INTRO TO SOLID MODELLING

MARTTI MANTYLA

COMP. SCIENCE PRESS

Chapter 4

DECOMPOSITION MODELS

Decomposition models describe solids through a combination of some basic building blocks glued together. The kinds on basic objects used and the way the combination of basic objects is recorded leads to variations on this basic theme.

4.1 EXHAUSTIVE ENUMERATION

Earlier, we viewed solids as continuous point sets. While we cannot list all points belonging to a solid, we can easily list all, say, tiny cubes that are contained (completely or partially) in the solid. The cubes are assumed to be nonoverlapping and of uniform size and orientation, i.e., they form a *regular subdivision* of the space.

The resulting solid representation is called the *exhaustive enumeration*; see Figure 4.1 [28]. In the collection of cubes forming the exhaustive enumeration, each small cube can be completely described in terms of its corners. By regularity, it is sufficient to store the coordinates of just one corner for each cube of the collection. In the case that we can *a priori* limit our interest to some *space of interest* being a subset of E^3, the collection can be efficiently encoded as a three-dimensional array c_{ijk} of binary data. The array represents the "coloring" of each cube: if $c_{ijk} = 1$ ("*black*"), the cube ijk represents a solid region of the space, and otherwise it is empty ("*white*").

The binary array representation is, of course, the obvious way to represent a picture in *digital image processing*. Reference [112] surveys object

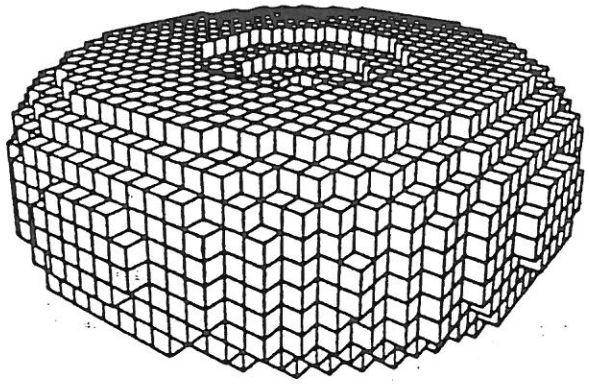


Figure 4.1 Exhaustive enumeration.

representations of this field; see also [137]. While image processing as such is beyond the scope of this book, the areas of interest of image processing and solid modeling do intersect, and much of what we shall say on the processing of exhaustive enumerations (or decomposition models in general) has its roots in image processing techniques.

4.1.1 Construction of Exhaustive Enumerations

An independent solid description mechanism for exhaustive enumerations would consist of a list of all cubes that are considered to form parts of the solid. In image processing applications this is indeed possible with image scanning devices. Digital tomography [123] can also supply three-dimensional digital information.

Unfortunately, in solid modeling we rarely have an image or an object to start with, and these direct solid description techniques cannot be used. It would also be very hard to describe realistic objects as directly enumerating its cubes. A more practical approach is to create exhaustive enumerations by a conversion from some other representation such as the constructive or boundary models to be discussed in the subsequent chapters. Other objects can then be created by Boolean set operations or other suitable algorithms for exhaustive enumerations.

The interesting point of exhaustive enumeration is that it is straightforward to produce algorithms that create new representations from existing representations. A prime example is the calculation of Boolean set operations on exhaustive enumerations. In the binary matrix representation, the algorithm becomes just the corresponding bitwise operation for all corresponding elements.

Simplistically, a binary matrix is always a valid solid representation; hence, all algorithms that produce them can be considered closed. However, often disconnected cells that do not have any neighbor cells are not desired, and algorithms such as the Boolean set operations must pay attention to disallowing them. Quite similarly, a single white cell within a block of black cells might not be considered correct. For various situations, various degrees of connectivity may be desired. Each 3-dimensional cell has six face neigbors, 12 edge neighbors, and eight vertex neighbors. The strictest connectivity criterion would require that all black cells have at least one black face neighbor.

Finding and filling connected components (with various degrees of connectivity) are standard image processing algorithms [92]. It might be possible to apply image processing techniques for solid modeling purposes also more generally. For instance, a *growing algorithm* (*dilatation*) could be used to calculate the "offset solid" of a given solid, i.e., the solid that contains all points at distance $d < r$ from any point of the original solid. This operation would be useful for the generation of data for numerically controlled (NC) machine tools and robotics.

4.1.2 Uses of Exhaustive Enumeration

Two- and three-dimensional variants of the exhaustive enumeration are the obvious representations for digital images, and are widely used in digital image processing. Through this connection, exhaustive enumeration has found its solid modeling uses in special applications, such as modeling of cell particles. In this area, a cube is allowed to have other "colors" beside mere "white" or "black," again raising the memory consumption.

Exhaustive enumeration is as much a representation of empty spaces as of spaces occupied by material. A coloring scheme can be used to distinguish various kinds of empty spaces; this is useful, for instance, for modeling buildings in heat transfer analysis. In this context, empty spaces in the interior and in the exterior of the building must be modeled differently.

Exhaustive enumeration is also used as an auxiliary scheme for speeding up operations on other representations. For instance, while being based on the half-space approach to be described in the next chapter, the solid modeler TIPS [91] uses a regular three-dimensional grid as a geometric directory to the elements of the half-space model in order to speed up various geometric algorithms. We shall return to this in Chapter 18.

