7-54. Consider faces F_1 of A and F_2 of B. Their underlying surfaces are S_1 and

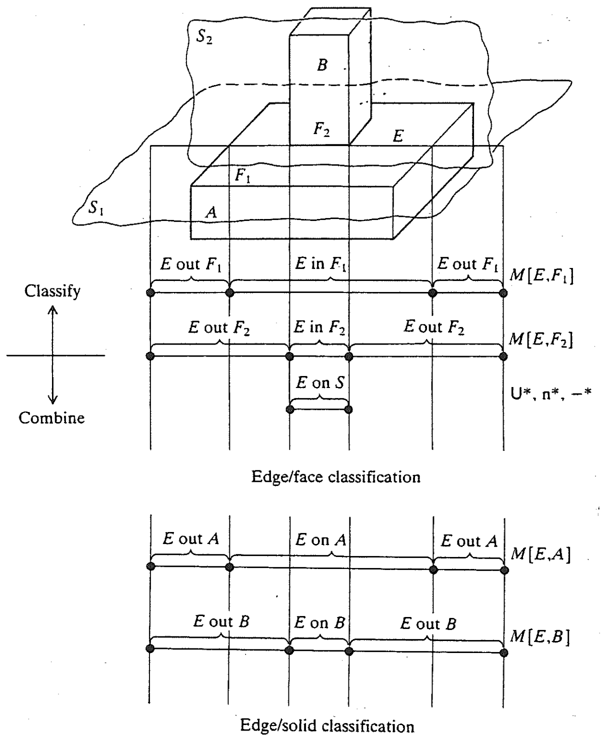


FIGURE 7-54
Cross-edge classification and combination.

S_2 respectively. Intersecting S_1 and S_2 is easier than interesting F_1 and F_2 but produces an "oversized" edge that must be reduced down to the edge between F_1 and F_2; that is, the "minimal" edge. This is achieved via an edge/face classifier (step 8). The "minimal" edge is the "E on S" segment shown in Fig. 7-54. This procedure (steps 7 and 8) is repeated for the other three cross-edges in this case (step 9).

The edge/face classification used in this example assumes that the B-rep of A and B is available in addition to their CSG. This assumption is acceptable for CSG modelers that use incremental boundary evaluations. Modelers that use nonincremental boundary evaluations can replace this classification by an edge/solid classification utilizing the divide-and-conquer paradigm. In this case, the classifier must eliminate the "on" segments, of the cross-edge, that belong to only one face, as mentioned in step 6 (see Fig. 7-54). This is normally done with the aid of neighborhoods and is not shown in Fig. 7-54 or discussed here because it requires new definitions of faces (maximum faces or m-faces) that may be confusing to the reader.

The reader is encouraged to apply the set-operation algorithms to other problems. However, the reader is advised that the algorithm needs refinements and details regarding neighborhoods to be universal.

The classification ideas used in this example and its related algorithm can be applied to algorithms based on B-rep (see Example 7.7). The major change to be done is to replace the divide-and-conquer paradigm that utilizes CSG tree structure by an algorithm that is based on a face/edge/vertex data structure.

7.8.3 Remarks

The CSG scheme is a very powerful representation scheme. It is not closely related to conventional drafting language and has many advantages. It is easy to construct out of primitives and boolean operations. It is concise and requires minimum storage to store solid definitions (the CSG graph). This is why it is slow to retrieve the model because it has to build a boundary from the CSG graph. It is also due to this fact that CSG is slow in generating wireframes, that is, line drawings. CSG must be converted internally into a B-rep (similar to the set-operation algorithm covered earlier) to display the model or generate its line drawings.

Application algorithms based on CSG schemes are very reliable and competitive with those based on B-rep schemes. However, the major disadvantage of CSG is in its inability to represent sculptured surfaces and half-spaces. This is an active area of research, and one would expect this limitation to go away with time.

7.9 SWEEP REPRESENTATION

Schemes based on sweep representation are useful in creating solid models of two-and-a-half-dimensional objects. The class of two-and-a-half-dimensional objects includes both solids of uniform thickness in a given direction and axisymmetric solids. The former are known as extruded solids and are created via linear or translational sweep; the latter are solids of revolution which can be created via rotational sweep. Sweeping is used in general as a means of entering object descriptions into B-rep or CSG-based modelers. There exists no sweeping-based modelers due to the limited modeling domain of sweep representations and the lack of a formal underlying theory of sweeping. For example, general validity and regularization conditions for sweep representations are not known and are usually left to the user.

Sweeping is based on the notion of moving a point, curve, or a surface along a given path. There are three types of sweep: linear, nonlinear, and hybrid sweeps. In linear sweep, the path is a linear or circular vector described by a linear, most often parametric, equation while in nonlinear sweep, the path is a curve described by a higher-order equation (quadratic, cubic, or higher). Hybrid sweep combines linear and/or nonlinear sweep via set operations and is, therefore, a means of increasing the modeling domain of sweep representations.

Linear sweep can be divided further into translational and rotational sweep. In translational sweep, a planar two-dimensional point set described by its boundary (or contour) can be moved a given distance in space in a perpendicular direction (called the directrix) to the plane of the set (see Fig. 7.55*a*). This is similar to entity projection and surface offsetting or translation in wireframe and surface representations respectively. The boundary of the point set must be closed otherwise invalid solids (open sets) result. In rotational sweep, the planar two-dimensional point set is rotated about an axis of rotation (axis of symmetry of the object to be created) by a given angle (see Fig. 7-55*a*). This is similar to entity rotation or a surface of revolution in wireframe and surface representations. Nonlinear sweep is similar to linear sweep but with the directrix being a curve instead of a vector (Fig. 7-55*b*). Hybrid sweep tends to utilize some form of set operations. Figure 7-55*c* shows the same object shown in Fig. 7-55*a* but with a hole. In this case two point sets are swept in two different directions and the two resulting swept volumes are glued together to form the final object. Invalid solids or nonregular sets may result if the sweeping direction is not chosen properly, as shown in Fig. 7-55*d*.

Sweeping operations are useful in engineering applications that involve swept volumes in space. Two widely known applications are simulations of material removal due to machining operations and interference detection of moving objects in space. In the first application, the volume swept by a moving cutter along a specific direction is intersected with the raw stock of the part. The intersection volume represents the material removed from the part. In interference detection, a moving object collides with a fixed one if the swept volume due to the motion of the first intersects the fixed object.

7.9.1 Basic Elements

Wireframe curves, both analytic and synthetic, covered in Chap. 5 are valid basic elements, or primitives, to create two-dimensional contours for sweep operations. However, a solid modeler may not allow its users to use all wireframe entities it supports in sweep operations. Such a limitation usually stems from the modeling domain of the internal representation the modeler supports. It is this representation that user sweep operations are converted to before being stored in the data-

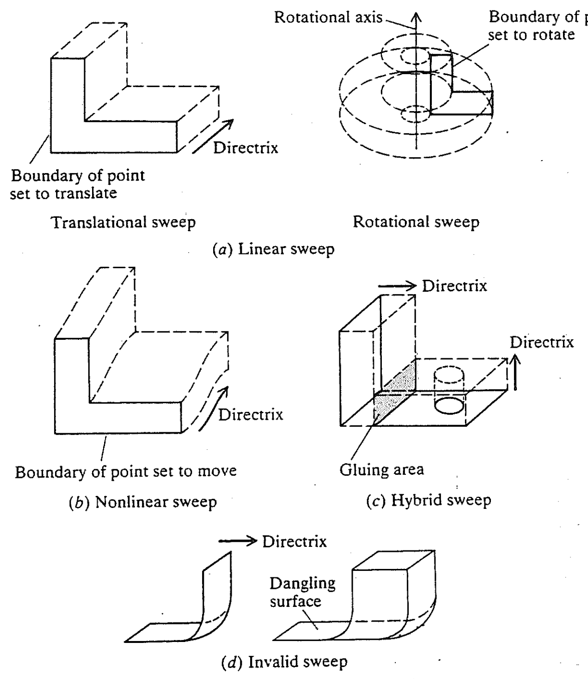


FIGURE 7-55
Types of sweep.

base of the model under construction. If the sweep operations are converted to a B-rep, the permissible wireframe entities are those that are the same as the underlying curves of edges that the B-rep supports. If they are converted to a CSG, the permissible entities are the ones that generate surfaces (boundaries) of supported half-spaces. Lines, arcs, circles, and B-splines are among the most widely used entities in sweep operations.

The boundary of a two-dimensional point set used in sweep operations can consist of nested contours up to one level only (one inner contour) within the outer contour. This is allowed to create holes in the resulting solid. There may also exist a maximum number of entities allowed to create any one contour. The number is usually adequate for practical uses. It is usually set for implementation purposes of the sweep algorithm.

7.9.2 Building Operations

The building operations of linear and nonlinear sweep models are simple: generate the boundary and sweep it. If hybrid sweep is available, these operations extend to include boolean operations. If there existed data structures designed only for sweep representation, algorithms to implement these boolean operations would have been different from those already discussed for B-rep and CSG. Practically, this is not the case and the sweep operations are used only as a user convenience and a boolean operation acts on the corresponding B-rep or the CSG data structure of the sweep operations.

How are sweep operations converted to a B-rep or CSG? It is well known that only linear sweep can be converted. In the case of a translational sweep, each entity in the swept boundary represents an edge and each corner point represents a vertex in the corresponding boundary model. For a one-contour boundary, each entity in the boundary indicates a face also. The number of faces of the model are equal to $(N + 2)$, where N is the number of entities of the boundary and the 2 accounts for the front and back faces (see Fig. 7-55*a*). The number of edges is equal to $(2N + M)$, where M is the number of corner points of the boundary and the number of vertices is equal to $2M$. These values satisfy the Euler equation (7.57), regardless of the number of entities of the boundary, if we notice that $N = M$ for a closed boundary. If the boundary has holes (nested contours), a similar relationship can be obtained. Rotational sweep operations can be converted in a similar fashion. In Sec. 7.7, we have discussed how a rotational sweep operator (see Fig. 7-32) based on approximate B-rep can be developed. The reverse of that discussion shows how a sweep model can be converted into an approximate B-rep. It is left to the reader to find the number of faces, edges, vertices, loops, bodies, and genus when the sweeping angle is a full 360° or a given range of it.

The linear sweep to CSG conversion must be based on unbounded CSG primitives, that is, half-spaces. Conversion to CSG based on bounded primitives might not be possible all the time and may be no better than the half-space alternative. In this case, each entity in the swept boundary represents a bounding surface of a corresponding half-space. A linear entity, for example, represents a planar surface and half-space and a circular entity or an arc represents a cylindrical surface and half-space. The CSG model is then composed of the union of these intersecting half-spaces.

In both conversions, the underlying surfaces must be oriented. This requires the direction of a surface normal. This can be achieved by choosing a direction (clockwise or counterclockwise) when inputting or creating the outer and inner contours of the given boundary. The interior of the boundary, which defines the interior of the solid, can be identified and the proper normal sign can be chosen accordingly. For example, one may choose to traverse the outer contour in a counterclockwise direction and then choose the positive normal sign to indicate the exterior of the solid. A vector that represents such a surface normal becomes very easy to create.

Example 7.10. Create the sweep model of the solid S shown in Fig. 7-20*a*.

Solution. The solid S is two-and-a-half dimensional, having uniform thickness in the negative Z direction. If it was not the cylindrical hole, the model could have been created via translational sweep exactly as shown in Fig. 7-55*a*. Due to the hole presence, hybrid sweep is utilized as shown in Fig. 7-55*c*. The bottom face with the circular hole is created and swept vertically and the left face is created and swept to the right. The two subsolids are then glued along the hatched gluing area shown.

7.9.3 Remarks

Sweep representation is useful once it develops. Its modeling domain can be extended beyond two-and-a-half-dimensional objects if nonlinear (sometimes called general) sweep is available. Nonlinear sweep may be useful in creating nonrigid objects and studying their deformation as they travel in space. Complex mechanical parts such as screws, springs, and other components that require helical and special loci can be represented by sweeping. In any one of these parts, a two-dimensional polygon can form the basis of the desired boundary.

7.10 ANALYTICAL SOLID MODELING (ASM)

The historical development of ASM is closely related to finite element modeling. Those who are familiar with finite element analysis (FEA) can easily recognize that the mathematical foundations of ASM follow similar guidelines to three-dimensional isoparametric formulation of FEA for 8- to 20-node hexahedral elements. ASM is developed to aid designers and engineers in the arduous task of modeling complex geometry commonly found in design applications. ASM can be viewed as more of a representation scheme for design than for manufacturing purposes due to its formulation, as seen in this section, which does not involve orientable surfaces as does B-rep or CSG.

While ASM originated from the need to solve the problem of finite element modeling, it has now a wide range of applications such as mass property calculations, composite material modeling, and computer animation. The widespread acceptance of ASM in the finite element and finite difference communities has been due to the efficiency and flexibility of mesh-generation algorithms that operate on hyperpatches (see the next section). A uniform transition, or nonuniform mesh, can be generated within a hyperpatch and, consequently, within the entire model. In addition, because a hyperpatch is a mapping of a unit cube, it is easy to subdivide it into hexahedral elements. The solid modeler PATRAN-G is based on ASM and interfaces to various FEA packages and other solid modelers.

7.10.1 Basic Elements

ASM is an extension of the well-established tensor product method, introduced to represent surfaces in Chap. 6, to three-dimensional parametric space with the parameters u, v, and w. Therefore, it involves only products of univariate basis polynomials and introduces no conceptual difficulty due to the higher dimensionality of a solid. The properties of tensor product solids can be easily deduced from properties of the underlying curve schemes. Thus, one can conceptually conceive and easily derive the representations of tricubic, Bezier, and B-spline solids analogous to bicubic, Bezier, and B-spline surfaces in two-dimensional parametric space (u, v) and analogous to cubic, Bezier, and B-spline curves in one-dimensional parametric space (u).

The tensor product formulation in three-dimensional parametric space is a mapping of a cubical parametric domain described by u, v, and w values into a solid described by x, y, and z in the cartesian (modeling) space, as shown in Fig.

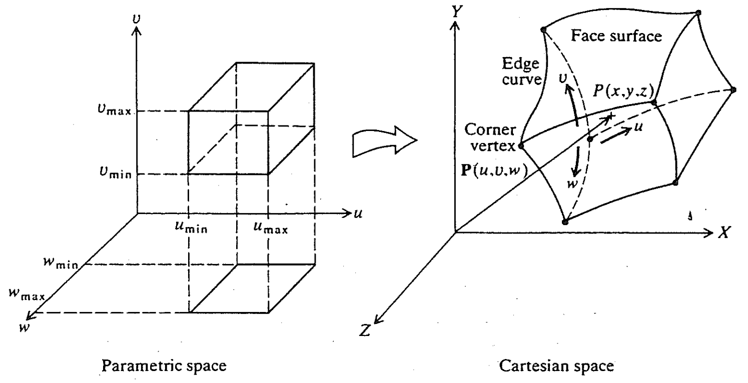


FIGURE 7-56
Hyperpatch representation.

7-56. The resulting solid is called a parametric solid or a hyperpatch (so called because hyperpatches are extensions of and bounded by surface patches) whose points in the interior or on the bounary are given by

$$\mathbf{P}(u, v, w) = [x \quad y \quad z] = [x(u, v, w) \quad y(u, v, w) \quad z(u, v, w)],$$
$$u_{min} \le u \le u_{max},\ v_{min} \le v \le v_{max},\ w_{min} \le w \le w_{max} \qquad (7.90)$$

As with curves and surfaces, Eq. (7.90) gives the coordinates of a point inside or on the hyperpatch as the components of its position vector. The equation uniquely maps the parametric space (E^3 in u, v, and w values) to the cartesian space. The parametric variables are constrained to intervals bounded by minimum and maximum values. In most hyperpatches, these intervals are [0, 1] which result in a unit cube in the parametric space.

Equation (7.90) suggests that ASM represents an object as an assembly of nonoverlapping hyperpatches. Each hyperpatch has six faces, each of which is any surface patch discussed in Chap. 6. Each face has four edge curves of the same type as the surface patches and four corner vertices (see Fig. 7-56). For example, a tricubic hyperpatch is bounded by six bicubic surface patches, each of which is bounded by four cubic splines. This hierarchy in topology, from a hyperpatch to patches, edges, and vertices, provides a means to construct a hyperpatch from given control points, curves, or patches. This feature makes ASM an extension of wireframe or surface modeling. For example, a user can create a Bezier hyperpatch by either entering its control points, Bezier curves, or Bezier patches.

While a hyperpatch can be described by a polynomial of any order, a cubic polynomial in each parameter is sufficient for practical design applications. As mentioned in Chap. 5, the higher the order, the more difficult it is to control the resulting shape. Analogous to Eqs. (5.74) and (6.39), a cubic hyperpatch can be given by the following equation:

$$\mathbf{P}(u, v, w) = \sum_{i=0}^{3} \sum_{j=0}^{3} \sum_{k=0}^{3} \mathbf{C}_{ijk} u^i v^j w^k, \quad 0 \le u \le 1, 0 \le v \le 1, 0 \le w \le 1 \qquad (7.91)$$

The face surfaces, edge curves, and corner vertices can be obtained by substituting the proper value(s) of the parameters into this equation. The face surfaces are given by $\mathbf{P}(0, v, w)$, $\mathbf{P}(1, v, w)$, $\mathbf{P}(u, 0, w)$, $\mathbf{P}(u, 1, w)$, $\mathbf{P}(u, v, 0)$, and $\mathbf{P}(u, v, 1)$. Similarly, the equation of any edge curve is obtained by fixing two of the three parametric variables and keeping the third one free. $\mathbf{P}(u, 0, 0)$, $\mathbf{P}(u, 1, 0)$, and $\mathbf{P}(0, v, 0)$ are three of the available 12 edge curves, and the eight corner vertices are obviously $\mathbf{P}(0, 0, 0)$, $\mathbf{P}(1, 0, 0)$, $\mathbf{P}(0, 1, 0)$, $\mathbf{P}(1, 1, 0)$, $\mathbf{P}(0, 0, 1)$, $\mathbf{P}(1, 1, 0)$, $\mathbf{P}(0, 1, 0)$, and $\mathbf{P}(1, 1, 1)$.

There are 64 $\mathbf{C}_{ijk}$ vector coefficients (called polynomial or algebraic coefficients, as they were called previously for curves and surfaces) that must be determined utilizing a given set of boundary conditions of a given hyperpatch. For a tricubic hyperpatch, these conditions are 8 position vectors (one $\mathbf{P}$ at each corner vertex), 24 tangent vectors [three ($\partial \mathbf{P}/\partial u$, $\partial \mathbf{P}/\partial v$, and $\partial \mathbf{P}/\partial w$) at each corner vertex], 24 twist vectors [three ($\partial^2 \mathbf{P}/\partial u\, \partial v$, $\partial^2 \mathbf{P}/\partial u\, \partial w$, and $\partial^2 \mathbf{P}/\partial v\, \partial w$) at each corner vertex], and 8 triple mixed partial derivatives [one ($\partial^3 \mathbf{P}/\partial u\, \partial v\, \partial w$) at each corner vertex]. For a cubic Bezier hyperpatch, there are 64 given control (data) points that form the characteristic (control) polyhedron of the hyperpatch. These points are arranged in a 4 × 4 × 4 mesh. Each face surface of the hyperpatch uses 16 of these control points to define its control polygon and each edge curve uses four control points to form its control polygon. In the case of a cubic B-spline hyperpatch, an $n \times m \times q$ mesh can be used where the number of control points in the u, v, and w directions are n, m, and q respectively and are not necessarily equal. *B*-spline surfaces and curves at the boundary of the hyperpatch can be deduced in a similar fashion to the Bezier hyperpatch. If a hyperpatch that interpolates the data points is desired, an interpolating polynomial must be used.

The reduction of Eq. (7.91) to a matrix form as we have done to Eqs. (5.74) and (6.39) [see Eqs. (5.77) and (6.40)] is possible but produces an awkward form. In addition, converting Eq. (7.91) (sometimes called the algebraic form) to a form similar to Eqs. (5.83) and (6.42) (sometimes called the geometric form), in the case of a tricubic hyperpatch, requires relating the algebraic coefficients $\mathbf{C}_{ijk}$ to the geometric coefficients, that is, the 64 boundary conditions. This conversion process is not different from the case of curves and surfaces and is very cumbersome for hyperpatches. Therefore, the developments of the geometric forms of Eq. (7.91) for a tricubic cubic Bezier and cubic B-spline hyperpatches are done here by extending (intuitively) the patterns we can identify in one- and two-dimensional parametric spaces into three-dimensional parametric space.

Let us look into developing the geometric form of a tricubic hyperpatch. If Eq. (5.83) is expanded and rearranged by collecting the terms of u^3, u^2, u^1, and u^0, we get the following equations:

$$\mathbf{P}(u) = \sum_{i=0}^{3} \sum_{j=1}^{4} u^i M_{H_{ij}} \mathbf{V}_j, \qquad 0 \le u \le 1 \qquad (7.92)$$

where $M_{H_{ij}}$ are the elements of the geometry matrix $[M_H]$ given by Eq. (5.84) and $\mathbf{V}_j$ are the elements of the vector $\mathbf{V}$ given by Eq. (5.85). It is more desirable to unify the summation limits of both the i and j indices in Eq. (7.92) by writing it as

$$\mathbf{P}(u) = \sum_{i=1}^{4} \sum_{j=1}^{4} u^{i-1} M_{H_{ij}} \mathbf{V}_j, \qquad 0 \le u \le 1 \qquad (7.93)$$

Equation (7.92) can also be written as

$$\mathbf{P}(u) = \sum_{i=0}^{3} \left(\sum_{l=1}^{4} M_{H_{il}} \mathbf{V}_l \right) u^i, \qquad 0 \le u \le 1 \tag{7.94}$$

Comparing Eqs. (7.94) and (5.74) gives

$$\mathbf{C}_i = \sum_{l=1}^{4} M_{H_{il}} \mathbf{V}_l \tag{7.95}$$

and, therefore, Eq. (7.93) becomes

$$\mathbf{P}(u) = \sum_{i=1}^{4} \mathbf{C}_i u^{i-1}, \qquad 0 \le u \le 1 \tag{7.96}$$

Equations (7.93) and (7.96) give the geometric and algebraic terms respectively of a Hermite cubic spline. Equation (7.95) relates the algebraic coefficients to the geometric ones.

Applying the same treatment to a bicubic surface, Eq. (6.42) can be rewritten as

$$\mathbf{P}(u, v) = \sum_{i=1}^{4} \sum_{j=1}^{4} \mathbf{C}_{ij} u^{i-1} v^{j-1}, \qquad 0 \le u \le 1, 0 \le v \le 1 \tag{7.97}$$

where

$$\mathbf{C}_{ij} = \sum_{l=1}^{4} \sum_{m=1}^{4} M_{H_{il}} M_{H_{jm}} \mathbf{b}_{lm} \tag{7.98}$$

and $\mathbf{b}_{lm}$ are the elements of the $[B]$ matrix given by Eq. (6.43).

Comparing Eqs. (7.96) and (7.97) shows that introducing an additional parametric variable amounts to adding an extra summation sign into the geometric form. Moreover, comparing Eqs. (7.95) and (7.98) shows the same effect, as well as the fact that the matrix $[M_H]$ always relates the algebraic coefficients to the geometric ones. On the basis of these observations, one can easily write the equation of a tricubic hyperpatch as

$$\mathbf{P}(u, v, w) = \sum_{i=1}^{4} \sum_{j=1}^{4} \sum_{k=1}^{4} \mathbf{C}_{ijk} u^{i-1} v^{j-1} w^{k-1}, \qquad 0 \le u \le 1, 0 \le v \le 1, 0 \le w \le 1 \tag{7.99}$$

where

$$\mathbf{C}_{ijk} = \sum_{l=1}^{4} \sum_{m=1}^{4} \sum_{n=1}^{4} M_{H_{il}} M_{H_{jm}} M_{H_{kn}} \mathbf{b}_{lmn} \tag{7.100}$$

Notice that Eq. (7.99) is the same as Eq. (7.91) with summation limits changed. The geometric coefficients $\mathbf{b}_{lmn}$ can be arranged in four $[B]$ matrices similar to the one given by Eq. (6.43). The first and second, say $[B_1]$ and $[B_2]$, are exactly like $[B]$ of Eq. (6.43) but for $w = 0$ and $w = 1$ respectively. For example, $\mathbf{b}_{122}$ and $\mathbf{b}_{211}$ are $\mathbf{P}_{110}$ and $\mathbf{P}_{001}$ respectively. The third and the fourth, say $[B_3]$ and $[B_4]$, are the derivatives of $[B_1]$ and $[B_2]$ respectively with respect to w. For example, $\mathbf{b}_{323}$ and $\mathbf{b}_{432}$ are $\mathbf{P}_{uvw000}$ and $\mathbf{P}_{uw011}$ respectively. If we choose the subscript l in Eq. (7.100) to correspond to the four $[B]$ matrices, we can easily

see that the four specific $\mathbf{b}$ elements we just mentioned become $\mathbf{b}_{122}$, $\mathbf{b}_{211}$, $\mathbf{b}_{323}$, and $\mathbf{b}_{432}$ respectively.

