12 Solid Modeling

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

12.1 REPRESENTING SOLIDS

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

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

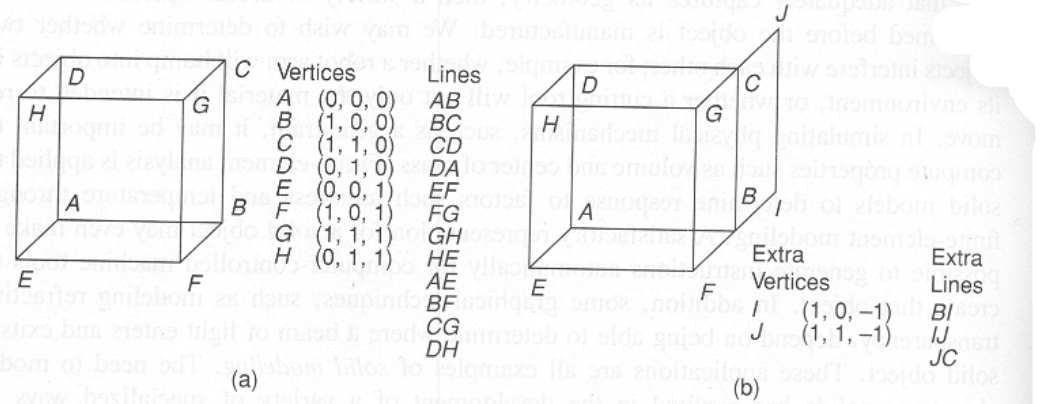

Fig. 12.1 (a) A wireframe cube composed of 12 straight lines. (b) A wireframe cube with an extra face.

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

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

12.2 REGULARIZED BOOLEAN SET OPERATIONS

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

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

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

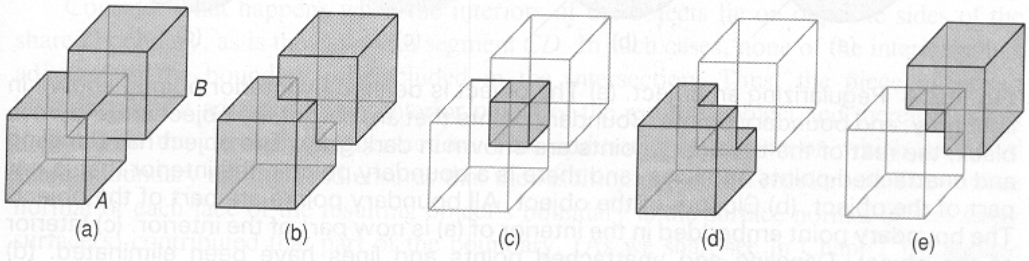

Fig. 12.2 Boolean operations. (a) Objects A and B, (b) $A \cup B$, (c) $A \cap B$, (d) $A - B$, and (e) $B - A$.

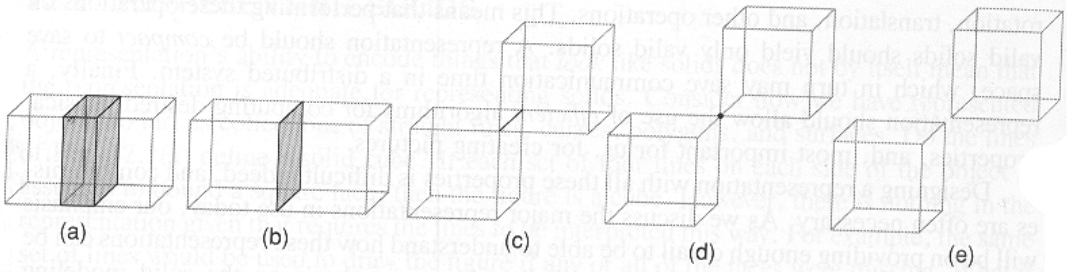

Fig. 12.3 The ordinary Boolean intersection of two cubes may produce (a) a solid, (b) a plane, (c) a line, (d) a point, or (e) the null set.