4.1.3 Properties of Exhaustive Enumeration

In summary, let us discuss the properties of the exhaustive enumerations according to the framework of Section 3.6.1.

Expressive power: Exhaustive enumeration is obviously *approximative*: surfaces that are not coplanar with any of the coordinate planes of the subdivision will be only approximately represented. With this restriction, however, the scheme is general: all kinds of objects can be represented under it.

Validity: The validity of exhaustive enumerations depends on the geometric interaction of individual cubes. Specifically, each pair of blocks may intersect only at a common vertex, edge, or face. If the binary matrix representation can be used, all exhaustive enumerations are valid if connectivity is not required.

Unambiguity and uniqueness: All valid exhaustive enumerations are unambiguous. They are also *unique*: in a fixed space of interest and resolution, each solid has just one representation.

Description languages: In image processing applications, exhaustive enumerations are generally created through scanning of an image. In the context of solid modeling proper, their generation is ordinarily based on conversion from other models.

Conciseness: Exhaustive enumerations tend to be fairly large: a resolution of 256^3 (which is only barely adequate) takes 16 million bits of storage. Storage requirements rise sharply with increased resolution.

Closure of operations: Exhaustive enumerations support many closed algorithms. Boolean set operations are a prime example.

Computational ease and applicability: Algorithms for this scheme tend to be extremely simple; however, the mere size of the object representations means that they are slow in an ordinary computer. Note, however, that the computation required is typically extremely *regular*: the calculation needs to consider just one cube or at most the cube and a few neighbor cubes. This regularity means that these representations are prime candidates for VLSI implementation. This can radically affect the applicability of exhaustive enumeration in the future.

4.2 SPACE SUBDIVISION SCHEMES

Exhaustive enumerations have many virtues: they are simple, general, and allow the use of a wide variety of algorithms. These good points are, however, offset by the huge memory consumption and the mediocre accuracy possible. To overcome this, many representations replace the underlying

regular space subdivision of the pure enumeration by a more efficient, adaptive subdivision.

These *adaptive subdivision schemes* are based on the simple observation that in the three-dimensional grid of an exhaustive enumeration, the neighbor cubes of a white cube are very likely to be white as well. By encoding this information into one combined node of the data structure considerable savings over the complete grid are possible.

Adaptive subdivisions use the fundamental property that the number of nodes needed for the representation of a solid is proportional to its surface area [82]. Hence the number of elements is proportional to the square r^2 of the resolution r used, whereas the size of exhaustive schemes is proportional to r^3.

4.2.1 The Octree Representation

Prime examples of adaptive space subdivision schemes are the *Octree representation* for solid objects [60,82] and the analogous *Quadtree representation* [106] for two-dimensional objects.

The octree representation uses a recursive subdivision of the space of interest into eight *octants* that are arranged into an 8-ary tree (hence the name). Figure 4.2(a) depicts the octant subdivision.

Usually the octree is thought of as being located around the origin of its local xyz-coordinate system, with its first-level octants corresponding to the octants of that space, and in particular octant 3 being the positive octant $x, y, z > 0$ of the space. Hence it is necessary to represent the actual space of interest separately in terms of an appropriate transformation.

Each node of an octree consists of a *code* and eight pointers towards eight sons, numbered 0 through 7. If *code* = *black*, the part of the space represented by the node is all material and the pointers 0 through 7 are nil, i.e., the node is a leaf. If *code* = *white*, the part of the space is empty and the node is again a leaf. The third possibility *code* = *grey* corresponds to the case where the part of the space is partly material and partly empty. In this case, the eight pointers point to eight children that correspond to a regular subdivision of their parent node. For instance, the object of Figure 4.2(b) would be represented by the 2-level octree shown in (c).

Program 4.1 outlines a data structure for representing octrees. Here the space of interest is an orthogonal box of the xyz-space, represented by a special "root" node.

It is easy to show that 7/8 of the nodes of an octree are leaves. Because of this, also leaves are usually represented with a special node to avoid allocating storage for the eight nil pointers.

