

Fig. 6.2 *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.

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. 6.3. 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. 6.3b.

Another alternative is shown in Fig. 6.3c, where the subtraction is performed first 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.

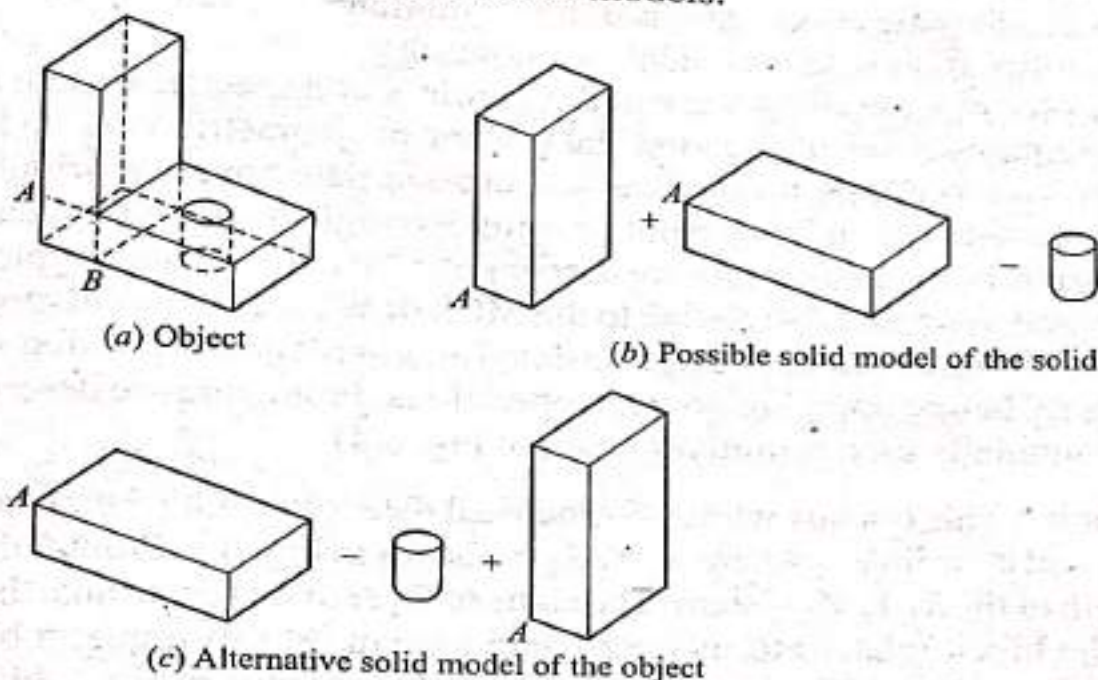


Fig. 6.3 *Nonuniqueness of Solid Model of an Object*

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.

6.3 ≡ 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 denning 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. 6.4):

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. 6.4. 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 negative value of W is displayed as if the block shown in Fig. 6.4 is mirrored about the $Y_L Z_L$ plane.
2. **Cylinder** This primitive is a right circular cylinder whose geometry is denned 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. 6.4). 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_1 and outer radius (or diameter) R_o .

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. 6.4 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.

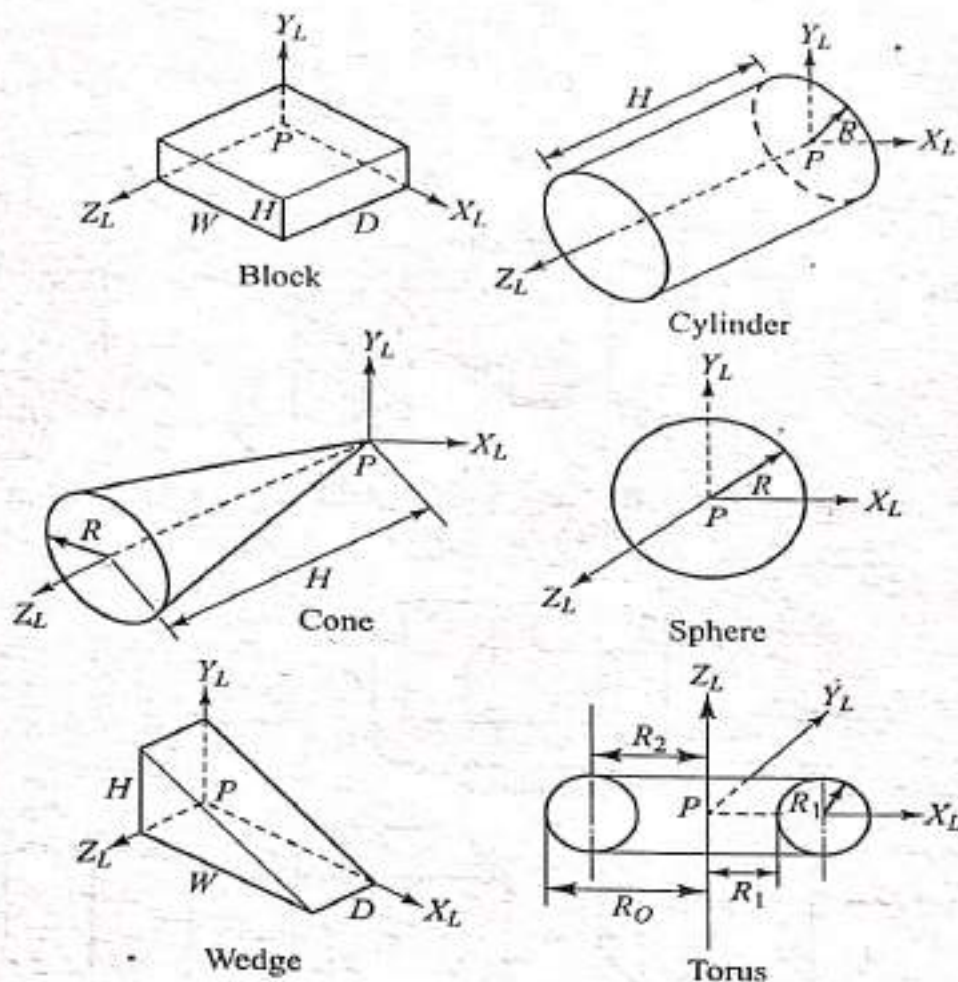


Fig. 6.4 Most Common Primitives

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 \cap) 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 6.5 shows boolean operations of a block A and a cylinder B .

Example 6.1 Create the solid model of the guide bracket shown in Fig. 4.2.

Solution The creation of the solid model of the guide bracket is much simpler than its wireframe and surface models created in Examples 4.1 and 5.1 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. 4.2 is used here). These primitives are

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 from a finite number of projections. Second, the edge representation itself is ambiguous and can correspond to more than one object. Algorithms for dis-ambiguating 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.

6.5 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. 6.2 and geometric closure is introduced in Sec. 6.4. 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. (6.1). They provide good rigorous mathematical foundations for developing and analyzing solids.

6.5.1 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

$$D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \quad (6.4)$$

and
$$D = \{x : x = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \quad (6.5)$$

Equation (6.4) reads as " D is equal to the set of elements 0, 1, 2, 3, 4, 5, 6, 7, 8, 9."

Equation (6.5) 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. (6.5) is sometimes replaced by a vertical bar, that is, $D = \{x | x = 0, 1, \dots, 9\}$. 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 $\not\subseteq$. 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 denoted as a set that has no elements or members. It is designated by the null set symbol \emptyset . 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,

$$cP = \{x : x \notin P\} \quad (6.6)$$

The shaded portion of the Venn diagram in Fig. 6.12a 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,

$$P \cup Q = \{x : x \in P \text{ or } x \in Q\} \quad (6.7)$$

The union is shown in Fig. 6.12b 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,

$$P \cap Q = \{x : x \in P \text{ and } x \in Q\} \quad (6.8)$$

The shaded portion in Fig. 6.12c shows the intersection of P and Q . It is easy to realize that $P \cap W = P$ and $P \cap cP = \emptyset$. Sets that have no common elements are termed disjoint or mutually exclusive.

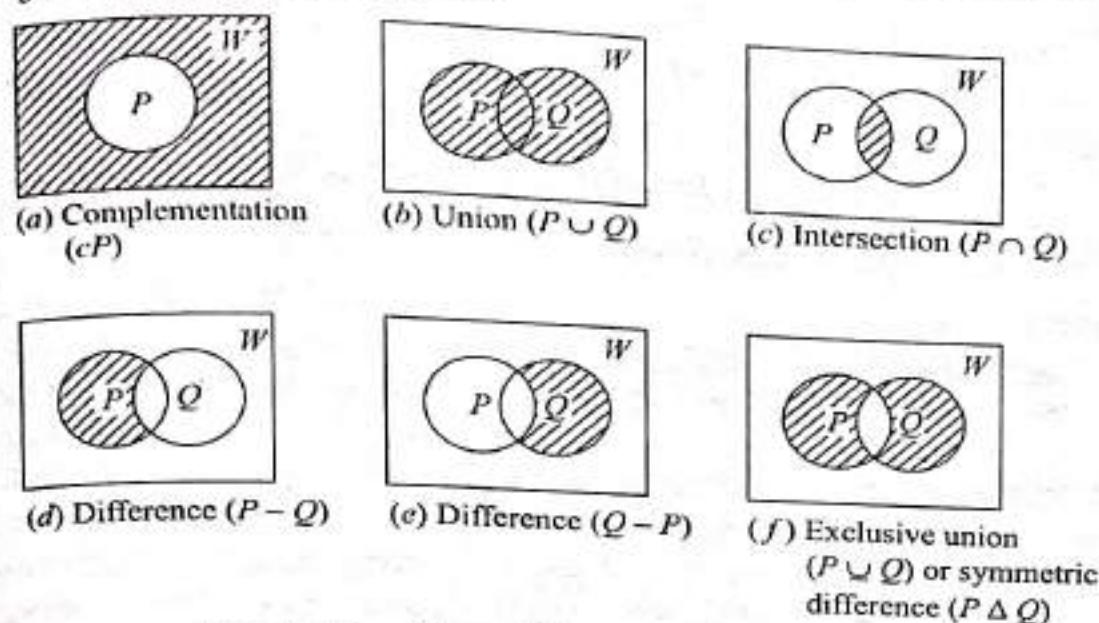


Fig. 6.12 Venn Diagram of Set Algebra

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,

$$P - Q = \{x: x \in P \text{ and } x \notin Q\} \quad (6.9)$$

or

$$Q - P = \{x: x \in Q \text{ and } x \notin P\} \quad (6.10)$$

Figure 6.12d and e shows the difference operator. The difference can also be expressed as

$$P - Q = P \cap cQ \quad (6.11)$$

The exclusive union (also known as symmetric difference) of two sets $P \cup Q$ (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,

$$P \cup Q = \{x: x \in P \cap Q\} \quad (6.12)$$

Figure 6.12f 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 $pq = qp$):

$$P \cup Q = Q \cup P \quad (6.13)$$

$$P \cap Q = Q \cap P \quad (6.14)$$

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

$$P \cup (Q \cap R) = (P \cup Q) \cap R \quad (6.15)$$

$$P \cap (Q \cup R) = (P \cap Q) \cup R \quad (6.16)$$

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

$$P \cup (Q \cap R) = (P \cup Q) \cap (P \cup R) \quad (6.17)$$

$$P \cap (Q \cup R) = (P \cap Q) \cup (P \cap R) \quad (6.18)$$

the idempotence law:

$$P \cap P = P \quad (6.19)$$

$$P \cup P = P \quad (6.20)$$

the involution law:

$$c(cP) = P \quad (6.21)$$

and

$$P \cup \emptyset = P \quad (6.22)$$

$$P \cap W = P \quad (6.23)$$

$$P \cup cP = W \quad (6.24)$$

$$P \cap cP = \emptyset \quad (6.25)$$

$$c(P \cup Q) = cP \cap cQ \quad (6.26)$$

$$c(P \cap Q) = cP \cup cQ \quad (6.27)$$

where Eqs. (6.26) and (6.27) are DeMorgan's laws and Eqs. (6.13) to (6.26) provide the tools necessary to manipulate and simplify sets. For example, using Eqs. (6.13), (6.17) and (6.19) one can prove that the set $(P \cup Q) \cap (P \cap Q)$ is equal to the set $P \cap Q$. 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

$$(a, b) = \{\{a\}, \{a, b\}\} \quad (6.28)$$

Equation (6.28) 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,

$$A \times B = \{(a, b): a \in A \text{ and } b \in B\} \quad (6.29)$$

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

$$A \times B \times C = (A \times B) \times C = \{(a, b, c): a \in A, b \in B, c \in C\} \quad (6.30)$$

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, \dots, a_n) . Ordered pairs and triples are considered 2-tuples and 3-tuples respectively.

Equations (6.29) and (6.30) 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 6.2 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

$$S = \{P_1, P_2, \dots, P_n\} \quad (6.31)$$

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

$$S = \{(x_1, y_1, z_1), (x_2, y_2, z_2), \dots, (x_n, y_n, z_n)\} = \{(x_i, y_i, z_i): 1 \leq i \leq n\} \quad (6.32)$$

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

$$A = \{x_1, x_2, \dots, x_n\} \quad (6.33)$$

$$B = \{y_1, y_2, \dots, y_n\} \quad (6.34)$$

$$C = \{z_1, z_2, \dots, z_n\} \quad (6.35)$$

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

$$P = A \times B \times C = \{(x_i, y_j, z_k): 1 \leq i \leq n, 1 \leq j \leq n, 1 \leq k \leq n\} \quad (6.36)$$

The point set S of the solid given by Eq. (6.32) 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 $A = \{a_1, a_2, \dots, a_n\}$ and $B = \{b_1, b_2, \dots, b_n\}$, then

$$A \otimes B = \{(a_i, b_i): a_i \in A, b_i \in B \text{ and } 1 \leq i \leq n\} \quad (6.37)$$

The ordered cartesian product of three sets is similarly given by

$$A \otimes B \otimes C = (A \otimes B) \otimes C = \{(a_i, b_i, c_i): a_i \in A, b_i \in B, c_i \in C \text{ and } 1 \leq i \leq n\} \quad (6.38)$$

Comparing Eqs. (6.32) and (6.38) shows that the *ordered* cartesian product of the three sets A , B and C given by Eqs. (6.33) to (6.35) 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.

6.5.2 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. (6.3)]. 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. 6.7 is an example of a regular set while Fig. 6.8 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

$$S = kiS \quad (6.39)$$

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 6.13a shows that set S is not regular because $S' = kiS$ 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

$$S = ikS \quad (6.40)$$

This equation states that a set is regular open if the interior of its closure is equal to the original set. Fig. 6.13b shows that S is not regular open because $S' = ikS$ 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.

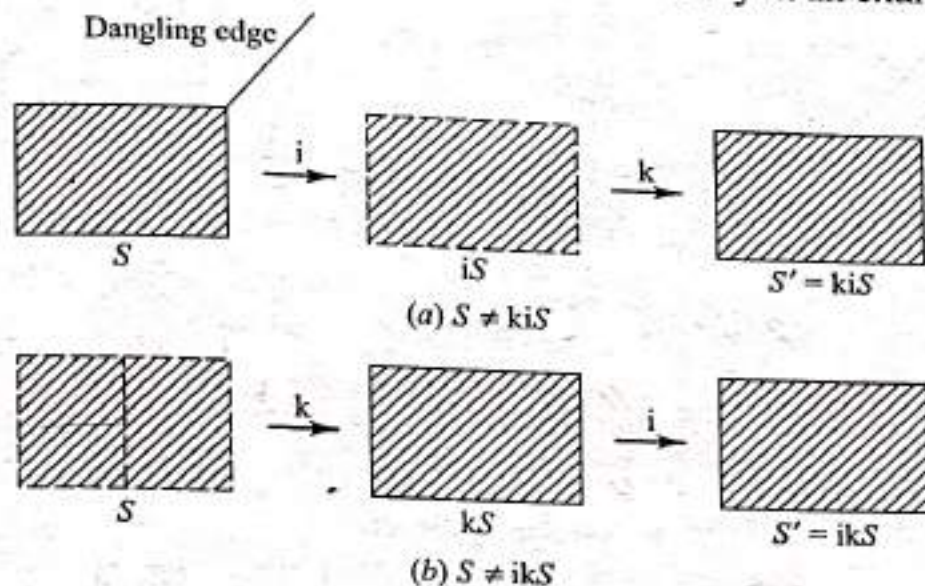


Fig. 6.13 Set Regularity

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

$$P \cup^* Q = ki(P \cup Q) \quad (6.41)$$

$$P \cap^* Q = ki(P \cap Q) \quad (6.42)$$

$$P -^* Q = ki(P - Q) \quad (6.43)$$

$$c^* P = ki(cP) \quad (6.44)$$

where the superscript $*$ to the right of each operator denotes regularization. The sets P and Q used in Eqs. (6.41) to (6.44) 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. (6.41) to (6.44) become

$$X \cup^* Y = X \cup Y \quad (6.45)$$

$$X \cap^* Y = X \cap Y \Leftrightarrow bX \text{ and } bY \text{ do not overlap} \quad (6.46)$$

$$X -^* Y = k\{X - Y\} \quad (6.47)$$

$$c^* X = k(cX) \quad (6.48)$$

If bX and bY overlap in Eq. (6.46), Eq. (6.42) is used and the result is a null object. Figure 6.14 illustrates Eqs. (6.41) to (6.48) geometrically. The figure does not include the complement operation.

Example 6.3 What are the results of applying the regularized set operations to objects A and B shown in Fig. 6.15?

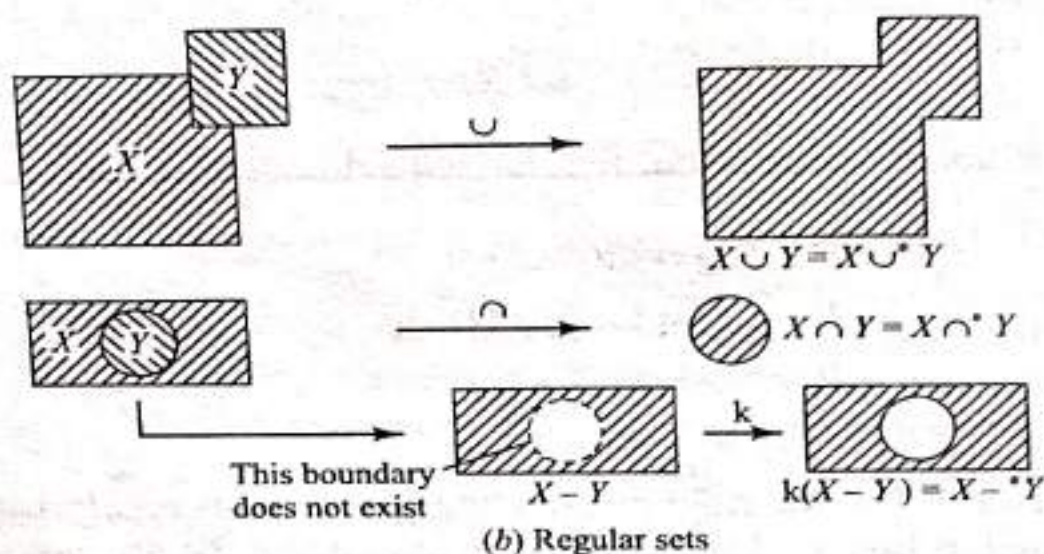
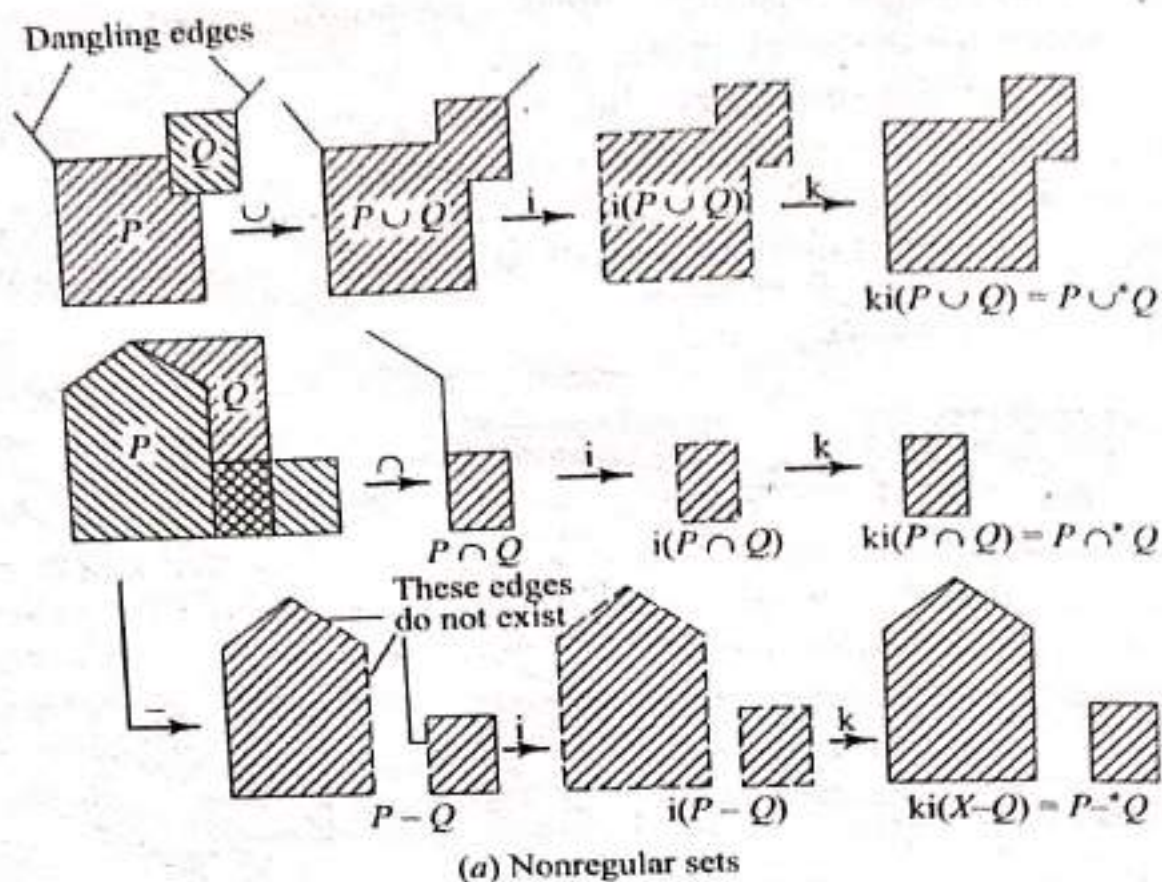


Fig. 6.14 Regularized set operators

Solution The positions of objects A and B shown in Fig. 6.15 are chosen to illustrate some tangency cases of objects. A and B are r -sets. The results of applying Eqs. (6.45) to (6.47) are shown in Table 6.2 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. 6.14. The reader should also try to use these cases to test any available solid modeling package.

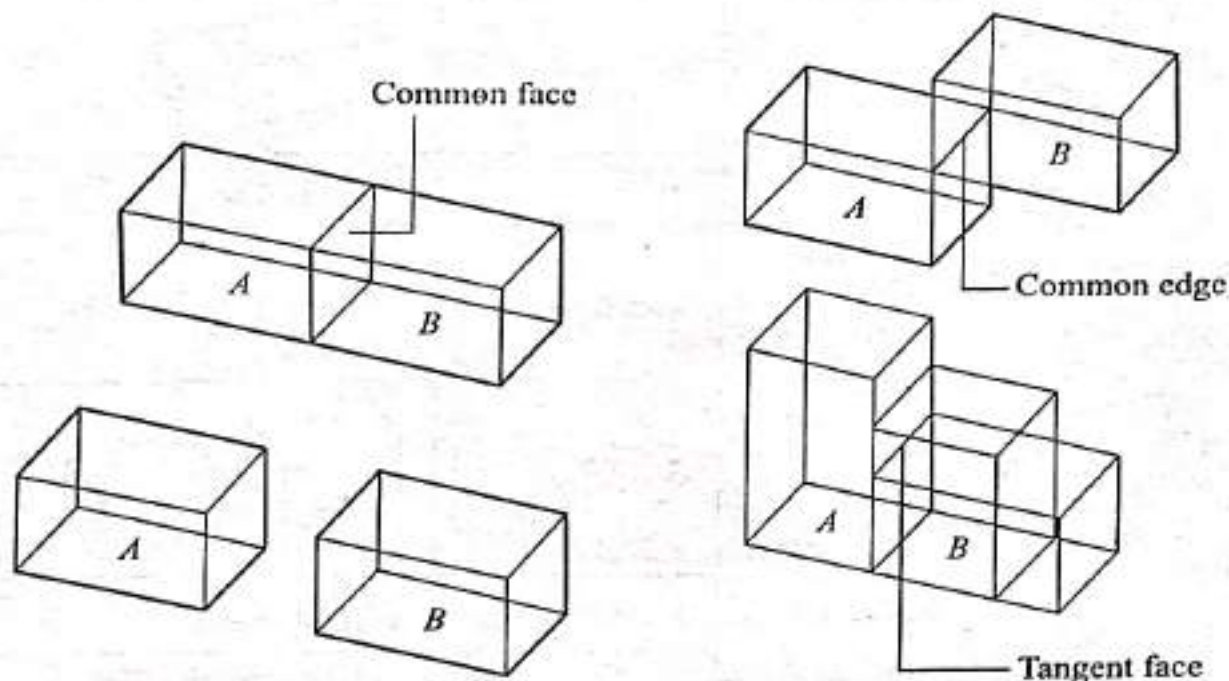


Fig. 6.15 Sample Objects

6.5.3 Set Membership Classification

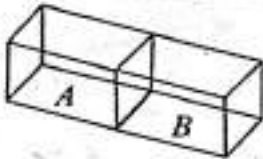
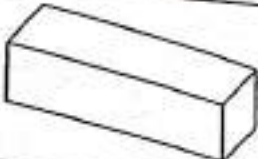

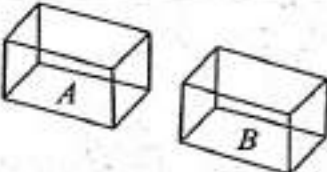
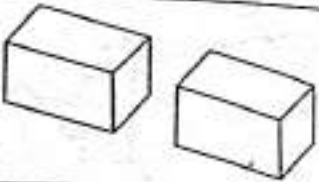

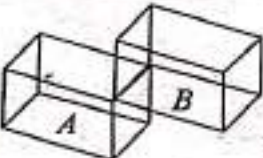
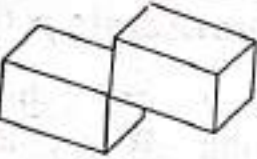

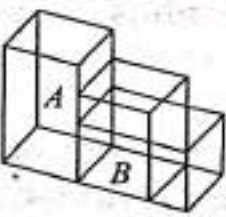
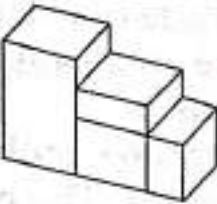

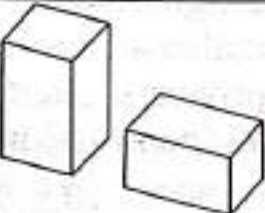
In various geometric problems involving solid models, we are often faced with the following question: given a particular solid, which point, line segment, or a portion of another solid intersects with such a solid? These are all geometric intersection problems. For a point/solid, line (curve)/solid,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

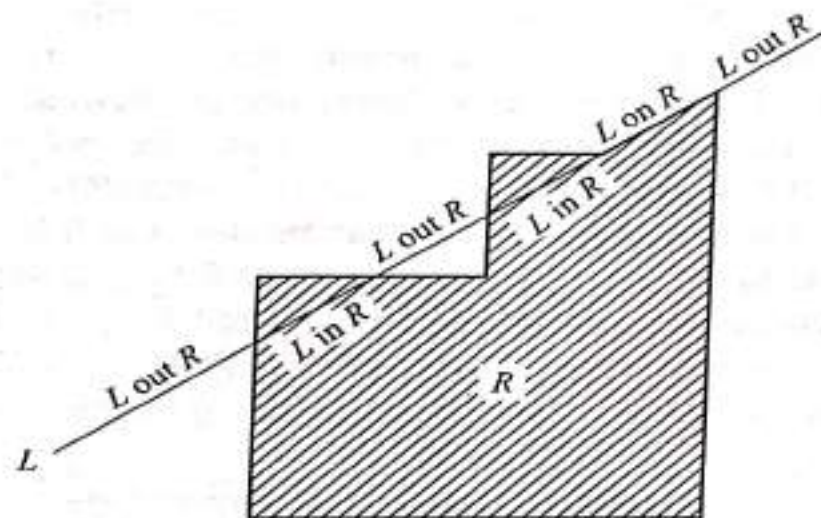
A function called a set membership classification function exists which provides a unifying approach to study the behavior of the candidate set X relative to the reference set S . The function is denoted by $M[.]$ and is defined as

$$M[X, S] = (X \text{ in } S, X \text{ on } S, X \text{ out } S) \quad (6.49)$$

Table 6.2 Results of Example 6.3

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

Equation (6.49) 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 6.16 shows an example of classifying a portion of a line L against the polygon R .