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

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

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

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

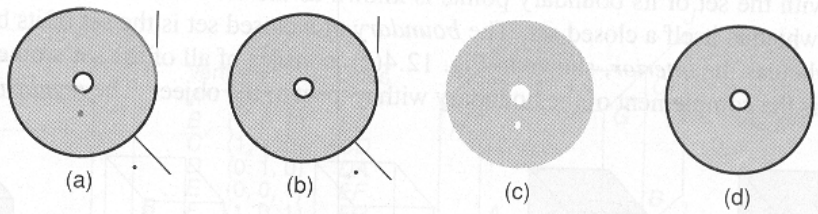

Fig. 12.4 Regularizing an object. (a) The object is defined by interior points, shown in light gray, and boundary points. Boundary points that are part of the object are shown in black; the rest of the boundary points are shown in dark gray. The object has dangling and unattached points and lines, and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

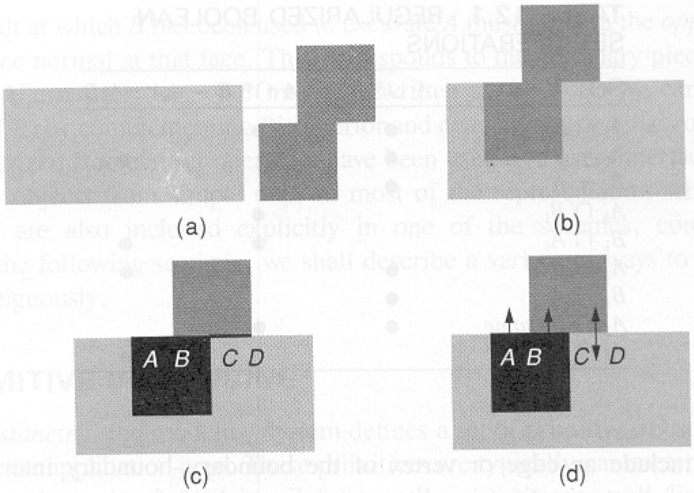

Fig. 12.5 Boolean intersection. (a) Two objects, shown in cross-section. (b) Positions of object prior to intersection. (c) Ordinary Boolean intersection results in a dangling face, shown as line *CD* in cross-section. (d) Regularized Boolean intersection includes a piece of shared boundary in the resulting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

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

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

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

TABLE 12.1 REGULARIZED BOOLEAN SET OPERATIONS

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

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

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

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

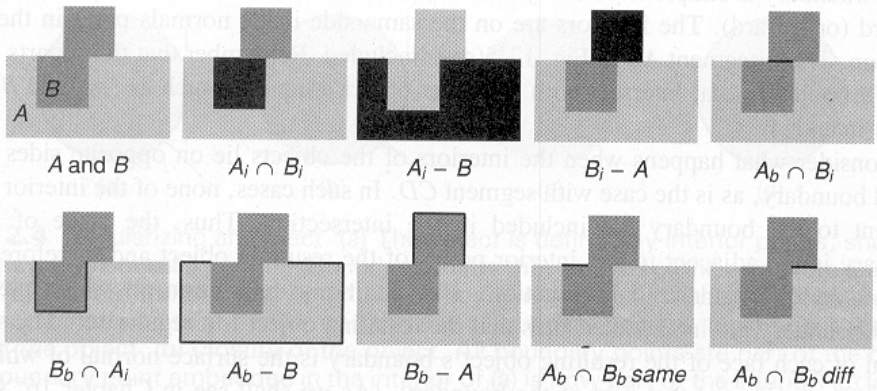

Fig. 12.6 Ordinary Boolean operations on subsets of two objects.

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

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

12.3 PRIMITIVE INSTANCING

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

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

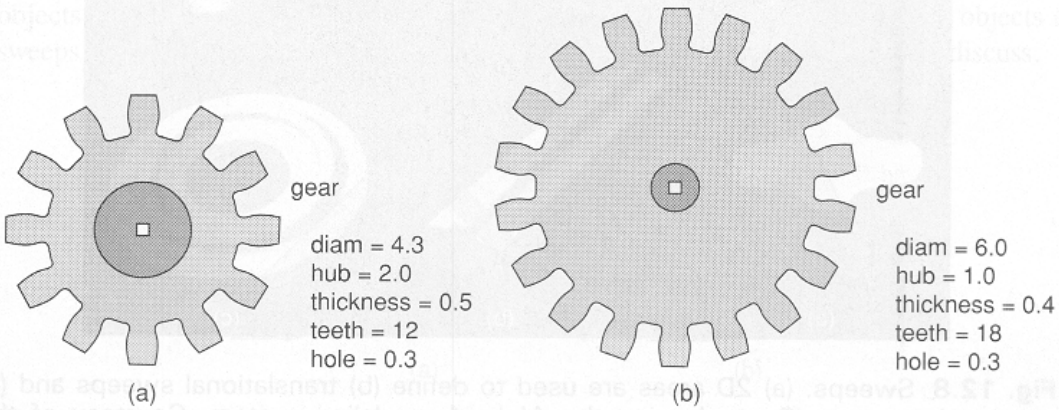

Fig. 12.7 Two gears defined by primitive instancing.