The tricubic hyperpatch suffers from all the disadvantages of the cubic curve and the bicubic surface. It even requires the input of $\partial^3\mathbf{P}/\partial u\, \partial v\, \partial w$ as a boundary condition. Therefore, let us look into a cubic Bezier hyperpatch. Following the same approach we used for the tricubic patch, the cubic Bezier curve (see Prob. 5.10), the cubic Bezier surface [see Eq. (6.73)], and the cubic Bezier hyperpatch can also be described by Eqs. (7.96), (7.97), and (7.99) respectively. Equations (7.95), (7.98), and (7.100) then become respectively:

$$\mathbf{C}_i = \sum_{l=1}^{4} M_{B_{il}} \mathbf{P}_l \tag{7.101}$$

$$\mathbf{C}_{ij} = \sum_{l=1}^{4} \sum_{m=1}^{4} M_{B_{il}} M_{B_{jm}} \mathbf{P}_{lm} \tag{7.102}$$

$$\mathbf{C}_{ijk} = \sum_{l=1}^{4} \sum_{m=1}^{4} \sum_{n=1}^{4} M_{B_{il}} M_{B_{jm}} M_{B_{kn}} \mathbf{P}_{lmn} \tag{7.103}$$

where the elements of the matrix $[M_B]$, given by Eq. (6.74), are used in these three equations. The control points $\mathbf{P}_{lmn}$ for the hyperpatch are arranged in the $l \times m \times n$ mesh. The $m \times n$ mesh for the Bezier surface can easily be extended to form the $l \times m \times n$ mesh. Thus any point $\mathbf{P}_{lmn}$ is much easier to locate and understand than the $\mathbf{b}_{lmn}$ coefficients used in Eq. (7.100) for the tricubic hyperpatch.

The cubic B-spline hyperpatch would be more advantageous to use in design applications over the cubic Bezier hyperpatch due to the local control characteristics of the former. The cubic B-spline curve (see Prob. 5.15), the cubic B-spline surface [see Eq. (6.82)], and the cubic B-spline hyperpatch can be described by Eqs. (7.96), (7.97), and (7.99) respectively. The maximum value of any of the parameters may exceed the value of 1. Equations (7.101) to (7.103) can be extended to the cubic B-spline hyperpatch by replacing the matrix $[M_B]$ with the matrix $[M_S]$ given by Eq. (6.81).

7.10.2 Building Operations

The creation of an ASM model of an object simply involves dividing the object into the proper assembly of nonoverlapping hyperpatches. Each hyperpatch can be constructed from curves and/or surface patches. For example, a cubic Bezier or a B-spline hyperpatch can be constructed by creating Bezier or B-spline curves and connecting the curves by surfaces. This process reflects the natural nesting of curves, surfaces, and solids.

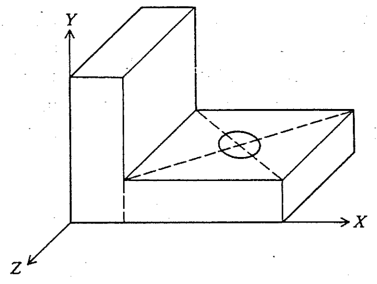
If the ASM model of an existing object is to be created, Bezier and B-spline hyperpatches introduced in Sec. 7.10.1 cannot be used because they extrapolate the given data points. Instead, we can use a B-spline hyperpatch that interpolates the data points or a cubic curve that interpolates through four data points at specified parametric locations can be used. If these points are at $u = 0, \frac{1}{3}, \frac{2}{3}$, and 1, they are sometimes referred to as the one-third points. Similarly 16 points and 64 points would be needed to create a bicubic surface and a tricubic hyperpatch respectively.

Other construction methods of ASM models can include ruled volumes and sweeping. A ruled volume can be created between two given surface patches by linearly interpolating between them as we did in developing ruled surfaces. Linear sweep creates hyperpatches that have uniform thickness normal to the surface patch. It is also possible to create hyperpatches that have thicknesses that vary bilinearly over the surface patch (see Sec. 6.6.6). Rotational and/or nonlinear sweep of surface patches can also create hyperpatches. The equations of the ruled or swept hyperpatches created by these construction methods are considered special cases of the general equations covered in Sec. 7.10.1 and can be obtained from them.

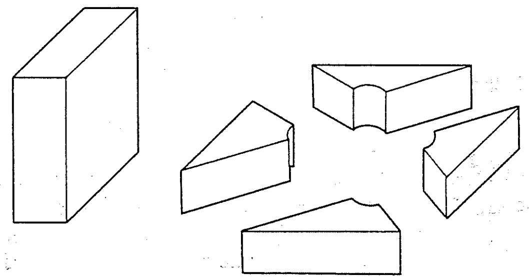
It is desirable, from a user point of view, for ASM schemes to support boolean operations. In such a case hyperpatches can be unioned, intersected, and differenced to model various objects. This would require developing intersecting algorithms of these hyperpatches as well as validity checks to ensure the creation of valid objects.

Example 7.11. Create the ASM model of the solid S shown in Fig. 7-20*a*.

Solution. The ASM model of this solid S consists mainly of hyperpatches that have planar face surfaces. Figure 7-57 shows the minimum number of hyperpatches that can be used to create the ASM model of the solid S.



(*a*) Subdivision of S into hyperpatches



(*b*) Individual hyperpatches

FIGURE 7-57
ASM model of solid S.

7.10.3 Remarks

ASM has a strong history in finite element modeling. The fact that a hyperpatch is given by Eq. (7.99) makes it appealing in design and analysis applications that require information inside as well as on the boundary of a given object. This is desirable, for example, in modeling and studying composite materials and fracture mechanics problems. However, ASM is not adequate for manufacturing applications such as tool path generation because face surfaces of hyperpatches are not explicitly stored and are not orientable; that is, normals to face surfaces (surface patches) cannot indicate the interior or exterior of the object.

7.11 OTHER REPRESENTATIONS

We have covered the four most popular representations (B-rep, CSG, sweep, and ASM) used in solid modeling in Secs. 7.7 to 7.10. Other representations exist. However, they are less popular because their modeling domain is limited and/or they do not support a wide range of applications. These representations are primitive instancing, cell decomposition, spatial occupancy enumeration, and octree encoding.

Primitive instancing is based on the notion of families of objects or family of parts. All objects that have the same topology but different geometry can be grouped into a family called generic primitive. Each individual object within a family is called a primitive instance. Take, for example, a block primitive which can be represented by its length L, width W, and height H. Each block primitive instance is defined by specific values of L, W, and H. Primitive instancing is similar in philosophy to group technology used in manufacturing. It promotes standardization. It is also an unambiguous, unique, and easy to use and validate scheme. However, its main drawbacks are its limited domain of modeling unless we use an enormous number of generic primitives, and the lack of generality to develop any algorithms to compute properties of represented solids.

In a cell decomposition scheme, an object can be represented as the sum of cells into which it can be decomposed. Each cell in the decomposition can always be represented. Thus, cell decomposition may enable us to model objects, which may not otherwise be representable, by their cells. Take the case of a cup with a handle. It can be decomposed into two cells: a body and a handle. The body and/or handle can be decomposed further if needed. Cell decompositions are unambiguous, nonunique, and are computationally expensive to validate. They have been historically used in structural analysis. ASM and finite element modeling are forms of cell decomposition.

In a spatial enumeration scheme, a solid is represented by the sum of spatial cells that it occupies. These cells (sometimes called voxels for "volume elements") are cubes of a fixed size that lie in a fixed spatial grid. Each cell can be represented by its centroid coordinates in the grid. The smaller the size of the cube, the more accurate the scheme in representing curved objects. It is exact for boxlike objects. The scheme is unambiguous, unique, and easy to validate, but it is verbose when describing an object, especially curved ones.

The octree encoding (quadtree encoding in two-dimensions) scheme can be considered a generalization of the spatial enumeration scheme in that the cubes

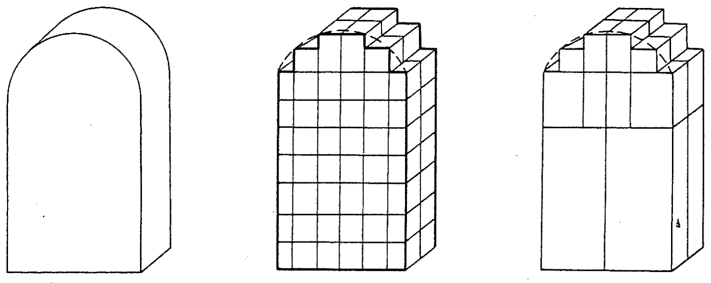


FIGURE 7-58
Quasi-disjoint decomposition of a solid.

may have variable sizes. Octrees are hierarchical structures that reflect the recursive subdivision of objects into variably sized cubes. Figure 7-58 shows the difference between spatial occupancy enumeration and octree encoding. In octree encoding, we enclose the object to be modeled inside a cube. If the object does not uniformly cover the cube, then we subdivide the cube into eight octants. If any of the resulting octants is full (completely inside the object) or empty (completely outside the object), no further subdivision is made. If any of the octants is partially full, we subdivide it again into octants. We continue to subdivide the partially full octants until the resulting octants are either full or empty or until some predetermined level of subdivision is reached. Quadtree encoding is exactly the same as octree encoding but it begins with a square and recursively subdivides it into quadrants. Quadtree and octree encodings were originally developed for use in image representation. They have been adapted to finite element modeling (refer to Chap. 18 for more details).

7.12 ORGANIZATION OF SOLID MODELERS

Each of the solid modeling representations described above has its advantages and disadvantages. Some are more suitable for certain applications than others. This leads to the idea of developing solid modelers with more than one internal representation. While this idea might extend the geometric coverage (modeling domain) of a modeler, the development and maintenance of such a modeler requires more time and effort than a modeler based on only one representation.

The generic architecture of a solid modeler is shown in Fig. 7-59. This architecture applies also to most of the existing CAD/CAM systems. The figure shows that a solid modeler can be divided into four major systems. The input system consists primarily of the user interface and its related commands. Users can input commands (sometimes called a symbol structure) to define a new object or input application commands that invoke an application algorithm such as mass property calculations. The GMS (geometric modeling system) is the heart of the solid modeler. The GMS translates the symbol structure that defines an object into the internal representation the solid modeler supports. The applica-

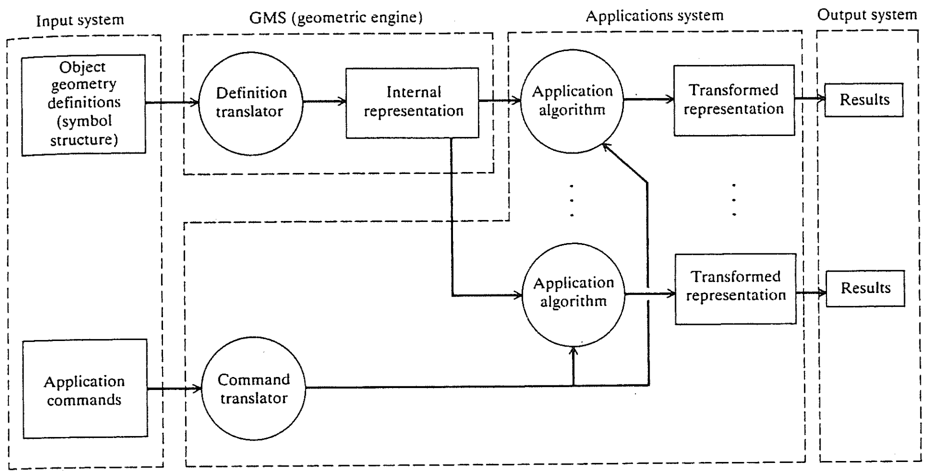


FIGURE 7-59
Architecture of a typical solid modeler.

tions system usually consists of various application algorithms. Each algorithm operates on the modeler's internal representation and transforms it into the proper geometric form (transformed representation) needed by the analysis procedure related to the application. In the case of finite element modeling, the mesh algorithm produces nodal (grid) points and elements. In general, the GMS is fixed once it is developed while the applications system is extending to accommodate new design and manufacturing applications. The engineering value of a solid modeler is usually assessed by the capabilities of its applications system. The output system displays the results in a graphical form. Its details have been discussed in Chapter 2. Typically, a solid modeler is a part of a total CAD/CAM system. In such a case, the architecture shown in Fig. 7-59 is an extension of that of the CAD/CAM system.

Solid modelers can be categorized into three types based on their GMSs as follows:

1. Single representation modelers. These modelers have only one internal representation they store. B-rep is usually such a representation. All modelers based on B-rep fall into this type. These modelers usually support CSG-like input and sweep operations to facilitate user input. These forms of input are not stored internally but are converted into the B-rep format before storing. Figure 7-60*a* shows the GMS of a single representation system.
2. Dual representation modelers. This type is very popular among modelers whose primary representation scheme is CSG (e.g., PADL-2). A modeler has both B-rep and CSG representations. However, B-rep is derived internally by the modeler from the CSG and the user has no control over it. In addition, the modeler does not usually store the B-rep. It only stores the CSG graph (and tree) and can always reevaluate the B-rep. The need for converting the CSG

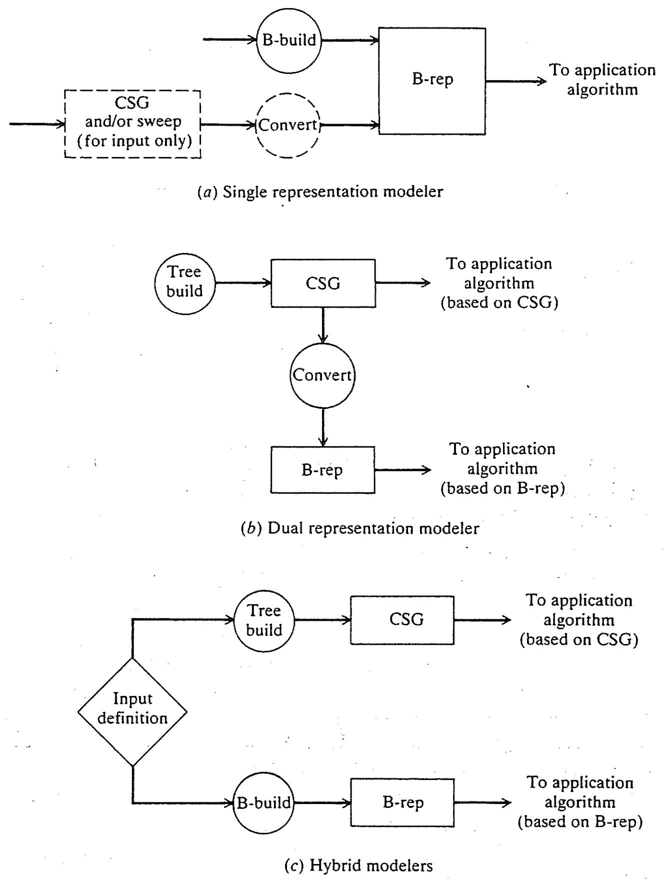

B-build
B-rep
To application algorithm
CSG and/or sweep (for input only)
Convert
(a) Single representation modeler
Tree build
CSG
To application algorithm (based on CSG)
Convert
B-rep
To application algorithm (based on B-rep)
(b) Dual representation modeler
Tree build
CSG
To application algorithm (based on CSG)
Input definition
B-build
B-rep
To application algorithm (based on B-rep)
(c) Hybrid modelers

FIGURE 7-60
Types of solid modelers.

into B-rep is natural and is needed for display and other graphics purposes. Dual-rep modelers can support a wide range of applications. However, their developments and maintenance are usually more complex than single-rep modelers. Figure 7-60*b* shows the structure of the related GMS.

3. Hybrid modelers. Here the modeler uses two independent internal representations, usually B-rep and CSG (see Fig. 7-60*c*). GMSOLID is a true hybrid modeler. Hybrid modelers are different from dual-rep modelers in a few ways. Neither of the internal representations supported is derived from the other. The user can choose which representation and input form to use. Furthermore, the user can use both representations to solve a given problem and compare them. In such cases, it is the user's responsibility to ensure the consistency between input data. The need for a hybrid modeler is clearly to increase

the modeling domain of the modeler. For example, sculptured surfaces can be modeled on a hybrid modeler using its B-rep while this cannot be done on a dual-rep system if it does not support sculptured surfaces.

7.13 SOLID MANIPULATIONS

Solid manipulations are useful during the design phase of a given part or object. It is beneficial if the manipulation concepts (refer to Chaps. 5 and 6) utilized by the wireframe and surface modeling techniques can be extended to solid modeling. Most of these concepts can be foreseen as solving intersection problems and/or using set membership classification. In addition, any solid manipulation involves manipulation of both its geometry (as in curves and surfaces) and its topology to ensure that the resulting solids are valid.

7.13.1 Displaying

Displaying a solid can take two forms: wire display and shaded images. The wire display requires the B-rep of the solid, in which case the metric information of edges and vertices are used to generate the wireframe model of the solid. This wireframe model can be displayed, edited, or produce line drawings. Editing of the wireframe model derived from a solid database does not, of course, affect the solid itself. If the underlying surfaces of the solid faces are utilized, a mesh can be added to the display to help visualization.

Displaying solids as shaded images provides realistic visual feedbacks to users of solid modelers. Shading is perhaps considered the oldest and most popular application of solid modeling. In fact, there exists the mistaken notion that a shaded image is a solid model. As one expects, shading algorithms are directly related to the representation schemes of solids. Shaded images can be generated from B-reps by a variety of visible-surface algorithms. Most often, these exact B-reps are converted to faceted (approximate or polygonal) B-reps because such algorithms become simple for the latter. Special-purpose tiling engines based on algorithms for displaying approximate B-reps are available commercially for faster displays.

Shading can be performed directly from CSG by means of ray-casting (also called ray-tracing) algorithms or depth-buffer (also called z-buffer) algorithms. Many improvements and alterations have been introduced to ray casting to speed up the algorithms. Shading algorithms for both spatial enumeration and octree have been implemented in special-purpose hardware: a voxel machine for the former and an octree machine for the latter. Visible-surface algorithms can be utilized to shade ASM models. Details of some of these shading algorithms are covered in Chap. 10.

7.13.2 Evaluating Points, Curves, and Surfaces on Solids

Applications that require information about the boundary of a solid would need to evaluate points, curves, and surfaces on this boundary. Take the popular application of generating tool paths from solid modeling databases. To drive a tool along the solid boundary, proper curves are evaluated on the underlying surfaces of the solid faces. Points are then generated on each curve to generate the required tool locations.

Evaluating points and curves on solids can be viewed as intersection problems. Solutions of curve/solid and surface/solid intersection problems generate points and curves on solids respectively. Curves and surfaces, utilized in the intersection problems, must be of the type that the solid modeler supports. More specifically, it suffices for these intersecting curves and surfaces to be lines and boundaries of planar half-spaces (i.e., planes) respectively. A plane/solid intersection is also useful in sectioning a solid to generate desired cross-sections. An algorithm that evaluates points and curves on a solid is dependent on the representation of the solid, but in the most part follows similar outlines as described in Examples 7.7 and 7.9.

Evaluating surfaces on a solid can be regarded as extracting the underlying surfaces of the solid faces. These surfaces can be bounded by the proper solid edges or other user-defined boundaries. To keep the solid topology and geometry intact, the geometry and other related information of these surfaces must be copied. The parametric equations of the surfaces might have to be stored in the given solid modeler to facilitate editing the extracted surfaces.

The inverse problem involves checking whether given points, curves, and surfaces lie in, on, or outside a given solid. The solution of this problem is achieved by the set membership classification and neighborhoods. To classify a point against a solid, one passes a line through the point, intersects it with the solid, and classifies it with respect to the solid. A similar approach can be followed for curve/solid and surface/solid classification.

7.13.3 Segmentation

The segmentation concept introduced for curves and surfaces is applicable to solids. Segmenting a solid is equivalent to splitting it into two or four valid subsolids depending on whether it is to be split by a plane or a point respectively. Each resulting subsolid should have its own topology and geometry. In a B-rep model, new vertices, edges, and faces are created. Steps 1 to 5 described in Example 7.7 shows how to split a B-rep model into two subsolids. In a CSG model, splitting the solid would also require splitting its CSG graph and tree. Splitting an ASM model is an extension of segmenting curves and surfaces discussed in Chaps. 5 and 6.

7.13.4 Trimming and Intersection

Trimming a solid entails intersecting the solid with the trimming boundaries, say surfaces, followed by the removal of the solid portions outside these boundaries. In trimming a solid, it is split into three subsolids, two of which are removed. The trimming surfaces must be of the type supported by the given solid modeler to be able to solve the resulting surface/solid intersection problem.

The intersection problem involving solids is trivial to perform if boolean operations are supported by the solid modeler. All that needs to be done is to use the intersection operator. No additional development or programming is required.

7.13.5 Transformation

Homogeneous transformations, or rigid motion, of solids involve translating, rotating, or scaling them. These transformations can be used on two different occasions. When constructing a solid, its primitives are positioned and oriented properly before applying boolean operations by using these transformations. Here, the local coordinate system of each primitive is positioned and oriented relative to the MCS or a WCS of the solid under construction. If the solid is to be transformed later after its complete construction, the transformation operation must be applied to all, say, its faces, edges, and vertices for a B-rep solid or its primitives for a CSG solid.

7.13.6 Editing

Editing a solid model is an important feature for the design process. Most new designs are not totally new but rather alterations of existing ones. Editing a solid involves changing its existing topological and geometrical information. An efficient means of solids editing is to use its CSG graph which is only a symbolic structure, as discussed in Sec. 7.8. This is natural for CSG models but for other models such as a B-rep model a CSG graph and a tree can be created; otherwise editing would have to be done on the face/edge/vertex structure which may be slow.

Solid modelers must provide users with fast visual feedbacks when solids are edited. This implies that boundary representations must be updated rapidly, because displays are typically generated from face, edge, and vertex data. Thus, editing is faster if only the part of the boundary representation that is affected by the user changes is updated. Some updating algorithms are based on structural and spatial localities and are not covered here.

7.14 SOLID MODELING-BASED APPLICATIONS

Applications based on solid modeling have been increasing rapidly. The underlying characteristic of all these applications is full automation. Current applications can be divided into four groups:

1. Graphics. This is considered the most complete group. It includes generating line drawings with or without hidden line removal, shading, and animation.
2. Design. The most well-understood application in this group is the mass property calculations. Other applications include interference analysis, finite element modeling, and kinematic and mechanism analysis. Some of these applications are more developed than the others.

3. Manufacturing. The most active application in this group is tool path generation and verification. Other applications include process planning, dimension inspection, implementing form features needed for manufacturing into solid modelers, and representing geometric features such as tolerances and surface finish.
4. Assembly. This is a useful group of applications to robotics and flexible manufacturing. Applications include assembly planning, vision algorithms based on solid modeling, and robotic kinematics and dynamics driven by solid models.

Some of the above applications are covered in the appropriate chapters of Part V of the book while other applications are not covered because they are either beyond the scope of the book or at an early stage of research.

7.15 DESIGN AND ENGINEERING APPLICATIONS

Following are some examples to show how the solid modeling theory can be utilized in design and engineering applications. Readers can utilize available solid modelers to them to rework these examples or develop new ones along the same line of thinking.

Example 7.12. Figure 7-61 shows a $3 \times 3 \times 1$ inch block. A curved slot of 0.3 inch deep is to be milled in the block using a ball-end mill of 0.25 inch diameter. The equation of the centerline of the slot on the top face of the block is given by $z = -(1.5 - \sqrt[3]{x^2/3.5})$. Create the solid model of the block with the slot in it and show the swept volume of the tool if it moves perpendicular to the top plane of the block.

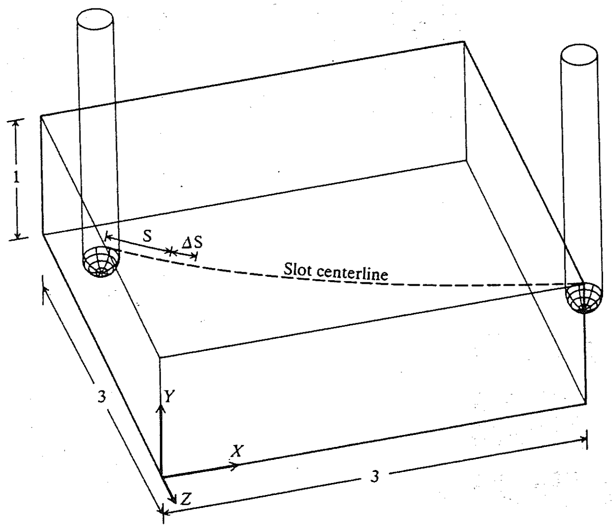


FIGURE 7-61
A block with a curved slot.

Solution. This example illustrates how complex shapes can be approximated to fit within the modeling domain of a given solid modeler. Here, we are assuming that the modeler supports boolean operations and has natural quadrics as its minimum set of primitives. If the block had a slot with a straight centerline in any orientation relative to it, its modeling would have been exact and trivial. In the case of a curved centerline, the tool motion is approximated by line segments along the centerline. The solid model of the block becomes a block primitive from which the tool, in its proper position and orientation, is subtracted. The swept volume of the tool is the union of the tool instances.