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

Fig. 6.16 Line/polygon Set Membership Classification

The implementation of the classification function given by Eq. (6.49) 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 6.17 shows the B-rep case. The line L is chosen such that no "on" segments result for simplicity.

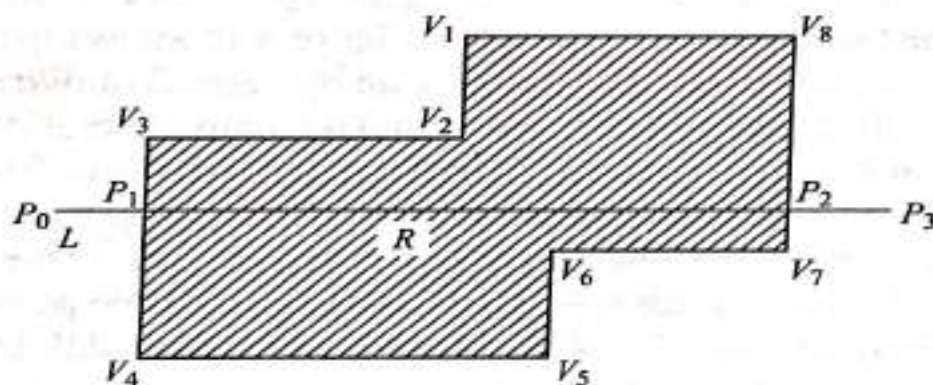


Fig. 6.17 Line/polygon Classification for B-rep

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

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

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

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

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. 6.17. 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 6.18 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 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. 6.8 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.

6.6 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

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,

$$S = \cup \left(\bigcap_{i=1}^n H_i \right) \quad (6.56)$$

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.

6.6.3 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.

6.7 BOUNDARY REPRESENTATION (B-rep)

Boundary representation is one of the two most popular and widely used schemes (the other is CSG discussed in Sec. 6.8) 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 6.22 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.

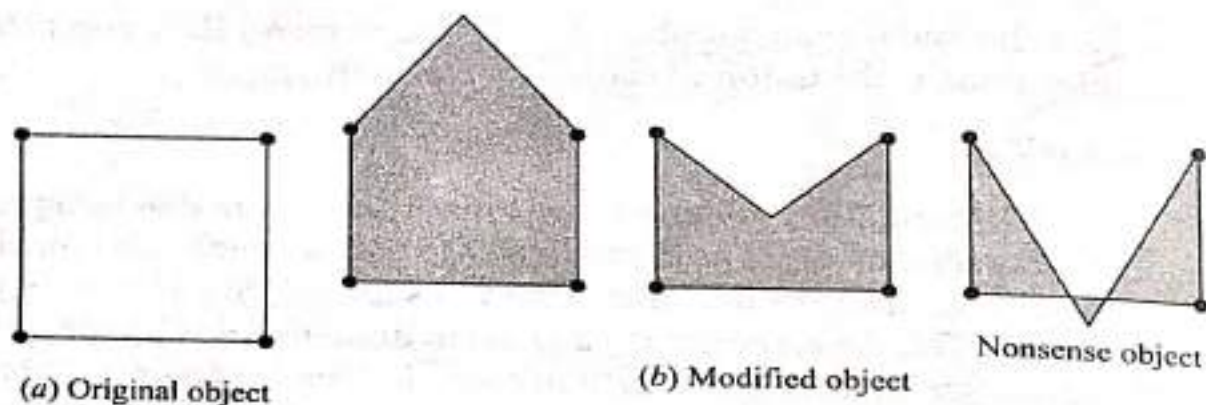


Fig. 6.22 *Effect of Topology and Geometry on Boundary Models*

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. 6.7.2. 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. 6.1 and shown in Table 6.1. The effect of a Boolean operation on a CSG model (see Sec. 6.8) 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.

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. 6.4 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

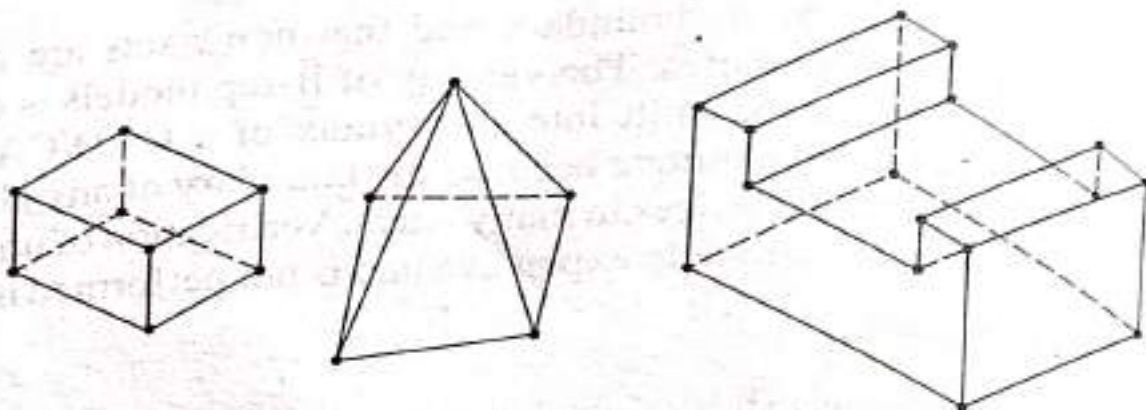
is defined unambiguously by its boundary and that non-r-sets are not defined unambiguously by their boundaries. The validity of B-rep models is ensured via Euler operations which can be built into the syntax of a CAD/CAM system. However, these models are not unique because the boundary of any object 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.

6.7.1 Basic Elements

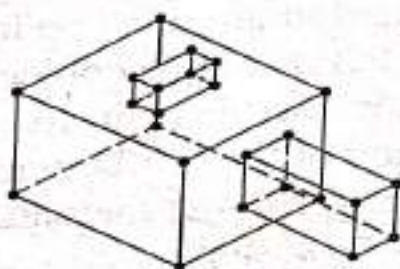
If a solid modeling system is to be designed, the domain of its representation scheme (objects that can be modeled) must be defined, the basic elements (primitives) needed to cover such modeling domain must be identified, the proper operators that enable the system users to build complex objects by combining 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.

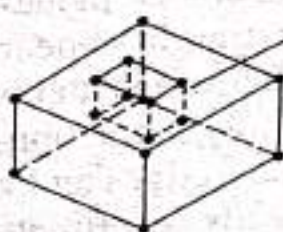
The reader might have jumped intuitively to the conclusion that the primitives of a B-rep scheme are faces, edges and vertices. This is true if we can answer the following two questions. First, what is a face, edge, or a vertex? Second, knowing the answer to the first question, how can we know that when 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. 6.23. Polyhedral objects can be classified into four classes. The first class (Fig. 6.23a) 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. 6.23b) is similar to the first 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. 6.23c) 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. 6.23d) includes objects that have holes that go through the entire objects. Topologically, these through holes are called handles.



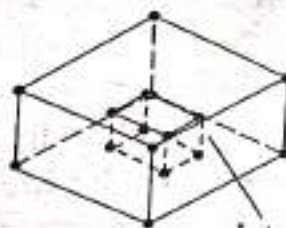
(a) Simple polyhedra



(b) Polyhedra with faces of inner loops

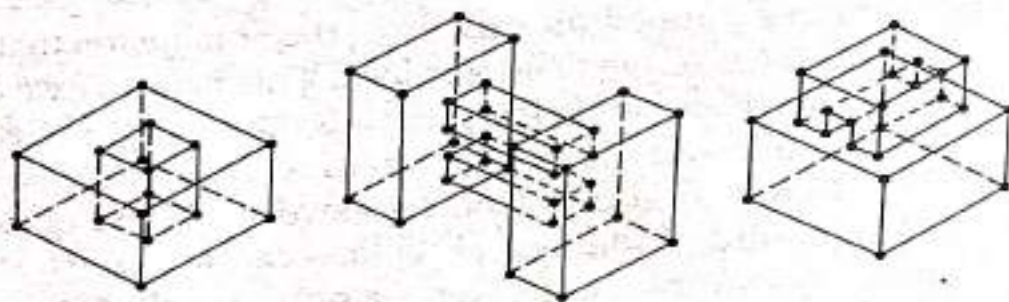


Boundary hole



Interior hole

(c) Polyhedra with not through holes



(d) Polyhedra with handles (through holes)

Fig. 6.23 *Types of Polyhedral Objects*

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. 6.23a, each face has one loop while the top and the right side faces of the object shown in Fig. 6.23b have two loops each (one inner and one outer). A "not" through hole is defined as

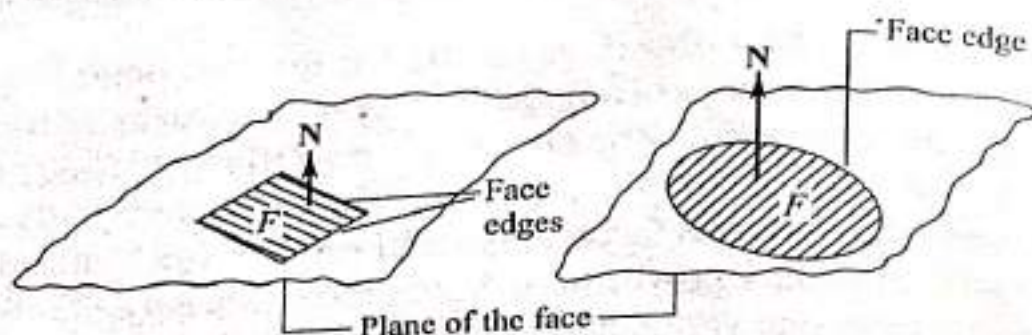
a depression in a face of an object. A handle (or through hole) is defined as a passageway that pierces the object completely. The topological name for the number of handles in an object is genus. The last item to be defined is a body (sometimes called a shell). It is a set of faces that bound a single 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. 6.23c has two bodies (the exterior and interior cubes) and any other object in Fig. 6.23 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 6.24 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.

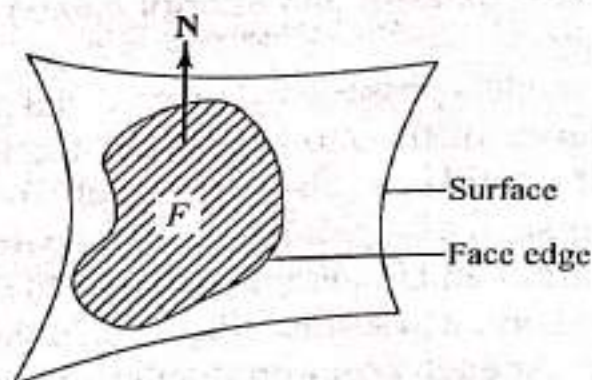
The representation of a face must ensure that both the face and solid interiors can be deduced from the representation. The direction of the face's surface normal can be used to indicate the inside or outside of the model. The surface equation must be consistent with the normal chosen convention. For 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 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 6.25 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:

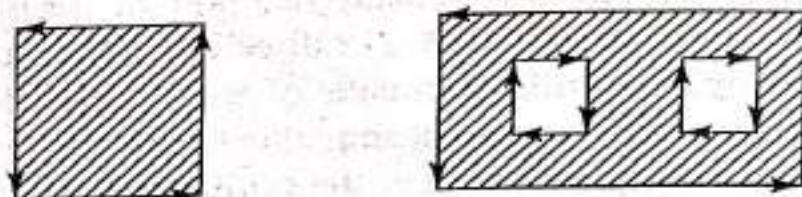
$$F - E + V - L = 2(B - G) \quad (6.57)$$



(a) Underlying surface is a plane



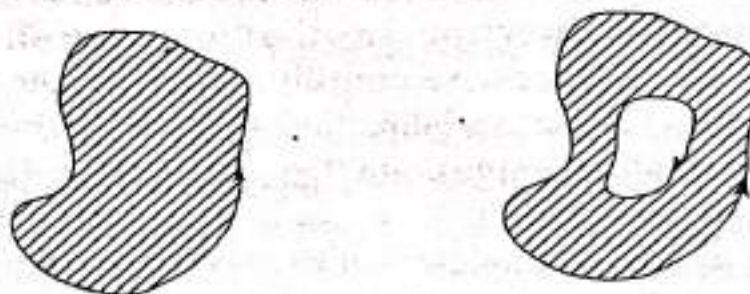
(b) A general underlying surface

Fig. 6.24 Underlying Surface of a Face

(a) Piecewise linear loops



(b) Circular loops



(c) General curve loops

Fig. 6.25 Traversal of Face's Loops

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 (6.57) 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. 6.23a. With Eq. (6.57) 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 point of view, to create the boundary model of a given object, the user identifies the proper number for all the variables of Eq. (6.57) 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 6.3 shows the counts of the various variables of Eq. (6.57) for polyhedra shown in Fig. 6.23. 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. (6.57) 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. (6.57). This class of objects includes open polyhedra 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:

$$F - E + V - L = B - G \quad (6.58)$$

Table 6.3 Counts of Polyhedral Values for Objects of Fig. 6.23

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

Figure 6.26 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. 6.26 have one body and only bodies of Fig. 6.26c have one genus each. It might be interesting to mention that Eq. (6.58) 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. 6.27. As shown in Fig. 6.27, a closed cylindrical face (and alike) has one edge and two vertices and a spherical face (and

6.8 ≡ 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. 6.6. 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 metaprimatives. 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. 6.38a. The model is represented by three bounded primitives (Fig. 6.38b) and seven half-spaces (Fig. 6.38c). 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.

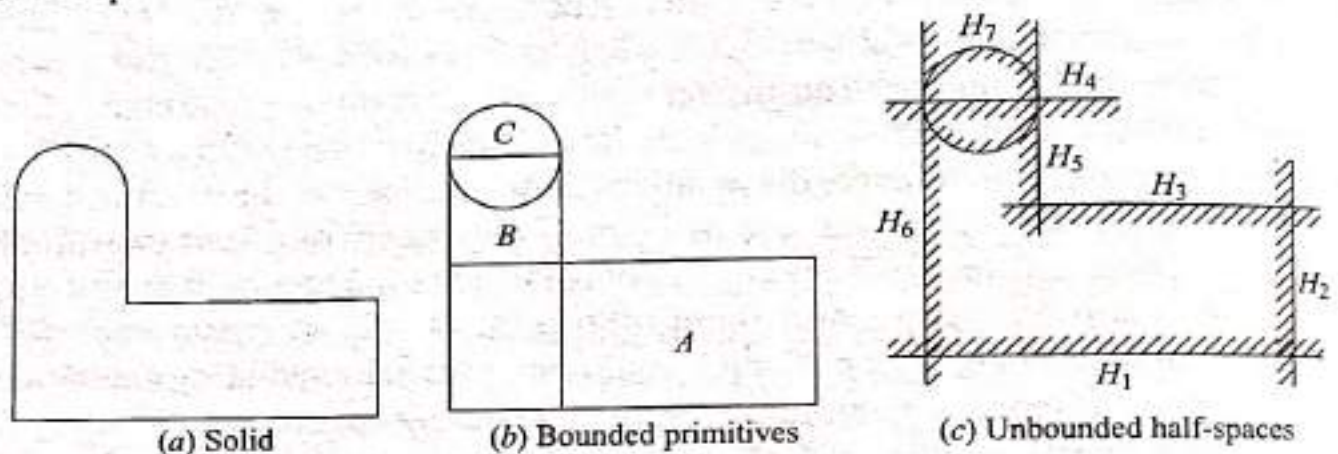


Fig. 6.38 *Bounded and Unbounded Primitives*

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 6.39a 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. 6.39b. 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. 6.39b is $\{(A, B), (A, C), (C, B), (B, E), (F, B), (B, G), (D, C), (E, C)\}$.

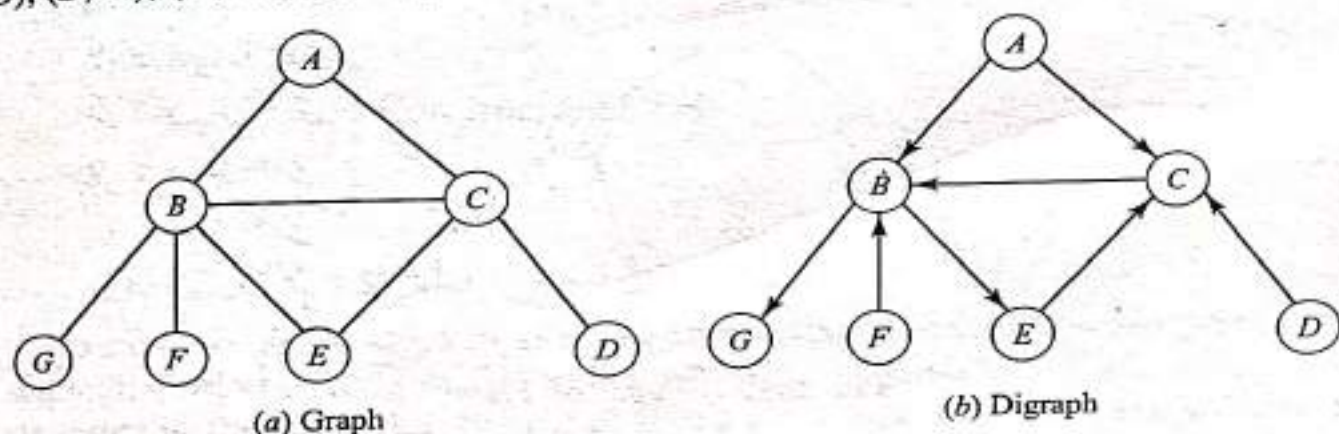


Fig. 6.39 Graphs and Digraphs

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. 6.39b 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, \dots, 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. 6.39b 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.

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. 6.39b is not a tree. However, its modification shown in Fig. 6.40a 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. 6.40b). Finally, if the arrow directions in a binary tree are reversed such that every node, except the root, in the tree has an out-degree of 1 and the root has a zero outdegree, the tree is called an inverted binary tree (see Fig. 6.40c). An inverted binary tree is very useful to understand the data structure of CSG models (sometimes called boolean models).

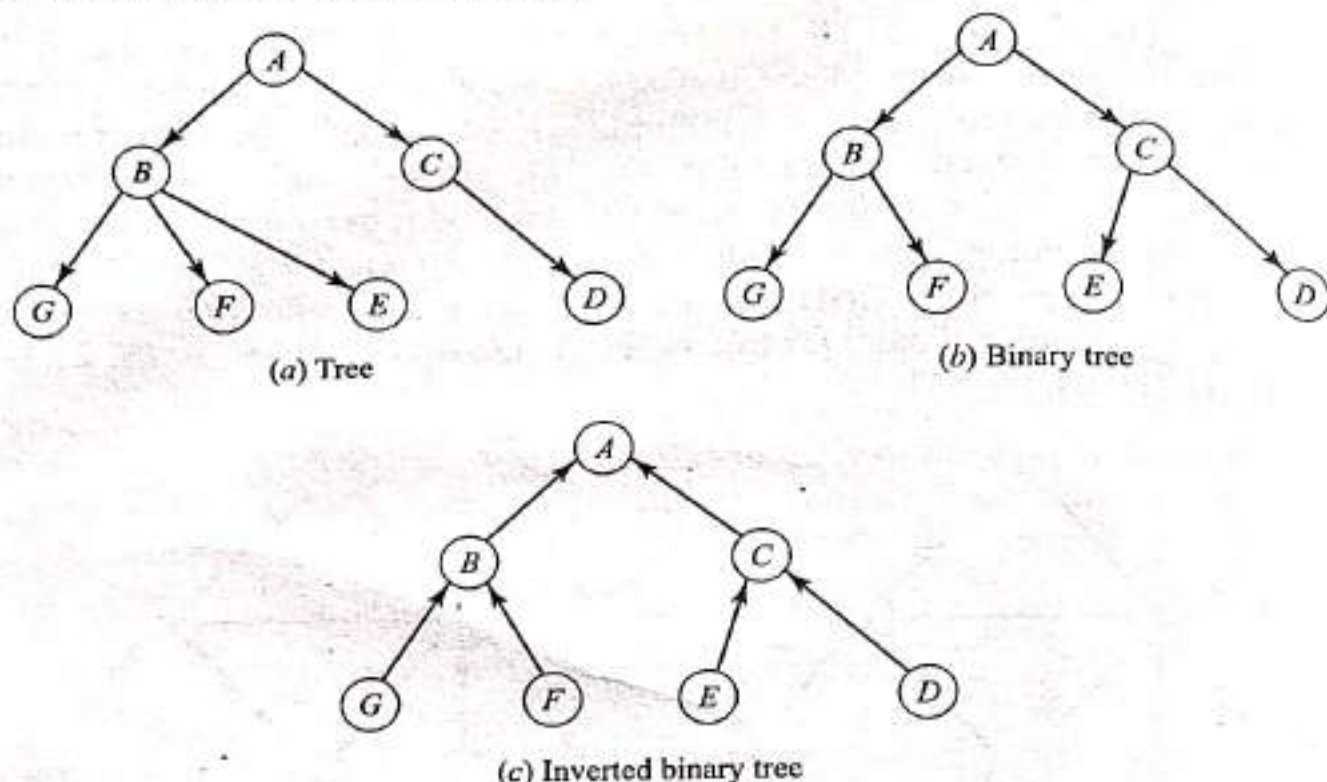


Fig. 6.40 Types of Trees

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. 6.40b, 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 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. 6.40b, 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. 6.41a with its MCS. A block and a cylinder primitive are enough to create the CSG model of the solid. Fig. 6.41b

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. 6.4 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:

$$\begin{array}{lcl}
 B_1 = \text{block positioned properly} & & \\
 B_2 = \text{block positioned properly} & & \\
 B_3 = \text{block} & & \\
 B_4 = B_3 \text{ moved properly in the } X \text{ direction} & & \\
 C_1 = \text{cylinder positioned properly} & & \\
 C_2 = C_1 \text{ moved properly in the } X \text{ direction} & & \\
 C_3 = \text{cylinder positioned properly} & & \\
 C_4 = C_3 \text{ moved properly in the } X \text{ direction} & \left. \vphantom{\begin{array}{l} B_1 \\ B_2 \\ B_3 \\ B_4 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{array}} \right\} & \text{Primitives' definitions} \\
 S_1 = B \cup^* B_3 & & \\
 S_2 = S_1 \cup^* C_1 & & \\
 S_3 = S_2 \cup^* C_3 & \left. \vphantom{\begin{array}{l} S_1 \\ S_2 \\ S_3 \end{array}} \right\} & \text{Construct left half} \\
 S_4 = B_2 \cup^* B_4 & & \\
 S_5 = C_2 \cup^* S_4 & & \\
 S_6 = C_4 \cup^* S_5 & \left. \vphantom{\begin{array}{l} S_4 \\ S_5 \\ S_6 \end{array}} \right\} & \text{Construct right half} \\
 S = S_3 \cup^* S_6 & & \\
 S = S_3 \cup^* S_6 & \text{) Model} &
 \end{array}$$

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. 6.42. 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

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

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. 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.

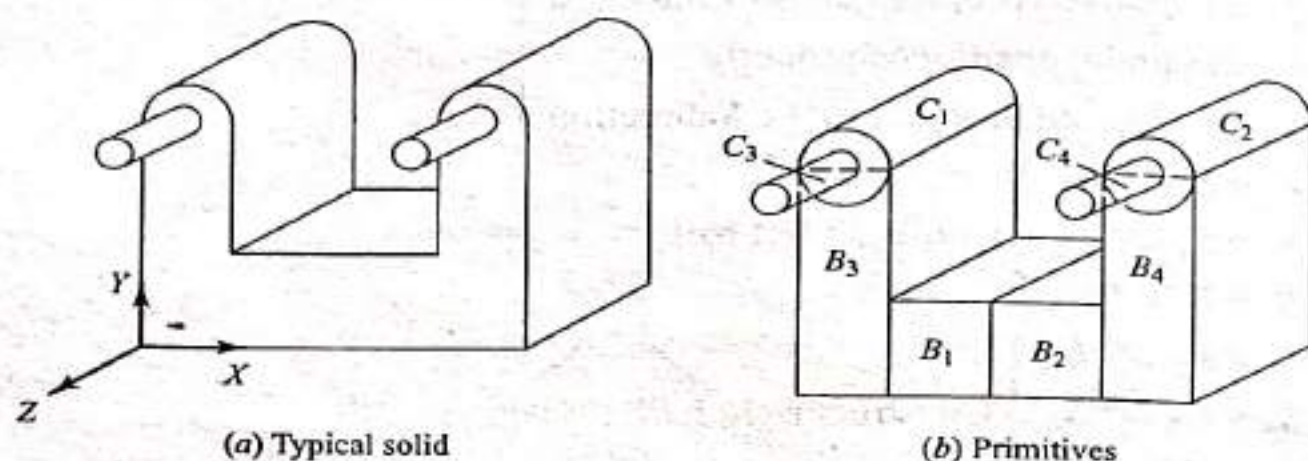


Fig. 6.41 A typical solid and its Building Primitives

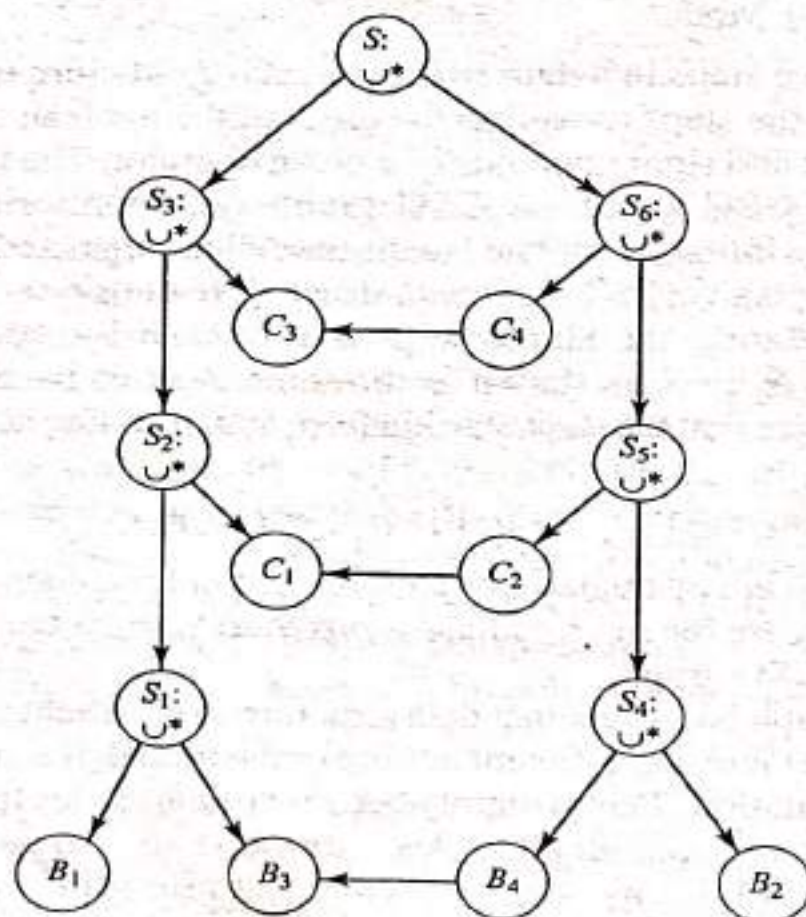


Fig. 6.42 CSG Graph of a typical solid

A CSG tree is defined as an inverted ordered binary tree whose leaf nodes are primitives and interior nodes are regularized set operations. Figure 6.43 shows the CSG tree derived from the CSG graph shown in Fig. 6.42. 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. 6.41*b*. In Fig. 6.43, 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. 6.42). 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.