12.4 SWEEP REPRESENTATIONS

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

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

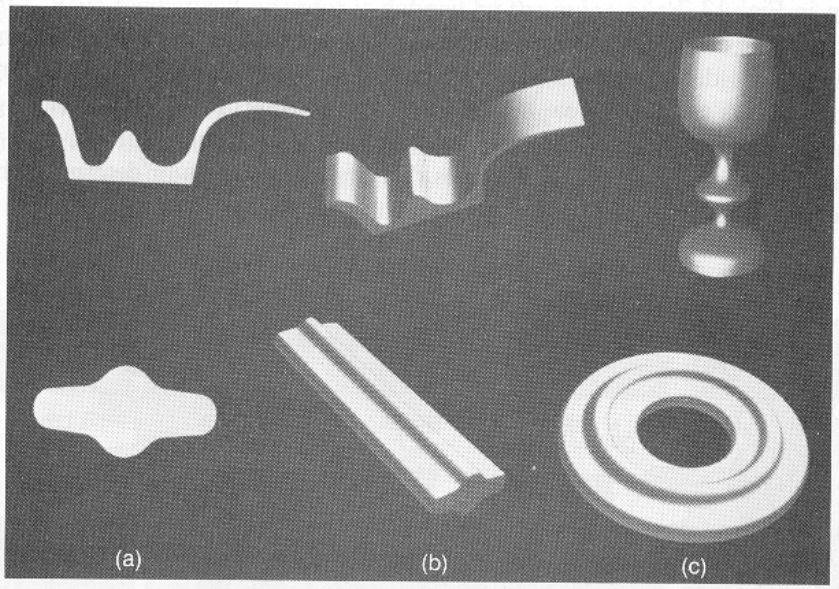

Fig. 12.8 Sweeps. (a) 2D areas are used to define (b) translational sweeps and (c) rotational sweeps. (Created using the Alpha_1 modeling system. Courtesy of the University of Utah.)

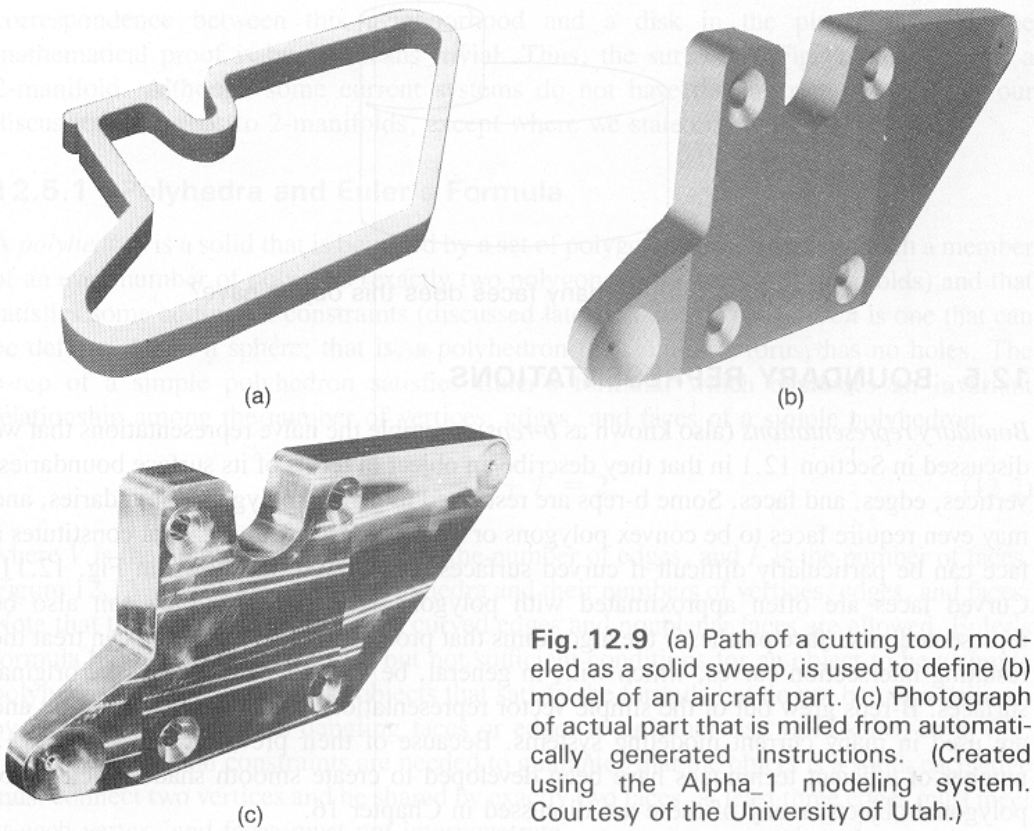

Fig. 12.9 (a) Path of a cutting tool, modeled as a solid sweep, is used to define (b) model of an aircraft part. (c) Photograph of actual part that is milled from automatically generated instructions. (Created using the Alpha_1 modeling system. Courtesy of the University of Utah.)