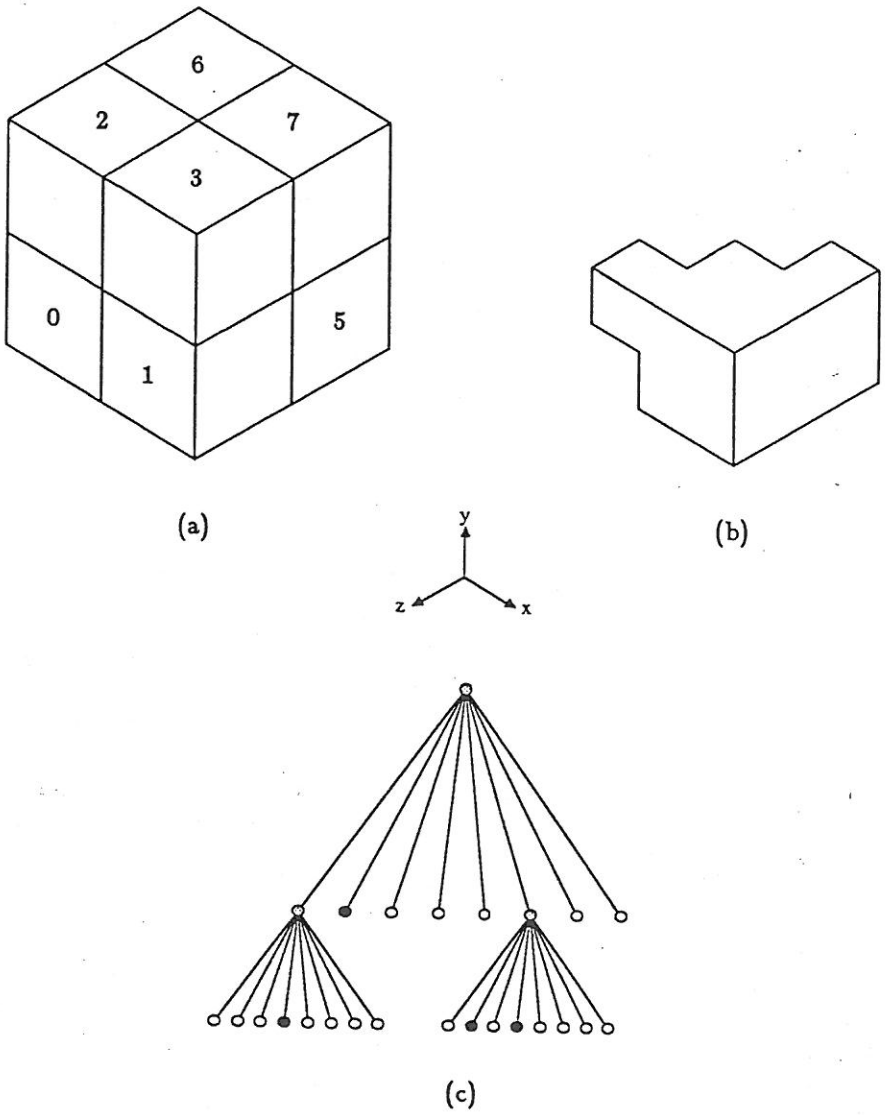


Figure 4.2 An octree model.


```
struct octreeroot
{
        float           xmin, ymin, zmin;  /* space of interest */
        float           xmax, ymax. zmax;
        struct octree   *root;             /* root of the tree */
};

struct octree
{
        char            code;              /* BLACK, WHITE, GREY */
        struct octree   *oct[8];           /* pointers to octants,
                                              present if GREY */
};
```

Program 4.1 Octree data structures.

Construction of Octree Representations

In the solid modeling context, octrees are ordinarily constructed from solid primitives. For each primitive type, a *classification procedure* between an instance of the primitive and an arbitrary node of the octree is needed. The classification must be capable of distinguishing between the following three cases:

1. The node considered is completely in the exterior of the primitive.
2. The node considered is completely in the interior of the primitive.
3. The node is partially in the interior of the primitive.

The classification procedure is used in a recursive fashion. Initially, the whole space of interest is represented by one node with *code* = *white*, and the algorithm is started at the single white node. Each node is classified against the primitive with the classification procedure. If the first or the second case above applies, the node is marked *white* or *black*, and the recursion terminates. Otherwise, the algorithm proceeds by marking the node *grey*, subdividing it into eight octants, and calling itself recursively for each octant. The subdivision is continued until a desired resolution has been reached, usually up to 6–12 levels.

The construction algorithm is outlined in Program 4.2. The actual solid primitives could be represented according to the primitive instancing scheme, i.e., in terms of a primitive type code and dimension and orientation parameters.

```
make_tree(p, t, depth)
primitive *p; /* p = the primitive to be modeled */
octree *t;    /* t = node of the octree, initially
                 the initial tree with one white node */
int depth;    /* initially max. depth of the recursion */
{
       int    i;
       switch(classify(p, t))
       {
           case WHITE:
               t->code = WHITE;
               break;
           case BLACK:
               t->code = BLACK;
               break;
           case GREY:
               if(depth == 0)
               {
                   t->code = BLACK;
               }
               else
               {
                   subdivide(t);
                   for(i=0; i<8; i++)
                        make_tree(p, t->oct[i], depth-1);
               }
               break;
       }
}

/* classify octree node against primitive */
classify(...)

/* divide octree node into eight octants */
subdivide(...)
```

Program 4.2 Construction of an octree.

The collection of possible primitives depends on whether a classification algorithm for the primitive can easily be implemented. Typical "easy" primitives include rectilinear blocks, half-spaces such as sphere, cylinder, and cone, and certain higher-order objects such as tori and objects bounded by so-called superquadrics [10].

Octrees and quadtrees can also be constructed from digital image information if such is available.

Algorithms for Octrees

A functionally complete octree modeler should include algorithms for the following categories of tasks:

1. *Tree generators* that create octrees from parameterized primitives or other types of geometric models.

2. *Set operations* that take two octrees (with identical spaces of interest) and calculate a new octree that gives the Boolean union, intersection, or set difference of the two arguments.

3. *Geometric operations* that take an octree and calculate a new octree that models the result of translating, rotating, or scaling the object modeled. Another type of geometric operation calculates a new octree that has been altered according to the *perspective transformation*. Some of these problems lead to unintuitive and complicated algorithms. For instance, the translation of an octree is a relatively hard operation.

4. *Analysis procedures* that calculate properties such as the volume or the surface area of an octree. A *connected components* procedure that labels each octree node with the identifier of the connected object it belongs to gives an example of a more involved analysis operation.