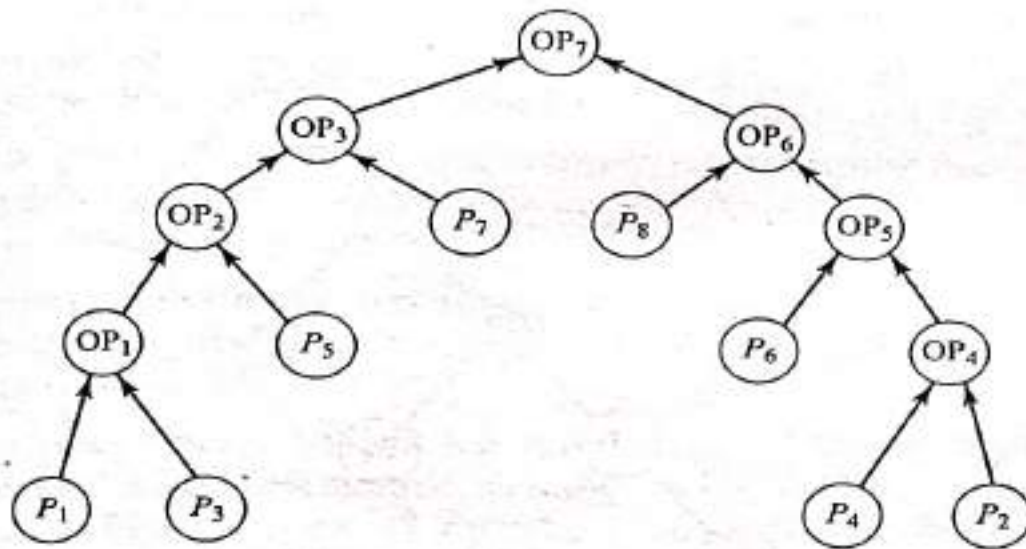


Fig. 6.43 CSG Tree of a Typical Solid

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 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

$$n_L + n_R = 2n - 2 \quad (6.59)$$

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:

$$n_L = n_R = n - 1 \quad (6.60)$$

Each subtree has $n/2$ leaf nodes (primitives) and $(n - 2)/2$ interior nodes (boolean operations). Figure 6.43 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. 6.43 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. 6.14b and 6.42) starting from the plane(s) of symmetry. Figure 6.44 shows an unbalanced tree of the same solid shown in Fig. 6.41. 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. 6.41b. 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.

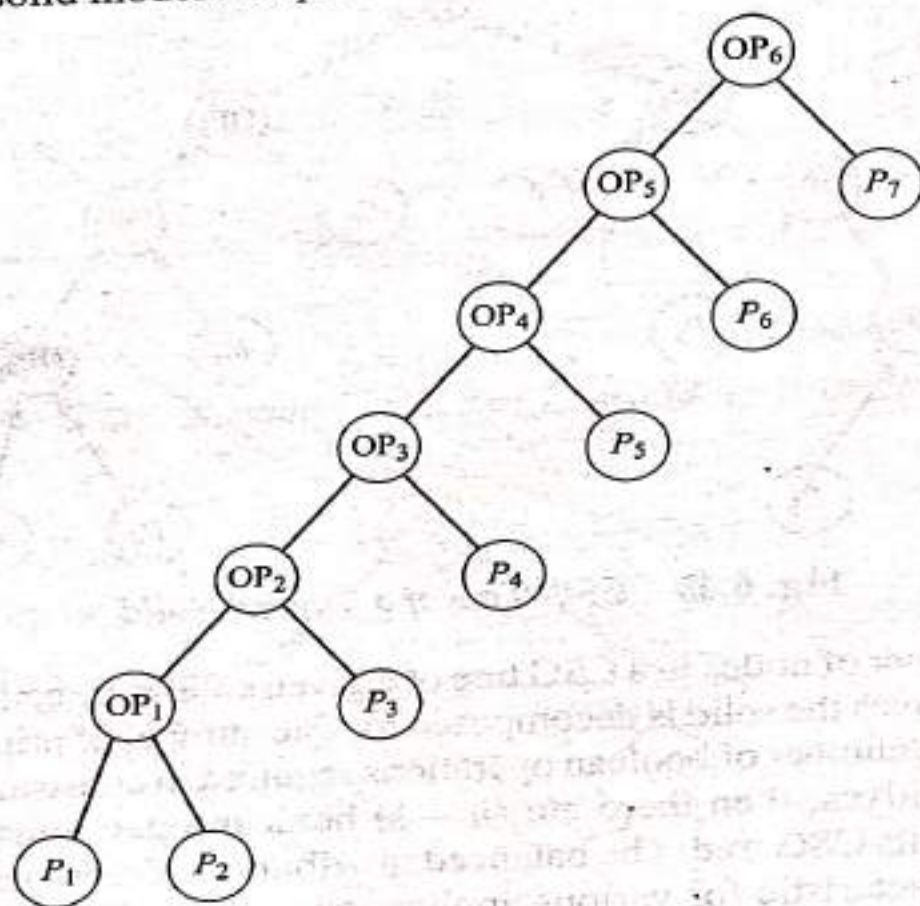


Fig. 6.44 *An Unbalanced CSG Tree*

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 denned 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 6.45 shows the preorder and its reverse, traversal of the tree shown in Fig. 6.43.

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. 6.46).

To traverse a tree in postorder:

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

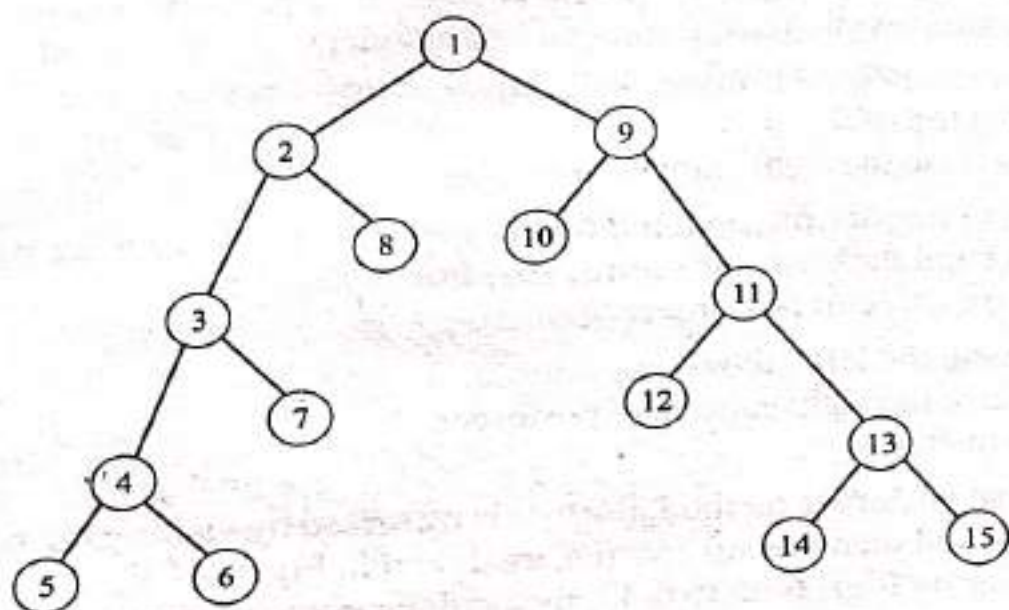
In the reverse postorder method, the tree is traversed by visiting the root, then the right subtree and then the left subtree, as shown in Fig. 6.47.

By comparing Figs. 6-45 to 6-47, 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.

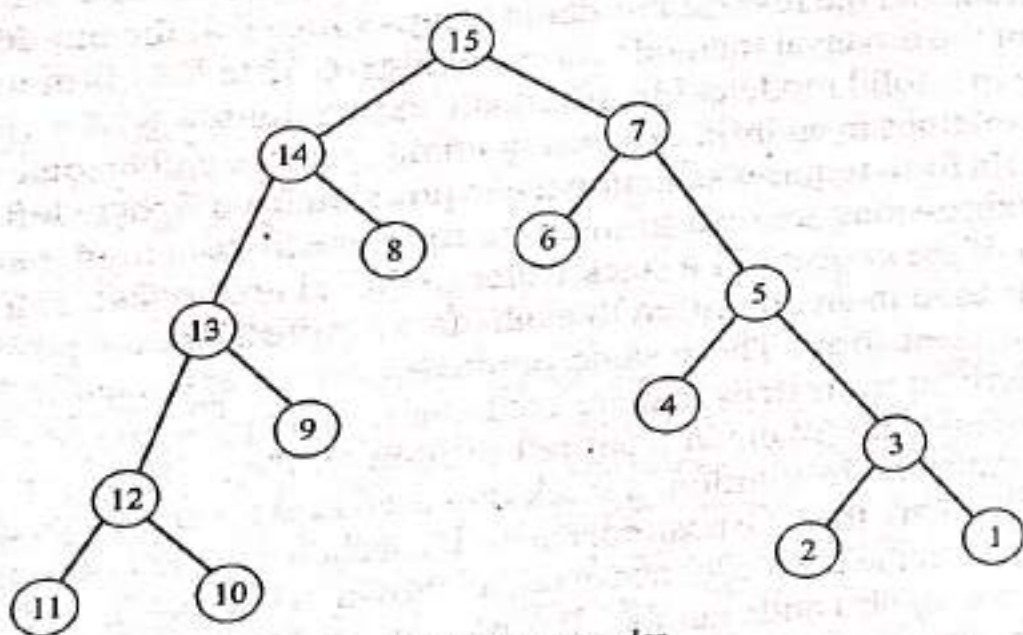
Which of the traversal methods shown in Figs. 6.45 to 6.47 is more suitable to store a tree in a solid modeler? In arithmetic expressions, e.g., $A + (B + C)D$, the order of operations in an infix expression might require cumbersome parentheses while a prefix form requires scanning the expression from 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. 6.47a, is that the root of the tree has the highest node number. It is more 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.

6.8.1 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. 6.4. Each one of these primitives 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. 6.4.

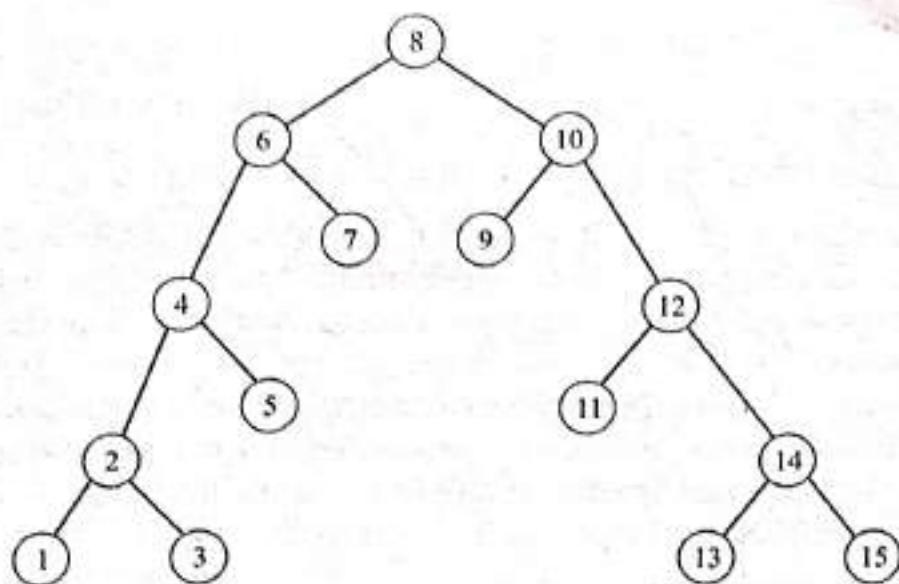


(a) Preorder

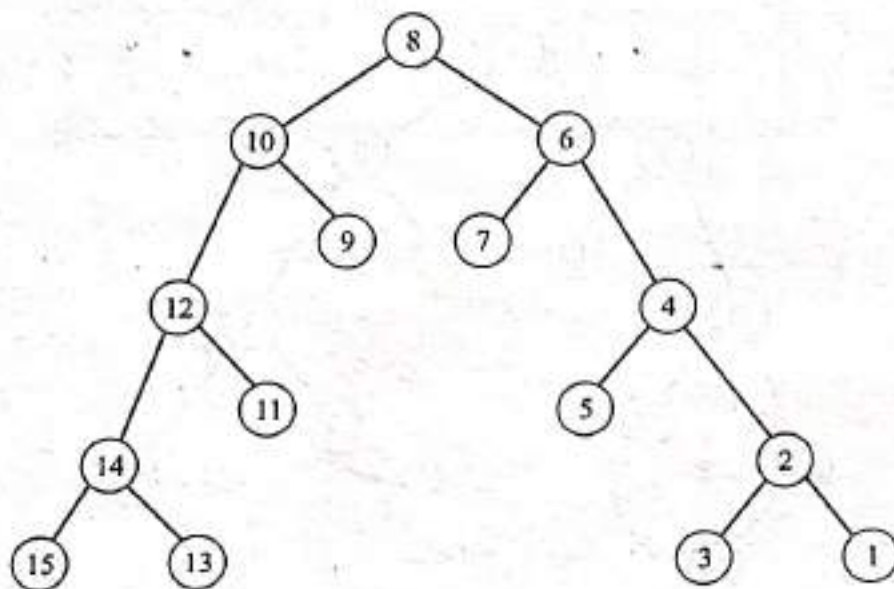


(b) Reverse preorder

Fig. 6.45 Preorder and Reverse Preorder Traversals of a Tree



(a) Inorder

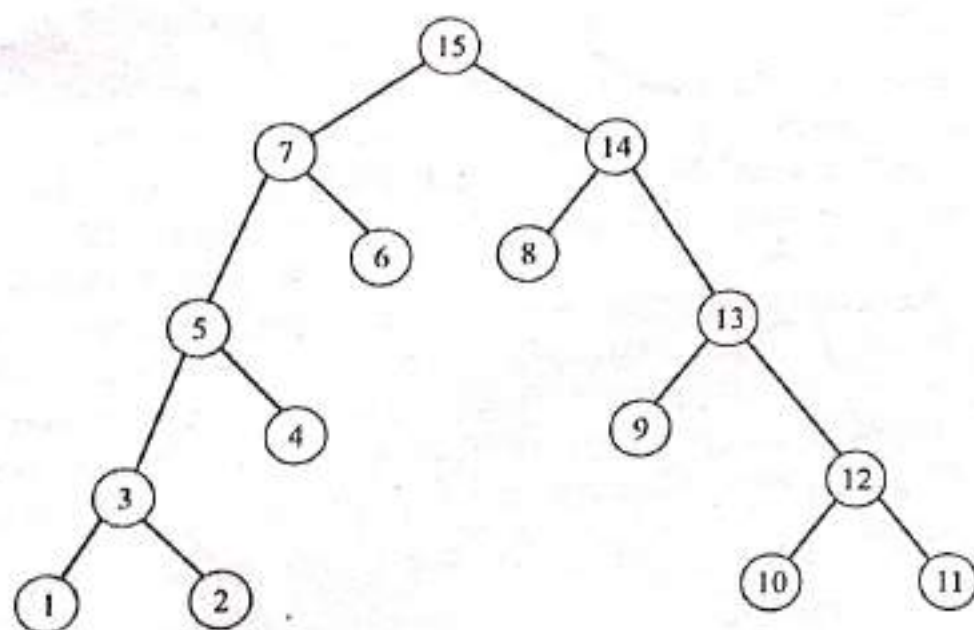


(b) Reverse inorder

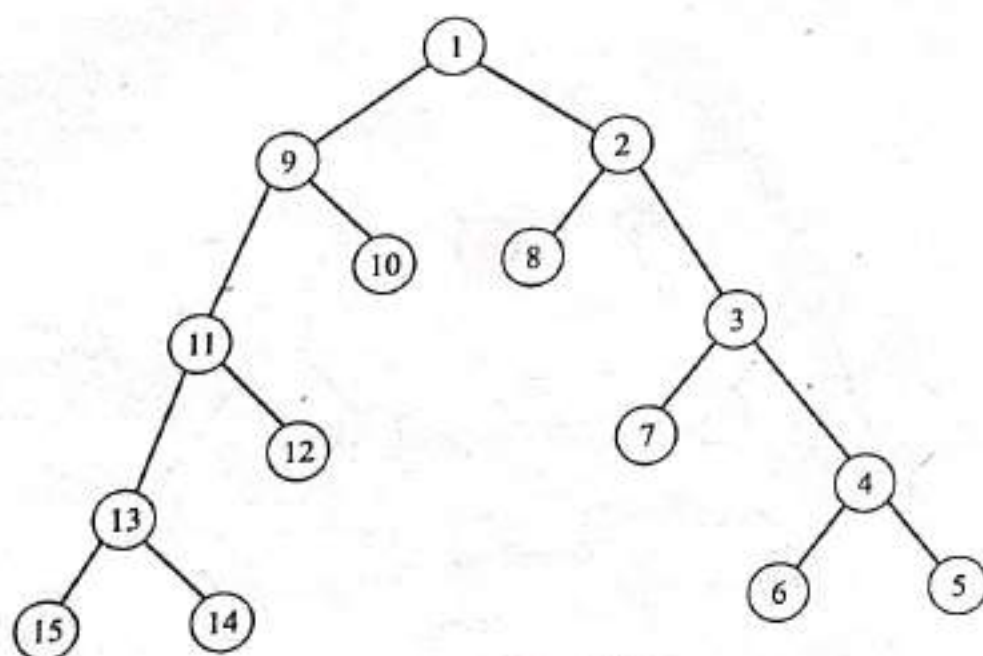
Fig. 6.46 Inorder and Reverse Inorder Traversals of a Tree

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



(a) Postorder



(b) Reverse postorder

Fig. 6.47 Postorder and Reverse Postorder Traversals of a Tree

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. 6.4, these point sets are given by:

$$\text{Block: } \{(x, y, z): 0 < x < W, 0 < y < H \text{ and } 0 < z < D\} \quad (6.61)$$

$$\text{Cylinder: } \{(x, y, z): x^2 + y^2 < R^2 \text{ and } 0 < z < H\} \quad (6.62)$$

$$\text{Cone: } \{(x, y, z): x^2 + y^2 < [(R/H)z]^2 \text{ and } 0 < z < H\} \quad (6.63)$$

$$\text{Sphere: } \{(x, y, z): x^2 + y^2 + z^2 < R^2\} \quad (6.64)$$

Wedge: $\{(x, y, z): 0 < x < W, 0 < y < H, 0 < z < D,$
 and $yW + xH < HW\}$ (6.65)

Torus: $\{(x, y, z): (x^2 + y^2 + z^2 - R_2^2 - R_1^2)^2 < 4R_2^2(R_1^2 - z^2)\}$ (6.66)

Comparing Eqs. (6.61) to (6.66) with the half-space equations (6.51) to (6.55), 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. (6.61). Similarly, a cylinder, cone and a wedge are the union of three, three and five half-spaces respectively. Fig. 6.48 shows two-dimensional illustrations of the half-spaces of each primitive shown in Fig. 6.4. Some half-spaces for the block and wedge primitives are not shown in the figure for clarity purposes.

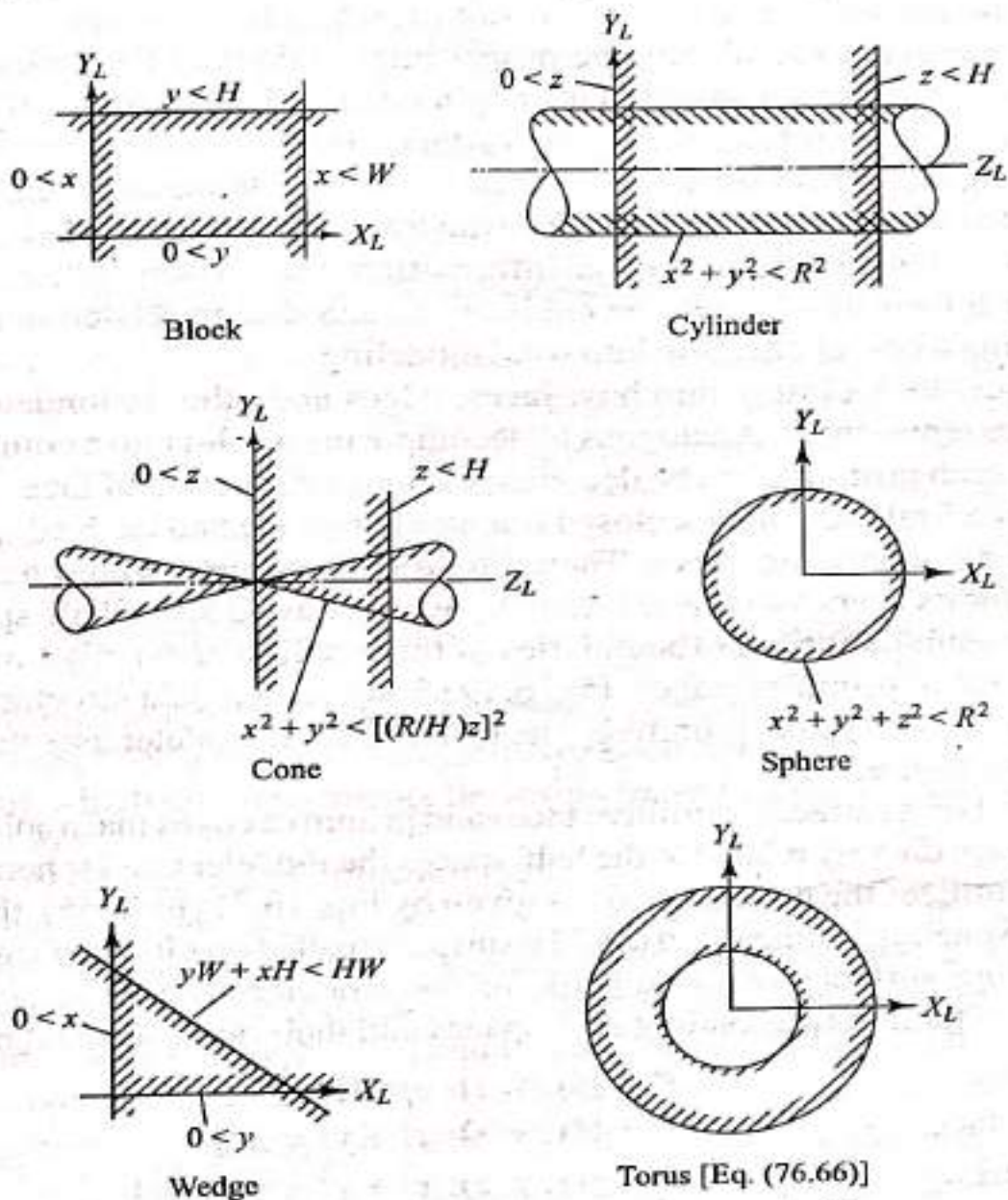


Fig. 6.48 Half-spaces of Bounded Primitives

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. 6.4. A user can specify its R_1 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 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. Fig. 6.49 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. (6.51) to (6.55), 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:

$$P = \{(x, y, z): z = 0\} \quad (6.67)$$

Cylinder surface:

$$P = \{(x, y, z): x^2 + y^2 = R^2\} \quad (6.68)$$

Spherical surface:

$$P = \{(x, y, z): x^2 + y^2 + z^2 = R^2\} \quad (6.69)$$

Conical surface:

$$P = \{(x, y, z): x^2 + y^2 = [(R/H)z]^2\} \quad (6.70)$$

Toroidal surface:

$$P = \{(x, y, z): (x^2 + y^2 + z^2 - R_2^2 - R_1^2)^2 = 4R_2^2(R_1^2 - z^2)\} \quad (6.71)$$

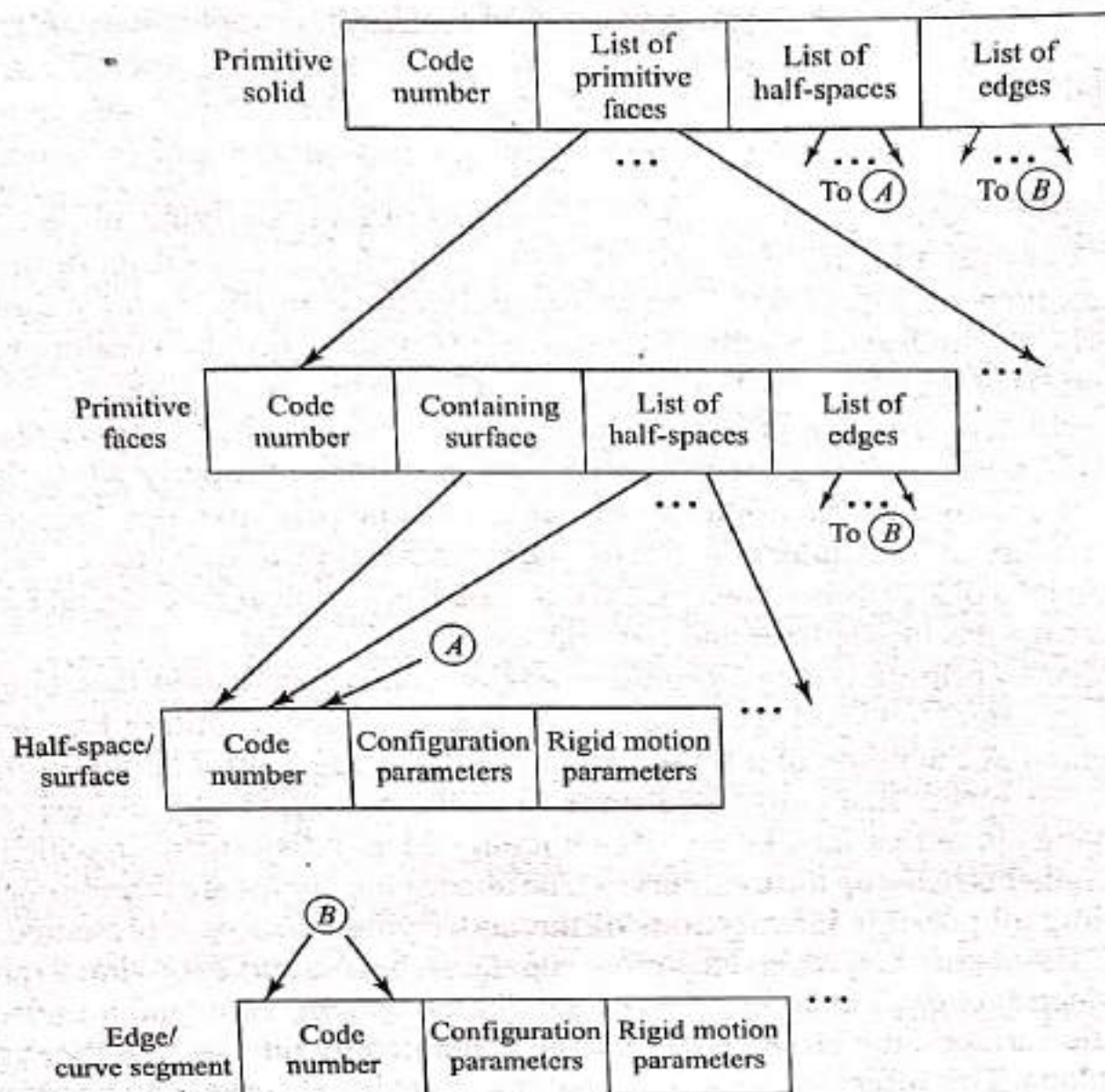


Fig. 6.49 Data Structure of a Typical Primitive Solid

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. 6.4 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: $F = \{(x, y, z): 0 < x < W, 0 < y < H \text{ and } z = 0\}$ (6.72)

Triplate Pface: $F = \{(x, y, z): 0 < x < W, 0 < y < H \text{ and } yW + xH < HW\}$ (6.73)

Disc Pface: $F = \{(x, y, z): x^2 + y^2 < R^2 \text{ and } z = 0\}$ (6.74)

Cylindrical Pface: $F = (x, y, z): x^2 + y^2 = R^2 \text{ and } 0 < z < H$ (6.75)

Spherical Pface: $F = \{(x, y, z): x^2 + y^2 + z^2 = R^2\}$ (6.76)

167 195-218
160, 195, 200, 207, 218 → module 2

243-249
module 3 → ~~245~~, 262, 266, 267, 268, 270, 277, 272, 287, 291

292

262-294

323-328
module 4 → 338, 351, 371 → 323-328 (338-345) (351-357) (371-378)

517-527
module 5 → 517, 537, 539, 550, 565 (517-527) (537-560) 565-569

module 6 →

The inconvenience associated with the partially hidden lines stems from the fact that the "trim entity" command, which is expected to be used, affects the entity permanently in all views; that is, it has a global effect on the entity, not just a local effect in a given view. This would force the user to copy the partially hidden line (model entity) first while the drafting mode is on (to obtain a drafting entity) and then to trim the resulting line. Some CAD/CAM systems may have a special solution to this problem.