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

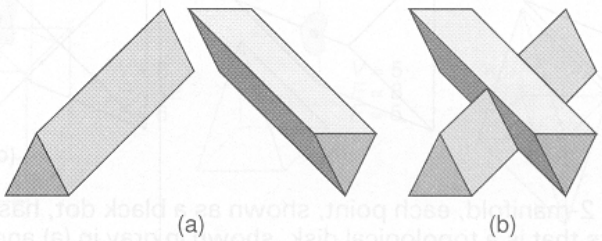

Fig. 12.10 (a) Two simple sweeps of 2D objects (triangles). (b) The union of the sweeps shown in (a) is not itself a simple sweep of a 2D object.

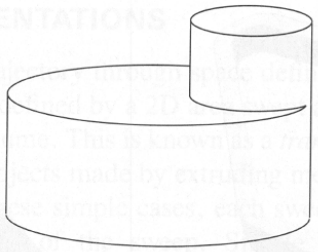

Fig. 12.11 How many faces does this object have?

12.5 BOUNDARY REPRESENTATIONS

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

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

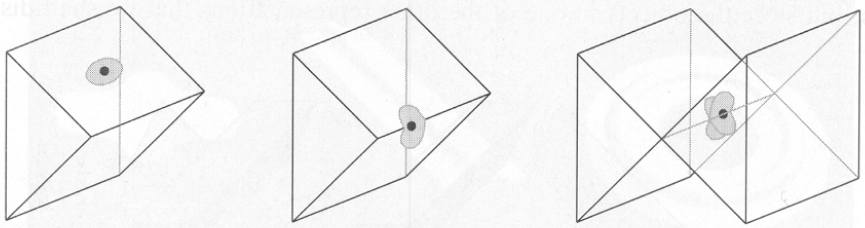

Fig. 12.12 On a 2-manifold, each point, shown as a black dot, has a neighborhood of surrounding points that is a topological disk, shown in gray in (a) and (b). (c) If an object is not a 2-manifold, then it has points that do not have a neighborhood that is a topological disk.

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

12.5.1 Polyhedra and Euler's Formula

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

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

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

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

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

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

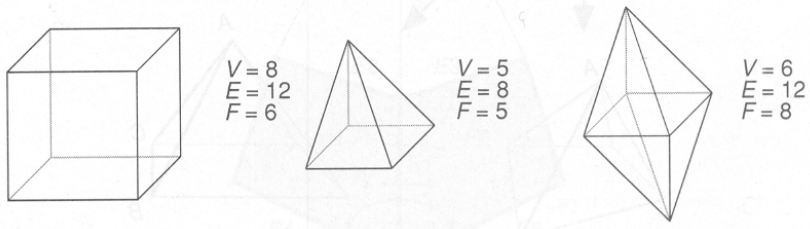

Fig. 12.13 Some simple polyhedra with their V, E, and F values. In each case $V - E + F = 2$.

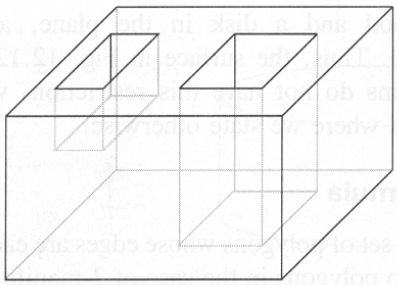

Fig. 12.14 A polyhedron classified according to Eq. (12.3), with two holes in its top face and one hole in its bottom face.

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

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

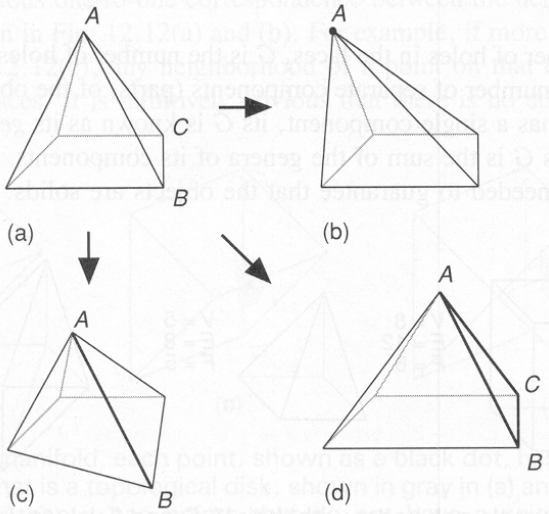

Fig. 12.15 (a) An object on which tweaking operations are performed to move (b) vertex *A*, (c) edge *AB*, and (d) face *ABC*.