The tool is the union of a sphere and a cylinder both positioned at 0.175 inch below the top face. This position (0.175) assumes the slot is created in the block by removing all the material in one cut. The original position of the centerline of the tool is at the beginning of the slot centerline, as shown in Fig. 7-61. In this position, the tool is oriented vertically along the Y axis. In order to obtain a fairly smooth slot, the tool is positioned every $d/4$, where d is the tool diameter, that is, every 0.0625 inch. The top view of the profile of the tool swept volume is shown in Fig. 7-62. The curve length S of the slot must be calculated to determine the required number of tool positions, N, to sweep the slot. This length is given by

$$S = \int \sqrt{1 + \left(\frac{dz}{dx}\right)^2}\, dx \tag{7.104}$$

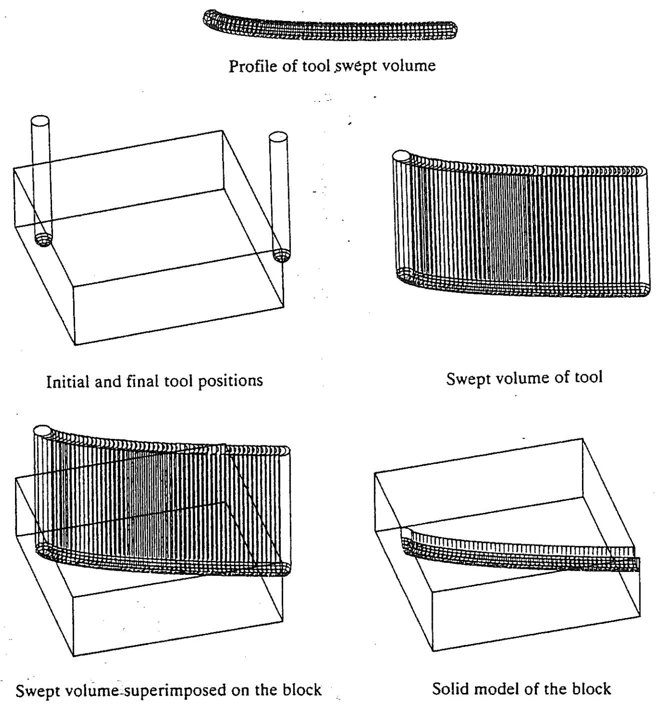

Profile of tool swept volume
Initial and final tool positions
Swept volume of tool
Swept volume superimposed on the block
Solid model of the block

FIGURE 7-62
Block and tool swept volume.

where dz/dx is calculated from the slot equation as

$$\frac{dz}{dx} = 0.439x^{-1/3} \tag{7.105}$$

Substituting Eq. (7.105) into (7.104), rearranging, and integrating, we obtain

$$S = \sqrt{(0.193 + x^{2/3})^3} \tag{7.106}$$

The curve length of the slot between $x = 0$ and $x = 3$ is $[S(x = 3) - S(x = 0)]$ 3.342 inch. Thus the number N is given by

$$N = \text{INT}\left(\frac{S}{\Delta S}\right) + 1 \tag{7.107}$$

where INT is the INTEGER function. For $\Delta S = 0.0625$, 54 tool instances are required.

Rearranging Eq. (7.106) to give x in terms of S enables the calculation of the tool position, that is,

$$x = \sqrt{(S^{2/3} - 0.193)^3} \tag{7.108}$$

In a recursive form, this equation becomes

$$x_{i+1} = \sqrt{(S_{i+1}^{2/3} - 0.193)^3}, \qquad 0 \le i \le (N-1) \tag{7.109}$$

and
$$S_{i+1} = S_i + \Delta S \tag{7.110}$$

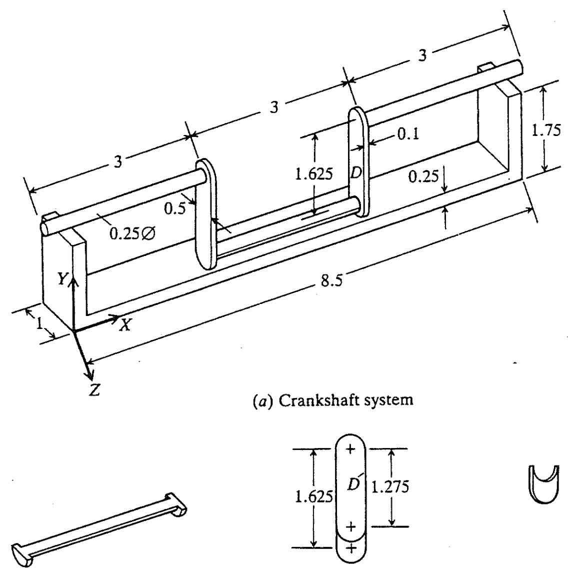
where the subscripts i and $i + 1$ indicate the previous and current positions respectively. For the initial position, $i = 0$, $S_0 = 0.085$ inch, and $x_0 = 0$. Having the value of x from Eq. (7.109), the z coordinate of the tool position can be obtained from the equation of the slot centerline. The y coordinate of the origins of the sphere and cylinder primitives that make up the tool (see Fig. 7.4) is 0.825.

The position of each primitive (sphere and cylinder) of the tool is given by (x, y, z) as calculated above. The sphere and cylinder are unioned together to give the tool which, in turn, is subtracted from the block. For example, the initial and final positions of the sphere and the cylinder are given by $(0, 0.825, -1.5)$ and $(3.0, 0.825, -0.13)$ respectively. Repeating this process 54 times produces the block with the slot. The results are shown in Fig. 7-62.

The above solution is approximate: the more the number of tool instances, the more accurate the solid model of the block and, of course, the more expensive to operate on the model. The exact solution of this problem would require a solid modeler that supports nonlinear (or general) sweep of moving objects besides the natural quadrics primitives.

Example 7.13. Figure 7-63 shows a crankshaft mounted on a bracket. The given dimensions cause interference between the crankshaft and the bracket to occur. Find the maximum volume of interference. The interference can be removed by either creating a depression in the bracket or decreasing the length of the crankshaft. Find the amount of material removed for the latter solution.

Solution. This problem illustrates how to use boolean operations for interference detection between components of a solid model. Given the dimensions shown, create the solid model of the crankshaft system. Designate the bracket and crankshaft as solids B and C respectively. The interference volume is the intersection of the two; that is, $B \cap^* C$ and is shown in Fig. 7-63b. The mass properties of this volume can easily be calculated and are not discussed now (refer to Chap. 17).



(a) Crankshaft system

(b) Interference volume

(c) Elimination of interference

FIGURE 7-63
Interference detection.

The crankshaft should be reduced by 0.35 inch (see Fig. 7-63c) so that the interference between the bracket and the crankshaft is removed, and to allow 0.1 inch clearance between the two. The amount of material removed is $2(D -^* D')$, where $D -^* D'$ is the difference between the original crankshaft D (see Fig. 7-63a) and the new one D' (Fig. 7-63c).

PROBLEMS

Part 1: Theory

7.1. A valid solid is defined as a point set that has an interior and a boundary as given by Eq. (7.1). A valid boundary must be in contact with the interior. Sketch a few two- and three-dimensional solids and identify iS and bS for each one. Is iS always joint for any S? Can bS be disjoint? What is your conclusion?

7.2. Three point sets in E^2 define three valid polygonal solids S_1, S_2, and S_3. The three solids are bounded by three boundary sets bS_1, bS_2, and bS_3 given by their corner points as: b$S_1 = \{(2, 2), (5, 2), (5, 5), (2, 5)\}$, b$S_2 = \{(3, 3), (7, 3), (7, 6), (3, 6)\}$, and b$S_3 = \{(4, 1), (6, 1), (4, 4), (6, 4)\}$. Find $S_1 \cup S_2 \cup S_3$, $S_1 \cap S_2 \cap S_3$, and $S_1 - S_2 - S_3$.

7.3. Reduce the following set expressions:
(a) $c(P \cap Q) \cup P$
(b) $c(P \cup Q) \cup (P \cap cQ)$
(c) $(P \cap Q) \cap c(P \cup Q)$
(d) $P - (P - Q)$
Use the set laws given by Eqs. (7.13) to (7.27) as well as the Venn diagram.

7.4. Using the set membership classification, classify the line L shown in Fig. P7-4 with respect to the solid shown if the solid given is a B-rep and a CSG.

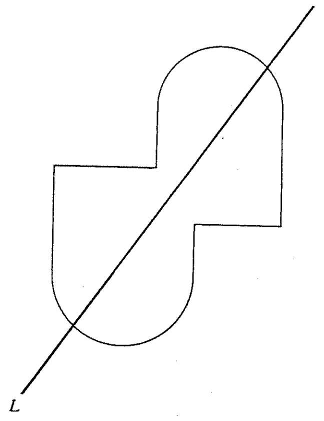


FIGURE P7-4

7.5. Implementing set operations given by Eqs. (7.41) to (7.48), or $S = A\langle \mathrm{OP}\rangle B$ in general, involves finding intersections between bA and bB, classifying the boundaries with respect to each operand, and combining classifications according to the rules:

$$\langle \mathrm{OP}\rangle = \cup^*: \mathrm{b}S = (A \text{ out } B \text{ UN}^* B \text{ out } A \text{ UN}^* A \text{ on } B+)$$

$$\langle \mathrm{OP}\rangle = \cap^*: \mathrm{b}S = (A \text{ in } B \text{ UN}^* B \text{ in } A \text{ UN}^* A \text{ on } B+)$$

$$\langle \mathrm{OP}\rangle = -^*: \mathrm{b}S = (A \text{ out } B \text{ UN}^* B \text{ in } A^{-1} \text{ UN}^* A \text{ on } B-)$$

where UN* is a regularized union operator in E^1. A on $B+$ consists of those parts of bA that lie on bB so that the face normals of the respective faces are equal, whereas A on $B-$ consists of the overlapping parts where the normals are opposite. B in A^{-1} denotes the complement of B in A, that is, B in A with the orientation of all faces reversed. Notice that the $+$ and $-$, and the exponent -1, are a form of neighborhoods. Using the above equations:

(*a*) Explain the concept of closure used in Eqs. (7.41) to (7.48) and shown in Fig. 7-14. What should be done to implement the concept of interior to eliminate nonregular sets?

(*b*) Find $S = A\langle \mathrm{OP}\rangle B$ for A and B shown in Fig. P7-5.

7.6. Solids in E^2 (i.e., two-dimensional solids) and their solid modelers are valuable in understanding many of the concepts needed to handle solids in E^3 (i.e., three-dimensional solids). A two-dimensional solid is a point set of ordered pairs (x, y). A two-dimensional solid modeler based on half-spaces is to be developed utilizing linear and disc half-spaces. Find the equations of the two half-spaces.

Hint: See Eqs. (7.51) and (7.52).

Develop a parametric equation for their intersection.

7.7. Apply Euler laws, Eqs. (7.57) and/or (7.58), to models shown in Prob. 7.17 of Part 2. Using Euler operators, write the construction sequence needed to create them.

7.8. Following Example 7.7, write a procedure to split a solid with a plane.

7.9. Create an input sequence to create a CSG model of each object shown in Prob. 7.17 in Part 2. Based on this sequence, find the model CSG graph and tree. It is preferred to obtain balanced trees if possible.

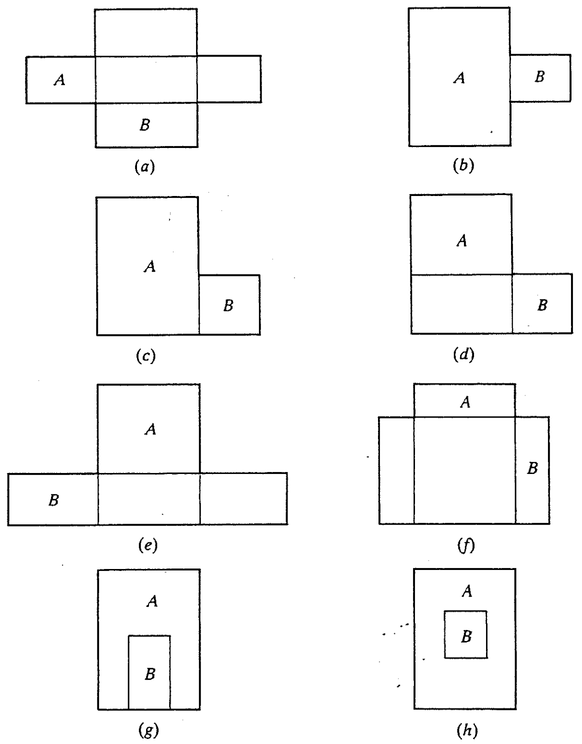


FIGURE P7-5

7.10. Apply the various available traversal methods to traverse each tree developed in Prob. 7.9.

7.11. It is desired to develop bounded primitives for a two-dimensional solid modeler based on the CSG scheme. A plate (rectangular plate and triplate) and disc primitives are to be developed. Find the mathematical definitions of these primitives.

Hint: See Eqs. (7.61), (7.62), and (7.65).

Develop intersection equations (refer to Prob. 7.6).

7.12. Apply the algorithms that perform boolean operations and described or used in Examples 7.7 (B-rep) and 7.9 (CSG) to the two primitive blocks shown in Fig. 7-36*b*.

7.13. How can you use a cylinder primitive to generate a sphere?

7.14. How can you generate a torus using other natural quadrics?

7.15. Repeat Prob. 7.8 but for a CSG model (follow Example 7.9).

7.16. Problems 7.6 and 7.11 introduced the mathematical foundations of a two-dimensional solid modeler. The further development of such a modeler requires representations of orientable surfaces, that is, represent surface normals, neighborhoods, classifications, and combining the classifications. Discuss the following for half-space, B-rep, and CSG schemes:

(*a*) How to represent surface normals and neighborhoods

(*b*) How to develop a classification algorithm

(*c*) How to combine classifications

Part 2: Laboratory

7.17. Create the solid models of the objects of Prob. 5-20. Similarly, create the solid models of the objects shown in Prob. 6.16. Use layers and colors provided by your CAD/CAM system to be able to manage the primitives and resulting subsolids. How do you compare the amount of effort it takes to create the wireframe, surface, and solid models of a given object?

7.18. Perform the set operations on the two objects shown in Fig. 7-15 on your CAD/CAM system. Compare the results with those of Example 7.3. Did your system fail to perform any of the operations? What is your conclusion?

Part 3: Programming

Write a procedure to:

7.19. Classify lines with respect to two-dimensional solids for B-rep and CSG (see Figs. 7-17 and 7-18).

7.20. Combine the classifications that result from the above procedure.

7.21. Implement the results of Prob. 7.6.

7.22. Convert a user input into a CSG graph and a tree traversed in a reverse postorder.

7.23. Implement the results of Prob. 7.11.

7.24. Create two-dimensional primitives based on B-rep schemes (assume that low-level Euler operators exist).

7.25. Implement results of Prob. 7.16.

7.26. Implement results of Prob. 7.13 into an existing solid modeler.

7.27. Implement results of Prob. 7.14 into an existing solid modeler.

BIBLIOGRAPHY

Baer, A., C. Eastman, and M. Henrion: "Geometric Modeling: A Survey," *Computer Aided Des. (CAD) J.*, vol. 11, no. 5, pp. 253–272, 1979.

Baumgart, B. G.: "Winged Edge Polyhedron Representation," AIM-179, report STAN-CS-320, Stanford University, 1972.

Baumgart, B. G.: "Geometric Modeling for Computer Vision," AIM-249, report CS-463, Artificial Intelligence Laboratory, Stanford University, 1974.

Berlin, E.: "Solid Modeling on a Microcomputer," *Computer Graphics World*, pp. 39–42, November 1984.

Bin, H.: "Inputting Constructive Solid Geometry Representations Directly from 2D Orthographic Engineering Drawings," *CAD J.*, vol. 18, no. 3, pp. 147–155, 1986.

Bobrow, J. E.: "NC Machine Tool Path Generation from CSG Part Representation," *CAD J.*, vol. 17, no. 2, pp. 69–76, 1985.

Bowerman, R. G.: "Drafting Links Up with Solid Modeling," *Mach. Des.*, pp. 46–49, September 12, 1985.

Boyse, J. W.: "Interference Detection among Solids and Surfaces," *Commun. ACM*, vol. 22, no. 1, pp. 3–9, 1979.

Braid, I. C.: "The Synthesis of Solids Bounded by Many Faces," *Commun. ACM*, vol. 18, no. 4, pp. 209–218, 1975.

Bronsvoort, W. F., J. J. Wijk, and F. W. Jansen: "Two Methods for Improving the Efficiency of Ray Casting in Solid Modeling," *CAD J.*, vol. 16, no. 1, pp. 51–55, 1984.

CAM-I, Inc.: "Design of an Experimental Boundary Representation and Management System for Solid Objects," report R-80-GM-02, CAM-I, Arlington, Tex., 1980.

CAM-I, Inc.: "Boundary File Design," report R-81-GM-02.1, CAM-I, Arlington, Tex., 1981.

CAM-I, Inc.: "Boundary Representation for Solid Objects," report R-82-GM-02.1, CAM-I, Arlington, Tex., 1982.

CAM-I, Inc.: "Extended Geometric Facilities for Surface and Solid Objects," report R-83-GM-02.1, CAM-I, Arlington, Tex., 1983.

CAM-I, Inc.: "Solid Modeling Applications; The Real Payback," *Proc. CAM-I's 3rd Geometric Modeling Seminar*, March 19–20, 1985, Nashville, Tenn.

Casale, M.: "Free-Form Solid Modeling with Trimmed Surface Patches," *IEEE CG&A*, pp. 33–43, January 1987.

Casale, M. S., and Stanton, E. L.: "An Overview of Analytic Solid Modeling," *IEEE CG&A*, pp. 45–56, February 1985.

Childress, R. L.: *Sets, Matrices, and Linear Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1974.

Chiyakura, H., and F. Kimura: "A Method of Representing the Solid Design Process," *IEEE CA&A*, pp. 32–41, April 1985.

Choi, B. K., M. M. Barash, and D. C. Anderson: "Automatic Recognition of Machined Surfaces from a 3D Solid Model," *CAD J.*, vol. 16, no. 2, pp. 81–86, 1984.

Clark, A. L.: "Roughing It: Realistic Surface Types and Textures in Solid Modeling," *Computers In Mechanical Engineering (CIME) Mag.*, pp. 12–16, March 1985.

Congdon, R. M., and D. C. Gossard: "Interactive Graphic Input of Plane-Faced Solid Models," *Proc. Conf. on CAD/CAM Technology in Mechanical Engineering*, March 24–26, 1982, pp. 350–360, MIT, Cambridge, Mass.

Crocker, G. A.: "Screen-Area Coherence for Interactive Scanline Display Algorithms," *IEEE CA&G*, pp. 10–17, September 1987.

Doctor, L. J., and J. G. Torborg: "Display Techniques for Octree-Encoded Objects," *IEEE CG&A*, pp. 29–38, July 1981.

Eastman, C., and M. Henrion: "GLIDE: A Language for Design Information Systems," *Proc. First Annual Conf. on Computer Graphics in CAD/CAM Systems*, April 9–11, 1979, pp. 24–33, MIT, Cambridge, Mass.

Eastman, C., and K. Weiler: "Geometric Modeling Using the Euler Operators," *Proc. First Annual Conf. on Computer Graphics in CAD/CAM Systems*, April 9–11, 1979, pp. 248–259, MIT, Cambridge, Mass.

Farr, R., and G. Fredrickson: "Interactive Solid Modeling," *CAE Mag.*, pp. 46–48, November 1986.

Hakala, D. G., R. C. Hillyard, B. E. Nourse, and P. J. Malraison: "Natural Quadrics in Mechanical Design," in *AUTOFACT WEST*, pp. 363–378, Anaheim, Calif., November 17–20, 1980.

Halmos, P. R.: *Naive Set Theory*, Litton Educational Publishing, New York, 1960.

Hillyard, R.: "The Build Group of Solid Modelers," *IEEE CG&A*, pp. 43–52, March 1982.

Holt, M. G.: "Experiences with CAD Solids Modeling and Its Role in Engineering Design," *Computer-Aided Engng J.*, pp. 38–44, April 1985.

Hook, T. V.: "Advanced Techniques for Solid Modeling," *Computer Graphics World*, pp. 45–54, November 1984.

Hrbacek, K., and T. Jech.: *Introduction to Set Theory*, Marcel Dekker, New York, 1978.

Johnson, R. H.: "Product Data Management with Solid Modeling," *Computer-Aided Engng J.*, pp. 129–132, August 1986.

Kalay, Y. E.: "A Relational Database for Nonmanipulative Representation of Solid Objects," *CAD J.*, vol. 15, no. 5, pp. 271–276, 1983.

Kirk, D. B.: "Curved Surfaces in Solid Modeling: New Hardware Improves the View," *CIME Mag.*, pp. 10–14, May 1986.

Krouse, J. K.: "Solid Models for Computer Graphics," *Mach. Des.*, pp. 50–55, May 20, 1982.

Krouse, J. K.: "Sorting Out the Solid Modelers," *Mach. Des.*, pp. 94–101, February 10, 1983.

Krouse, J. K.: "Solid Modeling Catches On," *Mach. Des.*, pp. 60–64, February 7, 1985.

Lee, Y. C., and K. S. Fu: "Machine Understanding of CSG: Extraction and Unification of Manufacturing Features," *IEEE CG&A*, pp. 20–32, January 1987.

Levin, J.: "A Parametric Algorithm for Drawing Pictures of Solid Objects Composed of Quadratic Surfaces," *Commun. ACM*, vol. 19, no. 10, pp. 555–563, 1976.

Manty, M. la: "Boolean Operations of 2-Manifolds Through Vertex Neighborhood Classification," *ACM Trans. on Graphics*, vol. 5, no. 1, pp. 1–29, 1986.

Manty, M. la, and R. Sulonen: "GWB: A Solid Modeler with Euler Operators," *IEEE CG&A*, pp. 17–31, September 1982.

Manty, M. la, and M. Tamminen: "Localized Set Operations for Solid Modeling," *ACM Computer Graphics*, vol. 17, no. 3, pp. 279–288, 1983.

Miller, J. R., D. R. Starks, and M. D. Hastings: "An Evolving Volume Modeling-Based CAD/CAM System," *Proc. Conf. on CAD/CAM Technology in Mechanical Engineering*, March 24–26, 1982, pp. 33–53, MIT, Cambridge, Mass.

Mortenson, M. E.: *Geometric Modeling*, John Wiley, New York, 1985.

Myers, W.: "An Industrial Perspective on Solid Modeling," *IEEE CG&A*, pp. 86–97, March 1982.

Patnaik, L. M., R. S. Shenoy, and D. Krishnan: "Set Theoretic Operations on Polygons Using the Scan-Grid Approach," *CAD J.*, vol. 18, no. 5, pp. 275–279, 1986.

Pickett, M. S., and J. W. Boysl (Eds.): "Solid Modeling by Computers: From Theory to Applications," *Proc. Symposium on Solid Modeling*, September 25–27, 1983, General Motors Research Laboratories, Warren, Mich.

Post, F. H., and F. Klok: "Deformations of Sweep Objects in Solid Modeling," in *Eurographics '86* (Ed. A. A. G. Requicha), pp. 103–115, Elsevier Science, New York, 1986.

Pratt, M. J.: "Solid Modeling and the Interface Between Design and Manufacture," *IEEE CG&A*, pp. 52–59, July 1984.

Putnam, L. K., and P. A. Subrahmanyam: "Boolean Operations on *n*-Dimensional Objects," *IEEE CG&A*, pp. 43–51, June 1986.

Requicha, A. A. G. (Ed.): *Eurographics '86*, Elsevier Science, New York, 1986.

Requicha, A. A. G., and S. C. Chan: "Representation of Geometric Features, Tolerances, and Attributes in Solid Modelers Based on Constructive Geometry," *IEEE J. of Robotics and Automation*, vol. RA-2, no. 3, pp. 156–166, 1986.

Requicha, A. A. G., and H. B. Voelcker: "Solid Modeling: A Historical Summary and Contemporary Assessment," *IEEE CG&A*, pp. 9–24, March 1982.

Requicha, A. A. G., and H. B. Voelcker: "Solid Modeling: Current Status and Research Directions," *IEEE CG&A*, pp. 25–37, October 1983.

Requicha, A. A. G., and H. B. Voelcker: "Boolean Operations in Solid Modeling: Boundary Evaluation and Merging Algorithms," *Proc. IEEE*, vol. 73, no. 1, pp. 30–44, 1985.

Rossignac, J. R., and A. A. G. Requicha: "Offsetting Operations in Solid Modeling," *Computer Aided Geometric Des.*, vol. 3, pp. 129–148, 1986.

Rossignac, J. R., and A. A. G. Requicha: "Depth-Buffering Display Techniques for Constructive Solid Geometry," *IEEE CG&A*, pp. 29–39, September 1986.

Roth, S. D.: "Ray Casting as a Method for Solid Modeling," report GMR-3466, General Motors Research Laboratories, Warren, Mich., 1980.

Rouse, N. E.: "Linking Solids and Surfaces," *Mach. Des.*, pp. 82–86, May 7, 1987.

Sarraga, R. F.: "Algebraic Methods for Intersections of Quadric Surfaces in GMSOLID," *Computer Vision, Graphics, and Image Processing*, vol. 22, pp. 222–238, 1983.

Stewart, I. P.: "Quadtrees: Storage and Scan Conversion," *The Computer J.*, vol. 29, no. 1, pp. 60–75, 1986.

Tamminen, M., O. Karonen, and M. la Manty: "Ray-Casting and Block Model Conversion Using a Spatial Index," *CAD J.*, vol. 16, no. 4, pp. 203–208, 1984.

Tan, S. T., and M. M. F. Yuen: "Integrating Solid Modeling with Finite Element Analysis," *Computer-Aided Engng J.*, pp. 133–137, August 1986.

Tan, S. T., M. F. Yuen, and K. C. Hui: "Modeling Solids with Sweep Primitives," *Computers In Mechanical Engineering (CIME) Mag.*, pp. 60–73, September/October 1987.

The Merrit Company: *Solid Modeling Today*, vol. 1, no. 1–8, The Merrit Company, Santa Monica, Calif., 1986.