While the manual model clean-up as described above is applicable to wireframe models, its extension to surface models becomes practically useless. In some instances, the surface may have to be replaced by its boundaries (wireframe entities) before the clean-up process begins. Surface manual clean-up is usually not common because surfaces are seldom used in engineering drawings. Automatic hidden line and hidden surface removal algorithms, discussed in the following sections, are typically used instead.

9.3 ≡ HIDDEN LINE REMOVAL

Since the early development of computer graphics, there is always a demand for images (of objects) enhanced by removing the hidden parts that would not be seen if objects were constructed from opaque material in real life. Edges and surfaces of a given object may be hidden (invisible) or visible in a given image depending on the viewing direction. The determination of hidden edges and surfaces is considered one of the most challenging problems in computer graphics. Its solution typically demands large amounts of computer time and memory space. Techniques to reduce these demands and improve efficiencies of related algorithms exist and are discussed here.

The solution to the problem of removing hidden edges and surfaces draws on various concepts from computing, mainly sorting and geometric modeling, mainly projection and intersection. This problem can also be viewed as a visibility problem. Therefore, a clear understanding of it and its solution is useful and can be extended to solve relevant engineering problems. Consider, for example, the vision and path planning problems in robotics applications. In the vision problem, the camera location and orientation provide the viewing direction which, in turn, can be used to determine the hidden edges and surfaces of objects encountered in the robot working environment. In the path planning problem, the knowledge of when a given surface changes from visible to hidden (via finding silhouette edges and curves as seen later in this section) can be utilized to find the minimum path of the robot end effector. Points on the surface where its status changes from visible to invisible or vice versa can be considered as critical points which the path planning algorithm can use as an input. Another example is the display of finite element meshes where the hidden elements are removed. In this case, each element is treated as a planar polygon and the collection of elements that forms the meshed object, from a finite element point of view, forms a polyhedron from a computer graphics viewpoint.

A wide variety of hidden line and hidden surface removing (visibility) algorithms is in existence today. The development of these algorithms is influenced by the

types of graphics display devices they support (whether they are vector or raster) and by the type of data structure or geometric modeling they operate on (wireframe, surface, or solid modeling). Some algorithms utilize parallel, over the traditional serial, processing to speed up their execution. The formalization and generalization of these algorithms are useful and are required if one attempts to design and build special-purpose hardware to support hidden line and hidden surface removal, which is not restricted to a single algorithm. However, it is not a trivial task to convert the different algorithmic formulations into a form that allows them to be mapped onto a generalized scheme.

Algorithms that are applied to a set of objects to remove hidden parts to create a more realistic image are usually classified into hidden line and hidden surface algorithms. The former supports line-drawing devices such as vector displays and plotters, while the latter supports raster displays. Hidden line algorithms can, of course, be used with raster displays because they support line drawings. However, hidden surface algorithms are not applicable to vector displays. From a geometric modeling point of view, this classification is both confusing and deceiving. Hidden line removal does not mean (as the name may imply) that it is applicable to wireframe models only. Similarly, hidden surface removal is not only applicable to surface models. As a matter of fact, algorithms to remove hidden parts from an image cannot be applied to wireframe or surface models directly. They require an unambiguous data structure that represents an object as orientable faces. This means that each face has a surface normal with a consistent direction (say positive if face edges are input in a counterclockwise direction); that is, polyhedral objects are represented by orientable flat polygons. These polygons can be obtained from a wireframe, surface, or solid model. Users would have to input extra information to identify faces and orientation for wireframes or orientation for surface models. Solid models provide such information automatically. In spite of the above misleading classification, we have hidden line, hidden surface and hidden solid removal sections in this chapter. This is merely done to reflect the historical order of the development of the related algorithms.

Hidden line and hidden surface algorithms have been classified as object-space methods, image-space methods, or a combination of both (hybrid methods). Image-space algorithms can be further divided into raster and vector algorithms. The raster algorithms use a pixel-matrix representation of the image and the vector algorithms use endpoint coordinates of line segments in representing the image. An object-space algorithm utilizes the spatial and geometrical relationships among the objects in the scene to determine hidden and visible parts of these objects. An image-space algorithm, on the other hand, concentrates on the final image to determine what is visible, say, within each raster pixel in the case of raster displays. Most hidden surface algorithms use raster image-space methods while most hidden line algorithms use object-space methods.

The two approaches (object-space and image-space) to achieve visual realism exhibit different characteristics. Object-space algorithms are more accurate than image-space algorithms. The former perform geometric calculations (such as intersections) using the floating-point precision of the computer hardware, while the latter perform calculations with accuracy equal to the resolution of the display

screen used to present them. Therefore, enlargement of an object-space image does not degrade its quality of display as does the enlargement of an image-space image. As the complexity of the scene increases (large numbers of objects in the scene), the computation time grows faster for object-space algorithms than for image-space algorithms.

9.3.1 Visibility of Object Views

The visibility of parts of objects of a scene depends on the location of the viewing eye, the viewing direction, the type of projection (orthogonal or perspective) and the depth or the distance from various faces of various objects in the scene to the viewing eye. The hidden line removal of perspective views (see Fig. 8.21) is a fairly complex problem to solve. Many lines of sight (rays) from the viewing eye must be considered and their points of intersection with objects' faces have to be calculated. The complexity of the problem is considerably reduced if orthographic views are utilized because no intersections would be necessary. Therefore, it is common practice to apply the perspective transformation given by Eq. (8.116) to the set of the points in the scene and then apply orthographic hidden line visibility algorithms to the resulting (transformed) set of points. This is equivalent to saying that the orthographic viewing of the transformed (perspective) objects is identical to the perspective viewing of the original (untransformed) objects. Hence, only orthographic hidden line algorithms are discussed in this book.

The depth comparison is the central criterion utilized by hidden line algorithms to determine visibility. Depth comparisons are typically done after the proper view transformation given by Eqs (8.105) and (8.116) for orthographic and perspective projections respectively. While these two equations destroy the depth information (the z_v coordinate of projected points) to generate views, such information can be saved [by replacing the element t_{33} of $[T]$ in both equations by 1 instead of the current 0; for Eqs. (8.107) and (8.108), set t_{32} and t_{31} respectively to 1] for depth comparisons by hidden line algorithms.

The depth comparison determines if a projected point $P_{1v}(x_{1v}, y_{1v})$ in a given view obscures another point $P_{2v}(x_{2v}, y_{2v})$. This is equivalent to determining if the two original corresponding points P_1 and P_2 lie on the same projector as shown in Fig. 9.3 (the MCS and VCS are shown as the XYZ and the $X_v Y_v Z_v$ systems respectively). For orthographic projections, projectors are parallel. Therefore, two points P_1 and P_2 are on the same projector if $x_{1v} = x_{2v}$ and $y_{1v} = y_{2v}$. If they are, a comparison of z_{1v} and z_{2v} decides which point is closer to the viewing eye. Utilizing the VCS shown in Figs. 8.19 and 9.3, the point with the larger z_v coordinate lies closer to the viewer. Applying this depth comparison to points P_1 , P_2 and P_3 of Fig. 9.3 shows that point P_1 obscures P_2 (i.e., P_1 is visible and P_2 is hidden) and P_3 is visible. If the depth comparison is to be performed utilizing the ECS (see Fig. 8-21), the transformation matrix given by Eq. (8.113) must be applied to the projected points before the comparison. This simply reverses the comparison to say that points with smaller z_v coordinates lie closer to the viewer.

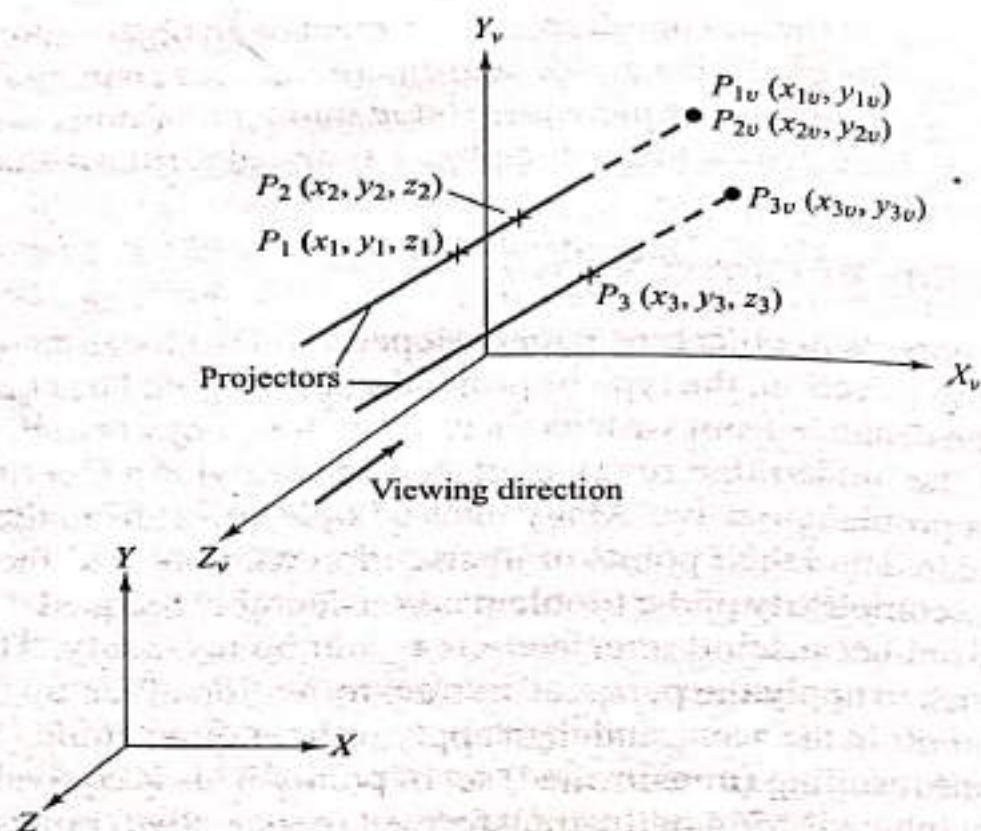


Fig. 9.3 Depth Comparison for Orthographic Projection

9.3.2 Visibility Techniques

If the depth comparison criterion is used solely with no other enhancements, the number of comparisons grows rapidly [for n points, $\binom{n}{2}$ tests are required] which leads to difficulties storing and managing the results by the corresponding hidden line algorithm. As a result, the algorithm might be slow in calculating the final image. Various visibility techniques exist to alleviate these problems. In general, these techniques attempt to establish relationships among polygons and edges in the viewing plane. The techniques normally check for overlapping of pairs of polygons (sometimes referred to as lateral comparisons) in the viewing plane (the screen). If overlapping occurs, depth comparisons are used to determine if part or all of one polygon is hidden by another. Both the lateral and depth comparisons are performed in the VCS.

9.3.2.1 Minimax Test

This test (also called the overlap or bounding box test) checks if two polygons overlap. The test provides a quick method to determine if two polygons do not overlap. It surrounds each polygon with a box by finding its extents (minimum and maximum x and y coordinates) and then checks for the intersection for any two boxes in both the X and Y directions. If two boxes do not intersect, their corresponding polygons do not overlap (see Fig. 9.4). In such a case, no further testing of the edges of the polygons is required.

If the minimax test fails (two boxes intersect), the two polygons may or may not overlap, as shown in Fig. 9.4. Each edge of one polygon is compared against all the edges of the other polygon to detect intersections. The minimax test can be applied first to any two edges to speed up this process.

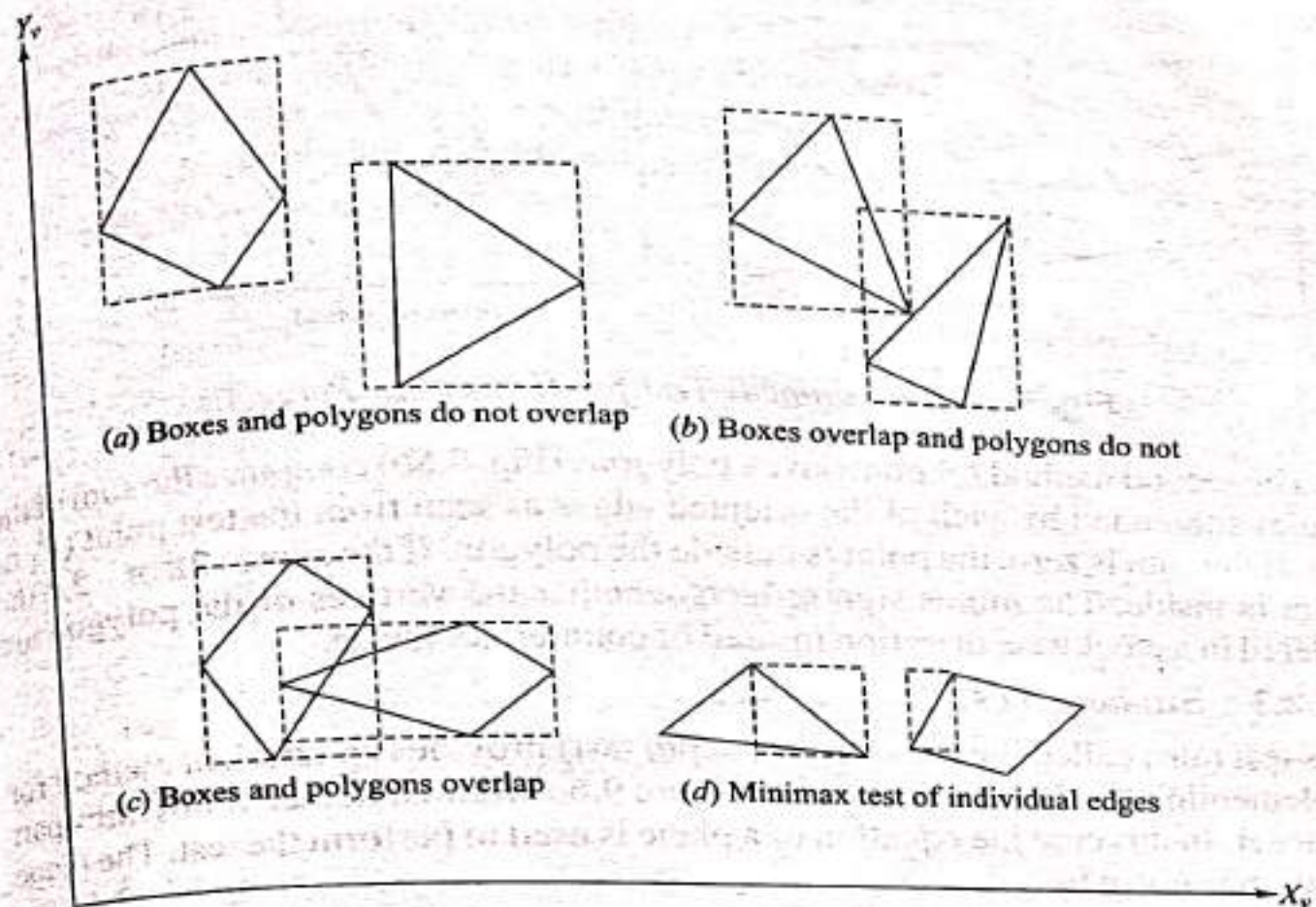


Fig. 9.4 Minimax Tests for typical Polygons and Edges

The minimax test can be applied in the Z direction to check if there is no overlap in this direction. In all tests, finding the extents themselves is the most critical part of the test. Typically, this can be achieved by iterating through the list of vertex coordinates of each polygon and recording the largest and the smallest values for each coordinate.

9.3.2.2 Containment Test

Some hidden line algorithms depend on whether a polygon surrounds a point or another polygon. The containment test checks whether a given point lies inside a given polygon or polyhedron. There are three methods to compute containment or surroundedness. For a convex polygon, one can substitute the x_v and y_v coordinates of the point into the line equation of each edge. If all substitutions result in the same sign, the point is on the same side of each edge and is therefore surrounded. This test requires that the signs of the coefficients of the line equations be chosen correctly.

For nonconvex polygons, two other methods can be used. In the first method, we draw a line from the point under testing to infinity as shown in Fig. 9.5a. The semi-infinite line is intersected with the polygon edges. If the intersection count is even, the point is outside the polygon (P_2 in Fig. 9.5a). If it is odd, the point is inside (P_1 in the figure). If one of the polygon edges lies on the semi-infinite line, a singular case arises which needs special treatment to guarantee the consistency of the results.

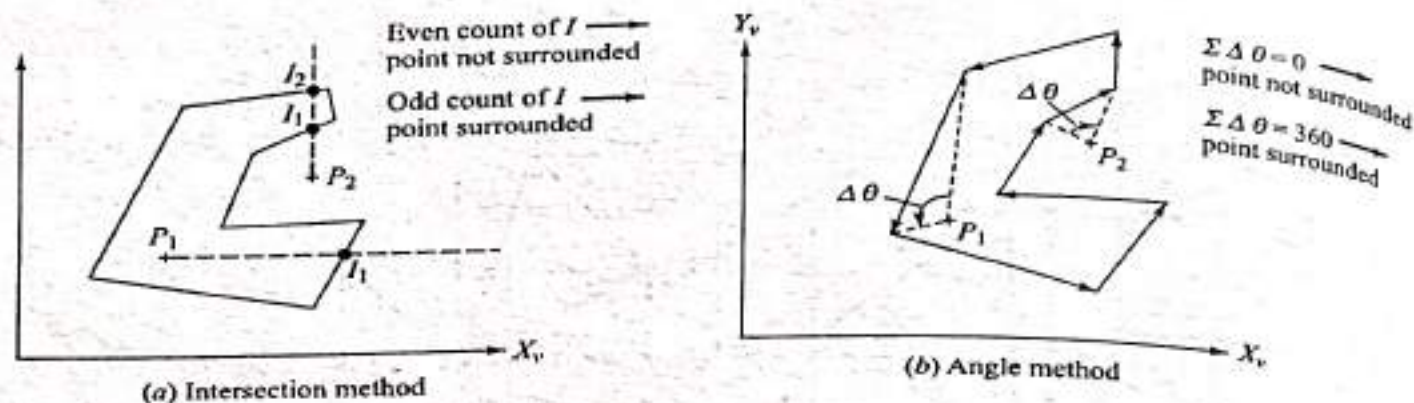


Fig. 9.5 Containment Test for Nonconvex Polygons

The second method for nonconvex polygons (Fig. 9.5b) computes the sum of the angles subtended by each of the oriented edges as seen from the test point (P_1 or P_2). If the sum is zero, the point is outside the polygon. If the sum is 2π or -2π , the point is inside. The minus sign reflects whether the vertices of the polygon are ordered in a clockwise direction instead of counterclockwise.

9.3.2.3 Surface Test

This test (also called the back face or depth test) provides an efficient method for implementing the depth comparison. Figure 9.6a shows that face B obscures part of face A . In this case the equation of a plane is used to perform the test. The plane equation is given by

$$ax_v + by_v + cz_v + d = 0 \quad (9.1)$$

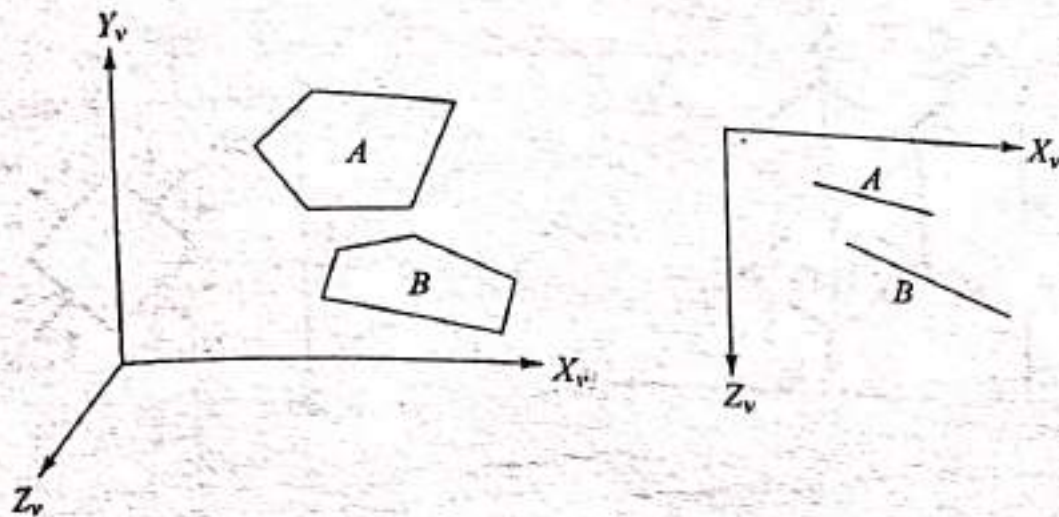
If a given point (x_v, y_v, z_v) is not on the plane, the sign of the left-hand side of the above equation is positive if the point lies on one side of the plane and negative if it lies on the other side. The equation coefficients a, b, c and d can be arranged so that a positive value indicates a point outside the plane. The plane equation can also be used to compute the depth z_v of a face at a given point (x_v, y_v) . The depths of two faces can, therefore, be computed at given points to decide which one is closer to the viewing eye.

Another important use of the plane equation in hidden line removal is achieved by using the normal vector to the plane. The first three coefficients a, b and c of Eq. (9.1) represent the normal to the plane and the vector $[a, b, c, d]$ represents the homogeneous coordinates of this normal. The coefficient d is found by knowing a point on the plane. In Chap. 5, we have discussed the various ways of finding the plane equation. Figure 9.6b shows how the normal to a face can be used to decide its visibility. The basic idea of the test (Fig. 9.6b) is that faces whose outward normal vector points toward the viewing eye are visible (face F_1) while others are invisible (face F_2). This test is implemented by calculating the dot product of the normal vector \mathbf{N} and the line-of-sight vector \mathbf{S} (Fig. 9.6b) as

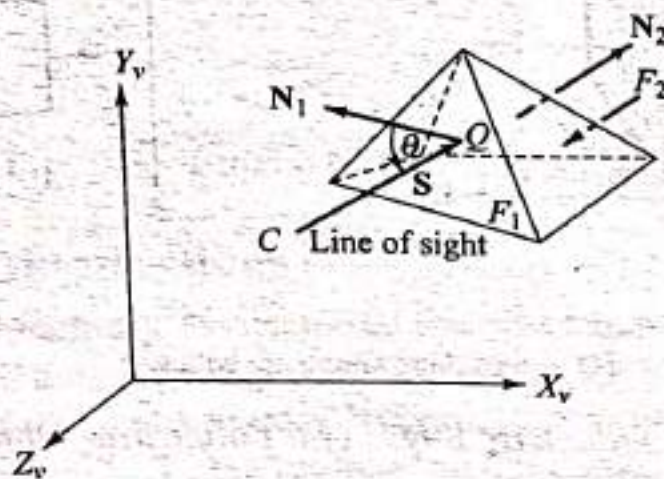
$$\mathbf{N} \cdot \mathbf{S} = |\mathbf{N}| |\mathbf{S}| \cos \theta \quad (9.2)$$

If, in this equation, we assume that \mathbf{N} points away from the solid and θ is measured from \mathbf{N} to \mathbf{S} , the dot product is positive when \mathbf{N} points toward the viewing eye or when the face and its edges are visible. The right-hand side of Eq. (9.2) gives the component of \mathbf{N} along the direction of \mathbf{S} . For orthographic projection, this direction

coincides with the Z_v axis. Thus the surface test can be stated as follows. Faces whose normal has a positive component in the Z_v direction are visible and those whose normal has a negative Z_v component are not visible. The surface test by itself cannot solve the hidden line problem except for single convex polyhedra. Even for convex polyhedra, the test may fail for perspective projection if more than one polyhedron exists in the scene.



(a) Utilizing plane equation



(b) Utilizing normals to planes

Fig. 9.6 Surface Test

9.3.2.4 Computing Silhouettes

A set of edges that separates visible faces from invisible faces of an object with respect to a given viewing direction is called silhouette edges (or silhouettes). The signs of the Z_v components of normal vectors of the object faces can be utilized to determine the silhouette. An edge that is part of the silhouette is characterized as the intersection of one visible face and one invisible face. An edge that is the intersection of two visible faces is visible, but does not contribute to the silhouette. The intersection of two invisible faces produces an invisible edge. Figure 9.7a shows the silhouette of a cube.