5. *Display generators* that create a graphical image of the object modeled by the octree.

The generation of octrees was already outlined in the above. In the following, we shall briefly discuss some of the other areas.

Graphical Output The essential property of octrees is that they record the shape information of an object in a spatially ordered manner. If this can be exploited in algorithm design, it is possible to create very simple algorithms that just scan their argument trees and perform relatively simple operations at each node.

Generation of graphical output for a frame buffer raster display is a particularly good example of this. Observe that by traversing the nodes of the octree in a suitable order, the parts of the image generated from far nodes can be painted before those of the near ones. Hence the close parts will overlay the far away parts in the frame buffer, and no sorting is required for the generation of an image with hidden surfaces removed.

The proper ordering depends on the location of the viewer with respect to the tree. For instance, if the viewer is located in the octant $x, y, z > 0$ of the space (such as in the view of Figure 4.2(a)), the ordering 4, 0, 5, 1, 6, 2, 7, 3 of the subtrees will produce the correct result. Reference [31] describes techniques for the generation of more advanced images based on this approach.

Analysis Many analysis operations can also be performed according to a similar node-by-node paradigm. For instance, integral properties such as volume, center of mass, and moments of inertia can all be calculated by simple tree traversal algorithms.

Set Operations for Octrees Also set operations for octrees lead to a tree traversal algorithm. The set operations algorithm takes two argument octrees, and produces a third octree that represents the desired Boolean set operation (union, intersection, set difference) of the arguments.

The argument trees are traversed in a synchronous fashion, and a case analysis as for the operation and the types of the corresponding nodes is performed. For instance, when calculating the set intersection, the following cases would occur in the processing of two corresponding nodes n_1 and n_2:

1. Nodes n_1 and n_2 are both leaves. In this case, the corresponding node of the result octree is black if both n_1 and n_2 are black; otherwise, it is white.

2. Either n_1 or n_2 is a leaf. In this case, if the leaf node is black, the subtree of the nonleaf is copied to the result octree. Otherwise, the node of the result tree is white.

3. Nodes n_1 and n_2 are both nonleaves. In this case, the algorithm considers recursively their children as above.

Observe that if the both octrees are already present (i.e., the time needed for their construction can be ignored), the complexity of this algorithm is at most proportional to the size of the smaller tree.

Limitations of tree traversal algorithms Unfortunately, not all algorithms for octrees can be brought in the form of a simple tree traversal. In graphics, visual effects such as smoothly shaded surfaces and transparency require the capability of accessing the neighbors of a node. Unfortunately, accessing a neighbor node can in the worst case require a traversal up to the root of the tree, and down to the neighbor. Similar lines apply also to various image processing algorithms such as "growing" and connected component labeling.

The complexity of accessing neighbor nodes can be somewhat cured by introducing further pointers in the octree data structure. As this raises the cost of the generation of octrees, careful analysis of the frequency of the various operations is needed for maintaining the proper balance.

Properties of Octrees

The properties of octree representations are similar to those of the exhaustive enumeration, with some noticeable exceptions.

Expressive power: As were exhaustive enumerations, octrees are approximative representations, and model exactly only rather peculiar objects. However, if a classification routine for a primitive can be written, arbitrarily accurate octrees for it can be generated (at the cost of high storage use).

Validity: If no special conenctivity requirement are posed, all octrees are valid representations of some solid.

Unambiguity and uniqueness: Up to the limits of resolution, all octrees unambiguously define a solid. On a fixed resolution, the representation is also unique: an object has just one compacted[1] octree representation with at most n levels.

Description languages: The same lines apply as for exhaustive enumerations. Octrees are usually formed by conversion from other representations, of which constructive representations have an especially important role. In image processing applications, quad- and octrees are also formed directly from rasterized image data.

Conciseness: In general, the number of nodes in the octree representation of a solid object is proportional to the surface area of the object [82]. Hence octree models are not quite as large as exhaustive representations but still take a fair amount of storage. An octree representation of an "average" engineering object easily takes more than 1 million bytes of memory.

[1] Algorithms such as set operations can create octrees with unnecessary nodes (e.g., an internal node whose children are all black). Such nodes can be removed with a relatively simple tree traversal algorithm.

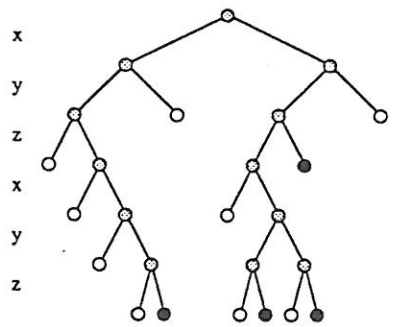


Figure 4.3 Binary subdivision tree.

Closure of Operations: Octree models support closed algorithms for problems such as translation, rotation, and Boolean set operations.

Computational ease and applicability: Many algorithms for octrees take the form of a tree traversal where a relatively simple operation is performed at each node of the tree. To these algorithms, the same notes apply as to exhaustive enumeration, including the potential role of VLSI.

Octree and quadtree representations have won considerable interest, and their literature is large and grows rapidly. For a comprehensive survey, the reader is referred to [106].

4.2.2 Binary Space Subdivision

As an alternative to the octree formulation, a *binary space subdivision* is also possible. As the name implies, this approach divides a grey node in two halves, instead of eight octants. The subdivision is performed successively in the direction of x, y, and z (see Figure 4.3).