Tilove, R. B.: "Set Membership Classification: A Unified Approach to Geometric Intersection Problems," *IEEE Trans. on Computers*, vol. C-29, no. 10, pp. 874–883, 1980.

Tilove, R. B.: "A Null-Object Detection Algorithm for Constructive Solid Geometry," *Commun. ACM*, vol. 27, no. 7, pp. 684–694, 1984.

Tilove, R. B., and A. A. G. Requicha: "Closure of Boolean Operations on Geometric Entities," *CAD J.*, vol. 12, no. 5, pp. 219–220, 1980.

Tilove, R. B., A. A. G. Requicha, and M. R. Hopkins: "Efficient Editing of Solid Models by Exploiting Structural and Spatial Locality," *Computer Aided Geometric Des.*, vol. 1, pp. 227–239, 1984.

Toriya, H., T. Satoh, K. Ueda, and H. Chiyokura: "UNDO and REDO Operations for Solid Modeling," *IEEE CA&A*, pp. 35–42, April 1986.

Vandoni, C. E. (Ed.): *Eurographics '85*, Elsevier Science, New York, 1985.

Varady, T., and M. J. Pratt: "Design Techniques for the Definition of Solid Objects with Free-Form Geometry," *Computer Aided Geometric Des.*, vol. 1, pp. 207–225, 1984.

Voelcker, H. B., and A. A. G. Requicha: "Geometric Modeling of Mechanical Parts and Processes," *Computer*, pp. 48–57, December 1977.

Wagner, P. M.: "Solid Modeling for Mechanical Engineering," *Computer Graphics World*, pp. 10–24, September 1984.

Wang, W. P., and K. K. Wang: "Geometric Modeling for Swept Volume of Moving Solids," *IEEE CG&A*, pp. 8–17, December 1986.

Weiler, K.: "Edge-Based Data Structures for Solid Modeling in Curved-Surface Environment," *IEEE CG&A*, pp. 21–40, January 1985.

"What's Holding Back Solid Modeling?," *CAE Mag.*, pp. 46–52, December 1986.

Williams, N. H.: *Combinatorial Set Theory*, North-Holland, 1977.

Wilson, P. R.: "Euler Formulas and Geometric Modeling," *IEEE CG&A*, pp. 24–36, August 1985.

Woo, T. C.: "Computer Aided Recognition of Volumetric Designs," in *Advances in Computer-Aided Manufacture* (Ed. D. McPherson), pp. 121–136, North-Holland, 1977.

Woo, T. C.: "Feature Extraction by Volume Decomposition," *Proc. Conf. on CAD/CAM Technology in Mechanical Engineering*, March 24–26, 1983, pp. 76–94, MIT, Cambridge, Mass.

Woo, T. C.: "Interfacing Solid Modeling to CAD and CAM: Data Structures and Algorithms for Decomposing a Solid," *Computer*, pp. 44–49, December 1984.

Woo, T. C.: "A Combinatorial Analysis of Boundary Data Structure Schemata," *IEEE CG&A*, pp. 19–27, March 1985.

Woodwark, J. R.: "Generating Wireframes from Set-Theoretic Solid Models by Spatial Division," *CAD J.*, vol. 18, no. 6, pp. 307–315, 1986.

Wyvill, G., T. Kunii, and Y. Shirai: "Space Division for Ray Tracing in CSG," *IEEE CG&A*, pp. 28–34, April 1986.

Yamaguchi, F., and T. Tokieda: "A Solid Modeler with a 4 × 4 Determinant Processor," *IEEE CG&A*, pp. 51–59, April 1985.

Yerry, M. A., and M. S. Shephard: "A Modified Quadtree Approach to Finite Element Mesh Generation," *IEEE CG&A*, pp. 39–46, January/February 1983.

1985 European Conference on Solid Modeling, September 9–10, 1985, London.

CHAPTER 9

GEOMETRIC TRANSFORMATIONS

9.1 INTRODUCTION

A crucial software module of a CAD/CAM system is its graphics package. Such a package contains many graphics concepts that produce the functionality and interactivity of the system. Some of these concepts are geometrical transformations, viewing in two and three dimensions, modeling and object hierarchy, algorithms for removing hidden edges and surfaces, shading and coloring, and clipping and windowing.

Geometric transformations play a central role in model construction and viewing. They are used in modeling to express locations of objects relative to others. In generating a view of an object, they are used to achieve the effect of different viewing positions and directions. Typical CAD/CAM construction commands to translate, rotate, zoom, and mirror entities are all based on geometric transformations covered in this chapter. Some of these commands have been utilized in Chaps. 5 and 6 to construct typical models. Once the model construction is complete, its viewing in its modeling space is achieved via geometric transformations again. Orthographic views for engineering drawings as well as perspective views of a geometric model can be obtained by projecting the model onto the proper plane. In addition, the model itself can be rotated or scaled up and down to view it in its three-dimensional space.

Geometric transformation can also be used to create animated files of geometric models to study their motion. For example, the motion of a spatial mechanism can be animated by first calculating its motion (displacements and/or rotations) using the proper kinematic and dynamic equations. The geometric model of the mechanism at the initial position is then constructed and transformed incrementally using the calculation results. The resulting configurations of the mechanism are grouped together and redisplayed to convey the continuous motion effect. Similarly, transformation can be applied to display vibrations and deformations of modeled objects.

Geometric transformations are ideally suited for computer graphics applications and object modeling because the utilized geometry is point-based. Chapters 5, 6, and 7 have shown that displaying and/or transforming a given entity require the transformation of its key points first. In applications where the viewpoint changes rapidly or where objects move fast in relation to each other, transformation of these points must be carried out rapidly and repeatedly. It is, therefore, necessary to find efficient ways of performing three-dimensional transformations. Most of these transformations are implemented at the hardware level, and firmware that perform them are commonly provided by CAD/CAM systems.

Geometric transformations is a well-established subject. However, this chapter covers them from a new perspective. A unified vector treatment of the subject is presented. This enables both two- and three-dimensional transformations to be handled at once. Sections 9.2 and 9.3 cover both transformations and mappings of geometric models respectively. Basic transformations such as translation, reflection, and rotation as well as their concatenations are covered. The recast of these transformations in terms of homogeneous coordinates is presented. In Sec. 9.3, it is shown how the same transformation equations are interpreted to map model representations from one coordinate system to another. Sections 9.4 and 9.5 describe the inverse operations and show how they are useful in a user's environment. Section 9.6 shows useful applications of geometric transformations.

9.2 TRANSFORMATIONS OF GEOMETRIC MODELS

By definition, geometric transformations are mappings from one coordinate system onto itself. In other words, the description of a geometric model of an object can change within its own MCS. This would imply that the geometric model must undergo motion relative to its MCS. The simplest motion is the rigid-body motion in which the relative distances between object particles remain constant; that is, the object does not deform during the motion. Geometric transformations that describe this motion are often referred to as rigid-body transformations and typically include translation, scaling, reflection, rotation, and any combination of them. These transformations can be applied directly to the parametric representations of objects such as points, curves, surfaces, and solids. Matrix notation provides a very expedient way of developing and implementing geometric transformations into graphic packages.

Transformation of a point represents the core problem in geometric transformation because it is the basic element of object representation. For example, a line is represented by its two endpoints, and a general curve, surface, or solid is represented by a collection of points as seen in the previous part of the book. The problem of transforming a point can be stated as follows. Given a point P that belongs to a geometric model that undergoes a rigid-body motion, find the corresponding point P^* in the new position such that

$$\mathbf{P}^* = f(\mathbf{P}, \text{transformation parameters}) \tag{9.1}$$

that is, the new position vector $\mathbf{P}^*$ should be expressed in terms of the old position vector $\mathbf{P}$ and the motion parameters. One of the characteristics of Eq. (9.1) that should be emphasized here is that geometric transformation should be unique. A given set of transformation parameters must yield one and only one new point for each old point. This characteristic is a direct outcome of the rigid-body motion requirement. Another characteristic is the concatenation, or combination, of transformations. Intuitively, two transformations can be concatenated to yield a single transformation which should have the same effect as the sequential application of the original two.

In order to implement Eq. (9.1) into graphics hardware or software, it is desirable to express it in terms of matrix notation as

$$\mathbf{P}^* = [T]\mathbf{P} \tag{9.2}$$

where $[T]$ is the transformation matrix. Its elements should be functions of the given transformation parameters. The matrix $[T]$ should have some important properties. It must apply to all rigid-body transformation (translation, scaling, reflection, and rotation) as well as clipping and windowing. It should also be applicable to both two- and three-dimensional graphics applications. As explained in the sections to follow, homogeneous representation of Eq. (9.2) is introduced in order to be able to recast translation in terms of this equation.

Applying Eq. (9.2) repeatedly to key points in a geometric model database or a particular entity enables the transformation of the model or the entity. For example, to transform a straight line, its two endpoints are transformed and then connected to produce the transformed line. Similarly, to transform a curve, points on the curve are generated utilizing its parametric equation, transformed, and then connected to give the transformed curve. Equation (9.2) may also be applied to the parametric equation of the entity as discussed in this chapter. The display of transformed entities entails displaying the line segments connecting the transformed key points as discussed in Part II of the book.

9.2.1 Translation

When every entity of a geometric model remains parallel to its initial position, the rigid-body transformation of the model is defined as translation. Translating a model implies that every point on it moves an equal given distance in a given direction. Translation can be specified by a vector, a unit vector and a distance, or two points that denote the initial and final positions of the model to be translated. Figure 9-1 shows a curve translated by a vector **d**.

To relate the final position vector $\mathbf{P}^*$ of a point P to its initial position vector $\mathbf{P}$ after being translated by a vector $\mathbf{d}$, consider the triangle shown in Fig. 9-1. In this case Eq. (9.1) takes the form

$$\mathbf{P}^* = \mathbf{P} + \mathbf{d} \tag{9.3}$$

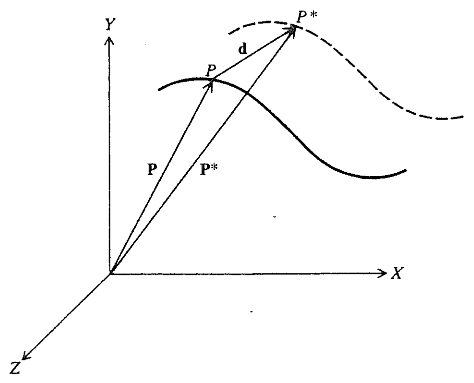


FIGURE 9-1
Translation of a curve.

This equation is applicable to both two- and three-dimensional points and can be written in a scalar form for the three-dimensional case as

$$\begin{aligned} x^* &= x + x_d \\ y^* &= y + y_d \\ z^* &= z + z_d \end{aligned} \tag{9.4}$$

If Eq. (9.3) is applied to each point on a curve, that is, pointwise transformation, the curve is then translated by the vector **d**. However, it is more efficient and useful to relate the translation of an entity (curve, surface, or solid) to its geometric representation whether it is analytic or synthetic. For example, translating a circle or an ellipse requires translating its center only, and translating a parabola or hyperbola requires translating its vertex. Example 9.1 as well as problems at the end of the chapter provide more details.

As expected intuitively, translating a curve does not change its tangent vector at any of its points. This can be seen by differentiating Eq. (9.3) with respect to the parameter u to obtain $\mathbf{P}^{*\prime} = \mathbf{P}'$ because the translation vector **d** is constant.

Example 9.1. Given a Hermite cubic spline, show that its pointwise translation and translating its geometric representation are identical.

Solution. As shown in Chap. 5, the geometric representation of a Hermite cubic spline is define by its two endpoints and two end slopes. Let us assume that the spline is defined in its initial and final positions by the geometry vectors $\mathbf{V} = [\mathbf{P}_0 \ \ \mathbf{P}_1 \ \ \mathbf{P}'_0 \ \ \mathbf{P}'_1]^T$ and $\mathbf{V}^* = [\mathbf{P}^*_0 \ \ \mathbf{P}^*_1 \ \ \mathbf{P}^{*\prime}_0 \ \ \mathbf{P}^{*\prime}_1]^T$ respectively, as shown in Fig. 9-2. Because the spline undergoes translation only, $\mathbf{V}^*$ becomes

$$\mathbf{V}^* = [\mathbf{P}_0 + \mathbf{d} \ \ \mathbf{P}_1 + \mathbf{d} \ \ \mathbf{P}'_0 \ \ \mathbf{P}'_1]^T = \mathbf{V} + \mathbf{D} \tag{9.5}$$

where $\mathbf{D} = [\mathbf{d} \ \ \mathbf{d} \ \ 0 \ \ 0]^T$.

Utilizing Eq. (5.83), the spline equation in the translated position is given by

$$\mathbf{P}^*(u) = \mathbf{U}^T[M_H]\mathbf{V}^*, \qquad 0 \le u \le 1 \tag{9.6}$$

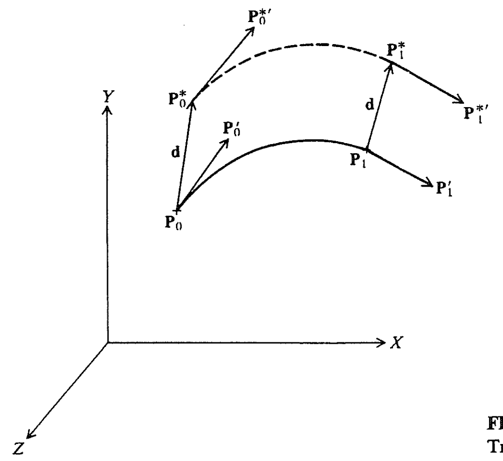


FIGURE 9-2
Translating a Hermite cubic spline.

Substituting Eq. (9.5) into Eq. (9.6) gives

$$\mathbf{P}^*(u) = \mathbf{U}^T[M_H](\mathbf{V} + \mathbf{D}) = \mathbf{U}^T[M_H]\mathbf{V} + \mathbf{U}^T[M_H]\mathbf{D}$$

or

$$\mathbf{P}^*(u) = \mathbf{P}(u) + \mathbf{U}^T[M_H]\mathbf{D} \tag{9.7}$$

Substituting Eq. (5.84) into the second term of Eq. (9.7) and reducing the result gives

$$\mathbf{P}^*(u) = \mathbf{P}(u) + \mathbf{d} \tag{9.8}$$

Equation (9.8) simply implies that each point on the translated spline is obtained by translating its corresponding point on the initial spline by the vector **d**. Therefore, pointwise translation of a Hermite spline is identical to translating its geometry vector—more specifically its endpoints. This is beneficial because the geometric characteristics of the curve can be preserved and no intermediate points are needed to be calculated on the curve to translate it.

9.2.2 Scaling

Scaling is used to change, increase or decrease, the size of an entity or a model. Pointwise scaling can be performed if the matrix $[T]$ in Eq. (9.2) is diagonal, that is,

$$\mathbf{P}^* = [S]\mathbf{P} \tag{9.9}$$

where $[S]$ is a diagonal matrix. In three dimensions, it is given by

$$[S] = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \tag{9.10}$$

Thus (9.9) can be expanded to give

$$x^* = s_x x \qquad y^* = s_y y \qquad z^* = s_z z \tag{9.11}$$

The elements s_x, s_y, and s_z of the scaling matrix $[S]$ are the scaling factors in the X, Y, Z directions respectively. Scaling factors are always positive (negative factors produce reflection). If the scaling factors are smaller than 1, the geometric model or entity to which scaling is applied is compressed; if the factors are greater than 1, the model is stretched. If the scale factors are equal, that is, $s_x = s_y = s_z = s$, the model changes in size only and not in shape; this is the case of uniform scaling. For this case, Eq. (9.9) becomes

$$\mathbf{P}^* = s\mathbf{P} \tag{9.12}$$

Unlike translation, scaling proportionally changes tangent vectors by the factor s as differentiating Eq. (9.12) with respect to u, for a curve, gives $P^{*\prime} = sP'$. However, uniform scaling does not change the slope, or direction cosines, at any point.

Differential scaling occurs when $s_x \neq s_y \neq s_z$; that is, different scaling factors are applied in different directions. Differential scaling changes both the size and the shape of a geometric model or curve. It also changes the direction cosines at any point. Differential scaling is seldom used in practical applications.

The scaling discussed above is said to be about the origin, that is, a model or a curve changes size and location with respect to the origin of the coordinate system, as shown in Fig. 9-3. The model or the curve gets closer to or further from the origin depending on whether the scaling factor is smaller or greater than one respectively (see Fig. 9-3). Scaling about any other point than the origin is possible, and its development is assigned in Prob. 9.3.

Uniform scaling is available on CAD/CAM systems in the form of a "zoom" command. The command requires users to input the scale factor s and digitize the entity or the view to be zoomed. The zoom, or scaling, function is useful if a user needs to magnify a dense graphics area, on the screen, to be able

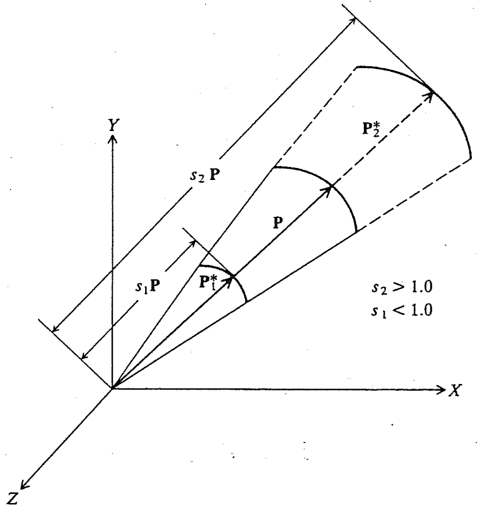


FIGURE 9-3
Scaling a curve relative to the origin.

to visually identify the geometry in the area for picking and selection purposes. If a view is zoomed, a "set view" or "reset view" command is usually required to make the view scaling permanent or to return the view to its original size respectively.

Example 9.2. Show that scaling the geometric representation of a Hermite cubic spline is identical to its pointwise scaling.

Solution. The solution is similar to that followed in Example 9.1. Here the geometry vector, based on Eq. (9.12), is given by

$$\mathbf{V}^* = [s\mathbf{P}_0 \quad s\mathbf{P}_1 \quad s\mathbf{P}'_0 \quad s\mathbf{P}'_1]^T = s\mathbf{V}$$

Substituting the above equation into Eq. (9.6) gives

$$\mathbf{P}^*(u) = s\mathbf{U}^T[M_H]\mathbf{V} = s\mathbf{P}(u)$$

Therefore, scaling the geometry vector $\mathbf{V}$ of a Hermite cubic spline is identical to scaling each point on it.

9.2.3 Reflection

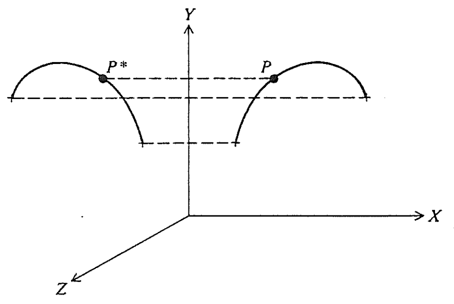
Reflection (or mirror) transformation is useful in constructing symmetric models. If, for example, a model is symmetric with respect to a plane, then only half of its geometry is created which can be copied by reflection to generate the full model. A geometric entity can be reflected through a plane, a line, or a point in space, as illustrated in Fig. 9-4. Reflecting an entity through a principal plane ($x = 0$, $y = 0$, $z = 0$ plane) is equivalent to negating the corresponding coordinate of each point on the entity. Reflection through the $x = 0$, $y = 0$, or $z = 0$ plane can be achieved by negating the x, y, or z coordinate respectively. Reflection through an axis is equivalent to reflection through two principal planes intersecting at the given axis. As shown in Fig. 9-4*b*, an entity is reflected through the Y axis by reflection through the $z = 0$ plane followed by a reflection through the $x = 0$ plane. In this case, reflection is accomplished by negating the x and z coordinates of each point on the entity. Similarly, reflection through the X and Z axes requires negating the y and z and the x and y coordinates respectively. Reflection through the origin is equivalent to reflection through the three principal planes that intersect at the origin. Figure 9-4*c* shows reflection of an entity through the origin which is accomplished by negating the three coordinates of any point on the entity.

Equation (9.9) can be used to describe reflection if the diagonal elements of $[S]$ are chosen to be ones. Thus, the reflection transformation can be expressed by the following equation:

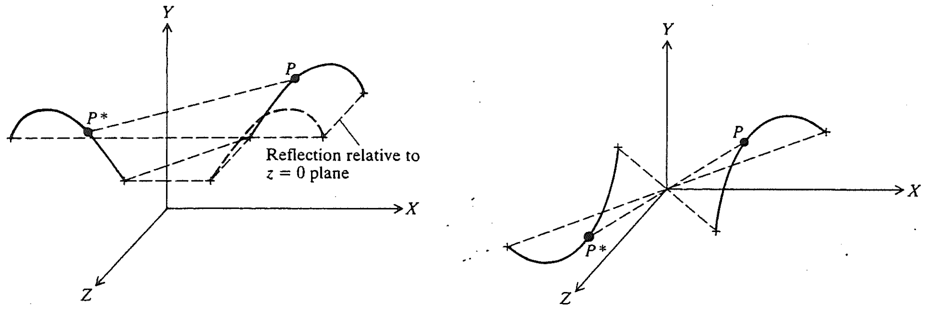
$$\mathbf{P}^* = [M]\mathbf{P} \tag{9.13}$$

where $[M]$ (mirror matrix) is a diagonal matrix with elements of ± 1, that is,

$$[M] = \begin{bmatrix} m_{11} & 0 & 0 \\ 0 & m_{22} & 0 \\ 0 & 0 & m_{33} \end{bmatrix} = \begin{bmatrix} \pm 1 & 0 & 0 \\ 0 & \pm 1 & 0 \\ 0 & 0 & \pm 1 \end{bmatrix} \tag{9.14}$$



(*a*) Reflection through a principal plane ($x = 0$ plane)

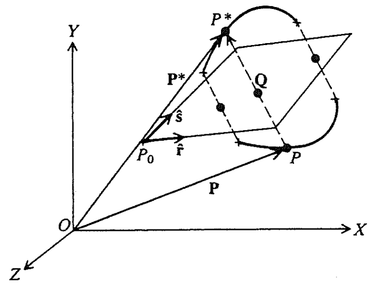


(*b*) Reflection through an axis (Y axis) (*c*) Reflection through the origin

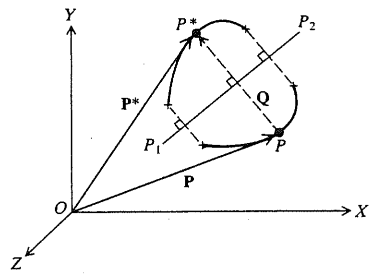
FIGURE 9-4
Reflecting a curve relative to a coordinate system.

The reflection (or mirror) matrix $[M]$ given by Eq. (9.14) applies only to reflections relative to planes, axes, or the origin of a coordinate system. For reflection through the $x = 0$ plane, $m_{11} = -1$ and $m_{22} = m_{33} = 1$. Similarly, setting $m_{11} = m_{33} = 1$ and $m_{22} = -1$, or $m_{11} = m_{22} = 1$ and $m_{33} = -1$, produces reflection through the $y = 0$ or $z = 0$ plane respectively. Reflection through the X axis requires $m_{11} = 1$ and $m_{22} = m_{33} = -1$, through the Y axis requires $m_{11} = m_{33} = -1$ and $m_{22} = 1$, and through the Z axis requires $m_{11} = m_{22} = -1$ and $m_{33} = 1$. Selecting all the diagonal elements to be negative, that is, $m_{11} = m_{22} = m_{33} = -1$, produces reflection through the origin. For the latter case, Eq. (9.14) becomes $\mathbf{P}^* = -\mathbf{P}$ and, therefore, $\mathbf{P}^{*\prime} = -\mathbf{P}'$; that is, magnitudes of tangent vectors remain constant but their directions are reversed.

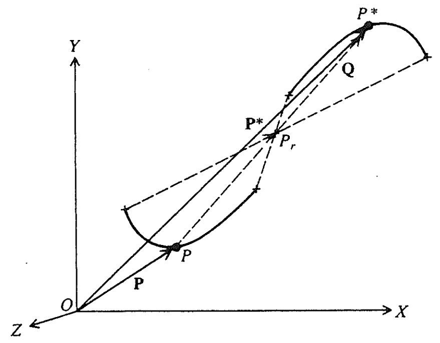
While reflections relative to a coordinate system have been discussed above, other reflections through general planes, lines, and points are possible and useful in practice. Figure 9-5 illustrates this general reflection of an entity. As seen from the figure, the common characteristic of the general reflection (the same as in Fig. 9-4) is that the distance from any point P to be reflected to the reflection mirror (plane, line, or point) is equal to that from the mirror to the image



(*a*) Reflection through a plane (P_0, $\hat{\mathbf{r}}$, $\hat{\mathbf{s}}$)



(*b*) Reflection through a line (P_1, P_2)



(*c*) Reflection through a point P_r

FIGURE 9-5
General reflection of a curve.

(reflected) point P^*. In the three cases, the triangle OPP^* can be identified and can be used to relate the coordinates of P^* to those of P as follows:

$$\mathbf{P}^* = \mathbf{P} + \mathbf{Q} \tag{9.15}$$

where $\mathbf{Q}$ is the vector connecting P and P^*.