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

12.5.2 The Winged-Edge Representation

Simple representations make certain computations quite expensive. For example, discovering the two faces shared by an edge (e.g., to help prove that a representation encodes a valid solid) requires searching the edge lists of all the faces. More complex b-reps have been designed to decrease the cost of these computations. One of the most popular is the *winged-edge* data structure developed by Baumgart [BAUM72; BAUM75]. As shown in Fig. 12.16, each edge in the winged-edge data structure is represented by pointers to its two vertices, to the two faces sharing the edge, and to four of the additional edges emanating from its vertices. Each vertex has a backward pointer to one of the edges emanating from it, whereas each face points to one of its edges. Note that we traverse the vertices of an edge in opposite directions when following the vertices of each of its two faces in clockwise order. Labeling the edge's vertices n and p, we refer to the face to its right when traversing the edge from n to p as its p face, and the face to its right when traversing the edge from p to n as its n face. For edge $E1$ in Fig. 12.16, if n is $V1$ and p is $V2$, then $F1$ is $E1$'s p face, and $F2$ is its n face. The four edges to which each edge points can be classified as follows. The two edges that share the edge's n vertex are the next (clockwise) edge of the n face, and the previous (counterclockwise) edge of the p face, $E3$ and $E2$, respectively. The two edges that share the edge's p vertex are the next (clockwise) edge of the p face, and the previous (counterclockwise) edge of the n face, $E4$ and $E5$, respectively. These four edges are the "wings" from which the winged-edge data structure gets its name.

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

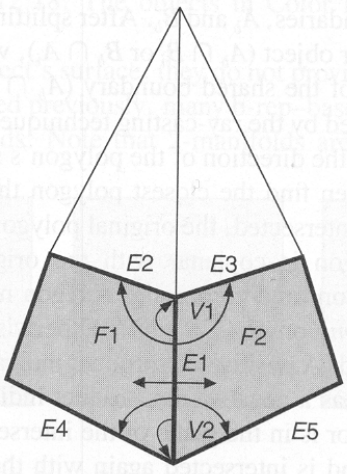

Fig. 12.16 Winged-edge data structure for $E1$. Each of $V1$, $V2$, $F1$, and $F2$ also have a backward pointer to one of their edges (not shown).

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

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

12.5.3 Boolean Set Operations

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

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

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

Vertex-adjacency information can be used to avoid the overhead of classifying each polygon in this way. If a polygon is adjacent to (i.e., shares vertices with) a classified

polygon and does not meet the surface of the other object, then it is assigned the same classification. All vertices on the common boundary between objects can be marked during the initial polygon-splitting phase. Whether or not a polygon meets the other object's surface can be determined by checking whether or not it has boundary vertices.

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

12.5.4 Nonpolyhedral b-Reps

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

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

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

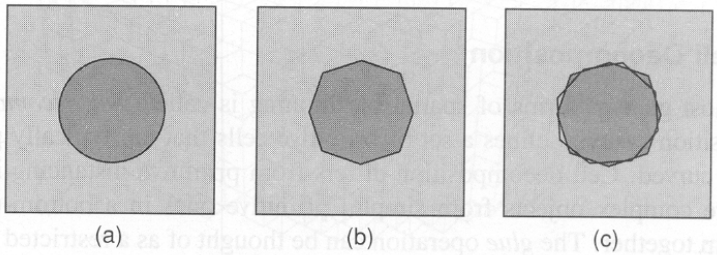

Fig. 12.17 (a) Cross-section of a cylinder in a round hole. (b) Polygonal approximation of hole and cylinder. (c) Interference occurs if approximated cylinder is turned relative to hole.

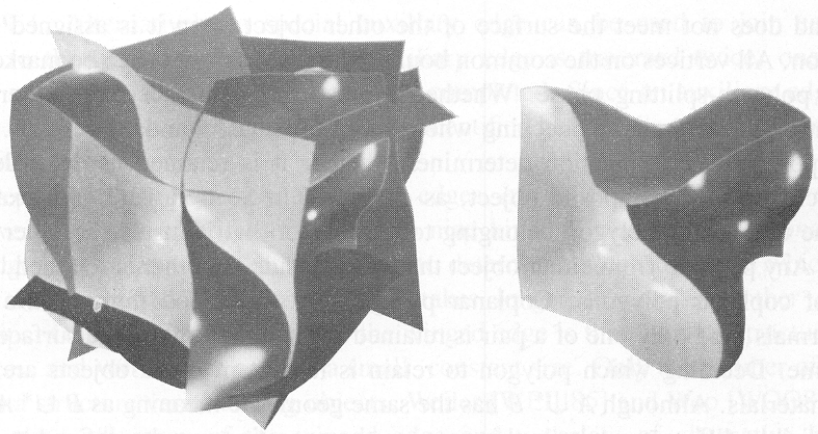

Fig. 12.18 Boolean set operations on partially bounded objects. (a) Six partially bounded sets. (b) Intersection of sets defines a wavy cube. (Courtesy of Spencer W. Thomas, University of Utah.)