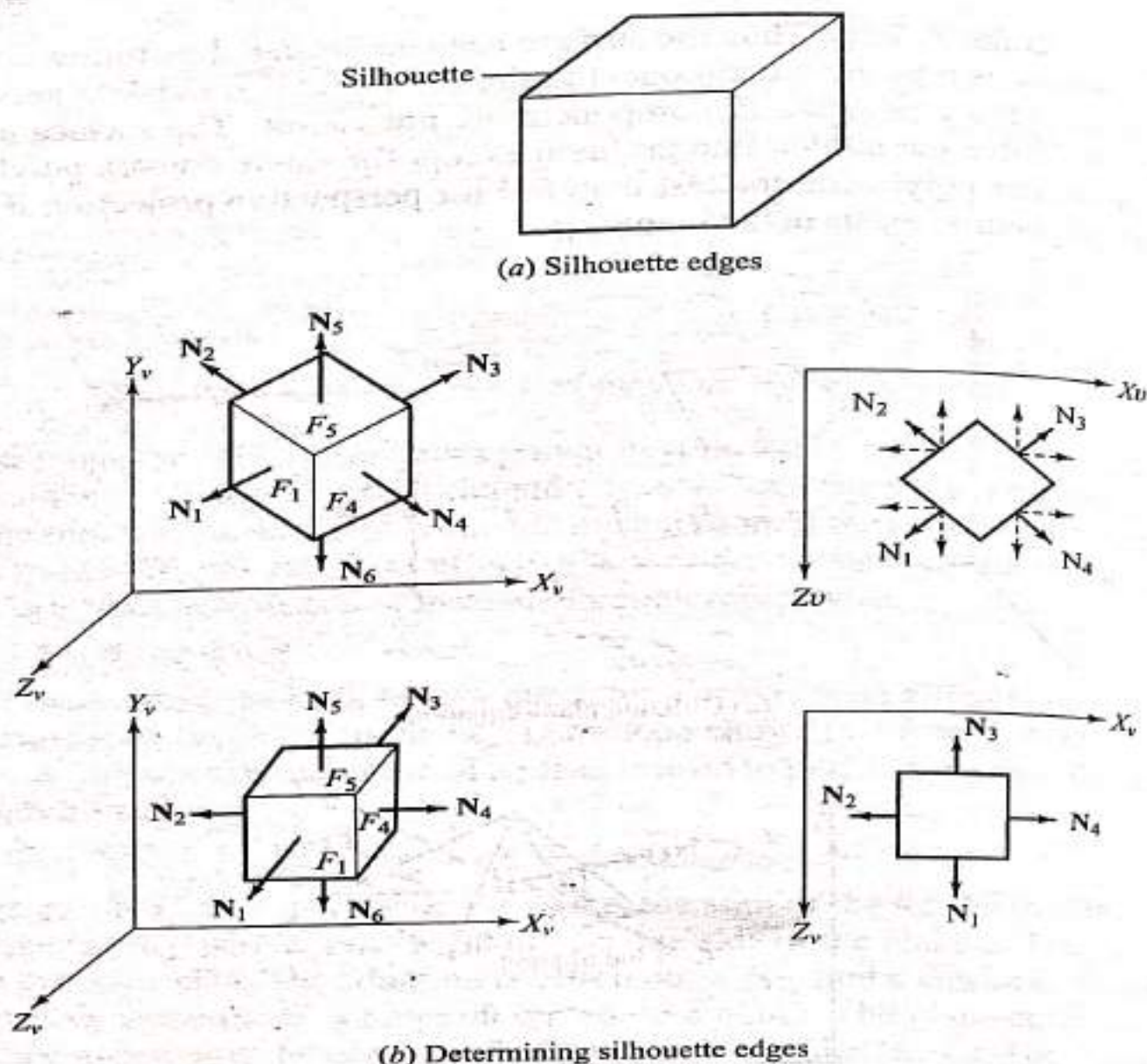


Fig. 9.7 Silhouette Edges of a Polyhedral Object

Figure 9.7b shows how to compute the silhouette edges. One cube is oriented at an angle with the axes of the VCS and the other is parallel to the axes. In the first case, the Z_v component of each normal is calculated (shown dashed in the figure). The edges between faces F_1 and F_2 , F_1 and F_6 , F_2 and F_5 , F_3 and F_5 , F_3 and F_4 and F_4 and F_6 are silhouette edges according to the above criterion. If a normal does not have a Z_v component, as in the second case of Fig. 9.7b, additional information is needed to compute the silhouette. This case implies that the corresponding face is parallel to the Z_v axis and either the X_v or Y_v axis and perpendicular to the remaining axis. For example, face F_4 is parallel to both the Z_v and Y_v axes and perpendicular to the X_v axis. Therefore, the face normal is parallel to one of the VCS axes. If this normal points in the positive direction of the axis, the face is visible. Face F_4 is visible and F_2 is not. Similarly, face F_5 is visible while F_6 is not.

Determining silhouette curves for curved surfaces follows a similar approach, but is more involved. The silhouette curve of a surface is a curve on the surface along which the Z_v component of the surface normal is zero, as shown in Fig. 9.8.

To obtain this curve, the equation of the Z_v component of the surface normal [Eq. (6.11)] is set to zero and solved for u and v . This approach is usually inconvenient because the resulting equation is difficult to solve. For a bicubic surface, the equation is a quintic polynomial in u and v . Other more efficient methods are available in the literature and are not discussed here.

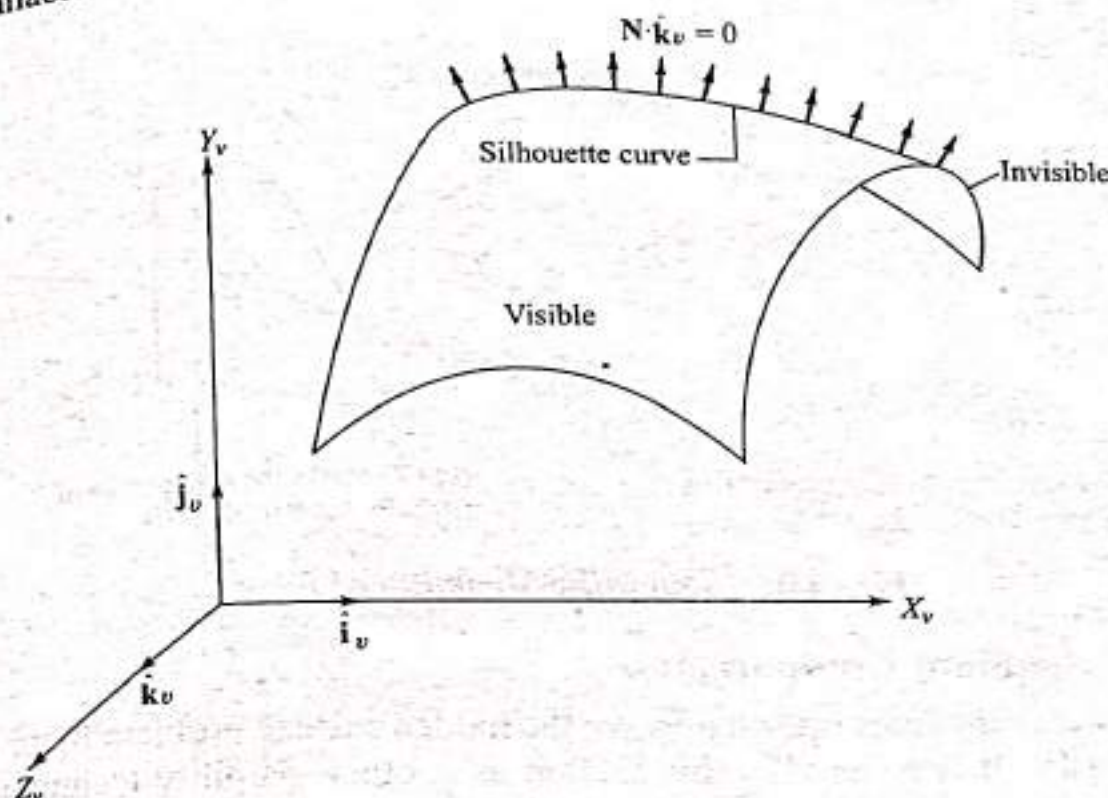
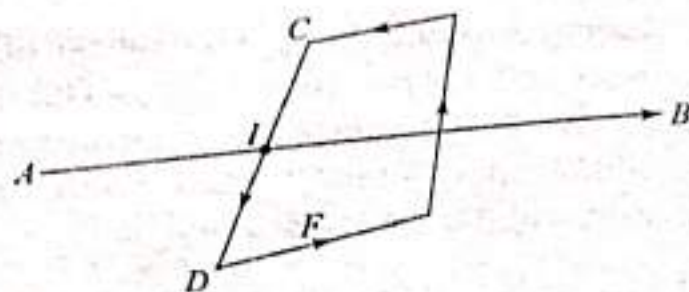
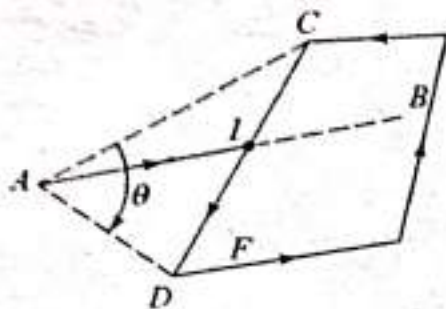
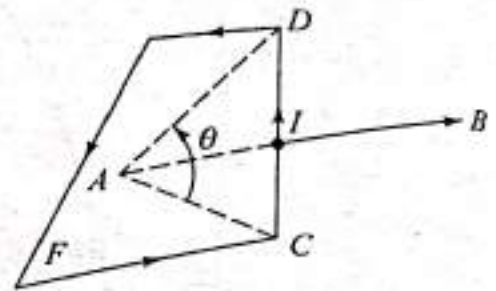


Fig. 9.8 Silhouette Curve of a Curved Surface

9.3.2.5 Edge Intersections

All the visibility techniques discussed thus far cannot determine which parts are hidden and which are visible for partially hidden edges. To accomplish this, hidden line algorithms first calculate edge intersections in two-dimensions, that is, in the $X_v Y_v$ plane of the VCS. These intersections are used to determine edge visibility. Consider the edge AB and the face F shown in Fig. 9.9. The edges of the face F are directed in a counter-clockwise direction. Let us consider the intersection I between AB and edge CD of face F . The visibility of AB with respect to F can fall into one of three cases: fully visible, I indicates the disappearance of AB , or I indicates the appearance of AB . In the first case, the depths z_v at point I are computed and compared. If it is considered a point on F , its x_v and y_v coordinates are substituted into the plane equation to find z_v . If it is a point on AB , the line equation is used instead to find the other depth. If the depth of the line is larger (we are dealing with a right-handed VCS) than the depth of the face, the line AB is fully visible (Fig. 9.9a). Otherwise, if the directed edge CD subtends a clockwise angle θ about A (Fig. 9.9b), the edge disappears. If, on the other hand, the edge subtends a counter-clockwise angle θ about A (Fig. 9.9c), the edge appears. Notice that if the face edges are directed clockwise, the angle criterion reverses. The angle criterion is sometimes referred to as the vorticity of edge CD with respect to point A .

(a) Fully visible edge AB (b) CD marks the disappearance of partially hidden edge AB (c) CD marks the appearance of partially hidden edge AB **Fig. 9.9** *Computing Visibility of Edges*

9.3.2.6 Segment Comparisons

This class of techniques is used to solve the hidden surface problem in the image (raster) space. It is covered in this section as another visibility technique. The techniques covered here are applicable to hidden surface and hidden solid algorithms as well. As discussed in Chap. 2, scan lines are arranged on a display screen from top to bottom, left to right. Therefore, instead of computing the whole correct image at once, it can be computed scan line by scan line, that is, in segments and displayed in the same order as the scan lines. Computationally, the plane of the scan line defines segments where it intersects faces in the image (see Fig. 9.10). Computing the correct image for one scan line is considerably simpler.

The segment comparisons are performed in the $X_V Z_V$ plane (Fig. 9.10). The scan line is divided into spans (dashed lines shown in the bottom of Fig. 9.10 define the bounds of the spans). The visibility is determined within each span by comparing the depths of the edge segments that lie within the span. Plane equations are used to compute these depths. Segments with maximum depth are visible throughout the span.

The strategy to divide a scan line into spans is a distinctive feature of any hidden surface algorithm. One obvious strategy is to divide the scan line at each endpoint of each edge segment (lines A , B , C and D in Fig. 9.10). A better strategy is to choose fewer spans. In Fig. 9.10, it is optimum to divide the scan line via line C into two spans only.

9.3.2.7 Homogeneity Test

The depth test described in Sec. 9.3.2.3 is concerned with comparing the depths of point sets (single points) to determine visibility. Computing homogeneity of point sets is another test to determine visibility. The notion of neighborhood (discussed

in Chap. 7) must be used to determine homogeneity. The neighborhood of a point P , denoted here by $N(P)$, in a data set is all points in the set lying inside a sphere (or a circle for two-dimensional point sets) around it.

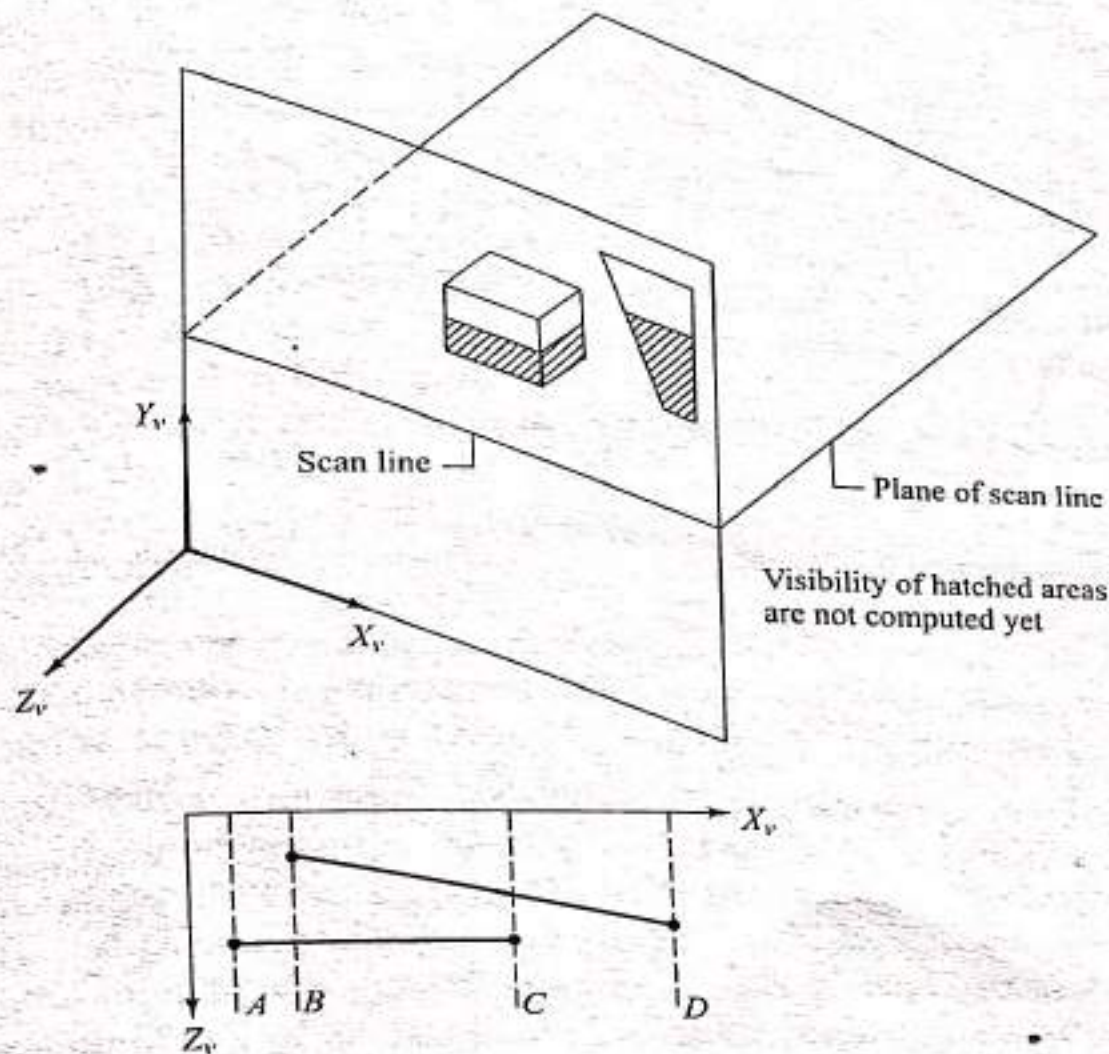


Fig. 9.10 Computing Visibility using Scan Lines

Three types of points can be identified based on computing homogeneity: homogeneously visible, homogeneously invisible and inhomogeneously visible. If a neighborhood of a point P can be bijectively projected onto a neighborhood of the projection of the point, then the neighborhood of P is visible or invisible and P is called homogeneously visible or invisible respectively. Otherwise, P is inhomogeneously visible or invisible. If we denote the projection of P by $\text{pr}(P)$, P is homogeneously visible or invisible if $\text{pr}(N(P)) = N(\text{pr}(P))$ and inhomogeneously visible or invisible if $\text{pr}(N(P)) \neq N(\text{pr}(P))$. Using this test, inner points of scenes are homogeneously visible (covering) or invisible (hidden) and contour (edge) points are inhomogeneously visible. Figure 9.11 shows an example. $N(P, F)$ denotes the neighborhood of a point P that belongs to face F . It is obvious that contour points (P_2) are inhomogeneously visible (covering) and inner points are homogeneously visible (P_1 on face F_2) or invisible (P_1 on face F_1).

Homogeneity is important for both covering and hiding. No point needs to be tested against any homogeneously visible (covering) point and no homogeneously hidden point needs to be tested against any other point, since these points are homogeneously invisible in both cases. Moreover, homogeneously invisible point

The two hidden line algorithms discussed in this section are sample algorithms to enable understanding of the basic nature of the hidden line removal problem. The area-oriented algorithm is more efficient than the priority algorithm because it hardly involves any unnecessary edge/face intersection.

9.3.7 Hidden Line Removal for Curved Surfaces

The hidden line algorithms described thus far are applicable to polyhedral objects with flat faces (planar polygons or surfaces). Fortunately, these algorithms are extendable to curved polyhedra by approximating them by planar polygons. The u - v grid offered by parametric surface representation (Chap. 5) offers such an approximation. This grid can be utilized to create a "grid surface" consisting of straight-edged regions (see Fig. 9.16) by approximating the u - v grid curves by line segments.

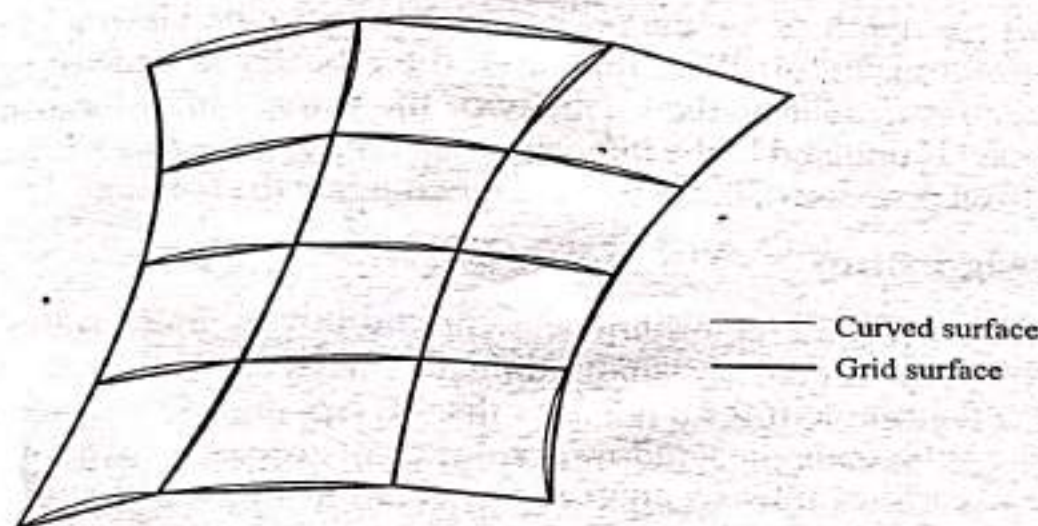


Fig. 9.16 Grid Surface Approximation of a Curved Surface

The overlay hidden line algorithm mentioned in Sec. 9.3.6 is suitable for curved surfaces. The algorithm begins by calculating the u - v grid using the surface equation. It then creates the grid surface with linear edges. Various criteria discussed previously can be utilized to determine the visibility of the grid surface.

There is no best hidden line algorithm. Many algorithms exist and some are more efficient and faster in rendering images than others for certain applications. Firmware and parallel-processing computations of hidden line algorithms make it possible to render images in real time. This adds to the difficulty of deciding on a best algorithm.

9.4 ■ HIDDEN SURFACE REMOVAL

The hidden surface removal and hidden line removal are identically one problem. Most of the concepts and algorithms described in Sec. 9.3 are applicable here and vice versa. While we limited ourselves to object-space algorithms for hidden line removal, we discuss image-space algorithms only for hidden surface removal. A wide variety of these algorithms exist. They include the z -buffer algorithm, Watkin's

algorithm, Warnock's algorithm and Painter's algorithm. Watkin's algorithm is based on scan-line coherence while Warnock's algorithm is an area-coherence algorithm. Painter's algorithm is a priority algorithm as described in the previous section for raster displays. Two sample algorithms are covered in this section.

9.4.1 The z-Buffer Algorithm

This is also known as the depth-buffer algorithm. In addition to the frame (refresh) buffer (see Chap. 2), this algorithm requires a z buffer in which z values can be sorted for each pixel. The z buffer is initialized to the smallest z value, while the frame buffer is initialized to the background pixel value. Both the frame and z buffers are indexed by pixel coordinates (x, y) . These coordinates are actually screen coordinates. The z -buffer algorithm works as follows. For each polygon in the scene, find all the pixels (x, y) that lie inside or on the boundaries of the polygon when projected onto the screen. For each of these pixels, calculate the depth z of the polygon at (x, y) . If $z > \text{depth}(x, y)$, the polygon is closer to the viewing eye than others already stored in the pixel. In this case, the z buffer is updated by setting the depth (x, y) to z . Similarly, the intensity of the frame buffer location corresponding to the pixel is updated to the intensity of the polygon at (x, y) . After all the polygons have been processed, the frame buffer contains the solution.

9.4.2 Warnock's Algorithm

This is one of the first area-coherence algorithms. Essentially, this algorithm solves the hidden surface problem by recursively subdividing the image into sub-images. It first attempts to solve the problem for a window that covers the entire image. Simple cases such as one polygon in the window or none at all are easily solved. If polygons overlap, the algorithm tries to analyze the relationship between the polygons and generates the display for the window.

If the algorithm cannot decide easily, it subdivides the window into four smaller windows and applies the same solution technique to every window. If one of the four windows is still complex, it is further subdivided into four smaller windows. The recursion terminates if the hidden surface problem can be solved for all the windows or if the window becomes as small as a single pixel on the screen. In this case, the intensity of the pixel is chosen equal to the polygon visible in the pixel. The subdivision process results in a window tree.

Figure 9.17 shows the application of Warnock's algorithm to the scene shown in Fig. 9.13a. One would devise a rule that any window is recursively subdivided unless it contains two polygons. In such a case, comparing the z depth of the polygons determines which one hides the other.

While the subdivision of the original window is governed by the complexity of the scene in Fig. 9.17, the subdivision of any window into four equal windows makes the algorithm inefficient. A more efficient way would be to subdivide a window according to the complexity of the scene in the window. This is equivalent to subdividing a window into four unequal subwindows.

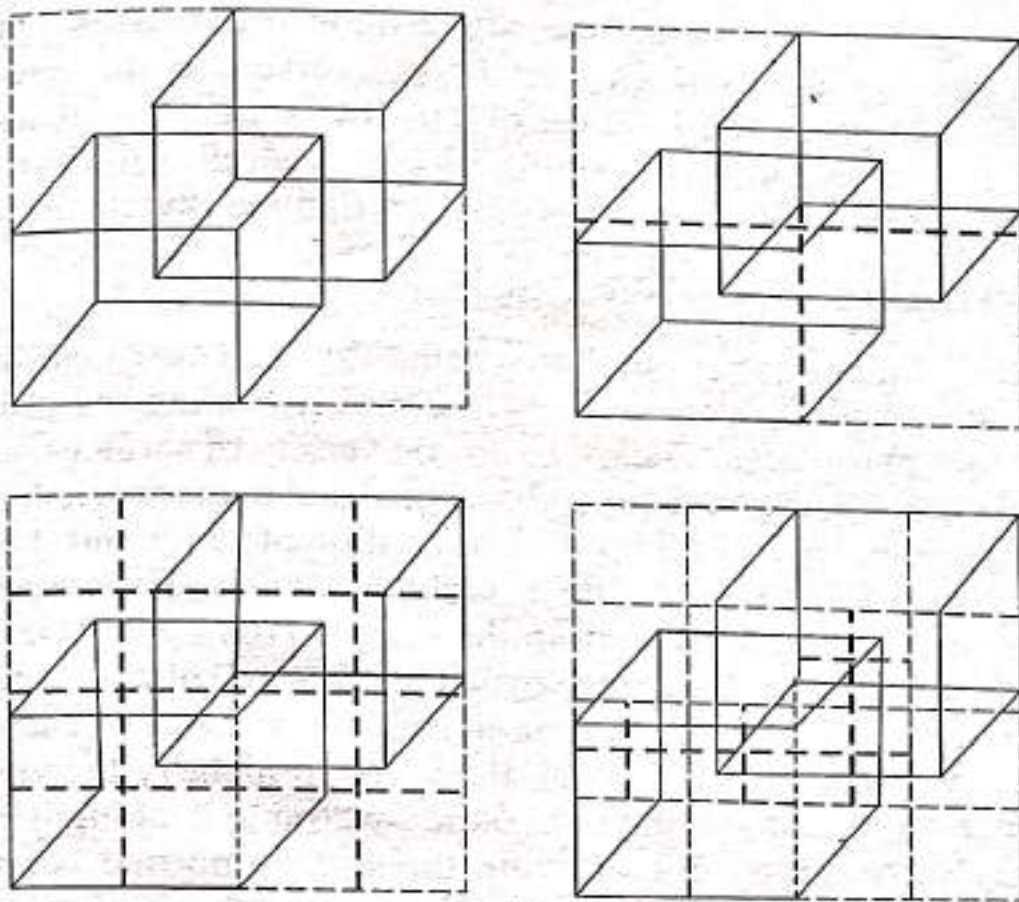


Fig. 9.17 Warnock's Algorithm

9.5 ■ HIDDEN SOLID REMOVAL

The hidden solid removal problem involves the display of solid models with hidden lines or surfaces removed. Due to the completeness and unambiguity of solid models as discussed in Chap. 6, the hidden solid removal is done fully automatically. Therefore, commands available on CAD/CAM systems for hidden solid removal, say a "hide solid" command, require minimum user input. In fact, all that is needed to execute the command is that the user identifies the solid to be hidden by digitizing it. The data structure of a solid model (see Fig. 6.29) has all the necessary information to solve the hidden line or hidden surface problem. All the algorithms discussed in Secs. 9.3 and 9.4 are applicable to hidden solid removal of B-rep models. Selected algorithms such as the z-buffer have been extended to CSG models.

For displaying CSG models, both the visibility problem and the problem of combining the primitive solids into one composite model have to be solved. There are three approaches to display CSG models. The first approach converts the CSG model into a boundary model that can be rendered with the standard hidden surface algorithms. The second approach utilizes a spatial subdivision strategy. To simplify the combinatorial problems, the CSG tree (half-spaces) is pruned simultaneously with the subdivision. This subdivision reduces the CSG evaluation to a simple preprocessing before the half-spaces are processed with standard rendering techniques.

The third approach uses a CSG hidden surface algorithm which combines the CSG evaluation with the hidden surface removal on the basis of ray classification.

The CSG ray-tracing and scan-line algorithms utilize this approach. The attractiveness of the approach lies in the conversion of the complex three-dimensional solid/solid intersection problem into a one-dimensional ray/solid intersection calculation. Due to its popularity and generality, the remainder of this section covers in more detail the ray-tracing (also called ray-casting) algorithm.

9.5.1 Ray-Tracing Algorithm

The virtue of ray tracing is its simplicity, reliability and extendability. The most complicated numerical problem of the algorithm is finding the points at which lines (rays) intersect surfaces. Therefore a wide variety of surfaces and primitives can be covered. Ray tracing has been utilized in visual realism of solids to generate line drawings with hidden solids removed, animation of solids and shaded pictures. It has also been utilized in solid analysis, mainly calculating mass properties.

The idea of ray tracing originated in the early 1970s by MAGI (Mathematic Applications Group, Inc.) to generate shaded pictures of solids. To generate these pictures, the photographic process is simulated in reverse. For each pixel in the screen, a light ray is cast through it into the scene to identify the visible surface. The first surface intersected by the ray, found by "tracing" along it, is the visible one. At the ray/surface intersection point, the surface normal is computed and knowing the position of the light source, the brightness of the pixel can be calculated.

Ray tracing is considered a brute force method for solving problems. The basic (straightforward) ray-tracing algorithm is very simple, but yet slow. The CPU usage of the algorithm increases with the complexity of the scene under consideration. Various alterations and refinements have been added to the algorithm to improve its efficiency. Moreover, the algorithm has been implemented into hardware (ray-tracing firmware) to speed its execution. In this book, we present the basic algorithm and some obvious refinements. More detailed work can be found in the bibliography at the end of the chapter.

9.5.1.1 Basics

The geometric reasoning of ray tracing stems from light rays and camera models. The geometry of a simple camera model is analogous to that of projection of geometric models. Referring to Fig. 8.18, the center of projection, projectors and the projection plane represent the focal point, light rays and the screen of the camera model respectively. For convenience, we assume that the camera model uses the VCS as described in Chap. 8 and shown in Figs. 8.19 and 8.21. For each pixel of the screen, a straight light ray passes through it and connects the focal point with the scene.