In comparison with octrees, explicit binary subdivision trees are slightly smaller. As the virtues of the binary subdivision approach are most apparent in linearized versions to be described below, we shall not discuss them further here.

4.2.3 Linearized Space Subdivision

While offering significant storage savings over exhaustive enumeration, octrees (and binary subdivision trees) are still quite large. This has led

researchers to investigate possibilities for compressing the octree representation further, and many alternative representations that replace the explicit tree structure with a pointer-free, linear data structure have been proposed. Obviously, such representations are also convenient for storing octrees into external storage.

Linear Octrees

Recall the numbering 0 ... 7 of octants of the octree. The octant numbers can be used to construct a *path address* for each node of the octree except the root. Clearly, the path address of a octree node of level i becomes a sequence of i digits $0, ..., 7$. A special "end-of-number" digit can be included to mark the tail of a number that has less digits than the maximal resolution.

The *linear octree* of Gargantini [42] is based on these observations. In her approach, the linear octree is simply the sorted list of path addresses to all black nodes. Hence, the linear octree that corresponds to the octree of Figure 4.2(c) is simply the list

$$\{03, 1X, 51, 53\}$$

where X denotes the "end-of-number" digit.

Another compact linear encoding of an octree is the so-called *DF-representation* [63]. It is formed by traversing the octree in preorder, and storing certain information of the nodes encountered. The encoding uses the alphabet "B", "W", and "(" denoting a black leaf, a white leaf, and an internal (nonleaf) node. Hence, the DF-representation of the octree of Figure 4.2(b,c) is the following string:

((WWWBWWWWBWWW(WBWBWWWWWW

As the alphabet has only three characters, two bits per each node are sufficient for the encoding.

Remarkably, mere linear representations are sufficient for many important algorithms. Boolean set operations on linear octrees end up in merging two linear strings of characters, a relatively straightforward operation. Algorithms for finding the nearest neighbor of a given node, for performing certain geometric operations, and for calculating connected components of objects also have been reported. For instance, reference [108] describes several algorithms for the two-dimensional case.

Bintree: Linear Binary Subdivision

Binary subdivision schemes lend themselves to very similar compact representations. In comparison with linear octrees, they are slightly smaller

because the number of leaves is smaller (compare Figures 4.2 and 4.3). Also, two leaves at the lowermost level of the tree under same parent can be encoded with one bit only because there are only two possibilities (black-white, white-black). These and other enhancements lead to linear representations having approximately one bit per tree node [115].

In three dimensions, Samet and Tamminen call the resulting solid representation a *bintree* [107,65]. Again, many algorithms can work without ever constructing an explicit tree. For instance, the paper [107] describes an algorithm for the evaluation of Boolean expressions of solid primitives. Display algorithms for bintrees are described by Koistinen *et al.* [65].

4.2.4 Geometric Search by Subdivision

Having an explicit octree available, it is very easy to check whether some particular location in the space of interest contains material or not. This point of view leads us to consider octrees and other space subdivision schemes as an efficient access method to geometric information.

Ordinarily, octree nodes only store a few bits of information encoding the material class of the cell. In the case of a geometric index, nodes may store (references to) other geometric entities, such as points, lines, or surfaces.

For the purposes of solid modeling, the prime example of this approach is the *extended octree* [7,22,86] that stores geometric elements of a boundary representation into an octree. As we interpret the extended octree as a hybrid representation scheme, we shall discuss its details in Chapter 7.

4.3 CELL DECOMPOSITIONS

Another approach to resolve the problems of exhaustive enumeration while preserving its nice properties is to use also other kinds of basic elements than just cubes. These schemes are called *Cell Decompositions*.

4.3.1 Representation of Cell Decompositions

A cell decomposition has a certain variety of basic *cell types* and a single combination operator *glue*. Individual cells are usually created as parameterized instances of cell types. Cells may be any objects that are topologically equivalent to a sphere (i.e., do not contain holes); in particular, they may include curved surfaces. A solid is modeled by means of a collection of semidisjoint cells, i.e., cells may "touch" each other along their bounding surfaces, but not have common interior points; see Figure 4.4 [35].

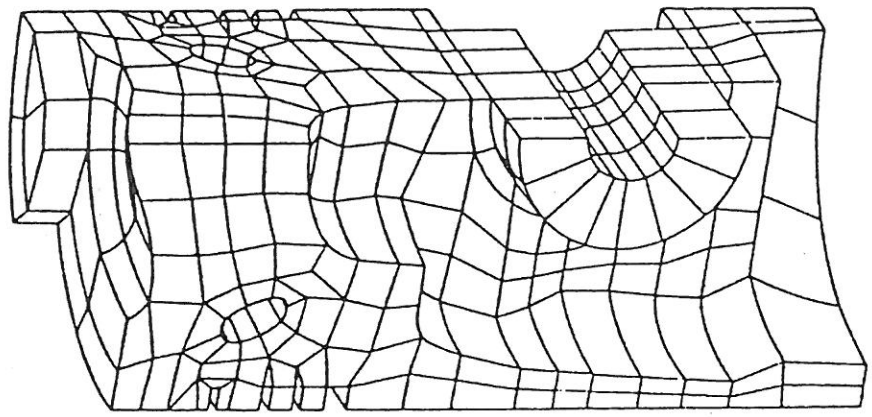


Figure 4.4 A cell decomposition.