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

12.6 SPATIAL-PARTITIONING REPRESENTATIONS

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

12.6.1 Cell Decomposition

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

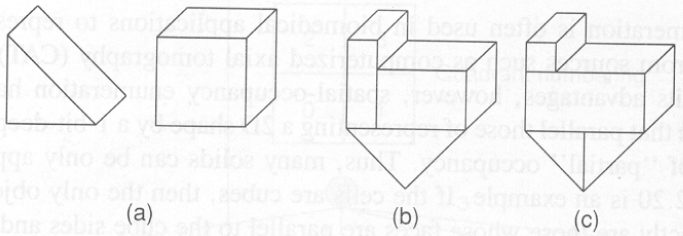

Fig. 12.19 The cells shown in (a) may be transformed to construct the same object shown in (b) and (c) in different ways. Even a single cell type is enough to cause ambiguity.

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

12.6.2 Spatial-Occupancy Enumeration

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

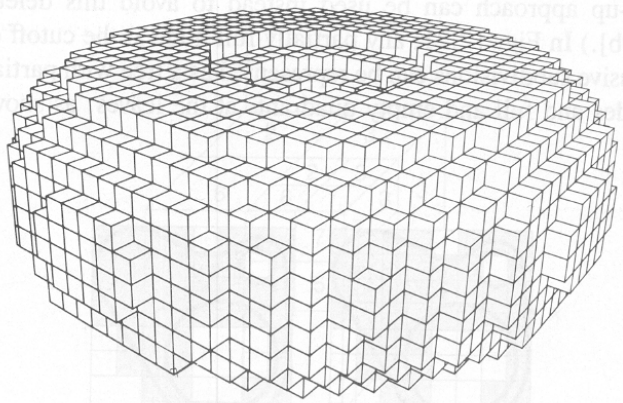

Fig. 12.20 Torus represented by spatial-occupancy enumeration. (By AHJ Christensen, SIGGRAPH '80 Conference Proceedings, *Computer Graphics* (14)3, July 1980. Courtesy of Association for Computing Machinery, Inc.)

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

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

12.6.3 Octrees

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

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

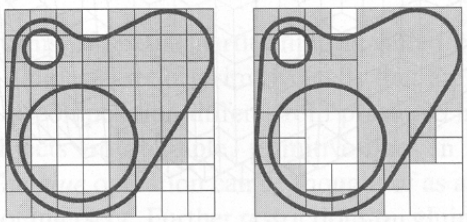

Fig. 12.21 An object represented using (a) spatial-occupancy enumeration (b) a quadtree.

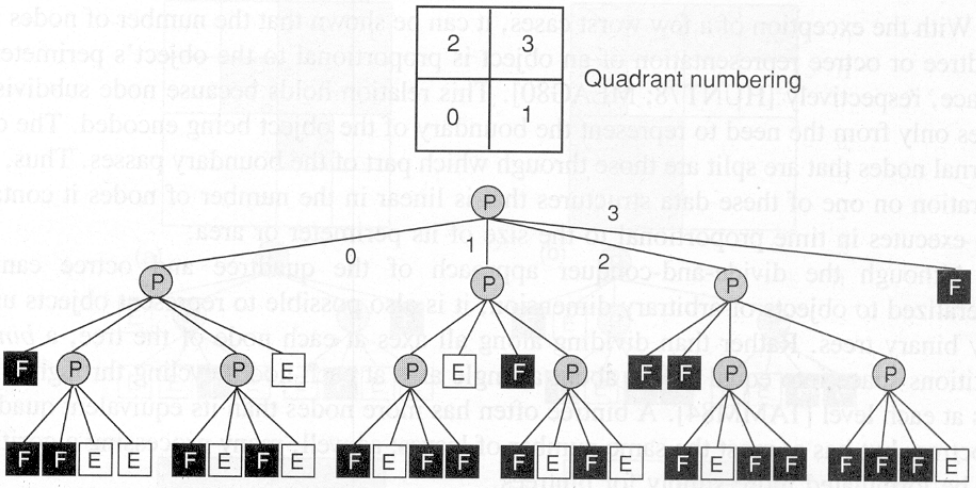

Fig. 12.22 Quadtree data structure for the object in Fig. 12.21. F = full, P = partially full, E = empty.