When the focal length, the distance between the focal point and screen, is infinite, parallel views result (Fig. 8.19) and all light rays become parallel to the Z_v axis and perpendicular to the screen (the $X_v Y_v$ plane). Figure 9.18 shows the geometry of a camera model as described here. The XYZ coordinate system shown is the same as the VCS. We have dropped the subscript v for simplicity. The origin of the XYZ system is taken to be the center of the screen.

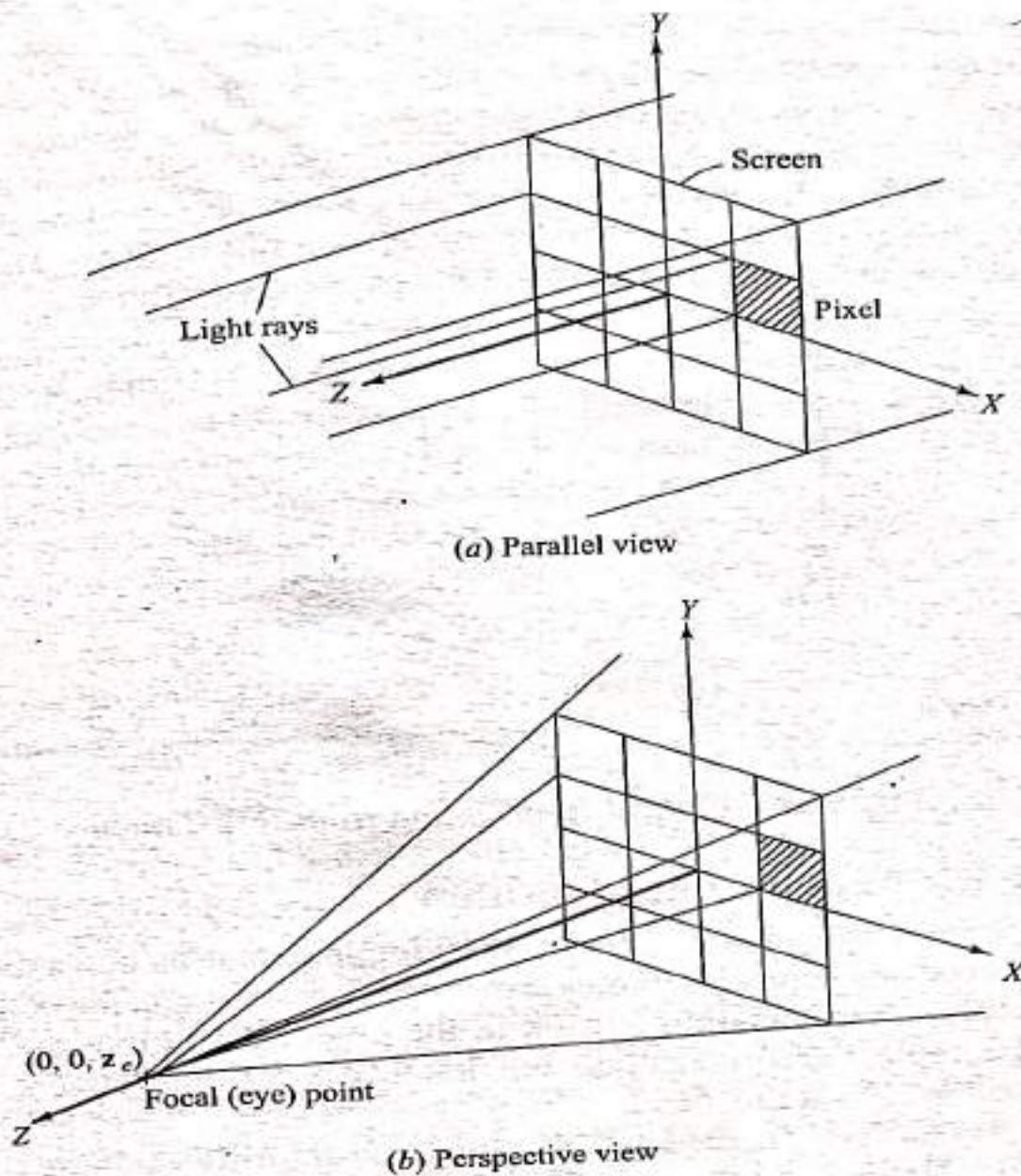


Fig. 9.18 Camera Model for Ray Tracing

A ray is a straight line which is best defined in a parametric form as a point (x_0, y_0, z_0) and a direction vector $(\Delta x, \Delta y, \Delta z)$. Thus, a ray is defined as $[(x_0, y_0, z_0) (\Delta x, \Delta y, \Delta z)]$. For a parameter t , any point (x, y, z) on the ray is given by

$$\begin{aligned} x &= x_0 + t\Delta x \\ y &= y_0 + t\Delta y \\ z &= z_0 + t\Delta z \end{aligned} \quad (9.8)$$

This form allows points on a ray to be ordered and accessed via a single parameter t . Thus, a ray in a parallel view that passes through the pixel (x, y) is defined as $[(x_0, y_0, z_0) (\Delta x, \Delta y, \Delta z)]$. In a perspective view, the ray is defined by $[(0, 0, z_e) (x, y, -z_e)]$ given the screen center $(0, 0, 0)$ and the focal point $(0, 0, z_e)$. In the parallel view, t is taken to be zero at the pixel location while it is zero at the focal point (and 1 at the pixel location) in the perspective view.

A ray-tracing algorithm takes the above ray definition given by Eqs. (9.8) as an input and output information about how the ray intersects the scene. Knowing the

camera model and the solid in the scene, the algorithm can find where the given ray enters and exits the solid, as shown in Fig. 9.19 for a parallel view. The output information is an ordered list of ray parameters, t_i , which denotes the enter/exit points and a list of pointers, S_i , to the surfaces (faces) through which the ray passes. The ray enters the solid at point t_1 , exits at t_2 , enters at t_3 and finally exits at t_4 . Point t_1 is closest to the screen and point t_4 is furthest away. The lists of ray parameters and surface pointers suffice for various applications.

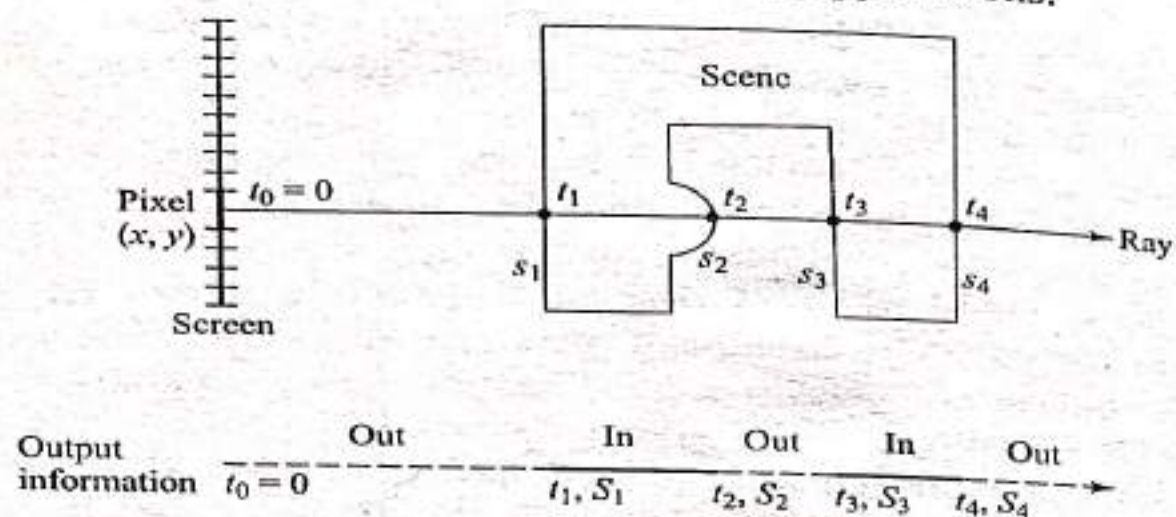


Fig. 9.19 *Output Information from Ray Tracing*

9.5.1.2 **Basic Ray-Tracing Algorithm**

While the basics of ray tracing is simple, their implementation into a solid modeler is more involved and depends largely on the representation scheme of the modeler. When boundary representation is used in the object definition, the ray-tracing algorithm is simple. For a given pixel, the first face of the object intersected by the ray is the visible face at that pixel.

When the object is defined as a CSG model, the algorithm is more complicated because CSG models are compositions of solid primitives. Intersecting the solid primitives with a ray yields a number of intersection points which requires additional calculations to determine which of these points are intersection points of the ray with the composite solid (object).

A ray-tracing algorithm for CSG models consists of three main modules: ray/primitive intersection, ray/primitive classification and ray/solid (or ray/object) classification.

Ray/primitive Intersection Utilizing the CSG tree structure, the general ray/solid intersection problem reduces to the ray/primitive intersection problem. The ray enters and exits the solid via the faces and the surfaces of the primitives.

For convex primitives (such as a block, cylinder, cone and sphere), the ray/primitive intersection test has four possible outcomes: no intersection (the ray misses the primitives), the ray is tangent to (touches) the primitive at one point, the ray lies on a face of the primitive, or the ray intersects the primitive at two different points. In the case of a torus, the ray may be tangent to it at one or two points and may intersect it at as many as four points.

The ray/primitive intersection is evaluated in the local coordinate system of the primitive (see Fig. 6.4) because it is very easy to find. The equation of a primitive

[Eqs. (6.61) to (6.66)] expressed in its local coordinate system is simple and independent of any rigid-body motion the primitive may undergo to be oriented properly in a scene. Arbitrary elliptic cylinders in the scene are all the same primitive in its local coordinate system: a right circular cylinder at the origin. Similarly, ellipsoids are the same sphere, elliptic cones are the same right circular cone and parallelepipeds are all the same block.

Given a ray originating in the screen coordinate system (SCS), it must be transformed into the primitive (local) coordinate system (PCS) via the scene (model) coordinate system (MCS) in order to find the ray/primitive intersection points. Each primitive has its local-to-scene transform and inverse, but there is only one scene-to-screen transform and inverse. The local-to-scene $[T_{LS}]$ and scene-to-screen $[T_{SS}]$ transformation matrices are determined from user input to orient primitives or define views of the scene respectively. The transformation matrix $[T]$ that transforms a ray from a local-to-screen coordinate system is given by

$$[T] = [T_{SS}][T_{LS}] \quad (9.9)$$

Therefore, a ray can be transformed to the PCS of a primitive by transforming its fixed point and direction vector:

$$\begin{bmatrix} x_0 & \Delta x \\ y_0 & \Delta y \\ z_0 & \Delta z \\ 1 & 1 \end{bmatrix}_{\text{local}} = [T]^{-1} \begin{bmatrix} x_0 & \Delta x \\ y_0 & \Delta y \\ z_0 & \Delta z \\ 1 & 1 \end{bmatrix}_{\text{screen}} \quad (9.10)$$

As discussed in Chap. 8, geometric transformation is a one-to-one correspondence. Thus, the parameter t that designates the ray/primitive intersection points need not be transformed once the intersection problem is solved in the PCS. Therefore, only rays need to be transformed between coordinate systems, not parameters.

The ray/plane intersection calculation is simple. For instance, to intersect the parameterized ray $[(x_0, y_0, z_0) (\Delta x, \Delta y, \Delta z)]$ with the XY plane, we simultaneously solve $z = 0$ and $z = z_0 + t\Delta z$ for t to get

$$t = -\frac{z_0}{\Delta z} \quad (9.11)$$

Having found t , the point of intersection is

$$[x_0 + (-z_0/\Delta z)\Delta x, y_0 + (-z_0/\Delta z)\Delta y, 0]$$

If Δz is zero, the ray is parallel to the plane, so they do not intersect. If the point of intersection lies within the bounds of the primitive, then it is a good ray/primitive intersection point. The bounds test for this point on the XY plane of a block given by Eq. (6.61) is

$$0 \leq (x_0 + t\Delta x) \leq W \quad \text{and} \quad 0 \leq (y_0 + t\Delta y) \leq H \quad (9.12)$$

Finding ray/quadric intersection points is slightly more difficult. Consider a cylindrical surface given by Eq (6.68). Substituting the x and y components of the ray's line equation into Eq. (6.68) yields

$$(x_0 + t\Delta x)^2 + (y_0 + t\Delta y)^2 = R^2 \quad (9.13)$$

Rearranging gives

$$r^2[(\Delta x)^2 + (\Delta y)^2] + 2t(x_0 \Delta x + y_0 \Delta y) + x_0^2 + y_0^2 - R^2 = 0 \quad (9.14)$$

Using the quadratic formula, we find t as

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \quad (9.15)$$

where

$$\begin{aligned} A &= (\Delta x)^2 + (\Delta y)^2 \\ B &= 2(x_0 \Delta x + y_0 \Delta y) \\ C &= x_0^2 + y_0^2 - R^2 \end{aligned} \quad (9.16)$$

Obviously, the ray will intersect the cylinder only if $A \neq 0$ and $(B^2 - 4AC) \geq 0$. Having found the one or two values of t , the bounds test for the cylindrical surface is

$$0 \leq (z_0 - t\Delta z) \leq H \quad (9.17)$$

Intersecting rays with a torus is more complicated because it is a quartic surface. It is left as an exercise (see the problems at the end of the chapter) for the reader.

Ray/primitive Classification The classification of a ray with respect to a primitive is simple. Utilizing the set membership classification function introduced in Sec. 6.5.3 and the ray/primitive intersection points, the "in," "out," and "on" segments of the ray can be found. As shown in Fig. 9.19, the odd intersection points signify the beginning of "in" segments and the end of "out" segments.

For the convex primitives, if the ray misses or touches the primitive at one point, it is classified as completely "out." If the ray intersects the primitive in two different points, it is divided into three segments: "out-in-out." If the ray lies on a face of the primitive, it is classified as "out-on-out." With respect to a torus, a ray is classified as "out," "out-in-out," or "out-in-out-in-out."

Ray/solid Classification Combining ray/primitive classifications produces the ray/solid (or ray/object) classification. Ray/solid classification produces the "in," "on," and/or "out" segments of the ray with respect to the solid. It also reorders ray/primitive intersection points and gives the closest point and surface of the solid to the camera. To combine ray/primitive classifications, a ray-tracing algorithm starts at the top of the CSG tree, recursively descends to the bottom, classifies the ray with respect to the solid primitives and then returns up the tree combining the classifications of the left and right subtrees. Combining the "on" segments requires the use of neighborhood information as discussed in Chap. 6. Figures 9.20 and 9.21 illustrate ray/solid classification. The solid lines in Fig. 9.21 are "in" segments.

The combine operation is a three-step process. First, the ray/primitive intersection points from the left and right subtrees are merged in sorted order, forming a segmented composite ray. Second, the segments of the composite ray are classified according to the boolean operator and the classifications of the left and right rays along these segments. Third, the composite ray is simplified by merging continuous segments with the same classification. Figure 9.22 illustrates these three steps for

the union operator. Combining classifications involves boolean algebra where the complement operator is replaced by the difference operator. Table 9.1 defines the combine rules.

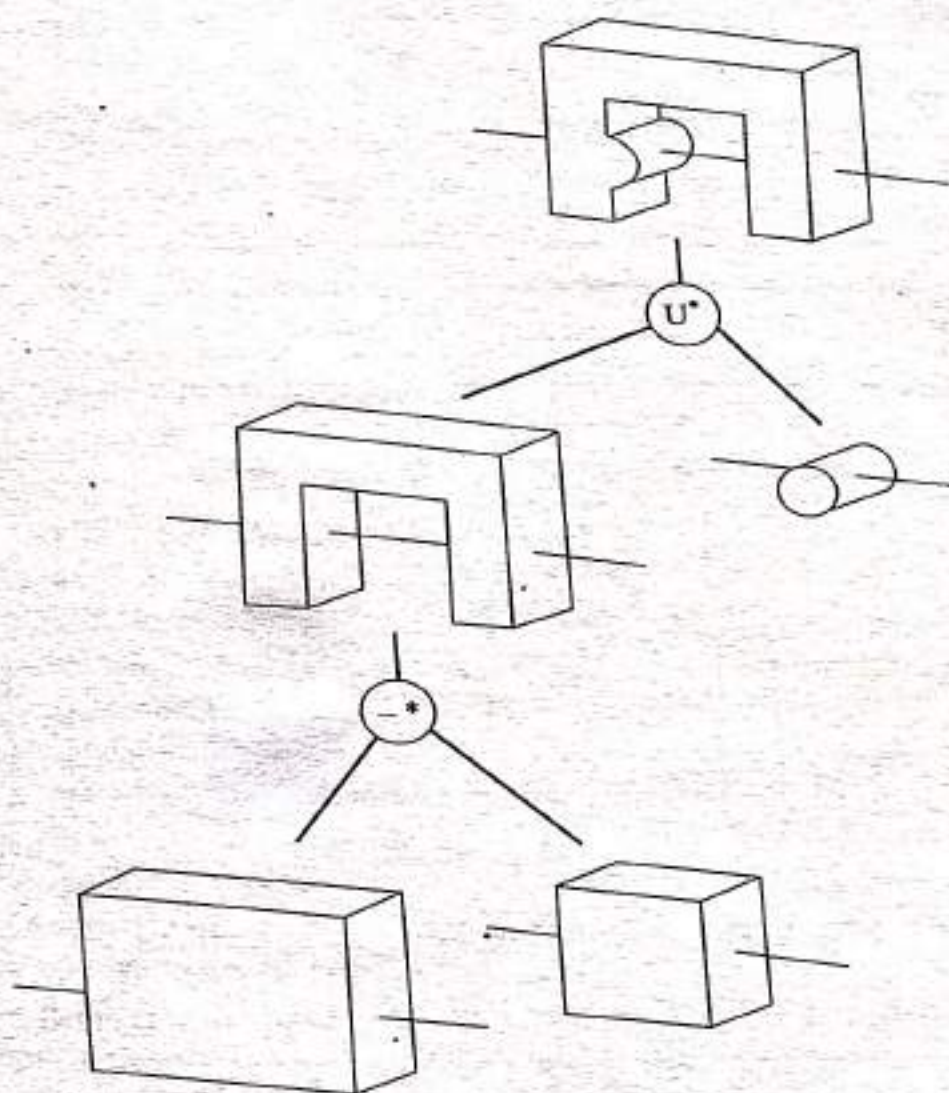


Fig. 9.20 Sample Ray and a CSG Tree

The Algorithm: To draw the visible edges of a solid, a ray per pixel is generated moving top-down, left-right on the screen. Each ray is intersected with the solid and the visible surface in the pixel corresponding to this ray is identified. If the visible surface at pixel (x, y) is different from the visible surface at pixel $(x - 1, y)$, then display a vertical line one pixel long centered at $(x - 0.5, y)$. Similarly, if the visible surface at pixel (x, y) is different from the visible surface at pixel $(x, y - 1)$, then display a horizontal line one pixel long centered at $(x, y - 0.5)$. The resulting line drawing with hidden solids removed will consist of horizontal and vertical edges only. Figure 9.23 shows a magnification of a drawing of a box with a hole. Figure 9.23a shows the pixel grid superimposed on the box and Fig. 9.23b shows the drawing only. As shown in Fig. 9.23a, the pixel-long horizontal and vertical lines may not coincide with the solid edges. However, the hidden solid still looks acceptable to the user's eyes because of the small size of each pixel.

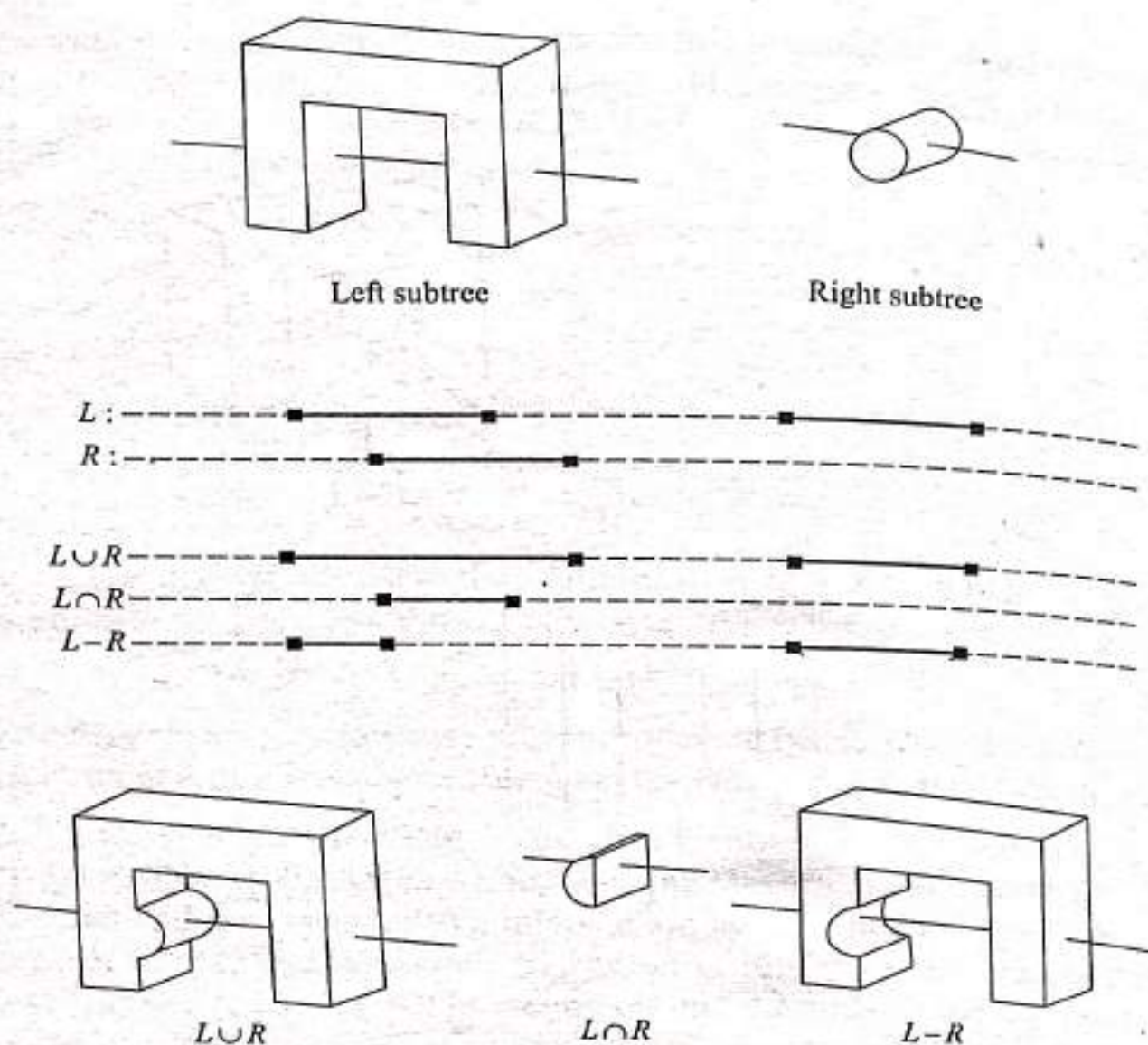


Fig. 9.21 Combining Ray Classifications

Table 9.1 The Combine Rules for Boolean Algebra

Operator	Left subtree	Right subtree	Combine
Union	IN	IN	IN
	IN	OUT	IN
	OUT	IN	IN
	OUT	OUT	OUT
Intersection	IN	IN	IN
	IN	OUT	OUT
	OUT	IN	OUT
	OUT	OUT	OUT
Difference	IN	IN	OUT
	IN	OUT	IN
	OUT	IN	OUT
	OUT	OUT	OUT

In pseudo code, a ray-tracing algorithm may be expressed as follows:

```

Procedure RAY TRACE
for each pixel (x, y) do
  generate a ray through pixel (x, y)
  if solid to be hidden is not a primitive {ray classification}
  then
    do {combine}
      classify ray against left subtree {L_classify}
      classify ray against right subtree {R_classify}
      combine (L_classify and R_classify)
    end {combine}
  else
    do {primitives}
      transform the ray equation from SCS to PCS
      branch to the proper primitive case:
      Block:
        do 6 ray-plane intersection tests;
      Sphere:
        do 1 ray-quadric intersection test;
      Cylinder:
        do 2 ray-plane & 1 ray-quadric intersection tests;
      Cone:
        do 1 ray-plane & 1 ray-quadric intersection tests;
      Torus:
        do 1 ray-quartic intersection test;
      end {branch}
      classify ray against primitive
    end {primitives}
  end {ray-classification}
  find the first visible surface  $S_1$  in pixel (x, y)
  if  $S_1$  in pixel (x,y) is different than  $S_1$  in pixel (x-1, y)
  then
    display a pixel-long vertical line centered at (x-0.5, y)
  else
    if  $S_1$  in pixel (x,y) is different than  $S_1$  in pixel (x, y-1)
    then
      display a pixel-long horizontal line centered at (x, y-0.5)
    end {if}
  end {if}
end {pixel loop}

```

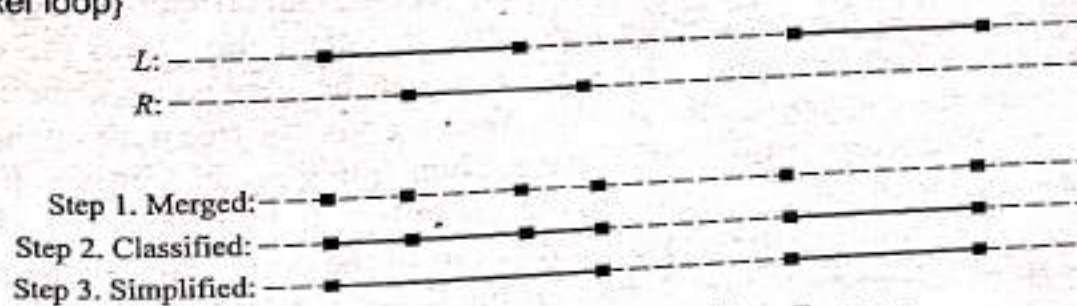


Fig. 9.22 The Three-step Combine Process

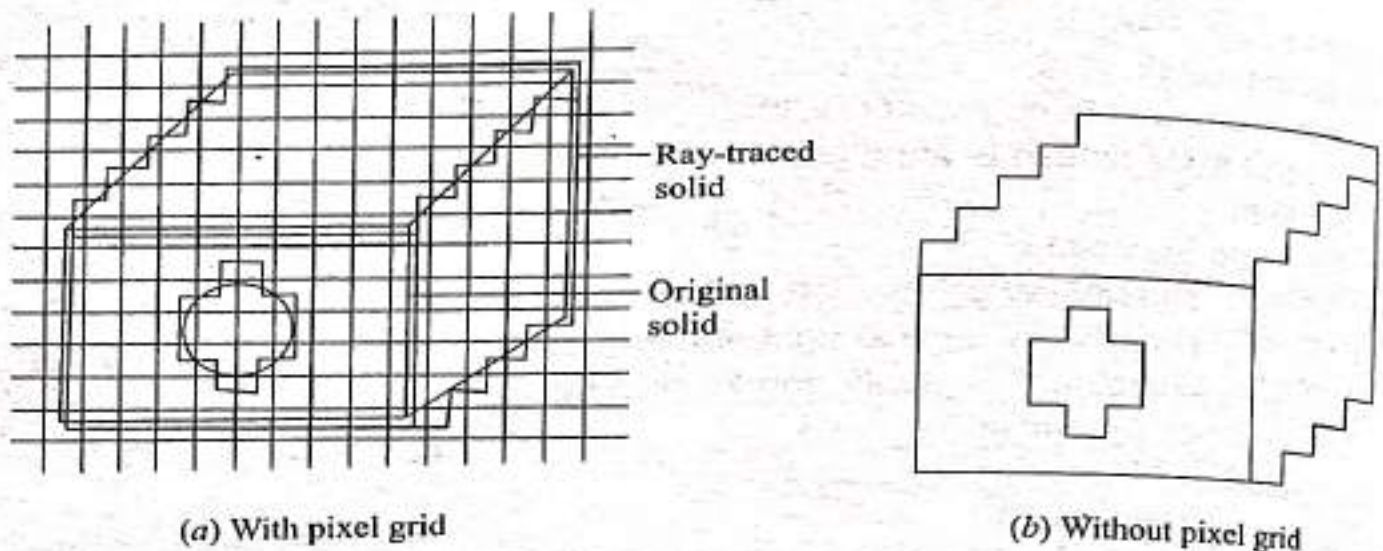


Fig. 9.23 *Solid Appearance after applying Ray-tracing Algorithm*

9.5.1.3 Improvements of the Basic Algorithm

The basic ray-tracing algorithm as described in the above section is very slow and its memory and CPU usage is directly proportional to the scene complexity, that is, to the number of primitives in the solid. In practice, the use of memory is not as much a concern as how fast the algorithm is. To appreciate the cost of using ray tracing, consider the scenario of a scene of a solid composed of 300 primitives drawn on a raster display of 500×500 pixels. Since the solid is composed of 300 primitives, its CSG tree has 300 (actually 299) composite solids, making a total of 600 solids which a ray must visit via 600 calls of the algorithm to itself. Thus $600 \times 500 \times 500$ calls of the ray-tracing algorithm are needed. At each composite solid in the tree, the left and right classifications must be combined, requiring $300 \times 500 \times 500$ classification combines. In addition, $300 \times 500 \times 500$ ray transformations from SCS to PCS are required. Finally, assuming an average of four surfaces per primitive, a total of $4 \times 300 \times 500 \times 500$ ray-intersection tests are performed. Therefore, the total cost of generating the hidden solid (or the shaded image) line drawing is the sum of these four costs.