A typical cell might be a curved polyhedron determined by 20 points. Eight points reside at its corners, and 12 points on the lines between the corners. Hence, each line has three points which define a quadratic curve, and each surface becomes a biquadratic surface patch determined by eight points (see Figure 4.5).

Typically, the collection of cells must satisfy also certain topological criteria in addition to being pairwise semidisjoint. Usually, any two cells are required either to be completely disjoint or meet in exactly one corner, along one line, or along one face. The case that two cells intersect otherwise is considered an incorrect cell decomposition. Such requirements make it quite difficult for a human being to construct valid cell decompositions directly.

In the important case of finite element models (FEM), the combination operation *glue* is usually represented implicitly through an encoding of "nodes" in the cell decomposition data structure. This is important for arranging the communication between a cell and its neighbors.

4.3.2 Properties of Cell Decompositions

Again, let us discuss the properties of cell decompositions according to the general scheme.

Expressive power: The modeling space is general and in the presence of cells with curved surfaces exact up to the degree of the cells, typically quadratic.

INTRO TO SOLID MODELLING

MARTII MANTYLA

COMP. SCI. PRESS

Chapter 5

CONSTRUCTIVE MODELS

Decomposition models discussed in the preceding chapter represent solids as a collection of basic elements, combined with a "gluing" operation. In contrast, the *constructive models* to be discussed in this chapter use much more powerful combination operations.

5.1 HALF-SPACE MODELS

All constructive models consider solids as point sets of E^3. Their basic idea is to start from some sufficiently simple point sets that can be represented directly, and model other point sets in terms of very general combinations of the simple sets. So-called *half-space* models apply this approach in a direct fashion.

5.1.1 Half-Spaces

Every point set A can be thought of as having a *characteristic function* $g_A(X) : X \rightarrow \{0,1\}$ which tells whether a point X is considered to be a member of A or not. In other words, the function g_A must satisfy

$$g_A(X) = 1 \Rightarrow X \in A$$

$$g_A(X) = 0 \Rightarrow X \notin A$$

For very general point sets characteristic functions do not offer much help, because their representation would be as hard as the representation

of the sets themselves. However, for an interesting class of point sets g_A can be represented in terms of a real-valued analytic function f of x, y, and z defined everywhere in E^3. The restriction to analytic functions excludes certain "pathological" objects [94] as explained in Chapter 3. All points $X = (x\ y\ z)$ such that $f(X) \geq 0$ are considered to belong to the point set, while $f(X) < 0$ defines its complement.

Because $f(X) = 0$ divides the whole space into two subsets, point sets defined by $f(X) \geq 0$ and $f(X) \leq 0$ are referred to as *half-spaces*. For instance, functions

$$ax + by + cz + d \geq 0$$

$$x^2 + y^2 - r^2 \leq 0$$

define useful point sets, namely the *planar half-space* that consists of all points on or in the positive side of the plane $ax + by + cz + d = 0$, and the *cylindrical half-space* that consists of all points or or inside an infinite cylinder whose axis = z-axis and radius = r.

Other half-spaces of interest include the remaining *natural quadratic surfaces* such as spheres and cones, and certain higher-order surfaces such as tori. Observe that this collection includes both unbounded half-spaces (such as the infinite cylinder above) and bounded half-spaces (such as the sphere $x^2 + y^2 + z^2 - r^2 \leq 0$).

Not all surfaces of interest are half-spaces. For instance, the various *surface patches* widely used in surface models are not half-spaces, because they do not divide the space into distinct subsets. This means that the modeling space of a half-space modeler cannot easily be extended to cover also objects bounded by surface patches.

5.1.2 Boolean Set Operations

Half-spaces form the basic modeling primitives of half-space models. As half-spaces are point sets, the natural modeling procedures for half-space models are the *Boolean set operations* union ($\cup$), intersection ($\cap$) and set difference ($\setminus$).

Hence, half-space models are constructed by combining instances of half-spaces with Boolean set operations. For instance, to describe a finite cylinder C of length h, we need one cylindrical half-space and two planar half-spaces, combined together with the set operation "$\cap$":

$$\begin{aligned} H_1 &: \quad x^2 + y^2 - r^2 \leq 0 \\ H_2 &: \quad z \geq 0 \\ H_3 &: \quad z - h \leq 0 \\ C &= \quad H_1 \cap H_2 \cap H_3 \end{aligned}$$

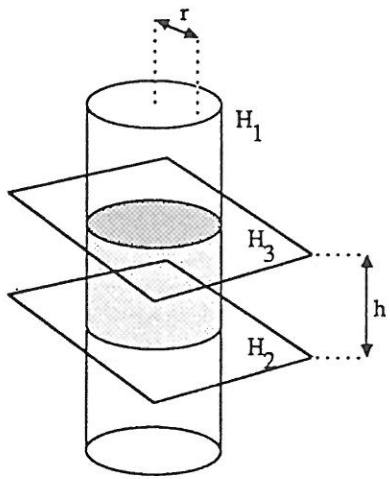


Figure 5.1 Half-space model of a finite cylinder.

This construction is illustrated in Figure 5.1.

Hence, the modeling space M of a half-space modeler is the class of *Boolean combinations* of the available half-spaces.