In order to obtain $\mathbf{P}^*$ from Eq. (9.15), the vector $\mathbf{Q}$ must be evaluated first from the given geometry. This is when the difference between the three cases comes into play. Consider first the case shown in Fig. 9-5*a*. The plane is defined by a point P_0 and two unit vectors $\hat{\mathbf{r}}$ and $\hat{\mathbf{s}}$. The vector $\mathbf{Q}$ is perpendicular to the plane and its magnitude is double the normal distance between P and the plane. Utilizing Eq. (6.34), the normal distance D can be obtained (compare Figs. 9-5*a* and 6-26) and we can write

$$\mathbf{Q} = 2D\hat{\mathbf{n}} \tag{9.16}$$

where $\hat{\mathbf{n}}$ is the surface unit normal to the plane and can be calculated from Eq. (6.27) by using $\hat{\mathbf{r}}$ and $\hat{\mathbf{s}}$ in place of $(\mathbf{P}_1 - \mathbf{P}_0)$ and $(\mathbf{P}_2 - \mathbf{P}_0)$ respectively in the equation.

The vector $\mathbf{Q}$ in the case of reflection through a general line shown in Fig. 9-5*b* can be evaluated by using the results of Example 5.9, case *b*. Comparing Fig. 5-20*b* and Fig. 9-5*b*, we can rewrite Eq. (5.22) to give

$$\mathbf{Q} = -2\{(\mathbf{P} - \mathbf{P}_1) - [(\mathbf{P} - \mathbf{P}_1) \cdot \hat{\mathbf{n}}_1]\hat{\mathbf{n}}_1\} \tag{9.17}$$

The factor -2 is used in Eq. (9.17) because the magnitude of $\mathbf{Q}$ is twice the normal distance between P and the line and its direction is opposite to $\hat{\mathbf{n}}_2$, shown in Fig. 5-20*b*.

In the case of reflection through a general point P_r, the vector $\mathbf{Q}$ can easily be written as (refer to Fig. 9-5*c*)

$$\mathbf{Q} = 2(\mathbf{P}_r - \mathbf{P}) \tag{9.18}$$

The effect of reflection on tangent vectors to a curve depends on each individual given case. Only for reflection through the origin or a general point can we write

$$\mathbf{P}^{*\prime} = -\mathbf{P}' \tag{9.19}$$

that is, tangent vectors reverse directions and their magnitudes remain unchanged. For other cases, only appropriate component(s) reverse directions.

Example 9.3. Show that reflecting the geometric representation of a Hermite cubic spline is identical to its pointwise reflection.

Solution. Let us consider the case of reflection through a general point. Utilizing Eqs. (9.15), (9.18), and (9.19), we can write the geometry vector $\mathbf{V}^*$ of the spline as

$$\begin{aligned}\mathbf{V}^* &= [2\mathbf{P}_r - \mathbf{P}_0 \quad 2\mathbf{P}_r - \mathbf{P}_1 \quad -\mathbf{P}'_0 \quad -\mathbf{P}'_1]^T \\ &= -[\mathbf{P}_0 \quad \mathbf{P}_1 \quad \mathbf{P}'_0 \quad \mathbf{P}'_1]^T + [2\mathbf{P}_r \quad 2\mathbf{P}_r \quad 0 \quad 0]^T = -\mathbf{V} + \mathbf{F}\end{aligned} \tag{9.20}$$

where $\mathbf{F} = [2\mathbf{P}_r \quad 2\mathbf{P}_r \quad 0 \quad 0]$. Substituting Eq. (9.20) into Eq. (9.6), we get

$$\mathbf{P}^*(u) = \mathbf{U}^T[M_H](-\mathbf{V} + \mathbf{F}) = -\mathbf{U}^T[M_H]\mathbf{V} + \mathbf{U}^T[M_H]\mathbf{F} = -\mathbf{P} + \mathbf{U}^T[M_H]\mathbf{F} \tag{9.21}$$

Substituting Eq. (5.84) into the second term of Eq. (9.21) and reducing we obtain

$$\mathbf{P}^*(u) = -\mathbf{P} + 2\mathbf{P}_r = \mathbf{P} + 2(\mathbf{P}_r - \mathbf{P}) = \mathbf{P} + \mathbf{Q} \tag{9.22}$$

This equation is the same as combining Eqs. (9.15) and (9.18). Therefore, reflecting the end conditions of a Hermite cubic spline results in the pointwise reflection of the spline. Proofs of other cases of reflections can follow the same outlines discussed in this example and are left as exercises to the reader.

9.2.4 Rotation

Rotation is an important form of geometrical transformation. It enables users to view geometric models from different angles and also helps many geometric operations. For example, it can be used to create entities arranged in a circular pattern (circular arrays) by creating the entity once and then rotating/copying it to the desired positions on the circumference. In a similar fashion, rotation can be used to construct axisymmetric geometric models.

Rotation has a unique characteristic that is not shared by translation, scaling, or reflection—that is, noncommutativeness. The final position and orientation of an entity after going through two subsequent translations, scalings, or reflections are independent of the order of the operations, that is, commutative. On the contrary, two subsequent rotations of the entity about two different axes produce two different configurations of the entity depending on the order of the rotations. The reader can verify this by simply marking an edge of a box and then rotating it about two of its other edges and observe the final configuration of the marked edge. The same experiment can be performed on a CAD/CAM system. Interpretations of these experiments are covered in Sec. 9.2.6.

9.2.4.1 ROTATION ABOUT COORDINATE SYSTEM AXES. Rotating a point a given angle θ about the X, Y, or Z axis is sometimes referred to as rotation about the origin. A convention for choosing signs of angles of rotations must be established. In this book, the right-hand convention is chosen. Therefore, a rotation angle about a given axis is positive in a counterclockwise sense when viewed from a point on the positive portion of the axis toward the origin.

To develop the rotational transformation of a point (or a vector) about one of the principal axes, let us consider the rotation of point P a positive angle θ about the Z axis, as shown in Fig. 9-6. This case is equivalent to two-dimensional rotation of a point in the XY plane about the origin. The final position of P after rotation is shown as point P^*. Equation (9.1) or (9.2) can be written here by relating the coordinates of P^* to those of P as follows:

$$\begin{aligned} x^* &= r\cos(\theta + \alpha) = r\cos\alpha\cos\theta - r\sin\alpha\sin\theta \\ y^* &= r\sin(\theta + \alpha) = r\sin\alpha\cos\theta + r\cos\alpha\sin\theta \\ z^* &= z \end{aligned} \tag{9.23}$$

where $r = |\mathbf{P}| = |\mathbf{P}^*|$. To eliminate the angle θ from Eqs. (9.23), we can write (refer to the trigonometry in Fig. 9-6)

$$x = r\cos\alpha \qquad y = r\sin\alpha \tag{9.24}$$

Substituting Eqs. (9.24) into (9.23) gives

$$\begin{aligned} x^* &= x\cos\theta - y\sin\theta \\ y^* &= x\sin\theta + y\cos\theta \\ z^* &= z \end{aligned} \tag{9.25}$$

Rewriting Eqs. (9.25) in a matrix form gives

$$\begin{bmatrix} x^* \\ y^* \\ z^* \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{9.26}$$

or

$$\mathbf{P}^* = [R_Z]\mathbf{P} \tag{9.27}$$

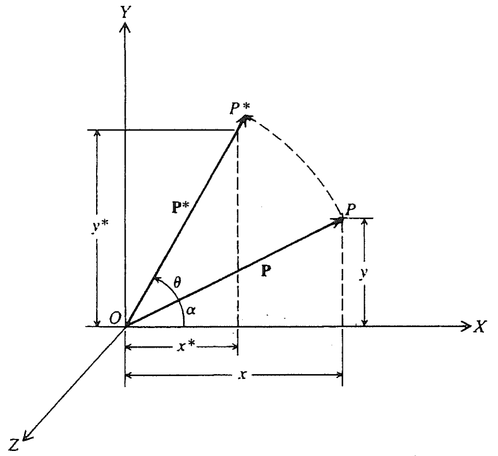


FIGURE 9-6
Rotation of a point about the Z axis.

where

$$[R_Z] = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{9.28}$$

Similarly, we can prove that matrices for rotations about X and Y axes are given by

$$[R_X] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \tag{9.29}$$

$$[R_Y] = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{9.30}$$

Thus, in general, we can write

$$\mathbf{P}^* = [R]\mathbf{P} \tag{9.31}$$

where $[R]$ is the appropriate rotational matrix. Equations (9.28) to (9.30) show some popular forms of $[R]$. Other forms are covered in the next two sections.

The columns of the rotation matrix $[R]$ have some useful characteristics. If we substitute the unit vector $[1 \quad 0 \quad 0]^T$ in the X direction into Eq. (9.26), we obtain the first column of $[R_Z]$ as the components of the transformed unit vector. This implies that if we rotate the unit vector in the X direction and angle θ about the Z axis, the first column of $[R_Z]$ gives the coordinates of the transformed unit vector. Similarly, the second and third columns of $[R_Z]$ are the new coordinates of the unit vectors $[0 \quad 1 \quad 0]^T$ and $[0 \quad 0 \quad 1]^T$ in the Y and Z directions respectively after rotating them the same angle θ. Therefore, the columns of a rotation matrix $[R]$ represent the unit vectors that are mutually orthogonal in a right-hand system, that is, $\mathbf{C}_1 \times \mathbf{C}_2 = \mathbf{C}_3$, $\mathbf{C}_2 \times \mathbf{C}_3 = \mathbf{C}_1$, and $\mathbf{C}_3 \times \mathbf{C}_1 = \mathbf{C}_2$, where $\mathbf{C}_1$, $\mathbf{C}_2$, and $\mathbf{C}_3$ are the first, second, and third columns of $[R]$ respectively. From linear algebra, a matrix with orthonormal columns is an orthogonal matrix and its inverse is equal to its transpose.

The effect of rotation on tangent vectors of a curve can be obtained from Eq. (9.31) as

$$\mathbf{P}^{*\prime} = [R]\mathbf{P}' \tag{9.32}$$

Thus, for a Hermite cubic spline, it is easily seen that $\mathbf{V}^* = [R]\mathbf{V}$. Therefore, the rotation of the spline about a given axis is equivalent to rotating its end conditions about the same axis.

9.2.4.2 TWO-DIMENSIONAL ROTATION ABOUT AN ARBITRARY AXIS. The rotation of a point, or an entity in general, about an axis passing through an arbitrary point that is not the origin occurs when one point rotates about another one. In fact, the rotation of a point about the origin covered in the previous section is considered a special case of this problem we are about to solve. Rotation of a point or an entity about a point is useful in simulations of mechanisms, linkages, and robotics where links or members must rotate about their respective joints.

Figure 9-7 shows the rotation of point P about point P_1 in the XY plane. Figure 9-7a shows P and its rotation, to its final position P^*, an angle θ about an axis parallel to the Z axis and passing through P_1. In order to develop the rotation matrix correctly for this case, we can use Eq. (9.27) to rotate the vector $(\mathbf{P} - \mathbf{P}_1)$(not $\mathbf{P}$) about P_1 to obtain $(\mathbf{P}^* - \mathbf{P}_1)$ (not $\mathbf{P}^*$). Thus, we can write

$$\mathbf{P}^* - \mathbf{P}_1 = [R_Z](\mathbf{P} - \mathbf{P}_1) \tag{9.33}$$

Rearranging Eq. (9.33) gives

$$\mathbf{P}^* = [R_Z](\mathbf{P} - \mathbf{P}_1) + \mathbf{P}_1 \tag{9.34}$$

Equation (9.34) can also be obtained by considering the rotation of point P about P_1 instead of considering the rotation of the vector $(\mathbf{P} - \mathbf{P}_1)$ about P_1 as we did. From this point of view, the rotation of P about P_1 can be achieved in three steps as shown in Fig. 9.7b to d. In the first step, translate P_1 to the origin O. In this position, we refer to point P_1 as P_{1t}. Also, translate point P to P_t by the translation vector $-\mathbf{P}_1$ as shown in Fig. 9.7b. Therefore,

$$\mathbf{P}_t = \mathbf{P} - \mathbf{P}_1 \tag{9.35}$$

In the second step, rotate P_t, in the XY plane, the angle θ about the origin, as shown in Fig. 9.7c. Consequently, Eq. (9.27) gives

$$\mathbf{P}_t^* = [R_Z]\mathbf{P}_t = [R_Z](\mathbf{P} - \mathbf{P}_1) \tag{9.36}$$

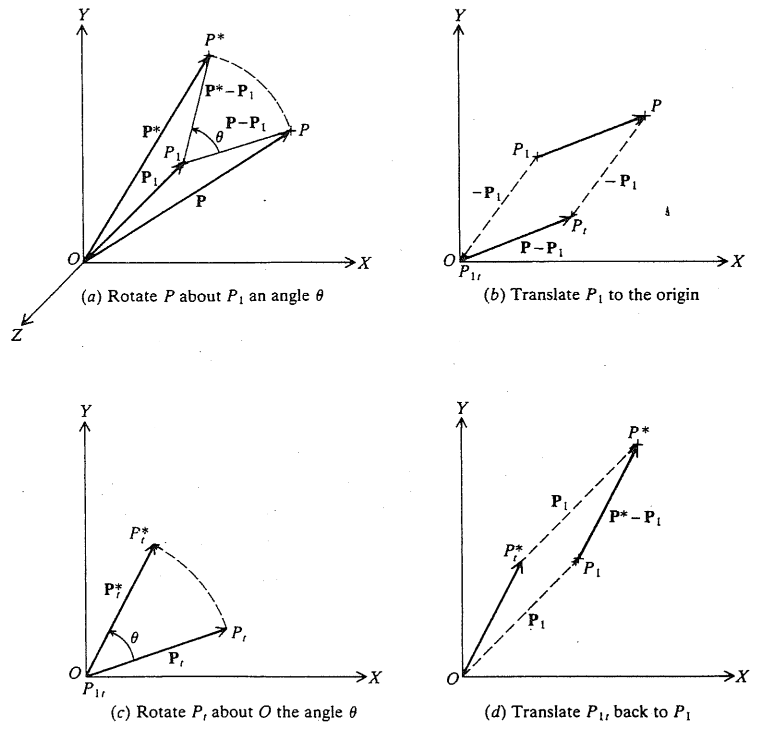

(a) Rotate P about P_1 an angle θ
(b) Translate P_1 to the origin
(c) Rotate P_t about O the angle θ
(d) Translate P_{1t} back to P_1

FIGURE 9-7
Two-dimensional rotation of a point about an arbitrary axis.

In the last step, translate points P_{1t} and P_t back to their original positions P_1 and P respectively by the translation vector $\mathbf{P}_1$. This would require translating point P_t^* by the same vector to the position P^*, as shown in Fig. 9.7*d*. Thus,

$$\mathbf{P}^* = \mathbf{P}_t^* + \mathbf{P}_1 = [R_Z](\mathbf{P} - \mathbf{P}_1) + \mathbf{P}_1 \tag{9.37}$$

Equation (9.37) is the same as Eq. (9.34). Equation (9.37) applies to two-dimensional rotations in the XZ or YZ plane by replacing $[R_Z]$ by $[R_Y]$ or $[R_X]$ respectively.

9.2.4.3 THREE-DIMENSIONAL ROTATION ABOUT AN ARBITRARY AXIS. Points undergoing rigid-body rotation describe arcs in a plane perpendicular to a fixed line, the axis of rotation. In planar two-dimensional rotation the axis is always perpendicular to the XY plane; that is, parallel to the Z axis. Consequently, the axis is completely defined by its intersection with the XY plane (the origin O in Fig. 9-6 and point P_1 in Fig. 9-7) and the orientation of the axis is implicitly defined (along the Z axis) and does not appear as a parameter in the

rotation matrix $[R]$. As a result, the angle of rotation θ is the only transformation parameter required to completely define a two-dimensional rotation and, therefore, the corresponding $[R]$. It will be shown later in this section that two-dimensional rotation is a special case of three-dimensional rotation.

In the general spatial (three-dimensional) case, rotation is not constrained to the XY plane and the axis of rotation may be oriented in any direction. Therefore, the orientation of the axis must be incorporated into the rotation matrix in addition to the angle of rotation. If we define the orientation by the unit vector $\hat{\mathbf{n}}$ (Fig. 9-8), Eq. (9.1) can be written as

$$\mathbf{P}^* = f(\mathbf{P}, \hat{\mathbf{n}}, \theta) \tag{9.38}$$

In this equation, it is assumed that the axis of rotation passes through the origin. If it does not, then a similar development to that presented in Sec. 9.2.4.2 should be followed. Equation (9.38) is derived below and recast in a matrix form for three-dimensional rotation about an arbitrary axis. Two cases are considered: the axis passes through the origin and the axis is in an arbitrary location.

Figure 9-8 shows the three-dimensional rotation of a point P an angle θ about an arbitrary axis that passes through the origin. The positions of the point before and after rotation are P and P^* respectively. The orientation of the axis of rotation is defined by the unit vector $\hat{\mathbf{n}}$ such that

$$\hat{\mathbf{n}} = n_x\hat{\mathbf{i}} + n_y\hat{\mathbf{j}} + n_z\hat{\mathbf{k}} = \cos\alpha\hat{\mathbf{i}} + \cos\beta\hat{\mathbf{j}} + \cos\gamma\hat{\mathbf{k}} \tag{9.39}$$

where $n_x = \cos\alpha$, $n_y = \cos\beta$, and $n_z = \cos\gamma$ are the direction cosines of $\hat{\mathbf{n}}$. If the axis of rotation is defined as a line connecting the origin O and any point, say A, on it (see Fig. 9-8), then $n_x = x_A/|\mathbf{A}|$, $n_y = y_A/|\mathbf{A}|$, and $n_z = z_A/|\mathbf{A}|$, where x_A, y_A, and z_A are the coordinates of point A and $|\mathbf{A}| = \sqrt{x_A^2 + y_A^2 + z_A^2}$.

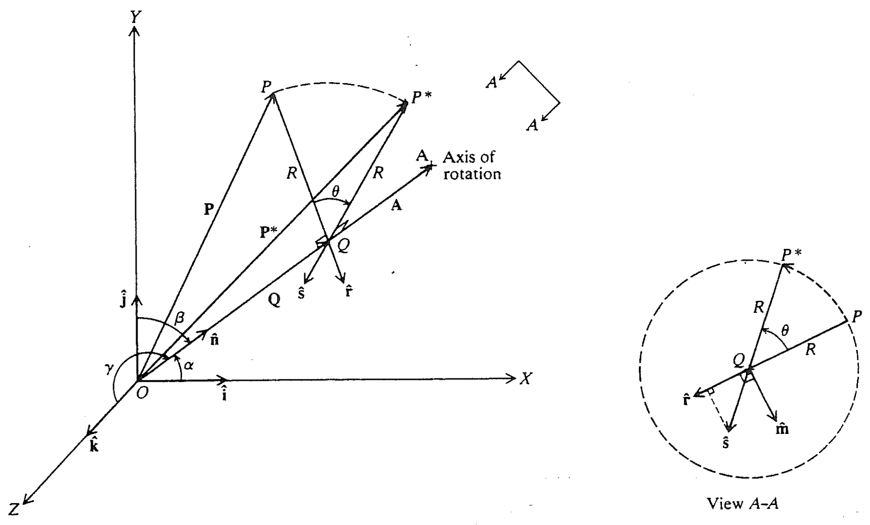


FIGURE 9-8
Three-dimensional rotation of a point about an arbitrary axis.

The rotation of P about the axis OA defines a circle whose plane is perpendicular to OA. Its center is point Q which is the intersection between the axis and the plane. Its radius is R which is the perpendicular distance between P and OA in any position, that is, $R = PQ = P^*Q$. The angle of rotation θ is chosen in Fig. 9-8 to be positive according to the agreed-upon convention adopted for two-dimensional rotation. View A-A shows θ counterclockwise, that is, positive, if the observer is placed at A-A, that is, on the positive portion of the axis.

In order to facilitate the development, let us define the directions of the lines PQ and P^*Q by the unit vectors $\hat{\mathbf{r}}$ and $\hat{\mathbf{s}}$ respectively, as shown in Fig. 9-8. From the figure, it is obvious that the final position vector $\mathbf{P}^*$ of point P is the resultant of three vectors, that is,

$$\mathbf{P}^* = \mathbf{P} + \mathbf{PQ} + \mathbf{QP}^* \tag{9.40}$$

where the notation $\mathbf{PQ}$ indicates a vector going from point P to point Q. Utilizing $\hat{\mathbf{r}}$, $\hat{\mathbf{s}}$, and R, Eq. (9.40) can be written as

$$\mathbf{P}^* = \mathbf{P} + R\hat{\mathbf{r}} - R\hat{\mathbf{s}} \tag{9.41}$$

The remainder of the development that follows centers around expressing $\hat{\mathbf{r}}$ and $\hat{\mathbf{s}}$ in terms of $\mathbf{P}$, $\hat{\mathbf{n}}$, θ, that is, in terms of the desired rotation parameters. Utilizing the right triangle OPQ, we can write

$$\mathbf{PQ} = \mathbf{Q} - \mathbf{P} \tag{9.42}$$

Observing that $\mathbf{Q}$ is the component of $\mathbf{P}$ along the axis of rotation, we can write

$$\mathbf{Q} = (\mathbf{P} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} \tag{9.43}$$

Substituting Eq. (9.43) into (9.42) and dividing the result by R (the magnitude of $\mathbf{PQ}$) gives

$$\hat{\mathbf{r}} = \frac{(\mathbf{P} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} - \mathbf{P}}{R} \tag{9.44}$$

In order to express $\hat{\mathbf{s}}$ in terms of $\mathbf{P}$ and $\hat{\mathbf{n}}$, we need to introduce the intermediate unit vector $\hat{\mathbf{m}}$ shown in view A-A in Fig. 9-8. The vector is chosen to be perpendicular to $\hat{\mathbf{r}}$ and lies in the plane of the circle; thus it is also perpendicular to $\hat{\mathbf{n}}$. Utilizing the cross-product definition of two vectors, we can write

$$\hat{\mathbf{m}} = \hat{\mathbf{n}} \times \hat{\mathbf{r}} \tag{9.45}$$

The unit vector $\hat{\mathbf{s}}$ can now be written in terms of its components in the $\hat{\mathbf{r}}$ and $\hat{\mathbf{m}}$ directions as

$$\hat{\mathbf{s}} = \cos\theta\hat{\mathbf{r}} + \sin\theta\hat{\mathbf{m}} \tag{9.46}$$

Substituting Eq. (9.45) into (9.46) and substituting the result together with Eq. (9.44) into (9.41), we obtain

$$\mathbf{P}^* = (\mathbf{P} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} + [\mathbf{P} - (\mathbf{P} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}] \cos\theta + (\hat{\mathbf{n}} \times \mathbf{P}) \sin\theta - \hat{\mathbf{n}} \times (\mathbf{P} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} \sin\theta \tag{9.47}$$

The last term in the above equation is equal to zero because it represents the cross product of two collinear vectors $\hat{\mathbf{n}}$ and $(\mathbf{P} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$ (also $\mathbf{Q}$). Thus we have

$$\mathbf{P}^* = (\mathbf{P} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} + [\mathbf{P} - (\mathbf{P} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}] \cos\theta + (\hat{\mathbf{n}} \times \mathbf{P}) \sin\theta \tag{9.48}$$

To write Eq. (9.48) in matrix form, we can write the following:

$$\mathbf{P} \cdot \hat{\mathbf{n}} = xn_x + yn_y + zn_z = [n_x \quad n_y \quad n_z] \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{9.49}$$

$$\begin{aligned} \hat{\mathbf{n}} \times \mathbf{P} &= \begin{vmatrix} \hat{\mathbf{i}} & \hat{\mathbf{j}} & \hat{\mathbf{k}} \\ n_x & n_y & n_z \\ x & y & z \end{vmatrix} = (n_y z - n_z y)\hat{\mathbf{i}} + (n_z x - n_x z)\hat{\mathbf{j}} + (n_x y - n_y x)\hat{\mathbf{k}} \\ &= \begin{bmatrix} 0 & -n_z & n_y \\ n_z & 0 & -n_x \\ -n_y & n_x & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \end{aligned} \tag{9.50}$$

Substituting Eqs. (9.49) and (9.50) into Eq. (9.48) and rearranging, we get

$$\mathbf{P}^* = \left\{ (1 - \cos\theta) \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} [n_x \quad n_y \quad n_z] + \cos\theta \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \sin\theta \begin{bmatrix} 0 & -n_z & n_y \\ n_z & 0 & -n_x \\ -n_y & n_x & 0 \end{bmatrix} \right\} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{9.51}$$

or

$$\mathbf{P}^* = [R]\mathbf{P} \tag{9.52}$$

After reducing Eq. (9.51) further, $[R]$ becomes

$$[R] = \begin{bmatrix} n_x^2 \,v\theta + c\theta & n_x n_y \,v\theta - n_z \,s\theta & n_x n_z \,v\theta + n_y \,s\theta \\ n_x n_y \,v\theta + n_z \,s\theta & n_y^2 \,v\theta + c\theta & n_y n_z \,v\theta - n_x \,s\theta \\ n_x n_z \,v\theta - n_y \,s\theta & n_y n_z \,v\theta + n_x \,s\theta & n_z^2 \,v\theta + c\theta \end{bmatrix} \tag{9.53}$$

where $c\theta = \cos\theta$, $s\theta = \sin\theta$, and $v\theta = \text{versine } \theta = 1 - \cos\theta$.

The general rotation matrix $[R]$ given by Eq. (9.53) has two important characteristics. First, it is skew symmetric because the third term in Eq. (9.51) is skew symmetric. Second, its determinant $|R|$ is equal to one. In general, $|R| = 1$ for any rotation matrix that describes rotation about the origin. The reader can verify this fact for Eqs. (9.28) to (9.30) and (9.53).