The above high cost of the basic ray-tracing algorithm is primarily due to the multipliers 300 and 500×500 . Many applications may require casting a ray from every other pixel (or more), thus reducing the latter multiplier to 250×250 . This is equivalent to using a raster display of resolution 250×250 instead of 500×500 . The former multiplier can be reduced significantly for the large class of solids using box enclosures.

By using minimum bounding boxes around the solids in the CSG tree, the extensive search for the ray/solid intersection becomes an efficient binary search. These boxes enable the ray-tracing algorithm to detect the "clear miss" cases between the ray and the solid. The CSG tree can be viewed as a hierarchical representation of the space that the solid occupies. Thus, the tree nodes would have enclosure boxes that are positioned in space. Then, quick ray/box intersection tests guide the search in the hierarchy. When the test fails at an intermediate node in the tree, the ray is guaranteed to be classified as out of the composite; thus recursing down the solid's subtrees to investigate further is unnecessary.

Figure 9.24 shows the tree of box enclosures for the solid shown in Fig. 9.20. The ray/box intersection test is basically two dimensional because rays usually start at the screen and extend infinitely into the scene. When rays are bounded in depth (as in mass property applications), a ray/depth test can be added. Unlike the union and intersection operators, the subtraction operator does not obey the usual rules of algebra. The enclosure of $A - B$ is equal to the enclosure of A , regardless of B .

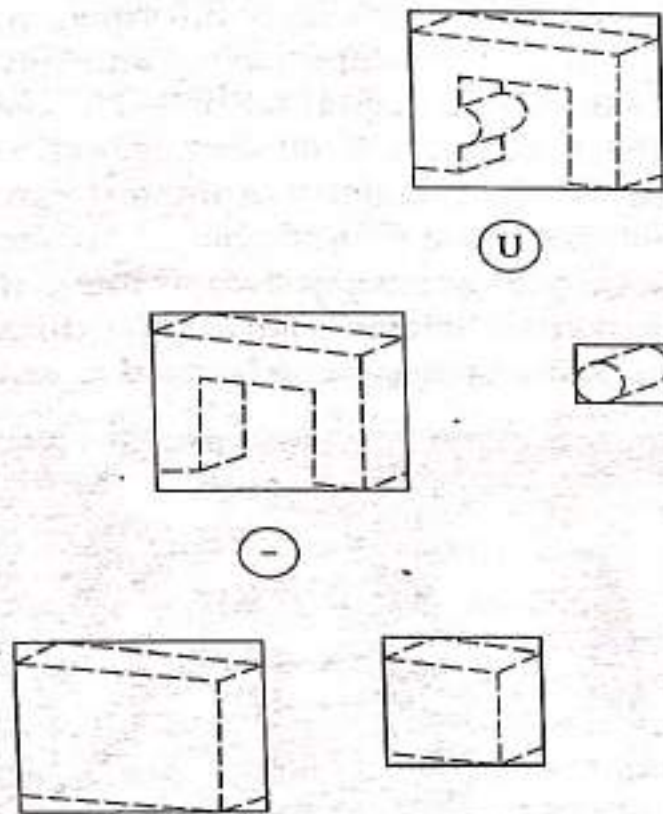


Fig. 9.24 Tree of Box Enclosures

9.5.1.4 Remarks

The ray-tracing algorithm to generate line drawings of hidden solids has few advantages. It eliminates finding, parameterizing, classifying and storing the curved edges formed by the intersection of surfaces. Finding the silhouettes of curved surface is a byproduct and can be found whenever the view changes.

The main drawbacks of the algorithm are speed and aliasing. Aliasing causes edges to be jagged and surface slivers may be overlooked. Speed is particularly important to display hidden solid line drawings in an interactive environment. If the user creates a balanced tree of the solid in the scene, the efficiency of ray tracing improves. Coherence of visible surfaces (surfaces visible at two neighboring pixels are more likely to be the same than different) can also speed up the algorithm. In addition, edges of the solid are only sought to generate line drawings. Thus, ray tracing should be concentrated around the edges and not in the open regions. This can be implemented by sparsely sampling the screen with rays and then locating (when neighboring rays identify different visible surfaces) the edges via binary searches. The sampling rate is under user control. As the sampling becomes sparser, the chance that solid edges and slivers may be overlooked becomes larger.

9.6 III SHADING

Line drawings, still the most common means of communicating the geometry of mechanical parts, are limited in their ability to portray intricate shapes. Shaded color images convey shape information that cannot be represented in line drawings. Shaded images can also convey features other than shape such as surface finish or material type (plastic or metallic look).

Shaded-image-rendering algorithms filter information by displaying only the visible surface. Many spatial relationships that are unresolved in simple wireframe displays become clear with shaded displays. Shaded images are easier to interpret because they resemble the real objects. Shaded images also have viewing problems not present in wireframe displays. Objects of interest may be hidden or partially obstructed from view, in which case various shaded images may be obtained from various viewing points. Critical geometry such as lines, arcs and vertices are not explicitly shown. Well-known techniques such as shaded-image/wireframe overlay (Fig. 9.25), transparency and sectioning can be used to resolve these problems.

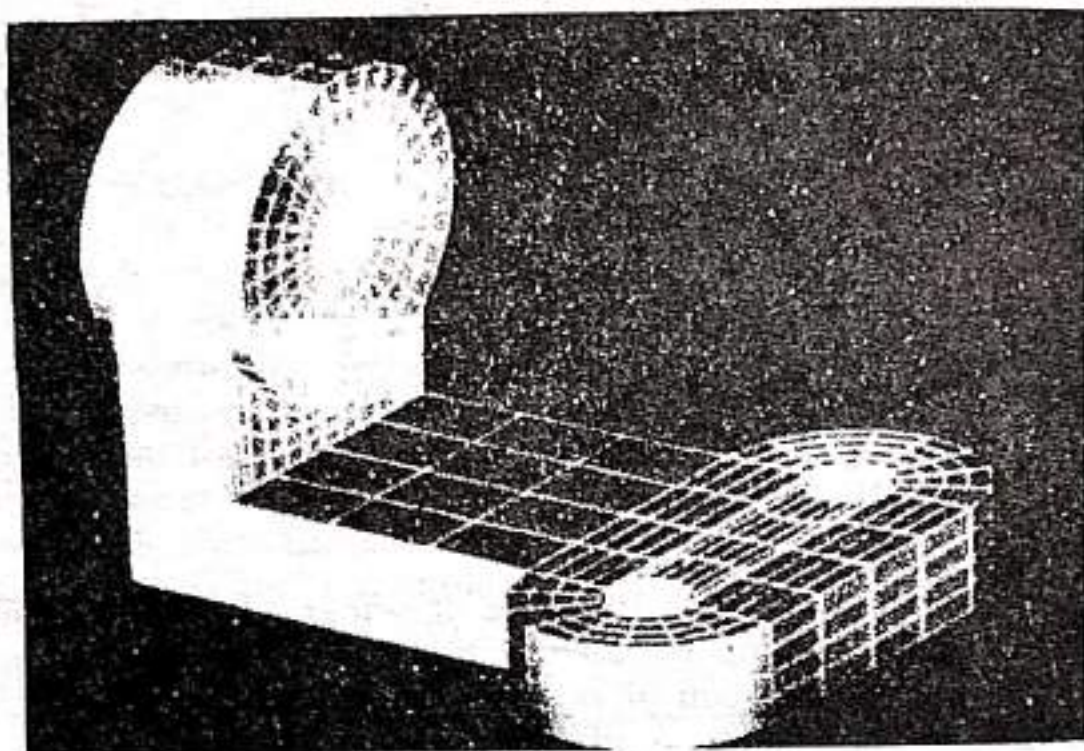


Fig. 9.25 *Shaded-image/Wireframe Overlay*

One of the most challenging problems in computer graphics is to generate images that appear realistic. The demand for shaded images began in the early 1970s when memory prices dropped enough to make the cost of raster technology attractive compared to the then-prevailing calligraphic displays. In shading a scene (rendering an image), a pinhole camera model is almost universally used. Rendering begins by solving the hidden surface removal problem to determine which objects and/or portions of objects are visible in the scene. As the visible surfaces are found, they must be broken down into pixels and shaded correctly. This process must take into account the position and color of the light sources and the position, orientation and surface properties of the visible objects.

Careful shading calculations can be distorted by defects in the hardware to display the image. Some of the common defects are noise, the spot size of the deflection beam and the fidelity of the display. The noise can occur either in the delivery of intensity (of pixels) information from the bit map of the display to its screen or in the deflection system that steers the beam over the pixel array. The spot size must be adjusted to minimize the user's perception of the raster array. An array of dots appears clearly if the spot size is too small but the sharpness of the image suffers if the spot size is too large. The fidelity of a display is a measure of how the light energy calculated by a shading model is reproduced on the screen. Nonlinearities in the intensity control circuits or in the phosphor response can distort the amount of energy actually emitted at a pixel.

9.6.1 Shading Models

Shading models simulate the way visible surfaces of objects reflect light. They determine the shade of a point of an object in terms of light sources, surface characteristics and the positions and orientations of the surfaces and sources. Two types of light sources can be identified: point light source and ambient light. Objects illuminated with only point light source look harsh because objects are illuminated from one direction only. This produces a flashlight-like effect in a black room. Ambient light is a light of uniform brightness and is caused by the multiple reflections of light from the many surfaces present in real environments.

Shading models are simple. The input to a shading model is intensity and color of light source(s), surface characteristics at the point to be shaded and the positions and orientations of surfaces and sources. The output from a shading model is an intensity value at the point. Shading models are applicable to points only. To shade an object, a shading model is applied many times to many points on the object. These points are the pixels for a raster display. To compute a shade for each point on a 1024×1024 raster display, the shading model must be calculated over one million times. These calculations can be reduced by taking advantage of shading coherence; that is, the intensity of adjacent pixels is either identical or very close.

Let us examine the interaction of light with matter to gain an insight into how to develop shading models. Particularly, we consider point light sources shining on surfaces of objects. (Ambient light adds a constant intensity value to the shade at every point.) The light reflected off a surface can be divided into two components: diffuse and specular. When light hits an ideal diffuse surface, it is reradiated equally in all directions, so that the surface appears to have the same brightness from all viewing angles. Dull surfaces exhibit diffuse reflection. Examples of real surfaces that radiate mostly diffuse light are chalk, paper and flat paints. Ideal specular surfaces reradiate light in only one direction, the reflected light direction. Examples of specular surfaces are mirrors and shiny surfaces. Physically, the difference between these two components is that diffuse light penetrates the surface of an object and is scattered internally before emerging again while specular light bounces off the surface.

The light reflected from real objects contains both diffuse and specular components and both must be modeled to create realistic images. A basic shading

model that incorporates both a point light source and ambient light can be described as follows:

$$\mathbf{I}_P = \mathbf{I}_d + \mathbf{I}_s + \mathbf{I}_b \quad (9.18)$$

where \mathbf{I}_P , \mathbf{I}_d , \mathbf{I}_s and \mathbf{I}_b are respectively the resulting intensity (the amount of shade) at point P , the intensity due to the diffuse reflection component of the point light source, the intensity due to the specular reflection component and the intensity due to ambient light. Equation (9.18) is written in a vector form to enable modeling of colored surfaces. For the common red, green and blue color system, Eq. (9.18) represents three scalar equations, one for each color. For simplicity of presentation, we develop Eq. (9.18) for one color and therefore refer to it as $I_P = I_d + I_s + I_b$ from now on (drop the vector notation).

To develop the intensity components in Eq. (9.18), consider the shading model shown in Fig. 9.26. The figure shows the geometry of shading a point P on a surface S due to a point light source. An incident ray falls from the source to P at an angle θ (angle of incidence) measured from the surface unit normal $\hat{\mathbf{n}}$ at P . The unit vector $\hat{\mathbf{i}}$ points from the light source to P . The reflected ray leaves P with an angle of reflection θ (equal to the angle of incidence) in the direction defined by the unit vector $\hat{\mathbf{r}}$. The unit vector $\hat{\mathbf{v}}$ defines the direction from P to the viewing eye.

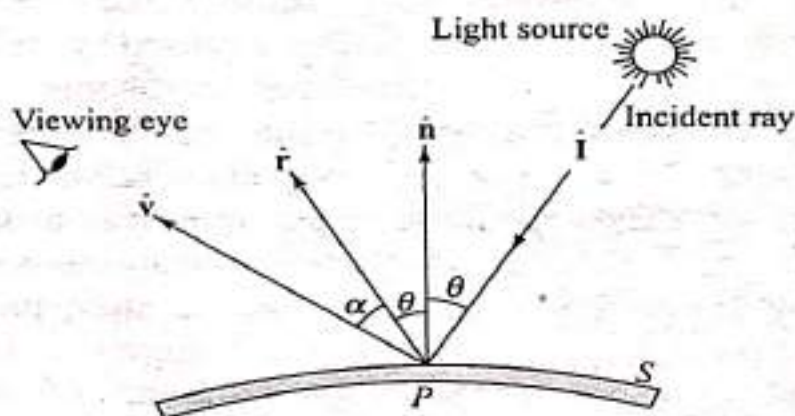


Fig. 9.26 The Geometry of Shading a Point

9.6.1.1 Diffuse Reflection

Lambert's cosine law governs the diffuse reflection. It relates the amount of reflected light to the cosine of the angle θ between $\hat{\mathbf{i}}$ and $\hat{\mathbf{n}}$. Lambert's law implies that the amount of reflected light seen by the viewer is independent of the viewer's position. The diffuse illumination is given by

$$I_d = I_L K_d \cos \theta \quad (9.19)$$

where I_L and K_d are the intensity of the point light source and the diffuse-reflection coefficient respectively. K_d is a constant between 0 and 1 and varies from one material to another. Replacing $\cos \theta$ by the dot product of $\hat{\mathbf{i}}$ and $\hat{\mathbf{n}}$, we can rewrite Eq. (9.19) as

$$I_d = I_L K_d (\hat{\mathbf{n}} \cdot \hat{\mathbf{i}}) \quad (9.20)$$

Note that since diffuse light is radiated equally in all directions, the position of the viewing eye is not required by the computations and the maximum intensity occurs when the surface is perpendicular to the light source. On the other hand, if the

angle of incidence θ exceeds 90, the surface is hidden from the light source and I_d must be set to zero. A sphere shaded with this model (diffuse reflection only) will be brightest at the point on the surface between the center of the sphere and the light source and will be completely dark on the far half of the sphere from the light.

Some shading models assume the point source of light to be coincident with the viewing eye, so no shadows can be cast. For parallel projection, this means that light rays striking a surface are all parallel. This means that $\hat{n} \cdot \hat{l}$ is constant for the entire surface, that is, the intensity I_d is constant for the surface, as shown by Eq. (9.20).

9.6.1.2 Specular Reflection

Specular reflection is a characteristic of shiny surfaces. Highlights visible on shiny surfaces are due to specular reflection while other light reflected from these surfaces is caused by diffuse reflection. The location of a highlight on a shiny surface depends on the directions of the light source and the viewing eye. If you illuminate an apple with a bright light, you can observe the effects of specular reflection. Note that at the highlight the apple appears to be white (not red), which is the color of the incident light.

The specular component is not as easy to compute as the diffuse component. Real objects are nonideal specular reflectors and some light is also reflected slightly off axis from the ideal light direction (defined by vector \hat{r} in Fig. 9.26). This is because the surface is never perfectly flat but contains microscopic deformations.

For ideal (perfect) shiny surfaces (such as mirrors), the angles of reflection and incidence are equal. This means that the viewer can only see specular reflected light when the angle α (Fig. 9.26) is zero. For nonideal (nonperfect) reflectors, such as an apple, the intensity of the reflected light drops sharply as α increases. One of the reasonable approximations to the specular component is an empirical approximation and takes the form

$$I_s = I_L W(\theta) \cos^n \alpha \quad (9.21)$$

For real objects, as the angle of incidence (θ) changes, the ratio of incident light to reflected light also changes and $W(\theta)$ is intended to model the change. In practice, however, $W(\theta)$ has been ignored by most implementors or very often is set to a constant K_s , which is selected experimentally to produce aesthetically pleasing results.

The value of n is the shininess factor and typically varies from 1 to 200, depending on the surface. For a perfect reflector, n would be infinite. $\cos^n \alpha$ reaches a maximum when the viewing eye is in the direction of \hat{r} ($\alpha = 0$). As n increases, the function dies off more quickly in the off-axis direction. Thus, a shiny surface with a concentrated highlight would have a large value of n , while a dull surface with the highlight covering a large area on the surface would have a low value of n , as shown in Fig. 9.27. Replacing $\cos \alpha$ by the dot product of \hat{r} and \hat{v} , we can rewrite Eq. (9.21) as

$$I_s = I_L W(\theta) (\hat{r} \cdot \hat{v})^n \quad (9.22)$$

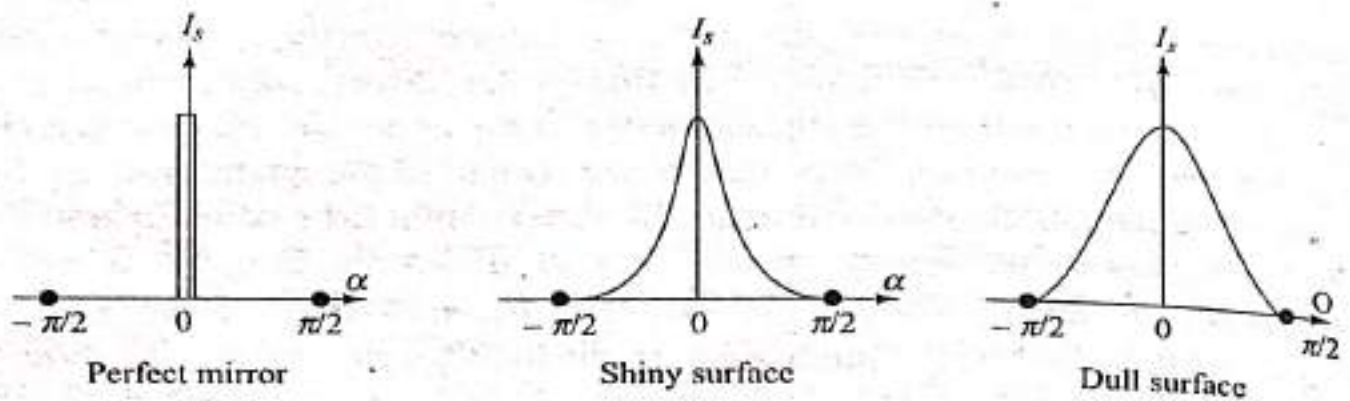


Fig. 9.27 *The Reflectance of Various Surfaces as a Function of α*

If both the viewing eye and the point source of light are coincident at infinity, $\hat{\mathbf{r}} \cdot \hat{\mathbf{v}}$ becomes constant for the entire surface. This is because $\hat{\mathbf{n}} \cdot \hat{\mathbf{v}}$, that is, $\cos(\theta + \alpha)$ and $\hat{\mathbf{n}} \cdot \hat{\mathbf{I}}$, that is, $\cos \theta$, become constant.

Other and more accurate, shading models for specular reflection have been developed and are available but are not discussed in this book. Among these realistic models are the Blinn and Cook and Torrance models.

9.6.1.3 Ambient Light

Ambient light is a light with uniform brightness. It therefore has a uniform or constant intensity I_a . The intensity at point P due to ambient light can be written as:

$$I_b = I_a K_a \quad (9.23)$$

where K_a is a constant which ranges from 0 to 1. It indicates how much of the ambient light is reflected from the surface to which point P belongs.

Substituting Eqs. (9.20), (9.22) and (9.23) into Eq. (9.18), we obtain

$$I_P = I_a K_a + I_L [K_d (\hat{\mathbf{n}} \cdot \hat{\mathbf{I}}) + W(\theta) (\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^n] \quad (9.24)$$

If $W(\theta)$ is set to the constant K_s , this equation becomes

$$I_P = I_a K_a + I_L [K_d (\hat{\mathbf{n}} \cdot \hat{\mathbf{I}}) + K_s (\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^n] \quad (9.25)$$

All the unit vectors can be calculated from the geometry of the shading model while constants and intensities on the right-hand side of the above equation are assumed by the model. Additional intensity terms can be added to the equation if more shading effects such as shadowing, transparency and texture are needed. Some of these effects are discussed later in this section.

9.6.2 Shading Surfaces

Once we know how to shade a point [Eq. (9.25)] we can consider how to shade a surface. To calculate shading precisely, Eq. (9.25) can be applied to each point on the surface. Relevant points on the surface have the same locations in screen coordinates as the pixels of the raster display. Determining these points is an outcome of hidden surface removal. The normal unit vector $\hat{\mathbf{n}}$ used in Eq. (9.25) depends on the surface geometry and can be computed a new for each point of the display. This would require a large number of calculations. Sometimes they are evaluated incrementally. If a bicubic surface is to be shaded in this way, its parametric equation

is used to subdivide it into patches whose sizes are equal to or less than the pixel size. After the visible surfaces are determined, the unit normal vector of each patch is calculated exactly and the patch is shaded using Eq. (9.25).

Most surfaces, including those that are curved, are described by polygonal meshes when the visible surface calculations are to be performed by the majority of rendering algorithms. The majority of shading techniques are therefore applicable to objects modeled as polyhedra. Among the many existing shading algorithms, we discuss three of them: constant shading, Gourand or first-derivative shading and Phong or second-derivative shading.

9.6.2.1 Constant Shading

This is the simplest and less realistic shading algorithm. Since the unit normal vector of a polygon never changes, polygons will have just one shade. An entire polygon has a single intensity value calculated from Eq. (9.25). Constant shading makes the polygonal representation obvious and produces unsmooth shaded images (intensity discontinuities). Actually, if the viewing eye or the light source is very close to the surface, the shade of the pixels within the polygon will differ significantly.

The choice of point **P** (Fig. 9.26) within the polygon becomes necessary if the light source and the viewing eye are not placed at infinity. Such a choice affects calculation of the vectors $\hat{\mathbf{I}}$ and $\hat{\mathbf{v}}$. **P** can be chosen to be the center of the polygon. On the other hand, $\hat{\mathbf{I}}$ and $\hat{\mathbf{v}}$ can be calculated at the polygon corners and the average of these values can be used in Eq. (9.25).

9.6.2.2 Gourand Shading

Gourand shading is a popular form of intensity interpolation or first-derivative shading. Gourand proposed a technique to eliminate (not completely) intensity discontinuities caused by constant shading. The first step in the Gourand algorithm is to calculate surface normals. When a curved surface is being broken down into polygons, the true surface normals at the vertices of the polygons are retained. If more than one polygon shares the same vertex as shown in Fig. 9.28a, the surface normals are averaged to give the vertex normal. If smooth shading between the four polygons shown is required, then

$$\mathbf{N}_v = \frac{1}{4} (\mathbf{N}_A + \mathbf{N}_B + \mathbf{N}_C + \mathbf{N}_D) \quad (9.26)$$

If shading discontinuities are to be introduced deliberately across an edge to show a crease or a sharp edge in the object, the proper surface normals can be dropped from the above equation. For example, shading discontinuities occur along the *AD* and *BC* boundaries shown in Fig. 9.28a if we average only two face normals.

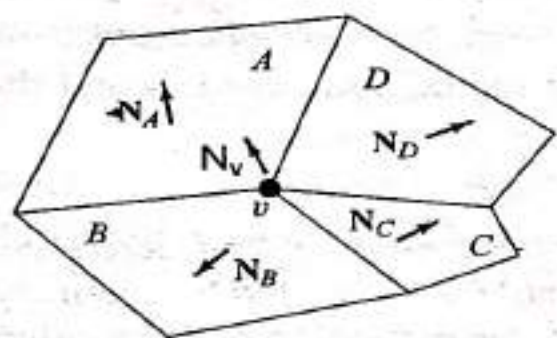
$\mathbf{N}_v = \frac{1}{2} (\mathbf{N}_A + \mathbf{N}_B)$ and $\mathbf{N}_v = \frac{1}{2} (\mathbf{N}_C + \mathbf{N}_D)$ are used to interpolate shades between polygons *A* and *B* and *C* and *D* respectively. Thus, smooth shading occurs along the *AB* and *CD* boundaries while discontinuous shading occurs along the *AD* and *BC* boundaries.

The third step in the Gourand algorithm (after calculating surface and vertex normals) is to compute vertex intensities using the vertex normals and the desired shading model [Eq. (9.25)]. The fourth and the last step is to compute the shade of each polygon by linear interpolation of vertex intensities. If the Gourand algorithm is utilized with a scan-line hidden surface algorithm, the intensity at any point P inside any polygon (Fig. 9.28b) is obtained by interpolating along each edge and then between edges along each scan line. This gives

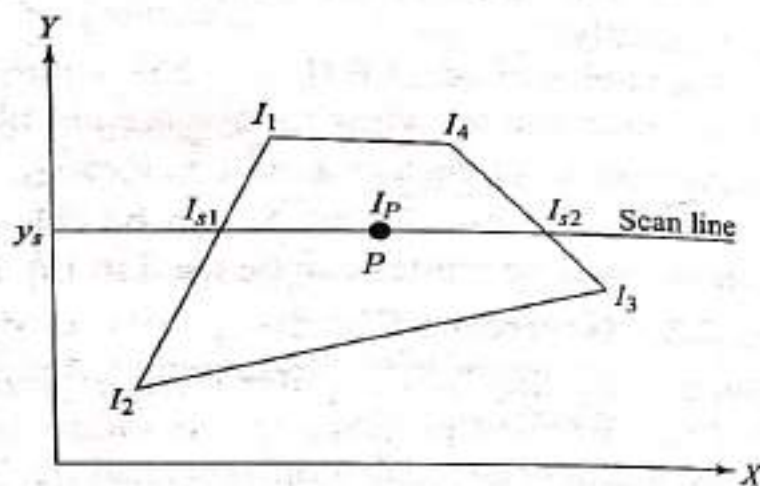
$$I_P = \frac{x_{s2} - x_P}{x_{s2} - x_{s1}} I_{s1} + \frac{x_P - x_{s1}}{x_{s2} - x_{s1}} I_{s2} \quad (9.27)$$

$$I_{s1} = \frac{y_s - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y_s}{y_1 - y_2} I_2 \quad (9.28)$$

$$I_{s2} = \frac{y_4 - y_s}{y_4 - y_3} I_3 + \frac{y_s - y_3}{y_4 - y_3} I_4 \quad (9.29)$$



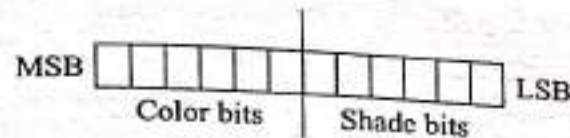
(a) Surface normals



(b) Intensity interpolation along polygon edges

Fig. 9.28 Gourand Shading

Gourand shading takes longer than constant shading and requires more planes of memory to get the smooth shading for each color. How are the bits of each pixel divided between the shading grade and the shade color? For example, one may use the red color to obtain a light red shade, a dark red shade, or any variation in between. Let us consider a display with 12 bits of color output, that is, 2^{12} or 4096 simultaneous colors. If we decide that 64 shading grades per color are required to obtain fairly smooth shades, then $4096/64 = 64$ gross different colors are possible. Actually only 63 colors are possible as the remaining one is reserved for the background. Within each color, 64 different shading grades are possible. This means that six bits of each pixel are reserved for colors and the other six for shades. The lookup table of the display would reflect this subdivision, as shown in Fig. 9.29. Usually, the six least significant bits (LSB) correspond to the shade and the six most significant bits (MSB) correspond to the color, so that when interpolation is performed to obtain shading grades the six most significant bits (i.e., the color) remain the same.