5.1.3 Representation of Half-Space Models

According to the above, the representation of a half-space model breaks into two parts:

1. *Representation of half-spaces*: The most common approach is to represent half-spaces as instances of half-space types, described with a half-spaces type code and a list of size parameters in some convenient coordinate system, and a transformation matrix that gives the actual location and orientation of the half-space.

   As an alternative to this approach, all quadratic half-spaces can be brought into the same form by writing their defining function $f(X) = f(x, y, z)$ as

$$f(x,y,z) = [x\ y\ z\ 1] \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

and storing the 16 coefficients A_{ij}, or the 10 coefficients derived by evaluating the matrix products. While this general form offers advantages, e.g., in ray casting (to be explained in Section 5.2.2 on page 92), the surface type coding is more commonly used because of the conveniences it offers, e.g., in the calculation of surface-surface intersections.

2. *Representation of Boolean combinations of half-spaces*: One possibility is offered by the fact that every Boolean expression may be brought into the "sum-of-products" form. TIPS [91], a solid modeler based on the pure half-space approach, follows this approach by representing a solid as the union of the intersections of the individual half-spaces. Hence, a solid S is expressed in TIPS as

$$S = \bigcup_i \bigcap_j H_{ij},$$

where H_{ij} denote the individual half-spaces of S. Other approaches to representing combinations of half-spaces are discussed later in this chapter.

5.1.4 Properties of Half-Space Models

Properties of pure half-space modelers can be described in the already familiar framework.

Expressive power: The modeling space of a half-space modeler is determined by the selection of half-spaces available, and of the generality of the operations available for combining them. Typically, modelers of this class include planar and quadratic half-spaces (such as spherical, cylindrical, and conical surfaces), sometimes even tori. Note, however, that patch surfaces are *not* half-spaces and cannot hence be included into the modeling space of half-space modelers.

Validity: Half-spaces are (usually) infinite point sets. Even a combination of them may be infinite, and hence not all combinations are valid solids (if finiteness is a required characteristic of a "solid" point set). TIPS circumvents this difficulty by enforcing the definition of a "box of interest" as a part of each solid description; only the part of space within the box is considered to form the solid.

Unambiguity and uniqueness: Each valid combination of half-spaces determines a solid. Hence half-space models are unambiguous. Half-space representations are not unique.

Description languages: Instantiation and combination of half-spaces leads easily to text-oriented, relatively simple solid descriptions. With some effort, it is also possible to create a drafting-type graphical user interface.

Conciseness: Half-space models are relatively concise: a few hundred half-spaces are usually sufficient to model realistic parts adequately (within the limitations of the modeling space).

Closure of operations: Any combination of two half-space models with a Boolean set operation defines a new valid model; hence these operations are closed.

Computational ease and applicability: The natural algorithms for half-space modeling are based on so-called *set membership classification*. In particular, the *ray casting approach* to be explained in the context of CSG models in Section 5.2.2 is a simple and general way of attacking the computational problems of half-space models. These families of algorithms are discussed in the next section in connection with the related CSG models.

5.2 CONSTRUCTIVE SOLID GEOMETRY

Pure half-space models offer a mathematically rigorous, easily understandable approach to solid modeling. For human users, however, it is easier to operate with bounded primitives instead of the unbounded half-spaces. Observe also that combinations of half-spaces may be unbounded point sets that do not quite match our notion of a valid solid. To avoid the generation of unbounded sets, the so-called *Constructive Solid Geometry* (CSG) approach to solid modeling uses only bounded point sets as its primitives.

5.2.1 Representation of CSG Models

CSG adopts the "building block" approach to solid modeling in its pure form. The user of a CSG modeler operates only on parameterized instances of *solid primitives* and Boolean set operations on them. Each primitive, in turn, is defined as a combination of half-spaces; the user has no direct access to individual half-spaces, however. For instance, the user may create planar and cylindrical half-spaces by means of a cylinder primitive, but he cannot directly manipulate them. Hence the mathematical modeling space of CSG models is exactly the same as that of pure half-space models, except that only finite point sets are included.

The most natural way to represent a CSG model is the so-called *CSG tree* that can be defined as follows:

<CSG tree> ::= *<primitive>* |
<CSG tree> <set operation> <CSG tree> |
<CSG tree> <rigid motion>

In the above, *<primitive>* is an instance of a solid primitive, represented through a primitive type identifier and a sequence of dimension