The rotation matrices given by Eqs. (9.28) to (9.30) can now be seen as special cases of Eq. (9.53) as follows. If the axis of rotation is the Z axis, then its orientation is given by $\hat{\mathbf{n}} = \hat{\mathbf{k}}$, that is, $n_x = n_y = 0$ and $n_z = 1$. Substituting these values into Eq. (9.53) gives Eq. (9.28). Similarly, the X and Y axes of rotation are given by $\hat{\mathbf{n}} = \hat{\mathbf{i}}$ ($n_x = 1$, $n_y = n_z = 0$) and $\hat{\mathbf{n}} = \hat{\mathbf{j}}$ ($n_x = n_z = 0$, $n_y = 1$) respectively and Eqs. (9.29) and (9.30) can be easily obtained from Eq. (9.53).

We now return to the case of three-dimensional rotation about an arbitrary axis that does not pass through the origin. This case is conceptually similar to the two-dimensional case covered in Sec. 9.2.4.2 and its development follows exactly the same steps. Therefore, Eq. (9.37) is applicable for the three-dimensional case after replacing $[R_Z]$ by $[R]$ given by Eq. (9.53), that is,

$$\mathbf{P}^* = [R](\mathbf{P} - \mathbf{P}_1) + \mathbf{P}_1 \qquad (9.54)$$

Here the point P_1 can be any point in space and is not restricted to the XY plane as in the two-dimensional case.

Example 9.4. Prove that if a point to be rotated about a given axis of rotation lies on the axis, the point does not change position in space and, therefore, its coordinates do not change.

Solution. Figure 9-9 shows this problem. It is expected that P and P^* are identical and, therefore, $\mathbf{P} = \mathbf{P}^*$ regardless of the angle of rotation θ. Substituting Eq. (9.53) into Eq. (9.52) and expanding the result, we obtain for the x coordinate:

$$x^* = (n_x^2 \, v\theta + c\theta)x + (n_x n_y \, v\theta - n_z \, s\theta)y + (n_x n_z \, v\theta + n_y \, s\theta)z \qquad (9.55)$$

If point P lies on the axis of rotation, then we can write (see Fig. 9-9)

$$x = |\mathbf{P}|n_x \qquad y = |\mathbf{P}|n_y \qquad z = |\mathbf{P}|n_z \qquad (9.56)$$

where $|\mathbf{P}|$ is the magnitude of $\mathbf{P}$. Substituting Eq. (9.56) into (9.55) and reducing the result, we obtain

$$x^* = |\mathbf{P}|[n_x \, v\theta(n_x^2 + n_y^2 + n_z^2) + n_x \, c\theta] \qquad (9.57)$$

Using the identity $n_x^2 + n_y^2 + n_z^2 = 1$, Eq. (9.57) becomes

$$x^* = |\mathbf{p}|[n_x(1 - \cos\theta) + n_x \cos\theta] = |\mathbf{P}|n_x = x \qquad (9.58)$$

Similarly, we can prove $y^* = y$ and $z^* = z$.

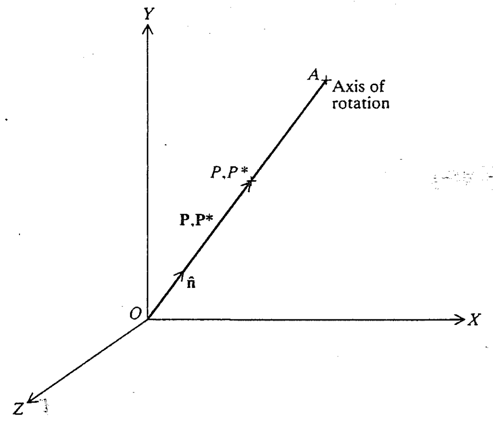


FIGURE 9-9
Rotation of a point about an axis passing through it and through the origin.

This example has useful practical implications. Typically, CAD/CAM systems let users define axes of rotations by inputting endpoints. If the axis of rotation happens to be an entity of a geometric model to be rotated, the coordinates of the endpoints of that entity should stay the same before and after rotation. The user can utilize the "verify entity" command available on the system before and after the rotation to display the coordinates and compare. If there are small differences, they usually result from the round-off errors. Coordinates should be the same within the given significant digits of the computer system used.

9.2.5 Homogeneous Representation

The various rigid-body geometric transformations have been developed in the previous section. Equations (9.3), (9.9), (9.13), and (9.52) represent translation, scaling, mirroring, and rotation respectively. While the last three equations are in the form of matrix multiplication, translation takes the form of vector addition. This makes it inconvenient to concatenate transformations involving translation. Equation (9.37) shows an example. It is desirable, therefore, to express all geometric transformations in the form of matrix multiplications only. Representing points by their homogeneous coordinates provides an effective way to unify the description of geometric transformations as matrix multiplications.

Homogeneous coordinates have been used in computer graphics and geometric modeling for a long time. With their aid, geometric transformations are customarily embedded into graphics hardware to speed their execution. Homogeneous coordinates are useful for other applications. They are useful to obtain perspective views of geometric models. The subjects of projective geometry, mechanism analysis and design, and robotics utilize them quite often in development and formulation. In addition, homogeneous coordinates remove many anomalous situations encountered in cartesian geometry such as representing points at infinity and the nonintersection of parallel lines. Also, they greatly simplify expressions defining rational parametric curves and surfaces.

In homogeneous coordinates, an n-dimensional space is mapped into $(n + 1)$-dimensional space; that is, a point (or a position vector) in n-dimensional space is represented by $(n + 1)$ coordinates (or components). In three-dimensional space, a point P with cartesian coordinates (x, y, z) has the homogeneous coordinates (x^*, y^*, z^*, h) where h is any scalar factor $\neq 0$. The two types of coordinates are related to each other by the following equations:

$$x = \frac{x^*}{h} \qquad y = \frac{y^*}{h} \qquad z = \frac{z^*}{h} \qquad (9.59)$$

Equations (9.59) are based on the fact that if the cartesian coordinates of a given point P are multiplied by a scalar factor h, P is scaled to a new point P^* and the coordinates of P and P^* are related by the above equations. Figure 9-10 shows point P scaled by the two factors h_1 and h_2 to produce the two new points P_1^* and P_2^* respectively. These two points could be interpreted in two different

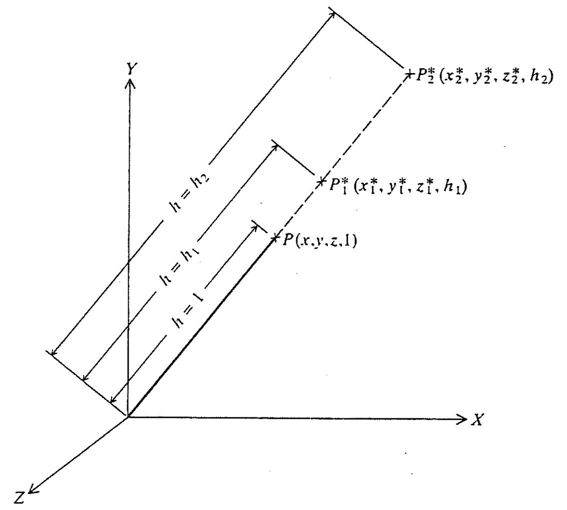

Y
X
Z
$P_2^*(x_2^*, y_2^*, z_2^*, h_2)$
$P_1^*(x_1^*, y_1^*, z_1^*, h_1)$
$P(x, y, z, 1)$
$h = h_2$
$h = h_1$
$h = 1$

FIGURE 9-10
Homogeneous coordinates of point P.

ways. From a cartesian-coordinates point of view, Eq. (9.12) can be used with $s = h_1$ and h_2. Once the cartesian coordinates of P_1^* and P_2^* are calculated, their relationships to P do not exist any more. Moreover, the three points still belong to the cartesian space. From a homogeneous-coordinates point of view, the original point P is represented by $(x, y, z, 1)$ and P_1^* and P_2^* are represented by $(x_1^*, y_1^*, z_1^*, h_1)$ and $(x_2^*, y_2^*, z_2^*, h_2)$ respectively according to Eqs. (9.59). More importantly, the three points belong to the homogeneous space, with the cartesian coordinates obtained when $h = 1$, and the relationship between P and P_1^* or P_2^* is maintained through the proper value of h. As a matter of fact, any two homogeneous-coordinates points P_1^* and P_2^* represent the same cartesian point if and only if $h_2 = ch_1$, for any nonzero constant c. Therefore, there is no unique homogeneous representation of a point. For the purpose of geometric transformations, the scalar factor h used in Eqs. (9.59) is taken to be unity to avoid unnecessary division.

The translation transformation given by Eq. (9.3) can now be written as a matrix multiplication by adding the component of 1 to each vector in the equation and using a 4×4 matrix as follows:

$$[x^* \quad y^* \quad z^* \quad 1]^T = \begin{bmatrix} 1 & 0 & 0 & x_d \\ 0 & 1 & 0 & y_d \\ 0 & 0 & 1 & z_d \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{9.60}$$

or

$$\mathbf{P}^* = [D]\mathbf{P} \tag{9.61}$$

where $[D]$ is the translation matrix shown in Eq. (9.60). While scaling, reflection, and rotation are already expressed in terms of matrix multiplication, their corresponding matrices are changed from 3×3 into 4×4 by adding a column and a row of zero elements except the fourth, which is 1. Thus, the scaling matrix [Eq. (9.10)] becomes

$$[S] = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.62}$$

Similarly, the reflection matrix [Eq. (9.14)] becomes

$$[M] = \begin{bmatrix} \pm 1 & 0 & 0 & 0 \\ 0 & \pm 1 & 0 & 0 \\ 0 & 0 & \pm 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.63}$$

and the rotation matrix [Eq. (9.53)] becomes

$$[R] = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.64}$$

Equations (9.28) to (9.30) can be rewritten in a similar fashion.

To illustrate the convenience gained from the homogeneous representation, Eq. (9.37) can be written as

$$\mathbf{P}^* = [D_1][R_Z][D_2]\mathbf{P} = [T]\mathbf{P} \tag{9.65}$$

where

$$[D_1] = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.66}$$

$$[R_Z] = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.67}$$

$$[D_2] = \begin{bmatrix} 1 & 0 & 0 & x_1 \\ 0 & 1 & 0 & y_1 \\ 0 & 0 & 1 & z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.68}$$

and

$$[T] = [D_1][R_Z][D_2] = \begin{bmatrix} \cos\theta & -\sin\theta & x_1(\cos\theta - 1) - y_1 \sin\theta & 0 \\ \sin\theta & \cos\theta & x_1 \sin\theta + y_1(\cos\theta - 1) & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (9.69)$$

A closer look at the transformation matrices given in Eqs. (9.61) to (9.64) shows that they can all be embedded into one 4 × 4 matrix. This matrix takes the form:

$$[T] = \left[\begin{array}{ccc|c} t_{11} & t_{12} & t_{13} & t_{14} \\ t_{21} & t_{22} & t_{23} & t_{24} \\ t_{31} & t_{32} & t_{33} & t_{34} \\ \hline t_{41} & t_{42} & t_{43} & t_{44} \end{array}\right] = \left[\begin{array}{c|c} T_1 & T_2 \\ \hline T_3 & 1 \end{array}\right] \qquad (9.70)$$

The 3 × 3 submatrix $[T_1]$ produces scaling, reflection, or rotation. The 3 × 1 column matrix $[T_2]$ generates translation. The 1 × 3 row matrix $[T_3]$ produces perspective projection, covered in Sec. 9.5.2. The fourth diagonal element is the homogeneous-coordinates scalar factor h used in Eq. (9.59) and is chosen to be unity, as mentioned earlier.

Equation (9.70) gives the explicit form of the transformation matrix $[T]$ used in Eq. (9.2). It is usually written for one geometric transformation at a time by using any of Eqs. (9.60) to (9.64). If more than one transformation is desired, the resulting matrices are multiplied to produce the total transformation, as discussed in Sec. 9.2.6 that follows.

While the homogeneous representation and the resulting transformation matrix $[T]$ given by Eq. (9.70) are useful and convenient to think of and write compact equations, a computer program to implement them to transform entities should be carefully designed to avoid wasting time multiplying ones and zeros. As a matter of fact, they may not even be used at all to simplify the related programming logic.

9.2.6 Concatenated Transformations

So far we have concentrated on one-step transformations of points such as rotating or translating a point. However, in practice a series of transformations may be applied to a geometric model. Thus, combining or concatenating transformations are quite useful. Concatenated transformations are simply obtained by multiplying the $[T]$ matrices [Eq. (9.70)] of the corresponding individual transformations. However, because matrix multiplication may not be commutative in all cases, attention must be paid to the order in which transformations are applied to a given geometric model. In general, if we apply n transformations to a point starting with transformation 1, with $[T_1]$, and ending with transformation

n, with $[T_n]$, then the concatenated transformation of the point is given by

$$\mathbf{P}^* = [T_n][T_{n-1}] \cdots [T_2][T_1]\mathbf{P} \qquad (9.71)$$

As an example, consider rotating a point, or its position vector, in the fixed coordinate system XYZ, that is, MCS, by the following rotations in the following order: α about the Z axis, β about the Y axis, and γ about the X axis. Substituting α, β, and γ in Eqs. (9.28), (9.30), and (9.29) respectively and multiplying, we obtain the concatenated transformation matrix as

$$[T] = [T_X][T_Y][T_Z] \qquad (9.72)$$

or

$$\left[\begin{array}{c|c} [R] & 0 \\ \hline 0 & 1 \end{array}\right] = \left[\begin{array}{c|c} [R_X] & 0 \\ \hline 0 & 1 \end{array}\right]\left[\begin{array}{c|c} [R_Y] & 0 \\ \hline 0 & 1 \end{array}\right]\left[\begin{array}{c|c} [R_Z] & 0 \\ \hline 0 & 1 \end{array}\right] \qquad (9.73)$$

or

$$[R] = [R_X][R_Y][R_Z] \qquad (9.74)$$

Expanding the above equation gives

$$[R] = \begin{bmatrix} c\alpha\, c\beta & -s\alpha\, c\beta & s\beta \\ s\alpha\, c\gamma + c\alpha\, s\beta\, s\gamma & c\alpha\, c\gamma - s\alpha\, s\beta\, s\gamma & -c\beta\, s\gamma \\ s\alpha\, s\gamma - c\alpha\, s\beta\, c\gamma & c\alpha\, s\gamma + s\alpha\, s\beta\, c\gamma & c\beta\, c\gamma \end{bmatrix} \qquad (9.75)$$

Example 9.5. Using the concatenated rotations about the axes of the coordinate system shown in Fig. 9-8, rederive Eq. (9.53).

Solution. The basic idea to solve this example is to rotate the axis of rotation OA shown in Fig. 9-8 to coincide with one of the axes, rotate the point P the angle θ about this coincident axis, and finally rotate OA in the opposite direction to its original position. The rotation of OA is achieved in two steps. In effect, this is equivalent to decomposing the rotation about the general axis into three rotations about the principal axes X, Y, and Z. Figure 9-11 shows one possible decomposition where point B is the projection of point A onto the XZ plane. In this decomposition, the following sequence of rotations is followed:

1. Rotate OA and point P about the Y axis an angle $-\phi$ so that OB is collinear with the Z axis, where

$$\tan\phi = \frac{x_A}{z_A} = \frac{x_A/|\mathbf{A}|}{z_A/|\mathbf{A}|} = \frac{n_x}{n_z} \qquad (9.76)$$

This then gives

$$\cos\phi = \frac{n_z}{\sqrt{n_x^2 + n_z^2}} \qquad \sin\phi = \frac{n_x}{\sqrt{n_x^2 + n_z^2}} \qquad (9.77)$$

Thus,

$$\mathbf{P}^* = [R_Y(-\phi)]\mathbf{P} \qquad (9.78)$$

2. Following the above rotation, rotate OA and P about the X axis an angle ψ so that OA is collinear with the Z axis, where

$$\sin\psi = \frac{y_A}{|\mathbf{A}|} = n_y \qquad \cos\psi = \sqrt{1 - n_y^2} \qquad (9.79)$$

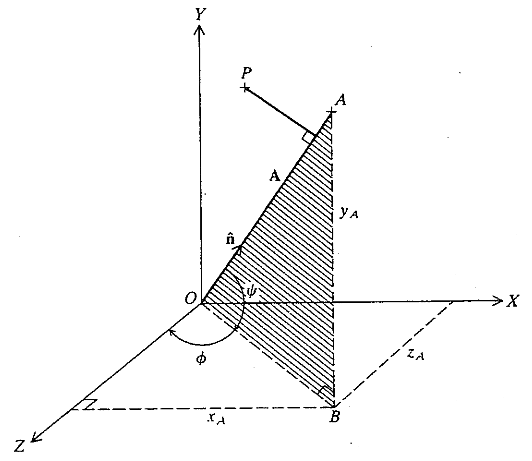


FIGURE 9-11
Decomposition of three-dimensional rotation of a point.

Equation (9.78) becomes

$$\mathbf{P}^* = [R_X(\psi)][R_Y(-\phi)]\mathbf{P} \tag{9.80}$$

2. Rotate point P about the Z axis an angle θ. Note that P is now given by Eq. (9.80). After rotating by the angle θ, it then becomes

$$\mathbf{P}^* = [R_Z(\theta)][R_X(\psi)][R_Y(-\phi)]\mathbf{P} \tag{9.81}$$

4. Reverse step 2, that is, rotate about the X axis an angle $-\psi$. This modifies Eq. (9.81) to

$$\mathbf{P}^* = [R_X(-\psi)][R_Z(\theta)][R_X(\psi)][R_Y(-\phi)]\mathbf{P} \tag{9.82}$$

5. Reverse step 1, that is, rotate about the Y axis an angle ϕ. This modifies Eq. (9.82) to

$$\mathbf{P}^* = [R_Y(\phi)][R_X(-\psi)][R_Z(\theta)][R_X(\psi)][R_Y(-\phi)]\mathbf{P} \tag{9.83}$$

Equation (9.83) should be the same as Eq. (9.51) but in a different form, and in comparing it with Eq. (9.52), we can write

$$[R] = [R_Y(\phi)][R_X(-\psi)][R_Z(\theta)][R_X(\psi)][R_Y(-\phi)] \tag{9.84}$$

If the matrix multiplications in the above equation are performed and the result is reduced, Eq. (9.53) will be obtained. The reader can carry out the details by using Eqs. (9.28) to (9.30) and using the identity $n_x^2 + n_y^2 + n_z^2 = 1$. If other sequences of decompositions of the rotation are used, the right-hand side of Eq. (9.84) changes but the final result, that is, $[R]$, stays the same. The reader is encouraged to try these sequences.

9.3 MAPPINGS OF GEOMETRIC MODELS

In the previous section, we concerned ourselves with rigid-body transformations of geometric models. Thus, we have discussed transforming a point (or a set of points) belonging to an object into another point (or another set of points), with both points (or sets) described in the same coordinate system. Thus, the model position and orientation change with respect to the origin of the coordinate system which stays unaltered in space. In this section, we think of rigid-body motion and its related matrices as mappings of geometric models between different coordinate systems. This is useful in geometric modeling as transformations (see Example 9.6). Mapping of a point (or a set of points) belonging to an object from one coordinate system to another is defined as changing the description of the point (or the set of points) from the first coordinate system to the second one. Thus, the model position and orientation stays unaltered in space with respect to the origins of both coordinate systems while only the description of such position and orientation changes. This is equivalent to transforming one coordinate system to another.

Mapping can be used in various applications. It is useful during model construction, as discussed in Chap. 3. When the user defines a WCS and creates geometry by inputting coordinates measured in this WCS, the software maps these coordinates to the MCS before storing them in the model database. Mapping is also useful in assemblies or model merging where one or more models, each defined in its own MCS, are combined or merged into a host model. The coordinates of each subassembly or merged model is expressed in terms of the MCS of the host assembly or model via mapping.

The same mathematical forms that we have developed for geometric transformations can be used to map points between coordinate systems. However, the interpretation of these forms is different. The problem of mapping a point from one coordinate system to another can be stated as follows. Given the coordinates of a point P measured in a given XYZ coordinate system, find the coordinates of the point measured in another coordinate system, say $X^*Y^*Z^*$, such that

$$\mathbf{P}^* = f(\mathbf{P}, \text{mapping parameters}) \tag{9.85}$$

where $\mathbf{P}$ and $\mathbf{P}^*$ are the position vectors of point P in the XYZ and $X^*Y^*Z^*$ systems respectively. The mapping parameters describe the relationship between the two systems and consist of the position of the origin and orientation of the $X^*Y^*Z^*$ system relative to the XYZ system. Equations (9.1) and (9.85) are the same, but their interpretations differ. Equation (9.85) can be expressed in the matrix form given by Eq. (9.2), where $[T]$ is referred to as the mapping matrix. This matrix describes the position of the origin and the orientation of one coordinate system relative to another one as expressed by Eq. (3.3). We now consider the three possible cases of mapping and develop their corresponding matrices.

9.3.1 Translational Mapping

When the axes of the two coordinate systems are parallel, the mapping is defined to be translational. In Fig. 9-12, the origins of the XYZ and the $X^*Y^*Z^*$ systems are different but their orientations in space are the same. The point P is described by the vectors $\mathbf{P}$ and $\mathbf{P}^*$ in the XYZ and $X^*Y^*Z^*$ systems respectively. The vector $\mathbf{d}$ describes the position of the origin of the former system relative to the

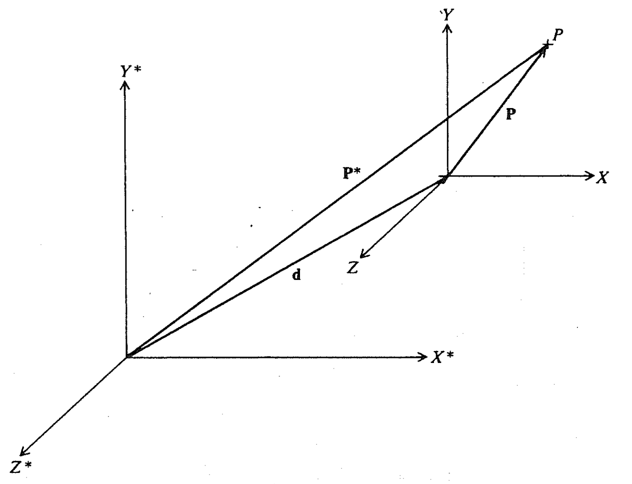


FIGURE 9-12
Translational mapping of a point.

latter. Equation (9.85) can be written exactly as Eq. (9.3) or (9.60) in the homogeneous form.

9.3.2 Rotational Mapping

Figure 9-13 shows rotational mapping between two coordinate systems. The two systems share the same origin and their orientations are different by the angle θ. In this figure, we assume that the XY and X^*Y^* planes are coincident. Utilizing

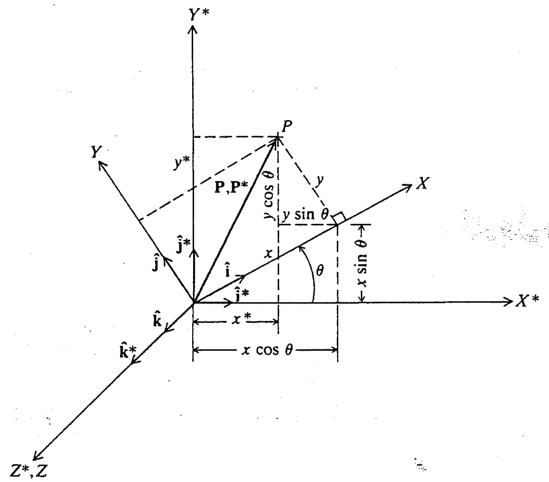


FIGURE 9-13
Rotational mapping of a point.

the trigonometric relationships shown in the figure, Eq. (9.25) can be derived. Therefore, the rotation matrix given by Eq. (9.28) or (9.67) is applicable for rotational mapping. Similarly, Eqs. (9.29), (9.30), (9.53), and (9.75) are applicable for their corresponding rotational mapping cases.

In rotational mapping, it is important to realize that the columns of a rotational matrix $[R]$ can be interpreted to describe the orientations of a given coordinate system in space. If we take the unit vectors $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$, and $\hat{\mathbf{k}}$ in the directions of the axes of the XYZ system as shown in Fig. 9-13, these vectors can be expressed in terms of the $X^*Y^*Z^*$ system as follows:

$$\begin{aligned}\hat{\mathbf{i}} &= \cos\theta\hat{\mathbf{i}}^* + \sin\theta\hat{\mathbf{j}}^* + 0\hat{\mathbf{k}}^*\\ \hat{\mathbf{j}} &= -\sin\theta\hat{\mathbf{i}}^* + \cos\theta\hat{\mathbf{j}}^* + 0\hat{\mathbf{k}}^*\\ \hat{\mathbf{k}} &= \hat{\mathbf{k}}^*\end{aligned} \tag{9.86}$$

Rewriting Eqs. (9.86) in a matrix form, we obtain:

$$[\hat{\mathbf{i}} \quad \hat{\mathbf{j}} \quad \hat{k}]^T = \begin{bmatrix}\cos\theta & -\sin\theta & 0\\ \sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{bmatrix}^T \begin{bmatrix}\hat{\mathbf{i}}^*\\ \hat{\mathbf{j}}^*\\ \hat{\mathbf{k}}^*\end{bmatrix} \tag{9.87}$$

The matrix in the above equation is $[R_Z]^T$ and each of its untransposed columns represents the components of a unit vector. Comparing the columns of the matrix $[R_Z]$ with Eqs. (9.86) shows that the first column represents the direction cosines (components) of the unit vector $\hat{\mathbf{i}}$. The second and the third columns represent the direction cosines of the unit vectors $\hat{\mathbf{j}}$ and $\hat{\mathbf{k}}$ respectively. Therefore, the columns of any rotational matrix $[R]$ represent orthogonal unit vectors. This observation is useful in building $[R]$ from user input, as explained in Example 9.6. The reader can show that the columns of any rotation matrix [Eqs. (9.28) to (9.30) or (9.53) or (9.75)] all have unit magnitude and that they are orthonormal.