Address	Color	Shade
0 ⋮ 63	Background	—
64 ⋮ 127	1	Light shade ⋮ Dark shade
128 ⋮ 191	2	Light ⋮ Dark
⋮	⋮	⋮
4032 ⋮ 4095	63	Light ⋮ Dark

Fig. 9.29 Split of Pixel Bits between Colors and Shades

9.6.2.3 Phong Shading

While Gourand shading produces smooth shades, it has some disadvantages. If it is used to produce shaded animation (motion sequence), shading changes in a strange way because interpolation is based on intensities and not surface normals that actually change with motion. In addition, Mach bands (a phenomenon related to how the human visual system perceives and processes visual information) are sometimes produced and highlights are distorted due to the linear interpolation of vertex intensities.

Phong shading avoids all the problems associated with Gourand shading although it requires more computational time. The basic idea behind Phong shading is to interpolate normal vectors at the vertices instead of the shade intensities and to apply the shading model [Eq. (9.25)] at each point (pixel). To perform the interpolation, Eq. (9.26) can be used to obtain an average normal vector at each vertex. Phong shading is usually implemented with scan-line algorithms. In this case Fig. 9.28b is applicable if we replace the intensities by the average normal vectors, N_v , at the vertices. Similarly, Eqs. (9.27) to (9.29) are applicable if the intensity variables are replaced by the normal vectors.

9.6.3 Shading Enhancements

The basic shading model described in Sec. 9.6.1 is usually enhanced to produce special effects for both artistic value and realism purposes. These effects include transparency, shadows, surface details and texture.

Transparency can be used to shade translucent material such as glass and plastics or to allow the user to see through the opaque material. Two shading techniques can be identified: opaque and translucent. In the opaque technique, hidden surfaces in every pixel are completely removed. In the translucent method, hidden surfaces are not completely removed. This allows some of the back pixels to show through, producing a screen-door effect.

Consider the box shown in Fig. 9.30. If the front face F_1 is made translucent, the back face F_2 can be seen through F_1 . The intensity at a pixel coincident with the locations of points P_1 and P_2 can be calculated as a weighted sum of the intensities at these two points, that is,

$$I = KI_1 + (1 - K)I_2 \quad (9.30)$$

where I_1 and I_2 are the intensities of the front and back faces respectively, calculated using, say, Eq. (9.25). K is a constant that measures the transparency of the front face: when $K = 0$, the face is perfectly transparent and does not change the intensity of the pixel; when $K = 1$, the front face is opaque and transmits no light. Sometimes transparency is referred to as X-ray due to the similarity in effect.

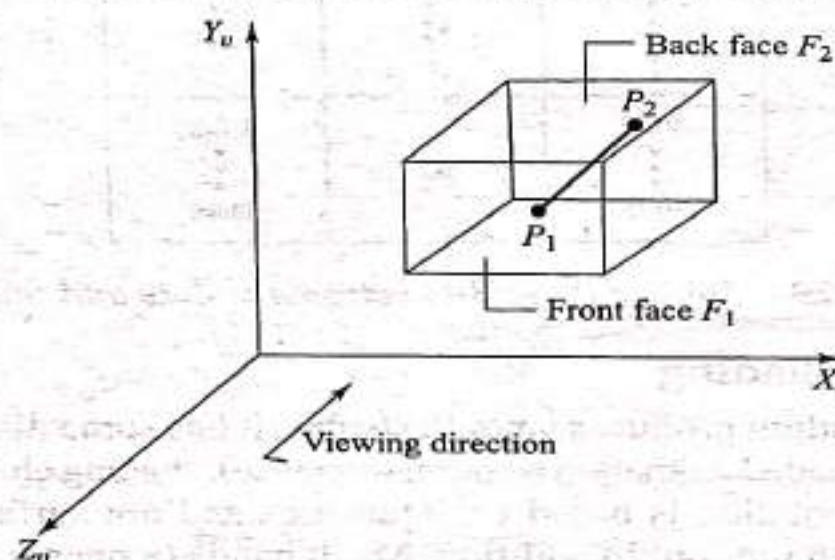


Fig. 9.30 Transparency and Visibility of Back Faces

Shadows are important in conveying realism to computer images. More importantly, they facilitate the comprehension of spatial relationships between objects of one image. The complexity of a shadow algorithm is related to the model of the light source. If it is a point source outside the field of view at infinity, the problem is simplified. Finding which objects are in shadow is equivalent to solving the hidden surface problem as viewed from the light source. If several light sources exist in the scene, the hidden surface problem is solved several times—every time one of the light sources is considered as the viewing point. The surfaces that are visible to both the viewer and the light source are not shaded. Those that are visible to the viewer but not to the light source are shaded.

Surface details that are usually needed to add realism to the surface image are better treated as shading data than as geometrical data. Consider, say, adding a logo of an object to its image. The logo can be modeled using “surface-detail” polygons. Polygons of the object geometric model point to these surface-detail

polygons. If one of the geometric model polygons is to be split for visibility reasons, its corresponding surface-detail polygon is split in the exact fashion. Surface-detail polygons obviously cover the surface polygons when both overlap. When the shaded image is generated, the surface details are guaranteed to be visible with their desired color attributes. Separating the polygons of geometric models and surface details speeds up the rendering of images significantly and reduces the possibility of generating erroneous images.

Texture is important to provide the illusion of reality. For example, modeling of a rough casting should include the rough texture nature of its surfaces. These objects, rich in high frequencies, could be modeled by many individual polygons, but as the number of polygons increases, they can easily overflow the modeling and display programs. Texture mapping (Fig. 9.31) is introduced to solve this problem and provide the illusion of complexity at a reasonable cost. It is a method of "wallpapering" the existing polygons. As each pixel is shaded, its corresponding texture coordinates are obtained from the texture map and a lookup is performed in a two-dimensional array of colors containing the texture. The value in this array is used as the color of the polygon at this pixel, thus providing the "wallpaper."

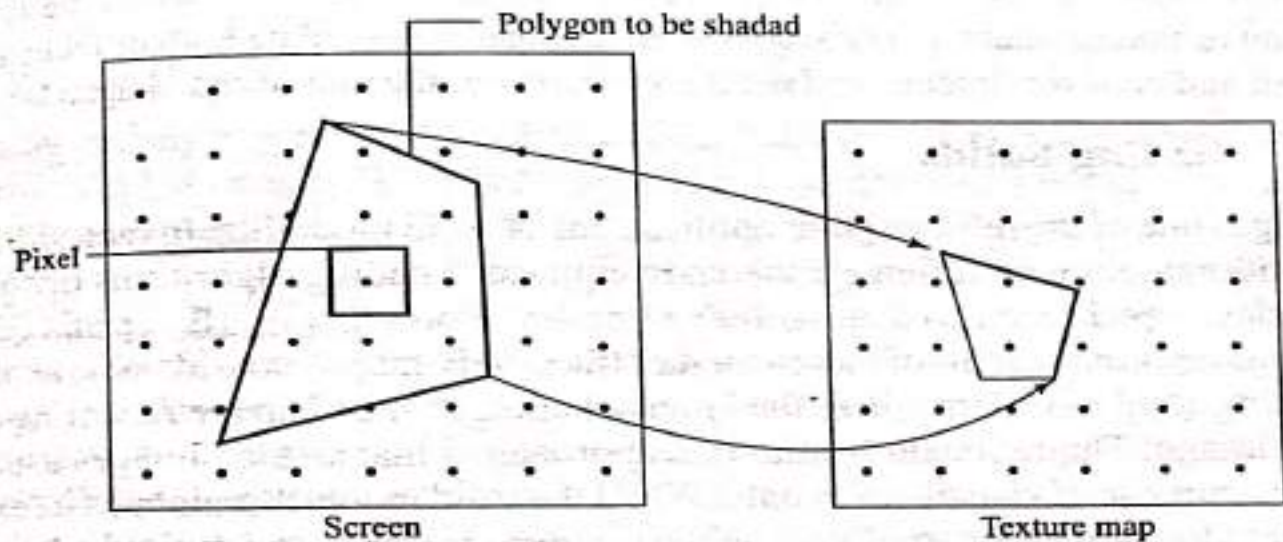


Fig. 9.31 *Texture Mapping*

Three-dimensional texture mapping is easier to use with three-dimensional objects than two-dimensional texture mapping. It is a function that maps the object's spatial coordinates into three-dimensional texture space and uses three-dimensional textures. Thus, no matter what the object's shape is, the texture on its surface is consistent. This is useful to model materials such as wood and marble. Due to the space required to store a three-dimensional array of pixels, procedural textures can be used. However, they are difficult to be antialiased. Texture mapping can contain other surface properties besides color to increase the illusion of complexity. For example, surface normal perturbations could be stored in the texture map (bump mapping) to enable the simulation of wrinkled surfaces.

Example 9.1 Apply the shading model given by Eq. (9.25) to the scene of the two boxes shown in Fig. 9.13a. Assume that the visible surfaces have been identified as shown in Fig. 9.13c. Use $n = 100$ for the specular reflection component.

Solution Let us assume that the point light source is placed at infinity, that is, coincident with the viewing eye. Table 9.2 can be derived to enable use of Eq. (9.25). The table shows the components of the vectors of each face in the VCS shown in Fig. 9.13a. Substituting the values in Table 9.2 into Eq. (9.25) gives

$$\text{Face } F_1: I_P = I_a K_a + I_L [-K_d + (K_s)^{100}]$$

$$\text{Face } F_2: I_P = I_a K_a + I_L (K_s)^{100}$$

$$\text{Face } F_3: I_P = I_a K_a + I_L (K_s)^{100}$$

$$\text{Face } F_4: I_P = I_a K_a + I_L [-K_d + (K_s)^{100}]$$

$$\text{Face } F_5: I_P = I_a K_a + I_L (K_s)^{100}$$

$$\text{Face } F_6: I_P = I_a K_a + I_L (K_s)^{100}$$

The intensities I_a and I_L are chosen based on the maximum intensity a pixel may have. The coefficients K_a , K_d and K_s are chosen based on experimental measurements. The above equations assume constant shading. Notice that the shading of F_1 and F_4 and F_2 , F_3 , F_5 and F_6 are equal. This will make F_2 and F_3 and F_5 and F_6 indistinguishable in the shaded image. A solution to this problem would be to use Gourand or Phong shading. The reader is encouraged to calculate both of them as an exercise and compare intensities (see the problems at the end of the chapter).

9.6.4 Shading Solids

Shading is one of the most popular applications of solid modeling. In fact, shading and solid modeling are often erroneously equated. Shading algorithms of solids can be developed based on exact solid's representation schemes (B-rep and CSG) or on some approximations of these schemes (faceted B-rep). A considerable number of existing solid modelers utilize the latter schemes to speed up the rendering of a shaded image. The rationale behind this approach is that shaded images usually serve the purpose of visualization only. While the solid modeler maintains its exact representation scheme internally for analysis purposes (mass property calculations, NC tool path generation and finite element modeling), an approximate representation (polygonal approximation of exact geometry) is derived for shading purposes. Sometimes, these exact and approximate (for visualization purposes) representations are referred to as analytic and visual solid modelers respectively.

Table 9.2 Vectors for Example 9.1

Face	\hat{n}	\hat{l}	\hat{r}	\hat{v}
F_1	(0, 0, 1)	(0, 0, -1)	(0, 0, 1)	(0, 0, 1)
F_2	(1, 0, 0)	(0, 0, -1)	(0, 0, 1)	(0, 0, 1)
F_3	(0, 1, 0)	(0, 0, -1)	(0, 0, 1)	(0, 0, 1)
F_4	(0, 0, 1)	(0, 0, -1)	(0, 0, 1)	(0, 0, 1)
F_5	(1, 0, 0)	(0, 0, -1)	(0, 0, 1)	(0, 0, 1)
F_6	(0, 1, 0)	(0, 0, -1)	(0, 0, 1)	(0, 0, 1)

and right (Fig. 9.35b) neighbors of the octant being processed (call it the active octant) are utilized. If the active octant has a void (completely empty) octant on the left side, a surface normal exists in this direction. If any other neighbors are void, surface normal vectors must lie along the same area as well. The computations of these vectors are easy because the spatial orientations of the octants in the octree are known.

Transparency can be modeled with octree structures. For opaque materials (as described above), the quadtree is simply overwritten if an octant is found to obscure it. For transparent materials, however, the intensity of the old quadrant is used to add a contribution to the existing quadrant by using, say, Eq. (9.30).

9.7 ≡ COLORING

The use of colors in CAD/CAM has two main objectives: facilitate creating geometry and display images. Colors can be used in geometric construction. In this case various wireframe, surface, or solid entities can be assigned different colors to distinguish them. Color is one of the two main ingredients (the second being texture) of shaded images produced by shading algorithms. In some engineering applications such as finite element analysis, colors can be used effectively to display contour images such as stress or heat-flux contours.

Black and white raster displays provide achromatic colors while color displays (or television sets) provide chromatic color. Achromatic colors are described as black, various levels of gray (dark or light gray) and white. The only attribute of achromatic light is its intensity, or amount. A scalar value between 0 (as black) and 1 (as white) is usually associated with the intensity. Thus, a medium gray is assigned a value of 0.5. For multiple-plane displays, different levels (scale) of gray can be produced. For example, 256 (2^8) different levels of gray (intensities) per pixel can be produced for an eight-plane display. The pixel value V_i (which is related to the voltage of the deflection beam) is related to the intensity level I_i by the following equation:

$$V_i = \left(\frac{I_i}{C} \right)^{1/\gamma} \quad (9.31)$$

The values C and γ depends on the display in use. If the raster display has no lookup table, V_i (e.g., 00010111 in an eight-plane display) is placed directly in the proper pixel. If there is a table, i is placed in the pixel and V_i is placed in entry i of the table. Use of the lookup table in this manner is called gamma correction, after the exponent in Eq. (9.31).

Chromatic colors produce more pleasing effects on the human vision system than achromatic colors. However, they are more complex to study and generate. Color is created by taking advantage of the fundamental trichromacy of the human eye. Three different colored images are combined additively at photo-receptors in the eye to form a single perceived image whose color is a combination of the three prime colors. Each of the three images is created by an electron gun acting on a color phosphor. Using shadow-mask technology, it is possible to make the images intermingle on the screen, causing the colors to mix together because of spatial

proximity. Well-saturated red, green and blue colors are typically used to produce the wide range of desired colors.

Color descriptions and specifications generally include three properties: hue, saturation and brightness. Hue associates a color with some position in the color spectrum. Red, green and yellow are hue names. Saturation describes the vividness or purity of a color or it describes how diluted the color is by white light. Pure spectral colors are fully saturated colors and grays are desaturated colors. Brightness is related to the intensity, value, or lightness of the color.

There exists a rich wealth of studies and methods of how to specify and measure colors. Some methods are subjective such as Munsell and pigment-mixing methods. The Munsell method is widely used and is based on visually comparing unknown colors against a set of standard colors. The pigment-mixing method is used by artists. Other methods used in physics are objective and treat visible light with a given wavelength as an electromagnetic energy with a spectral energy distribution. Our primary interest in this section is not to review these studies and methods but to describe some existing color models, so that application programs can choose the desired colors properly. We will also show how some of these models can be converted to red, green and blue since most of the commonly used CRTs demand three digital values, specifying an intensity for each of the colors.

9.7.1 Color Models

A color model or a space is a three-dimensional color coordinate system to allow specifications of colors within some color range. Each displayable color is represented by a point in a color model. There are quite a number of color models available. Some popular models are discussed here. These models are based on the red, green and blue (RGB) primaries. For any one of these models, coordinates are translated into three voltage values in order to control the display. This process is shown in Fig. 9.36, which summarizes the sequence of transformation for some models. The gamma correction is performed to obtain a linear relationship between digital RGB values and the intensity of light emitted by the CRT.

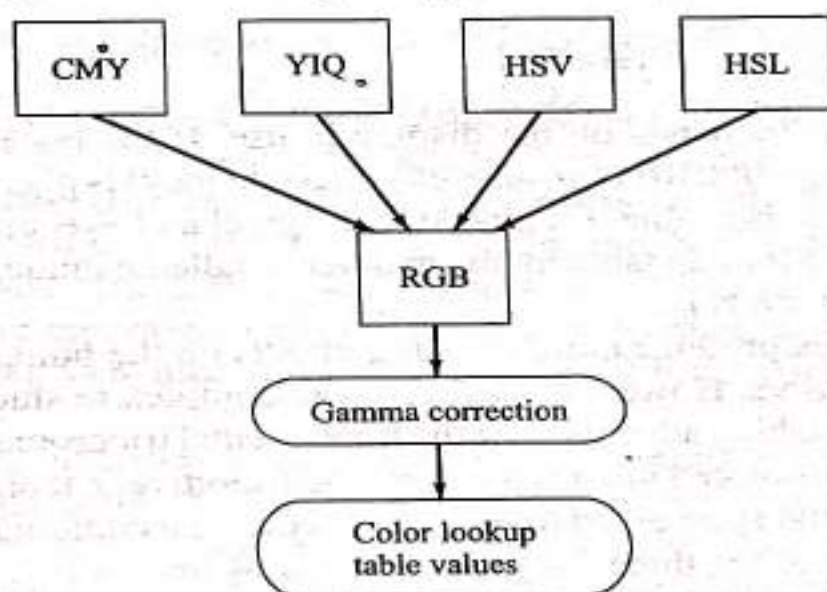


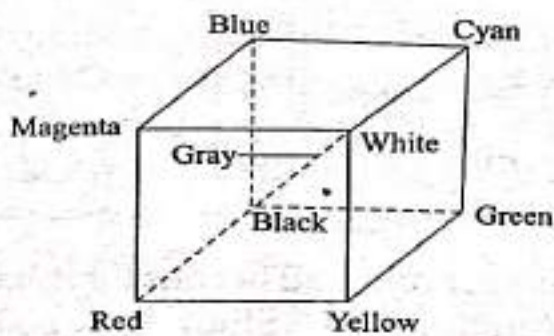
Fig. 9.36 Transformation of a Color Model to RGB

9.7.1.1 RGB Model

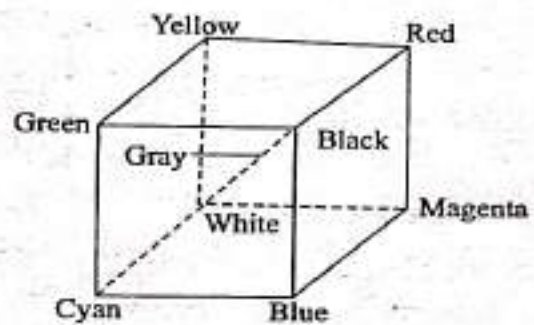
The RGB color space uses a cartesian coordinate system as shown in Fig. 9.37a. Any point (color) in the space is obtained from the three RGB primaries; that is, the space is additive. The main diagonal of the cube is the locus of equal amounts of each primary and therefore represents the gray scale or levels. In the RGB model, black is at the origin and represented by (0, 0, 0) and white is represented by (1, 1, 1). Thus in the RGB model, the lowest intensity (0 for each color) produces the black color and the maximum intensity (1 for each color) produces the white color.

9.7.1.2 CMY Model

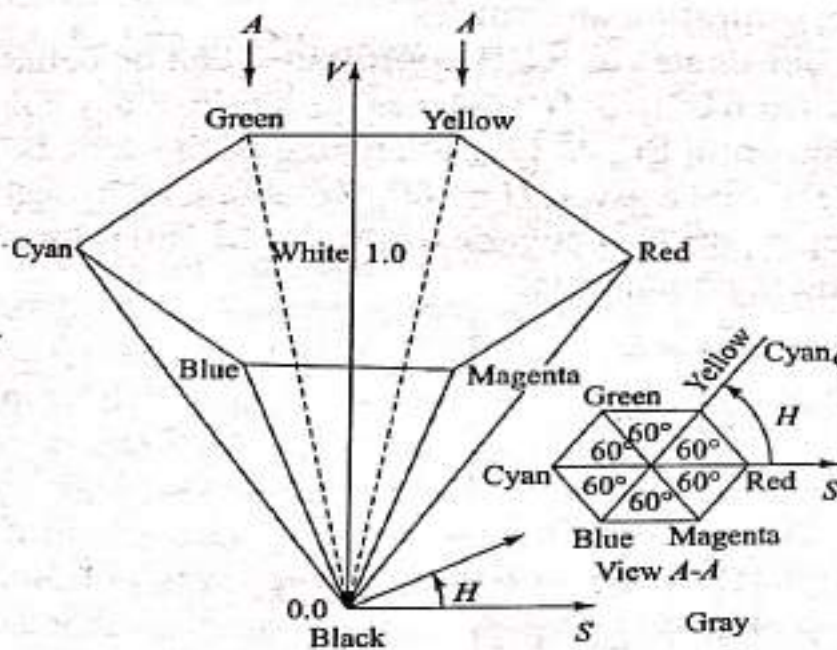
The CMY (cyan, magenta, yellow) model shown in Fig. 9.37b is the complement of the RGB model. The cyan, magenta and yellow colors are the complements of the red, green and blue respectively. The white is at the origin (0, 0, 0) of the model and the black is at point (1, 1, 1) which is opposite to the RGB model. The CMY model is considered a subtractive model because the model primary colors subtract some color from white light. For example, a red color is obtained by subtracting a cyan color from the white light (instead of adding magenta and yellow).



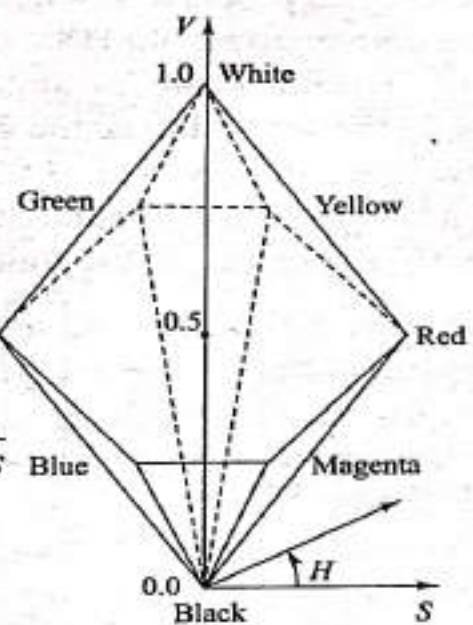
(a) RGB color space (unit cube)



(b) CMY color space (unit cube)



(c) HSV color space



(d) HSL color space

Fig. 9.37 Some Color Models

The conversion from CMY to RGB is achieved by the following equation:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix} \quad (9.32)$$

The unit column vector represents white in the RGB model or black in the CMY model.

9.7.1.3 YIQ Model

The YIQ space is used in raster color graphics. It has been in use as a television broadcast standard since 1953 when it was adopted by the National Television Standards Committee (NTSC) of the United States. It was designed to be compatible with black and white television broadcast. The Y axis of the color model corresponds to the luminance (the total amount of light). The I axis encodes chrominance information along a blue-green to orange vector and the Q axis encodes chrominance information along a yellow-green to magenta vector.

The conversion from YIQ coordinates to RGB coordinates is defined by the following equation:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.0 & 0.95 & -0.62 \\ 1.0 & -0.28 & -0.64 \\ 1.0 & -1.11 & 1.73 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix} \quad (9.33)$$

9.7.1.4 HSV Model

This color model (shown in Fig. 9.37c) is user oriented because it is based on what artists use to produce colors (Hue, Saturation and Value). It is contrary to the RGB, CMY and YIQ models which are hardware oriented. The model approximates the perceptual properties of hue, saturation and value.

The conversion from HSV coordinates to RGB coordinates can be defined as follows. The hue value H (range from 0° to 360°) defines the angle of any point on or inside the single hexacone shown in Fig. 9.37c. Each side of the cone bounds 60° as shown in view A-A. If we divide a given H by 60, we obtain an integer part i and a fractional part f . The integer part i is between 0 and 5 ($H = 360$ is treated as if $H = 0$). Let us define the following quantities:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = V \begin{bmatrix} 1 - S \\ 1 - Sf \\ 1 - S(1 - f) \end{bmatrix} \quad (9.34)$$

Then we can write:

$$(R, G, B) = \begin{cases} (V, c, a) & i = 0 \\ (b, V, a) & i = 1 \\ (a, V, c) & i = 2 \\ (a, b, V) & i = 3 \\ (c, a, V) & i = 4 \\ (V, a, b) & i = 5 \end{cases} \quad (9.35)$$

9.7.1.5 HSL Model

The HSL (Hue, Saturation, Lightness) color model shown in Fig. 9.37d forms a double hexacone space. It is used by Tektronix. The saturation here occurs at $V = 0.5$ and not 1.0 as in the HSV model. The HSL model is as easy to use as the HSV model. The conversion from HSL to RGB is possible by using the geometry of the double hexacone as we did with the HSV model.

9.8 **III** USER INTERFACE FOR SHADING AND COLORING

Major commercial CAD/CAM systems provide their users with shading packages that implement most of the shading and coloring concepts covered in Secs. 9.6 and 9.7. A shading command with the appropriate shading attributes and modifiers allows users to shade either surface or solid models. A generic syntax for a typical shading command may look as follows:

SHADE <geometric model type><shading modifiers><shading resolution>
<geometric model entities>

The type of geometric model could be a "surface" or "solid." The "shade surface" command would require the user to digitize all the surfaces of the model that are to be shaded. The back invisible surfaces of a model need not be digitized. Digitizing surfaces could be very time-consuming and error-prone for complex and dense models. Users can use the "window" modifier offered by their respective systems to select all the entities displayed in one view by defining a window around them. The "shade solid" command, on the other hand, identifies the solid to be shaded by digitizing it only once. This reflects the completeness and unambiguities of solid models as discussed in Chapter 6.

The shading modifiers of the "shade" command include all the input parameters and their values, required to define the desired shading model. These modifiers include a background color, a location of the point light source and a desired intensity of an ambient light (between 0 and 1). The background color is the color of the picture (image) background. The location of the point light source is the coordinates (x, y, z) of a point input relative to the MCS. Other modifiers to specify enhancements of shading models such as transparency and texture are also possible.

The shading resolution is an important input parameter because it controls the quality of the image to be generated. The higher the resolution of the image, the higher its quality is and the more CPU time and memory space it takes to generate it and store it. A low resolution is recommended when the user is experimenting in search for the best combination of colors and shading modifiers, especially the best location of the point light source to best illuminate the model. How can the resolution of an image be made different from the resolution of the graphics display? Let us define two types of resolution: hardware and software. A hardware resolution is the actual resolution of the display and is equal to the number of pixels in both the horizontal and vertical directions, as discussed in Chapter 2. A software resolution is a scale of hardware resolution and is used by shading software to control the quality of a shaded image. Consider, for example, a graphics display of