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

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

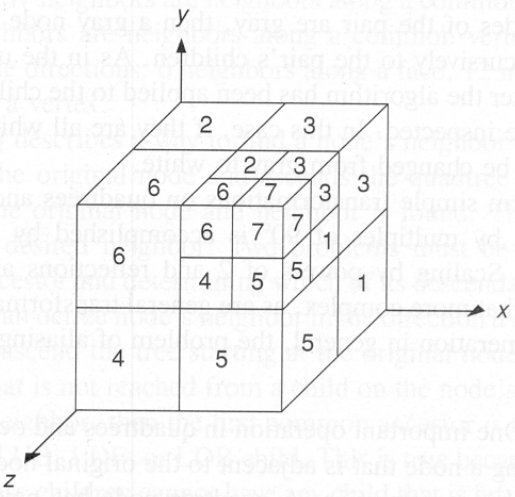

Fig. 12.23 Octree enumeration. Octant 0 is not visible.

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

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

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

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

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

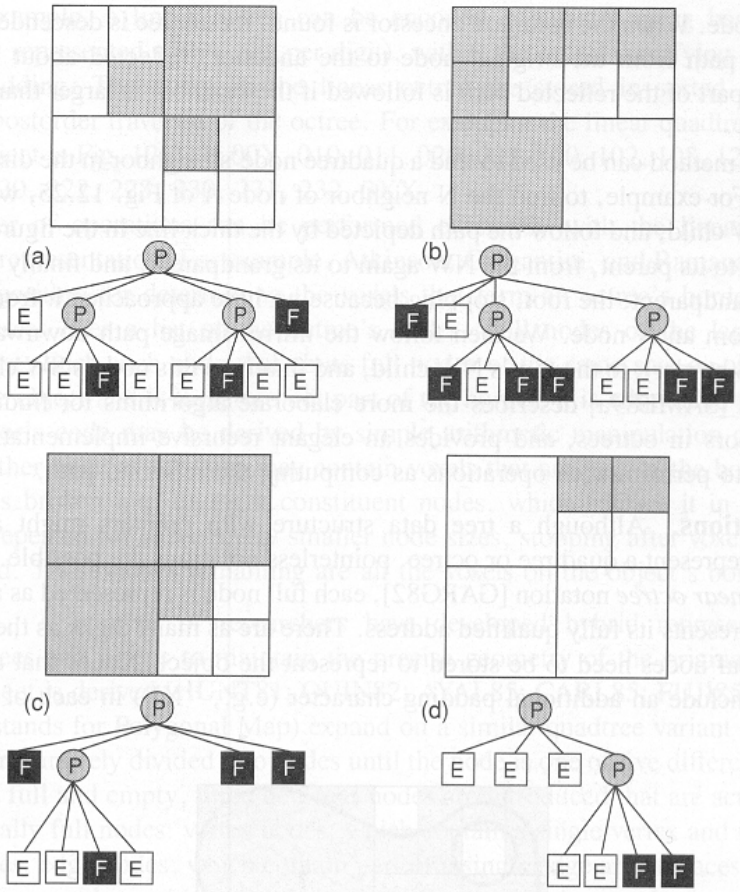

Fig. 12.24 Performing Boolean set operations on quadtrees. (a) Object S and its quadtree. (b) Object T and its quadtree. (c) $S \cup T$. (d) $S \cap T$.

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

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

the original node. When the common ancestor is found, its subtree is descended in a mirror image of the path from the original node to the ancestor, reflected about the common border. Only part of the reflected path is followed if the neighbor is larger than the original node.

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

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

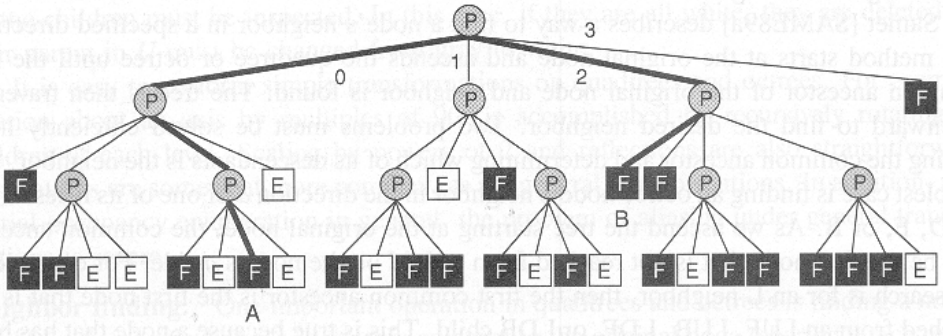

Fig. 12.25 Finding the neighbor of a quadtree node.