9.3.3 General Mapping

The general mapping combines both translational and rotational mappings as shown in Fig. 9-14. In this case, the general mapping matrix $[T]$ is given by Eq. (9.70) with the submatrix $[T_3]$ set to zero, that is,

$$[T] = \left[\begin{array}{ccc|c} r_{11} & r_{12} & r_{13} & x_d\\ r_{21} & r_{22} & r_{23} & y_d\\ r_{31} & r_{32} & r_{33} & z_d\\ \hline 0 & 0 & 0 & 1\end{array}\right] = \left[\begin{array}{c|c}[R] & \mathbf{d}\\ \hline 0 & 1\end{array}\right] \tag{9.88}$$

where $[R]$ and $\mathbf{d}$ are the rotational and translational mapping parts of $[T]$ respectively.

Due to the involvement of two coordinate systems in mapping geometric models, the correct interpretation of Eqs. (9.2) and (9.88) can be explained as follows. In Eq. (9.2), the coordinates of point P are measured in the XYZ coordinate system while its coordinates measured in the $X^*Y^*Z^*$ coordinate system are given by $\mathbf{P}^*$ or $[T]\mathbf{P}$. Therefore, Eq. (9.88) gives the position vector of the origin of the XYZ system as $\mathbf{d}$ and its orientation as $[R]$, both measured in the

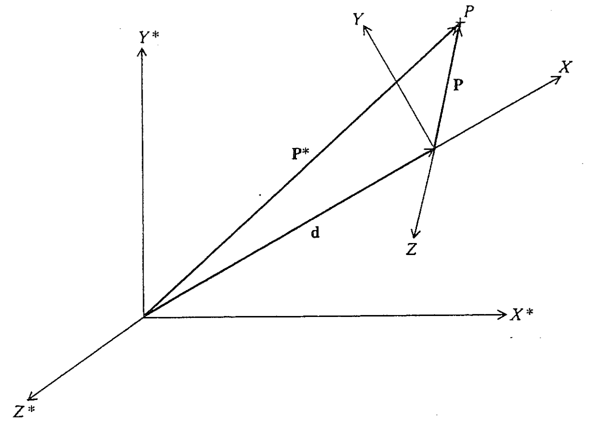


FIGURE 9-14
General mapping of a point.

$X^*Y^*Z^*$ system. The columns of $[R]$ are the components of the unit vectors of the XYZ system (along its axes) measured in the $X^*Y^*Z^*$ system. If one reverses the descriptions, that is, given **P** measured in the $X^*Y^*Z^*$ system, then the inverse of $[T]$ must be used (see Sec. 9.4). To emphasize this point, Eq. (9.2) can be written as ${}^B\mathbf{P} = {}^B_A[T]{}^A\mathbf{P}$, where A and B are the two coordinate systems involved in the mapping process. However, this notation is not used in the book to emphasize that we deal with one matrix $[T]$ for both mapping and transformation, but we interpret it differently.

9.3.4 Mappings as Changes of Coordinate System

In the previous section, 9.3.3, we have presented mappings as changing descriptions of points (or point sets) from one coordinate system to another. In this section we view the same mappings and their related matrices as changes of coordinate system. Let us assume that we are given a set of points described in a given coordinate system. Mapping this coordinate system to another is defined as changing the coordinate system so that the coordinates of the points in the transformed set with respect to the new coordinate system are the same as the coordinates of the points in the original set with respect to the original system.

Mappings as changes of coordinate system are useful in applications such as model merging or building solid models. The local coordinate system of each subassembly in model merging or of each primitive in solid modeling is positioned and oriented properly relative to a reference coordinate system. In both cases, the coordinates of the related point set stays the same with respect to its local coordinate system in its new configuration. The mapping matrix between the reference and the local coordinate systems can be derived as explained in the following section. This matrix can be used to find the coordinates of the point set in its final position, measured in the reference coordinate system as described in Sec. 9.3.3.

Figure 9-15 shows mapping as a change of coordinate system. Let us assume that the $X^*Y^*Z^*$ reference coordinate system and point P^* are changed to the XYZ system and point P respectively such that the coordinates of P^* relative to the $X^*Y^*Z^*$ system are the same as those of P relative to the XYZ system. In other words, the magnitudes and orientations of the vector $\mathbf{P}^*$ relative to $X^*Y^*Z^*$ and of the vector **P** relative to XYZ are the same. The mapping matrix given by Eq. (9.88) can be used to describe the relationship between the $X^*Y^*Z^*$ and XYZ systems. In terms of Fig. 9-15, the columns of this matrix are precisely the coordinates of the unit vectors $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$, and $\hat{\mathbf{k}}$, and the origin O of the XYZ systems, all measured relative to the reference system $X^*Y^*Z^*$. Therefore, this matrix maps the reference coordinate system into a new system (XYZ) whose origin, in reference system coordinates, is the point (x_O, y_O, z_O) and whose x, y, z direction cosines are given by (i_x, i_y, i_z), (j_x, j_y, j_z), and (k_x, k_y, k_z).

Once the mapping matrix $[T]$ given by Eq. (9.88) is established, it can be used to map coordinates of points from the XYZ system to the $X^*Y^*Z^*$ as described in Sec. 9.3.3. Equation (9.2) can be used to find the coordinates of P (vector $\mathbf{P}^{**}$ shown in Fig. 9-15) relative to the $X^*Y^*Z^*$ system if its coordinates (vector **P** in Fig. 9-15) relative to the XYZ system are given. Figure 9-16 shows the various cases of mappings as changes of coordinate system with their corresponding mapping matrices.

Example 9.6. For the geometric model shown in Fig. 9-17, a user defines the XYZ coordinate system shown as the MCS of the model database. The user later defines the two WCSs $\{W\}$ and $\{W_1\}$ shown to be able to construct the two circles whose centers are C and C_1. Both C and C_1 are the centerpoints of the respective faces of the model. Find:

(*a*) The mapping matrices $[T_W]$ and $[T_{W1}]$ that map points from any one of the two WCSs to the MCS.

(*b*) The coordinates of the centers C and C_1 as they are stored in the model database.

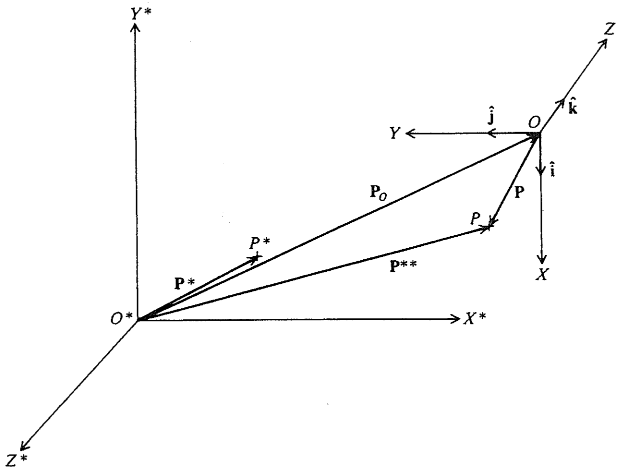
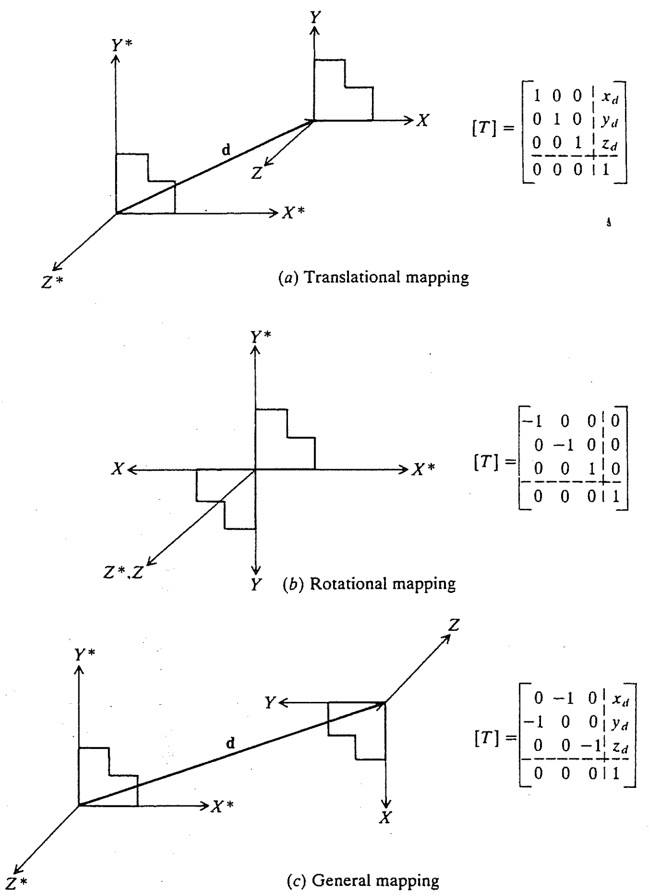


FIGURE 9-15
Mapping as a change of coordinate system.



$$[T] = \left[\begin{array}{ccc|c} 1 & 0 & 0 & x_d \\ 0 & 1 & 0 & y_d \\ 0 & 0 & 1 & z_d \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

(*a*) Translational mapping

$$[T] = \left[\begin{array}{ccc|c} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

(*b*) Rotational mapping

$$[T] = \left[\begin{array}{ccc|c} 0 & -1 & 0 & x_d \\ -1 & 0 & 0 & y_d \\ 0 & 0 & -1 & z_d \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

(*c*) General mapping

FIGURE 9-16
Cases of mappings as changes of coordinate system.

Solution

(*a*) Using a typical CAD/CAM system, the user creates all the lines of the model (assuming a wireframe construction). To create the circle with center C, the user defines the WCS $\{W\}$ by selecting the endpoints P_1, P_2, and P_3 of the existing lines shown such that P_1 defines the origin of the WCS, and the lines P_1P_2 and P_1P_3 define the X_W and Y_W axes respectively. The definition of this WCS is useful for two reasons. First, it defines the plane of the circle. Second, the coordinates of the center C relative to this WCS are obviously (1.5, 2, 0). In order to calculate $[T_W]$, we need to calculate $[R]$ and $\mathbf{d}$ given in Eq. (9.88). The coordinates of points P_1, P_2, and P_3 relative to the MCS are (5, 0, 0), (5, 0, −3), and (3, $2\sqrt{3}$, 0) respectively. The unit vectors $\hat{\mathbf{i}}_W$, $\hat{\mathbf{j}}_W$, and $\hat{\mathbf{k}}_W$ can be calculated as

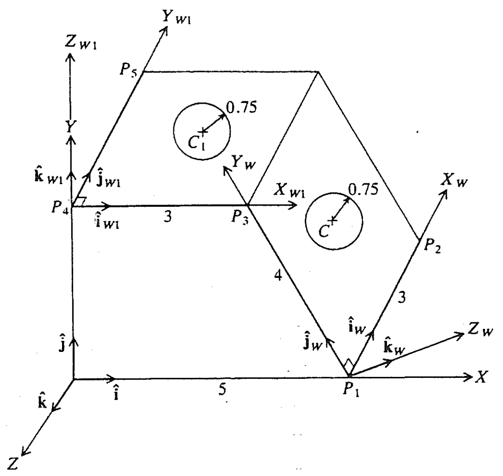


FIGURE 9-17
Utilizing general mapping in model construction.

follows:

$$\hat{\mathbf{i}}_W = \frac{\mathbf{P}_2 - \mathbf{P}_1}{|\mathbf{P}_2 - \mathbf{P}_1|} = -\hat{\mathbf{k}} \qquad (9.89)$$

Notice that this relation between $\hat{\mathbf{i}}_W$ and $\hat{\mathbf{k}}$ is easily seen from Fig. 9-15 without substituting $\mathbf{P}_1$ and $\mathbf{P}_2$ into the above equation.

$$\hat{\mathbf{j}}_W = \frac{\mathbf{P}_3 - \mathbf{P}_1}{|\mathbf{P}_3 - \mathbf{P}_1|} = \frac{1}{4}(-2\hat{\mathbf{i}} + 2\sqrt{3}\hat{\mathbf{j}}) = -0.5\hat{\mathbf{i}} + 0.866\hat{\mathbf{j}} \qquad (9.90)$$

$$\hat{\mathbf{k}}_W = \hat{\mathbf{i}}_W \times \hat{\mathbf{j}}_W = 0.866\hat{\mathbf{i}} + 0.5\hat{\mathbf{j}} \qquad (9.91)$$

Writing Eqs. (9.89) to (9.91) in a matrix form, we obtain

$$[R_W] = \begin{bmatrix} 0 & -0.5 & 0.866 \\ 0 & 0.866 & 0.5 \\ -1 & 0 & 0 \end{bmatrix} \qquad (9.92)$$

Substituting this equation into Eq. (9.88) and knowing that $\mathbf{d} = \mathbf{P}_1$, we obtain

$$[T_W] = \left[\begin{array}{ccc|c} 0 & -0.5 & 0.866 & 5.0 \\ 0 & 0.866 & 0.5 & 0 \\ -1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \qquad (9.93)$$

A similar way can be followed to find $[T_{W1}]$ by using the points P_3 (3, $2\sqrt{3}$, 0), P_4 (0, $2\sqrt{3}$, 0), and P_5 (0, 0, −3). However, by inspection we can see that $\hat{\mathbf{i}}_{W1} = \hat{\mathbf{i}}$, $\hat{\mathbf{j}}_{W1} = -\hat{\mathbf{k}}$, and $\hat{\mathbf{k}}_{W1} = \hat{\mathbf{j}}$. The vector $\mathbf{d}$ is equal to $\mathbf{P}_4$. Therefore,

$$[T_{W1}] = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3.464 \\ 0 & -1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \qquad (9.94)$$

(b) The coordinates of the centers C and C_1 are expressed in terms of the MCS before they are stored in the model database. The coordinates of C relative to the WCS $\{W\}$ are (1.5, 2, 0) and those of C_1 relative to the WCS $\{W_1\}$ are (1.5, 1.5, 0). Their MCS coordinates are given by utilizing Eqs. (9.93) and (9.94) as follows:

$$C = [T_W]\begin{bmatrix} 1.5 \\ 2.0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 4.0 \\ 1.732 \\ -1.5 \\ 1 \end{bmatrix}$$

and

$$C_1 = [T_{W1}]\begin{bmatrix} 1.5 \\ 1.5 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1.5 \\ 3.464 \\ -1.5 \\ 1 \end{bmatrix}$$

The reader can verify these results by constructing the model on a CAD/CAM system and using the "verify entity" command or its equivalence.

9.4 INVERSE TRANSFORMATIONS AND MAPPINGS

Calculating the inverse of transformations and mappings is useful in both theoretical and practical aspects of geometric modeling. For example, using inverse mappings, some CAD/CAM systems (e.g., Computervision and GE Calma) provide their users with functions that can take an existing entity and return its coordinates relative to a given WCS. Normally, the "verify entity" command returns the coordinates relative to the MCS.

All the transformation and mapping matrices developed in Secs. 9.2 and 9.3 have inverses. These matrices have been collected into one general matrix given by Eq. (9.70). Thus, it is appropriate to find the inverse $[T]^{-1}$ of this matrix and then try to relate the result to the various matrices of the previous two sections. Because $[T]$ is partitioned into four submatrices, then $[T]^{-1} = [A]$ also has four submatrices such that

$$[T][A] = [I] \tag{9.95}$$

where $[I]$ is the identity matrix. This equation can be written as

$$\left[\begin{array}{c|c} T_1 & T_2 \\ \hline T_3 & T_4 \end{array}\right]\left[\begin{array}{c|c} A_1 & A_2 \\ \hline A_3 & A_4 \end{array}\right] = \left[\begin{array}{c|c} I & 0 \\ \hline 0 & I \end{array}\right] \tag{9.96}$$

Here we replaced the element $t_{44} = 1$ of $[T]$ given by Eq. (9.70) by the submatrix T_4 of size 1×1. The partitioned form implies four separate matrix equations, two of which are $T_1 A_1 + T_2 A_3 = I$ and $T_3 A_1 + T_4 A_3 = 0$. These can be solved simultaneously for A_1 and A_3. The remaining two equations give A_2 and A_4 and lead to the following result:

$$[T]^{-1} = \left[\begin{array}{c|c} (T_1 - T_2 T_4^{-1} T_3)^{-1} & -T_1^{-1} T_2 (T_4 - T_3 T_1^{-1} T_2)^{-1} \\ \hline -T_4^{-1} T_3 (T_1 - T_2 T_4^{-1} T_3)^{-1} & (T_4 - T_3 T_1^{-1} T_2)^{-1} \end{array}\right] \tag{9.97}$$

The inverse $[T]^{-1}$ given by the above equation does not take full advantage of the inherent structure of $[T]$ itself. First, T_4 is one element equal to 1.

Therefore, $T_4^{-1} = 1$ also. Second, in both transformation and mapping, $[T_3]$ has zero elements, that is, $[T_3] = [0 \quad 0 \quad 0]$. Moreover, $[T_1]$ and $[T_2]$ are the rotation matrix $[R]$ and the translational vector $\mathbf{d}$ in both transformation and mapping. Substituting all these properties into Eq. (9.97) gives

$$[T]^{-1} = \left[\begin{array}{c|c} [R]^{-1} & -[R]^{-1}\mathbf{d} \\ \hline 0 \quad 0 \quad 0 & 1 \end{array}\right] \tag{9.98}$$

We have mentioned in Sec. 9.3.2 that the rotational matrix $[R]$ is orthogonal. Therefore, its inverse is equal to its transpose, that is,

$$[R]^{-1} = [R]^T \tag{9.99}$$

Substituting this equation into (9.98), we obtain the final form of $[T]^{-1}$ as

$$[T]^{-1} = \left[\begin{array}{c|c} [R]^T & -[R]^T\mathbf{d} \\ \hline 0 \quad 0 \quad 0 & 1 \end{array}\right] \tag{9.100}$$

This equation is general and extremely useful to compute the inverse of a homogeneous transformation or mapping. The derivation we just followed to obtain Eq. (9.100) is quite general and other special derivatives of it may exist.

We may now ask ourselves the following questions. Does Eq. (9.100) agree with one's intuition for simple cases? And if it does, how? Take the example of translation. If a translational transformation or mapping is given by Eq. (9.60), the corresponding inverse is obtained by reversing the translational vector $\mathbf{d}$, that is, by negating its components x_d, y_d, and z_d. Here, no rotation is involved and therefore $[R] = [R]^T = [I]$ and Eq. (9.100) yields exactly the same result. Similarly, for scaling $\mathbf{d} = \mathbf{0}$ and $[R]$ is a diagonal matrix, that is, $[S]$ given by Eq. (9.62). In this case, Eq. (9.98) must be used, because $[R]$ is not orthogonal, with the diagonal elements of $[R]^{-1}$ being the reciprocals of those of Eq. (9.62). For rotation, $\mathbf{d} = \mathbf{0}$ and $[R]^T$ to find the inverse is equivalent to negating the angle of rotation, as one would expect. The reader can check this for Eqs. (9.28) to (9.30), (9.53), and (9.75).

One last useful inverse transformation problem is that of determining the direction of the axis of rotation $\hat{\mathbf{n}}$ and the angle of rotation θ from a given rotation matrix $[R]$. In general, the elements of $[R]$ are as shown by the top left submatrix of Eq. (9.88). They are also as shown by Eq. (9.53) in terms of $\hat{\mathbf{n}}$ and θ. To solve for $\hat{\mathbf{n}}$ and θ, we equate both forms, that is,

$$[R] = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = \begin{bmatrix} n_x^2\, v\theta + c\theta & n_x n_y\, v\theta - n_z\, s\theta & n_x n_z\, v\theta + n_y\, s\theta \\ n_x n_y\, v\theta + n_z\, s\theta & n_y^2\, v\theta + c\theta & n_y n_z\, v\theta - n_x\, s\theta \\ n_x n_z\, v\theta - n_y\, s\theta & n_y n_z\, v\theta + n_x\, s\theta & n_z^2\, v\theta + c\theta \end{bmatrix} \tag{9.101}$$

Adding elements (1, 1), (2, 2), and (3, 3) on both sides of this equation to each other and simplifying the result, we obtain

$$\theta = \cos^{-1}\left(\frac{r_{11} + r_{22} + r_{33} - 1}{2}\right) \tag{9.102}$$

Subtracting element (2, 3) from (3, 2) on both sides yields

$$n_x = \frac{r_{32} - r_{23}}{2 \sin \theta} \tag{9.103}$$

Similarly, we can find n_y and n_z, and write

$$\hat{\mathbf{n}} = \left(\frac{1}{2 \sin \theta}\right) \begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \tag{9.104}$$

Equation (9.102) always computes a value of θ between 1 and 180°. Thus, for any axis-angle pair ($\hat{\mathbf{n}}$, θ) there is another pair ($-\hat{\mathbf{n}}$, $-\theta$) that results in the same orientation in space, with the same $[R]$ describing it. Therefore, a choice has to always be made when converting from a rotation matrix into an axis-angle representation. It is also obvious from Eq. (9.101) that the smaller the angle of rotation θ, the closer to zero the off-diagonal elements are, and consequently the more ill-defined the axis of rotation becomes as seen from Eq. (9.104). When $\theta = 0$ or 180°, the axis becomes completely undefined.

Example 9.7. An entity is rotated about the three principal axes of its MCS with equal angles of 45° each. Find the equivalent axis and angle of rotation.

Solution. Substituting $\alpha = \beta = \gamma = 45°$ into Eq. (9.75), we obtain

$$[R] = \begin{bmatrix} 0.5 & -0.5 & 0.707 \\ 0.854 & 0.146 & -0.5 \\ 0.146 & 0.854 & 0.5 \end{bmatrix}$$

Substituting these values into Eqs. (9.102) and (9.104), we obtain

$$\theta = 85.81°$$

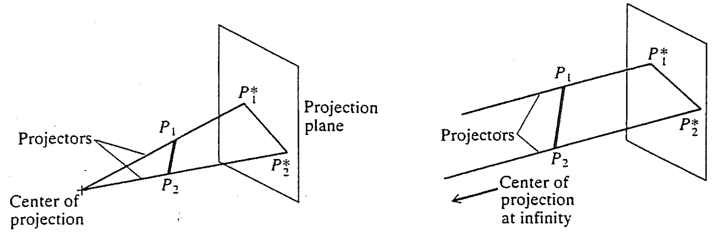
and

$$\hat{\mathbf{n}} = \begin{bmatrix} 0.679 \\ 0.281 \\ 0.679 \end{bmatrix}$$

9.5 PROJECTIONS OF GEOMETRIC MODELS

Databases of geometric models can only be viewed and examined if they can be displayed in various views on a display device or screen. Viewing a three-dimensional model is a rather complex process due to the fact that display devices can only display graphics on two-dimensional screens. This mismatch between three-dimensional models and two-dimensional screens can be resolved by utilizing projections that transform three-dimensional models onto a two-dimensional projection plane. Various views of a model can be generated using various projection planes.

To define a projection, a center of projection and a projection plane must be defined as shown in Fig. 9-18. To obtain the projection of an entity (a line connecting points P_1 and P_2 in the figure), projection rays (called projectors) are



(*a*) Perspective projection (*b*) Parallel projection

FIGURE 9-18
Projection definition.

constructed by connecting the center of projection with each point of the entity. The intersections of these projectors with the projection plane define the projected points which are connected to produce the projected entity. There are two different types of projections based on the location of the center of projection relative to the projection plane. If the center is at a finite distance from the plane, perspective projection results and all the projectors meet at the center. If, on the other hand, the center is at an infinite distance, all the projectors become parallel (meet at infinity) and parallel projection results. Perspective projection is usually a part of perspective, or projective, geometry. Such geometry does not preserve parallelism, that is, no two lines are parallel. Parallel projection is a part of affine geometry which is identical to euclidean geometry. In affine geometry, parallelism is an important concept and therefore is preserved.

Perspective projection creates an artistic effect that adds some realism to perspective views. As can be seen from Fig. 9-18*a*, the size of an entity is inversely proportional to its distance from the center of projection; that is, the closer the entity to the center, the larger its size is. Perspective views are not popular among engineers and draftsmen because actual dimensions and angles of objects, and therefore shapes, cannot be preserved, which implies that measurements cannot be taken from perspective views directly. In addition, perspective projection does not preserve parallelism.

Unlike perspective projection, parallel projection preserves actual dimensions and shapes of objects. It also preserves parallelism. Angles are preserved only on faces of the object which are parallel to the projection plane. There are two types of parallel projections based on the relation between the direction of projection and the projection plane. If this direction is normal to the projection plane, orthographic projection and views result. If the direction is not normal to the plane, oblique projection occurs.