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

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

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

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

12.6.4 Binary Space-Partitioning Trees

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

entirely inside or entirely outside the polyhedron. These homogeneous regions are called ''in'' cells and ''out'' cells. To account for the limited numerical precision with which operations are performed, each node also has a ''thickness'' associated with its plane. Any point lying within this tolerance of the plane is considered to be ''on'' the plane.

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

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

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

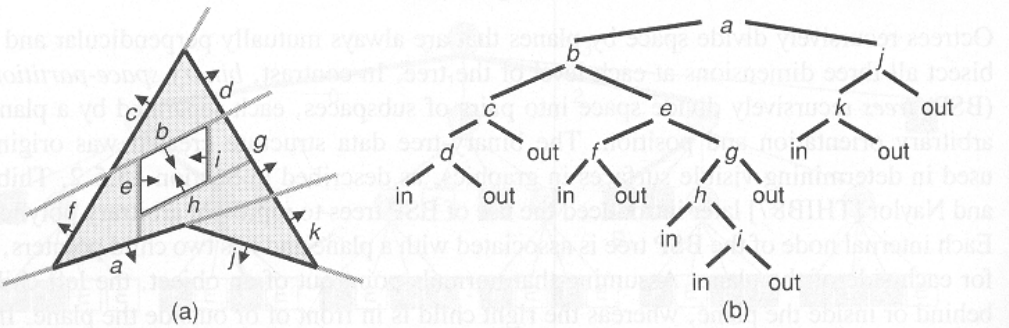

Fig. 12.26 A BSP tree representation in 2D. (a) A concave polygon bounded by black lines. Lines defining the half-spaces are dark gray, and ''in'' cells are light gray. (b) The BSP tree.

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

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

12.7 CONSTRUCTIVE SOLID GEOMETRY

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

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

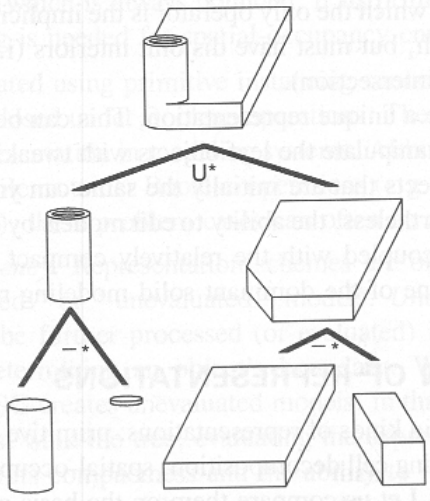

Fig. 12.27 An object defined by CSG and its tree.

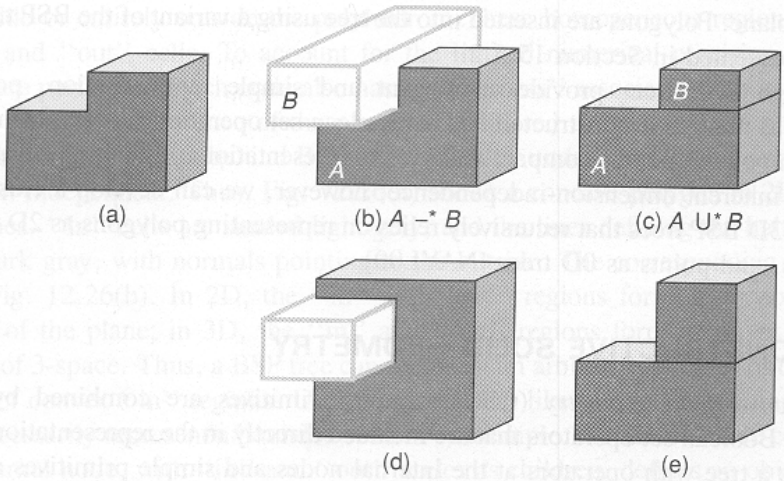

Fig. 12.28 The object shown in (a) may be defined by different CSG operations, as shown in (b) and (c). Tweaking the top face of (b) and (c) upward yields different objects, shown in (d) and (e).

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

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

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

12.8 COMPARISON OF REPRESENTATIONS

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

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

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

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

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

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

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

12.9 USER INTERFACES FOR SOLID MODELING

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

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

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

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

12.10 SUMMARY

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

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

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

EXERCISES

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

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

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

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

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

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

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

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