There are two types of orthographic projections. The most common type is the one that uses projection planes that are perpendicular to the principal axes of the MCS of the model; that is, the direction of projection coincides with one of these axes. The front, top, and right views that are used customarily in engineering drawings belong to this type. There are three other views that belong to this type and are typically provided by CAD/CAM systems. These are the bottom, rear, and left views. The other type of orthographic projection uses projection planes that are not normal to a principal axis and therefore show several faces of a model at once. This type is called axonometric projections. They preserve paralelism of lines but not angles. Thus, measurements can be made along each principal axis. Axonometric projections are further divided into trimetric, dimetric, and isometric projections. The isometric projection is the most common axonometric projection. The isometric projection has the useful property that all three principal axes are equally foreshortened, as will be seen in Sec. 9.5.1. Therefore measurements along the axes can be made with the same scale—thus the name: iso for equal, metric for measure. In addition, the normal to the projection plane makes equal angles with each principal axis and the principal axes make equal angles (120° each) with one another when projected onto the projection plane.

We may now ask the following question. How does the common practice of defining views, on CAD/CAM systems, of geometric models relate to both orthographic and isometric projections? Typically, a view definition requires a view origin, viewport (or view window), and a viewing direction, as shown in Fig. 9-19. The view origin defines the location of the origin of the MCS of the model (to be viewed) inside the view window. The viewing direction is the same as the projectors shown in Fig. 9-18*b*. The viewing plane is perpendicular to this direction and is the same as the projection plane. The viewport or view window defines the boundaries against which the view is clipped. Displayed graphics can always be zoomed in or out to scale within the viewport.

A view has a viewing coordinate system (VCS). It is a three-dimensional system with the X_v axis horizontal pointing to the right and the Y_v axis vertical pointing upward, as shown in Fig. 9-19. The Z_v axis defines the viewing direction. The positive Z_v axis has an opposite sense to the viewing direction to keep the VCS a right-handed coordinated system, even though a left-handed system may be more desirable here since its positive Z_v axis is in the direction of the lines of sight emitting from the viewing eye. (This leads to the logical interpretation of

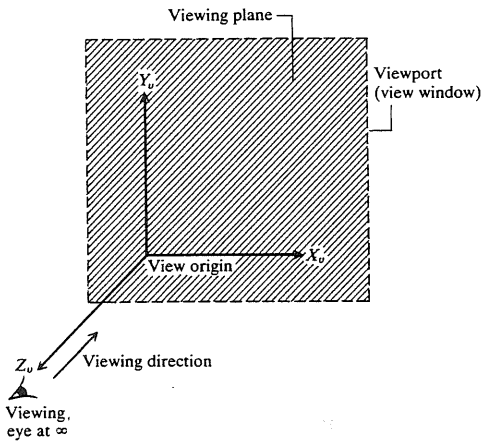


FIGURE 9-19
View definition.

larger z values being further from the viewing eye.) To obtain views of a model, the viewing plane, the $X_v Y_v$ plane, is made coincident with the XY plane of the MCS such that the VCS origin is the same as that of the MCS. Model views now become a matter of rotating the model with respect to the VCS axes until the desired model plane coincides with the viewing plane followed by projecting the model onto that plane. Thus, a view of a model is generated in two steps: rotate the model properly and then project it. These steps are usually performed when the user follows the view definition syntax on a given CAD/CAM system. Figure 9-20 shows the relationship between the MCS and VCS for typical views of a geometric model. We can apply the two-step procedure just described to the figure. For the front view, the XY and $X_v Y_v$ plane are identical. To obtain this view, we simply project the geometry onto the viewing plane. For the top view,

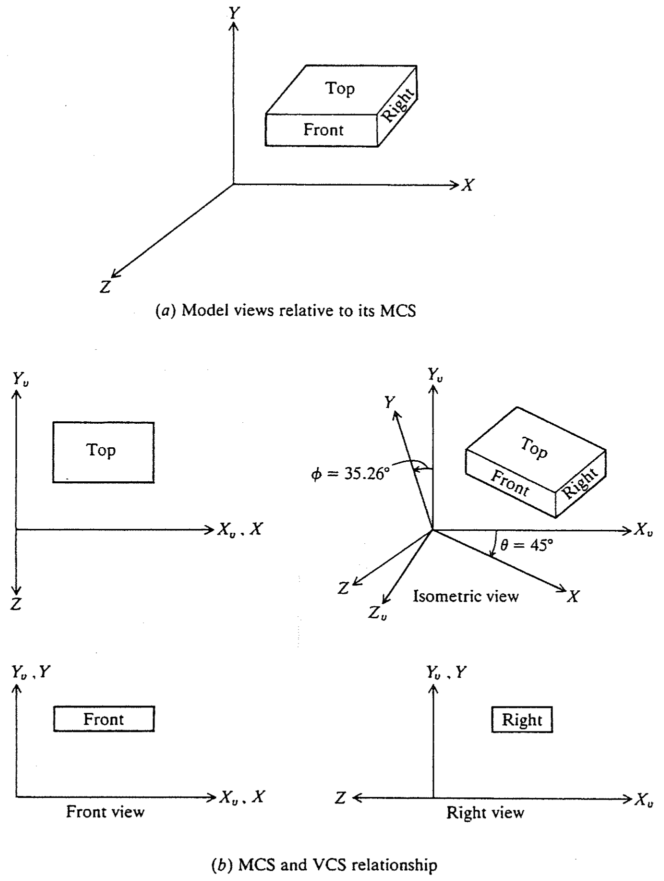


FIGURE 9-20
Relationship between MCS and VCS.

we must rotate the model about the X_v axis by 90° so that the XZ plane coincides with the $X_v Y_v$ plane. The other views can be obtained in a similar fashion. The MCS is shown in Fig. 9-20*b* as if it rotates in space with respect to the VCS exactly in the same way as the model rotates with respect to this VCS. This keeps the relationship between the model and its MCS unchanged in space. Another way to look at this observation is to say that views of a model are obtained by rotating the model and its MCS about the X_v and Y_v axes of the VCS.

The remainder of this section shows the underlying mathematics of projections and how views relate to geometric transformations.

9.5.1 Orthographic Projections

An orthographic projection (view) of a model is obtained by setting to zero the coordinate value corresponding to the MCS axis that coincides with the direction of projection (or viewing) after the model rotation. An orthographic view follows the definition shown in Fig. 9-19. To obtain the front view (see Fig. 9-20*b*), we only (no rotation is needed) need to set $z = 0$ for all the key points of the model. Thus, Eq. (9.70) becomes

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.105}$$

and Eq. (9.2) gives

$$\mathbf{P}_v = [T]\mathbf{P} \tag{9.106}$$

where P_v is the point expressed in the VCS. For the front view, Eq. (9.106) gives $x_v = x$ and $y_v = y$. For the top view, the model and its MCS are rotated by 90° about the X_v axis followed by setting the y coordinate of the resulting points to zero. The y coordinate is the one to set to zero because the Y axis of the MCS coincides with the projection direction. In this case, $[T]$ becomes

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.107}$$

and Eq. (9.106) gives $x_v = x$ and $y_v = -z$. If we use the above equation to transform the MCS itself, the X axis ($y = z = 0$) transforms to $x_v = x$ and the Y axis ($x = z = 0$) transforms to $y_v = -z$. This result agrees with Fig. 9-20*b*. The right view shown in the figure can be obtained by rotating the model and its MCS about the Y_v axis by $-90°$ and setting the x coordinate to zero. Thus,

$$[T] = \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.108}$$

which gives $x_v = -z$ and $y_v = y$.

Examining Eqs. (9.105), (9.107), and (9.108) shows that $[T]$ is a singular matrix with a column of zeros which corresponds to the MCS axis that coincides with the projection or viewing direction. These equations are obtained by rotation followed by setting a coordinate value to zero. They can also be obtained in the reverse order, that is, setting the coordinate value to zero followed by the rotation. Once the viewpoints P_v are generated, they are clipped against the viewport boundaries, and then mapped into the physical device coordinate system (SCS discussed in Chap. 3) to display the view.

To obtain the isometric projection or view, the model and its MCS are customarily rotated an angle $\theta = \pm 45°$ about the Y_v axis followed by a rotation $\phi = \pm 35.26°$ about the X_v axis. These angles have been used for years in conventional manual drafting. In practice, the angle ϕ is taken as $\pm 30°$ to enable the drafting (plastic) triangles in manual construction of isometric views. The values of these angles are based on the fact that the three axes are foreshortened equally in the isometric view. This can be explained as follows. The two rotations give

$$\mathbf{P}_v = [T_x][T_y]\mathbf{P}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{9.109}$$

Applying this equation to transform the unit vectors in the X direction $[1\ 0\ 0\ 1]^T$, in the Y direction $[0\ 1\ 0\ 1]^T$, and in the Z direction $[0\ 0\ 1\ 1]^T$, and ignoring the Z component because we are projecting onto the $z_v = 0$ plane, we obtain respectively:

$$\begin{aligned} x_v &= \cos\phi & y_v &= \sin\phi \sin\theta \\ x_v &= 0 & y_v &= \cos\theta \\ x_v &= \sin\phi & y_v &= -\cos\phi \sin\theta \end{aligned} \tag{9.110}$$

If the three axes are to be foreshortened equally, the magnitudes of the unit vectors given by the above equations must be equal. The first two equations give

$$\cos^2\phi + \sin^2\phi \sin^2\theta = \cos^2\theta \tag{9.111}$$

and the last two equations give

$$\sin^2\phi + \cos^2\phi \sin^2\theta = \cos^2\theta \tag{9.112}$$

Solving Eqs. (9.111) and (9.112) gives $\theta = \pm 45°$ and $\phi = \pm 35.26°$. The signs of the rotation angles θ and ϕ result in four possible orientations of isometric views. Figure 9-20*b* shows the most common orientation where $\theta = -45°$ and $\phi = 35.26°$.

Example 9.8. Find the rotations that are necessary to define the front, top, right, rear, bottom, and left views of a model if the XY plane of the MCS is (*a*) vertical, (*b*) horizontal.

Solution. In Chap. 3, we have obtained these six views as two-dimensional and isometric views, as shown in Figs. 3-47 and 3-48. To generate these views, use the following axes and angles of rotations in the view definition command available on the CAD/CAM system being used.

	Case (*a*)				Case (*b*)			
View	Two dimensions		Isometric		Two dimensions		Isometric	
Front	—	—	Y, X	θ, ϕ	X	-90	X, Y, X	$-90, \theta, \phi$
Top	X	90	X, Y, X	$90, \theta, \phi$	—	—	Y, X	θ, ϕ
Right	Y	-90	Y, Y, X	$-90, \theta, \phi$	Y	-90	Y, Y, X	$-90, \theta, \phi$
Rear	X	180	X, Y, X	$180, \theta, \phi$	X	90	X, Y, X	$90, \theta, \phi$
Bottom	X	-90	X, Y, X	$-90, \theta, \phi$	X	180	X, Y, X	$180, \theta, \phi$
Left	Y	90	Y, Y, X	$90, \theta, \phi$	Y	90	Y, Y, X	$90, \theta, \phi$

In this table, columns with X's and Y's show the order of rotation about the VCS axes and other columns show the corresponding angles of rotation. The angles θ and ϕ are $-45°$ and $35.26°$ as derived previously. The angles shown above are based on the assumption that the front and top views are the default views for cases (*a*) and (*b*) respectively. The reader is encouraged to test the proper case on the available CAD/CAM system.

9.5.2 Perspective Projections

One common way to obtain a perspective view is to place the center of projection along the Z_v axis of the VCS and project onto the $z_v = 0$ or the X_vY_v plane. Figure 9-21 shows this case. The center of projection C is placed at a distance d (measured along the Z_v axis) from the projection plane. In order to find the matrix $[T]$ for the case of perspective projection where the viewing eye lies on the Z_v axis let us develop it from the trigonometry shown in Fig. 9-21. The viewing eye is located at the center C. A new coordinate system called the eye coordinate system (ECS) is introduced relative to the line of sight (see Fig. 9-21). The ECS has an origin located at the same position as the viewing eye. Its X_e and Y_e axes are parallel to the X_v and Y_v axes of the VCS. However, it is a left-handed system. The Z_e axis is taken in the direction of the line of sight. Therefore, points with larger Z_e values are taken to be further from the viewing eye. The ECS is useful in the hidden line and surface removal algorithms (see Chap. 10). The transformation matrix of coordinates of points from the VCS to the ECS or vice versa can be written as

$$[T] = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{9.113}$$

This matrix simply inverts the sign of the z coordinate. In the orthographic views, the ECS is located at infinity. It is obvious that the ECS can be replaced by the

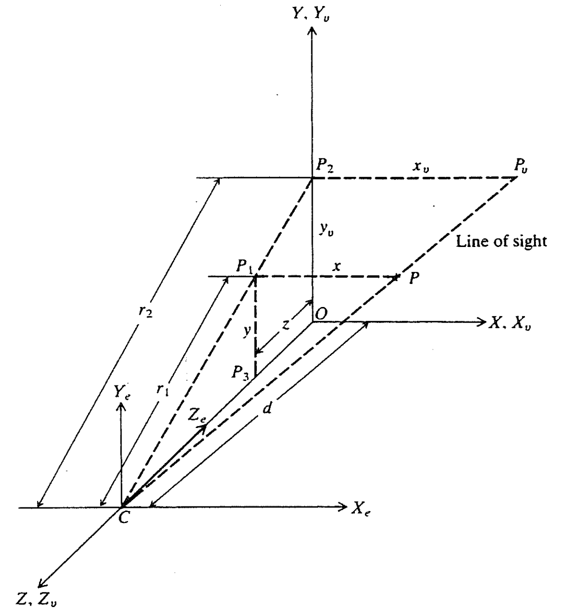


FIGURE 9-21
Perspective projection along the Z_v axis.

VCS. In this case, points with smaller z values are interpreted as being further from the viewing eye.

The figure shows the perspective projection of point P as point P_v. To find the y_v of P_v, the two similar triangles COP_2 and CP_3P_1 give

$$\frac{y_v}{y} = \frac{d}{d-z} = \frac{1}{1-z/d} \tag{9.114}$$

The two similar triangles CP_vP_2 and CPP_1 give x_v of P_v as

$$\frac{x_v}{x} = \frac{r_2}{r_1} = \frac{d}{d-z} = \frac{1}{1-z/d} \tag{9.115}$$

Rearranging Eqs. (9.114) and (9.115) to give y_v and x_v respectively and knowing $z_v = 0$, we can put the result in a homogeneous form as

$$\mathbf{P}_v = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1/d & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{9.116}$$

If this equation is expanded it gives $\mathbf{P}_v = [x \quad y \quad 0 \quad (1 - z/d)]^T$. This would require the division of x and y by $(1 - z/d)$ to obtain the corresponding cartesian coordinates of these homogeneous coordinates. Consequently, Eqs. (9.114) and (9.115) result. Thus, Eq. (9.116) gives the perspective projection onto the $z_v = 0$ plane when the center of projection is placed on the Z_v axis at a distance d from the origin. This result agrees with what was mentioned earlier that the matrix $[T_3]$ [Eq. (9.70)] produces perspective projection. If the center of projection is placed at a general point in space, the other elements of $[T_3]$ will be nonzero.

9.6 DESIGN AND ENGINEERING APPLICATIONS

Geometric transformations and mappings are useful in various design and engineering applications, especially those that are related to kinematics, mechanisms, linkages, and robotics. Most of these applications involve rotations and/or translations of various elements while maintaining the spatial and geometric constraints at the joints that connect these elements. The various configurations of these elements can be used to study their effects on the motion or kinematics of the corresponding geometric models, or they can be used to animate the motion of the models for visualization purposes.

When more than one degree-of-freedom model is to be transformed, such as in a robotics system, kinematic analysis is required to determine the relative motion between the elements. This motion can then be executed via geometric transformations. Due to the repetitive work to transform the elements (such as incremental rotation or translation), programming is usually more efficient than just typing commands. The following example shows how to use rotations in relation to a simple mechanism.

Example 9.9. Figure 9-22 shows a representation of a slider-crank mechanism. Find the locus of point D, the midpoint of the connecting rod BC. What is the angle θ at which the tangent to the locus at D becomes horizontal?

Solution. The idea here is to find enough points on the locus of D and then connect them with a closed B-spline. This is achieved by constructing the mechanism for $\theta = 0$. Then the crank AB is rotated incrementally, say by $\Delta\theta = 10°$, and the mechanism is constructed for each θ (10, 20, ...). At each configuration the position of point D is recorded by simply inserting a point at the origin of the line BC. The resulting positions are then connected with a closed B-spline command to produce the locus. To find the angle θ at which the tangent is horizontal, we construct a few tangents to the locus where they may be horizontal. Using the "measure angle" command, the angle that each tangent makes with the horizontal can be obtained

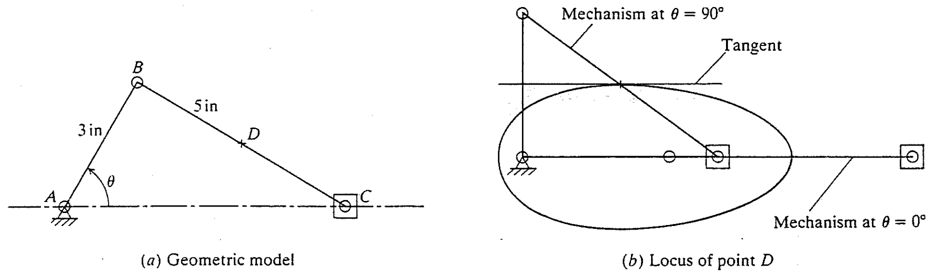


FIGURE 9-22
Slider-crank mechanism.

and compared to zero. The closest angle to zero with an allowable error gives the horizontal tangent and the corresponding angle is the solution. Figure 9-22*b* shows the locus and the tangent within an error of 2.6×10^{-6}°. The angle θ that corresponds to this tangent is 90°.

The reader can extend this method to study the effect of the lengths of the crank and the connecting rod on the locus and the tangent. Programming is useful in this study (refer to Chap. 15).

PROBLEMS

Part 1: Theory

9.1. A general curve such as a Bezier or B-spline is to be translated. Does translating the control points and then generating the curve give the same result as translating the original curve or not? Prove your answer.

9.2. Develop the translational transformation equation for a Hermite bicubic spline surface, a bicubic Bezier surface, and a bicubic B-spline surface. How can you extend the results to a cubic hyperpatch?

9.3. Derive the relationship between a point P and its scaled counterpart P^* if P is scaled uniformly about a given point Q which is not the origin.

9.4. How can a Bezier curve, B-spline curve, Hermite bicubic surface, Bezier surface, and B-spline surface be scaled uniformly?

9.5. Show that Eqs. (9.15) to (9.18) can reduce to Eqs. (9.13) and (9.14); that is, show that reflection relative to a coordinate system is a special case of general reflection.

9.6. Develop the reflection transformation equations for Bezier and B-spline curves and surfaces as well as a Hermite bicubic surface. Carry the developments for the case of reflection through a general point.

9.7. Figure P9-7 shows a cube of length 2 in. The cube is rotated an angle $\theta = 30°$ about the cube diagonal OD. If point B is the midpoint of side AD, find the coordinates of points A, B, and C before and after rotation. Verify your answer by solving the problem on your CAD/CAM system.

9.8. Show how the homogeneous representation can help represent points at infinity and can also be used to force parallel lines to intersect.

9.9. Show that parallel and perpendicular lines transform to parallel and perpendicular lines.

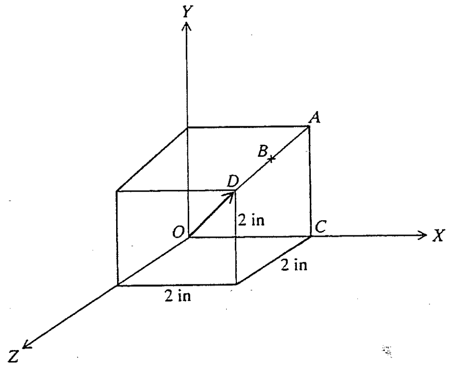


FIGURE P9-7

9.10. Show that the midpoint of a line transforms to the midpoint of the transformed lines.

9.11. A point is rotated about the Z axis by two successive angles θ_1 and θ_2. Show that this is equivalent to rotating the point about the same axis once with an angle $\theta = \theta_1 + \theta_2$.

9.12. Show that:
(*a*) Translation is commutative.
(*b*) Mirror and two-dimensional rotation about the Z axis are not commutative.
(*c*) Scaling and two-dimensional rotation about the Z axis are commutative.
(*d*) Three-dimensional rotations are not commutative.

9.13. Given a point $P = (2, 4, 8)$ and using the homogeneous representation:
(*a*) Calculate the coordinates of the transformed point P^* if P is rotated about the X, Y, and Z axes by angles 30, 60, and 90° respectively.
(*b*) If the point P^* obtained in part (*a*) is to be rotated back to its original position, find the corresponding rotation matrix. Verify your answer.
(*c*) Calculate P^* if P is translated by $\mathbf{d} = 3\hat{\mathbf{i}} - 4\hat{\mathbf{j}} - 5\hat{\mathbf{k}}$ and then scaled uniformly by $s = 1.5$.
(*d*) Calculate the orthographic projection P_v of P.
(*e*) Calculate the perspective projection P_v of P if the center of projection is at a distance $d = 10$ in from the origin along the Z_v axis.

9.14. Given three points P_1, P_2, and P_3 that belong to a geometric model and given three other points Q_1, Q_2, and Q_3, find the transformation matrix $[T]$ that:
(*a*) Transforms P_1 to Q_1.
(*b*) Transforms the direction of the vector $(\mathbf{P}_2 - \mathbf{P}_1)$ into the direction of the vector $(\mathbf{Q}_2 - \mathbf{Q}_1)$.
(*c*) Transforms the plane of the three points P_1, P_2, and P_3 into the plane of Q_1, Q_2, and Q_3.
This problem is sometimes called "three-point" transformation. It is useful to move two geometric models, mainly solids, to coincide with one another or to position entities in a geometric model.

9.15. Figure P9-15 shows the rotation of a point P about an arbitrary axis of rotation that passes through the origin and lies in the XZ plane. Derive the rotation matrix $[R]$ for this case. Verify your answer by substituting the proper values in the general matrix $[R]$ given by Eq. (9.53).

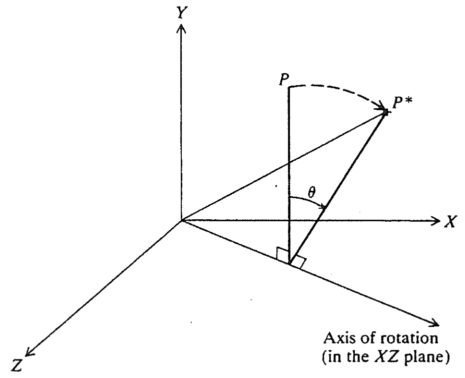


FIGURE P9-15

Part 2: Laboratory

9.16. Find the view definitions required by your CAD/CAM system to define the six two-dimensional views as well as the six isometric views described in Example 9.8.

9.17. Show that three-dimensional rotations are not commutative. Take a sequence of 90° rotations about the three principal axes and permutate them.

9.18. Redo Example 9.5 on your CAD/CAM system. Use $\hat{\mathbf{n}} = 0.5\hat{\mathbf{i}} + 0.707\hat{\mathbf{j}} + 0.5\hat{\mathbf{k}}$ and $P = (1, 3, 5)$.

9.19. A line is connecting the origin of the MCS of a model and a point $P(1, 2, 3)$. Find three different ways to rotate the line so that it coincides with the Z axis of the MCS.

Part 3: Programming

9.20. Write a program to implement Eq. (9.70). Use the form $\mathbf{P}^* = [T]\mathbf{P}$ where $\mathbf{P}$ is a vector of points, tangent vectors, or any other vectors of interest. For example, $\mathbf{P}$ could be four points, that is, $[\mathbf{P}_1 \;\; \mathbf{P}_2 \;\; \mathbf{P}_3 \;\; \mathbf{P}_4]^T$, or it could be $[\mathbf{P}_0 \;\; \mathbf{P}_1 \;\; \mathbf{P}'_0 \;\; \mathbf{P}'_1]^T$ as for the Hermite cubic spline. Write the program for rotation about the Z axis, translation, and scaling uniformly.

9.21. Using the program developed in Prob. 9.20, write a program to translate a cubic spline curve, a Bezier curve, and a B-spline curve.

BIBLIOGRAPHY

Demel, J. T., and M. J. Miller: *Introduction to Computer Graphics*, Brooks/Cole Engineering Division, Monterey, Calif., 1984.
Encarnacao, J., and E. G. Schlechtendahl: *Computer-Aided Design: Fundamental and System Architectures*, Springer-Verlag, New York, 1983.
Faux, I. D., and M. J. Pratt: *Computational Geometry for Design and Manufacture*, John Wiley, New York, 1981.
Foley, J. D., and A. van Dam: *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass., 1982.
Giloi, W. K.: *Interactive Computer Graphics: Data Structures, Algorithms, Languages*, Prentice-Hall, Englewood Cliffs, N.J., 1978.
Groover, M. P., and E. W. Zimmers: *CAD/CAM: Computer-Aided Design and Manufacturing*, Prentice-Hall, Englewood Cliffs, N.J., 1984.
Harrington, S.: *Computer Graphics: A Programming Approach*, McGraw-Hill, New York, 1983.
Harris, D.: *Computer Graphics and Application*, Chapman and Hall, New York, 1984.
Mortenson, M. E.: *Geometric Modeling*, John Wiley, New York, 1985.
Newman, W. M., and R. F. Sproull: *Principles of Interactive Computer Graphics*, 2d ed., McGraw-Hill, New York, 1979.
Park, C. S.: *Interactive Microcomputer Graphics*, Addison-Wesley, Reading, Mass., 1985.
Rogers, D. F., and J. A. Adams: *Mathematical Elements for Computer Graphics*, McGraw-Hill, New York, 1976.