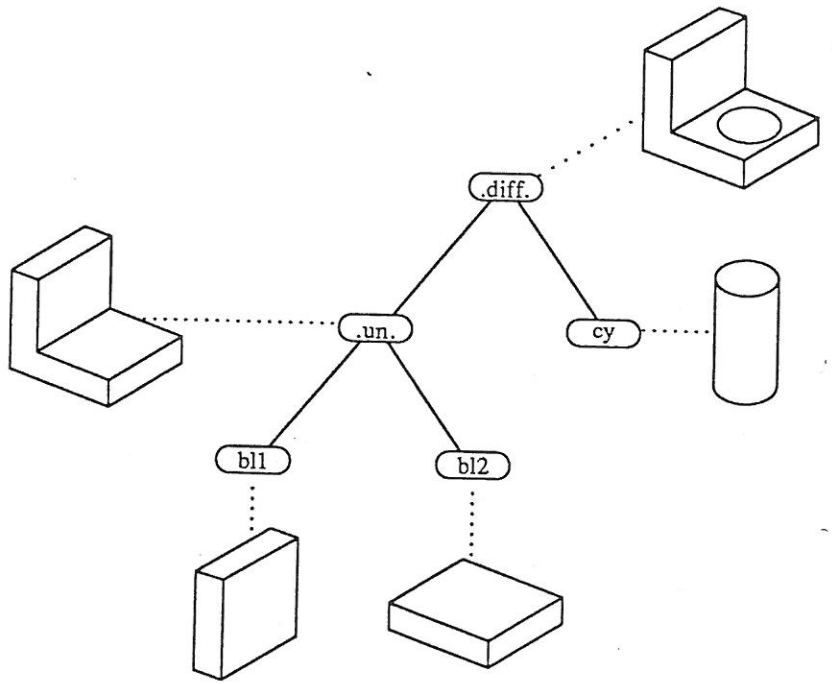


Figure 5.2 A CSG tree.

parameters. <*rigid motion*> is either a translation or a rotation, and <*set operation*> is one of $\cup$, $\cap$, and $\backslash$.

Hence, primitives are represented in the leaves of the CSG tree, while interior nodes are marked with either a Boolean set operation or with a rigid motion (see Figure 5.2). In this fashion, set operations and motions are interpreted to operate on CSG trees. This naturally leads to the creation CSG trees that model subassemblies of a design, which after appropriate transformations are used several times in the CSG tree representing the assembled design. In this case the binary tree actually becomes a *directed acyclic graph*.

Each primitive is chosen so as to define a bounded point set of E^3.

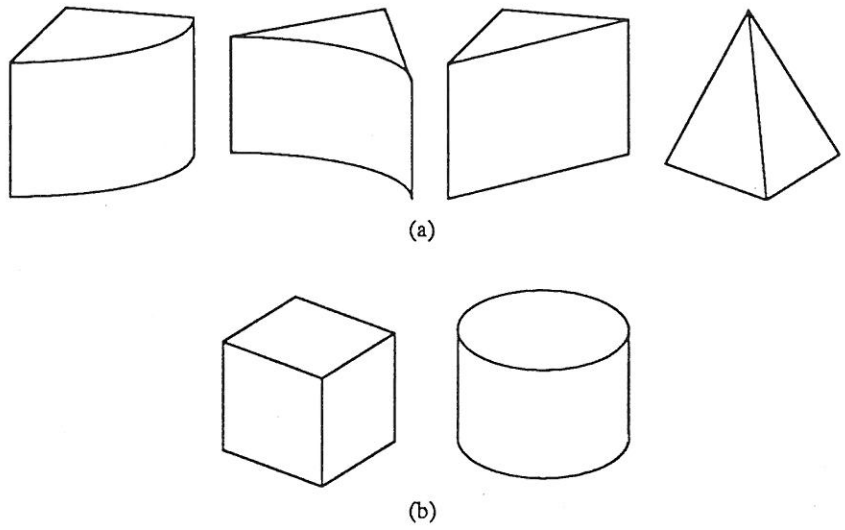


Figure 5.3 Two collections of CSG primitives.

As the set operations available cannot destroy boundedness, CSG models are guaranteed to define bounded sets. The ease of use of a CSG modeler depends much on the collection of primitives available. For instance, Figure 5.3 depicts two collections of CSG primitives. Note how collection (a) includes various wedges to aid the rounding of edges often needed in mechanical parts. Observe that all primitives of the figure can be expressed as a Boolean combination of simple half-spaces (in this case, planes and cylinders).

The actual domain of a CSG modeler depends on the variety of half-spaces available in its primitives, on the available rigid motions, and on the available set operations. Note that the two collections in Figure 5.3 have the same domain despite different primitives, because the underlying collection of half-spaces available is the same in both cases. (Actually, a CSG modeler with just one cylinder primitive has exactly the same domain.)

Regular Set Operations

Some combinations of CSG primitives (or half-spaces) do not quite satisfy our notion of "solidity." Consider, for instance, the case depicted in Figure 5.4. According to the ordinary definition of point set operations,

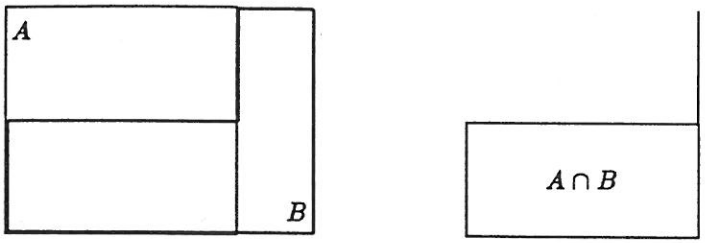


Figure 5.4 A nonregular set operation.

the intersection of the two objects consists of a rectangular object plus a "dangling" line segment (b). This effect is undesirable, for instance, if set operations are used to test the validity of an assembly by intersecting all component pairs and checking whether the result is an empty object. In that case, we would prefer that the intersection of merely touching components is empty.

In light of the point-set characterization of solidity, the problem of the resulting object is that it is not regular: the "dangling line" violates the Definition 3.3 of regularity on page 33.

The concept of regularity is introduced into CSG by considering its set operations to be the regularity-preserving variants of the usual point-set operations as defined below.

Definition 5.1 *The* regularized set operations *union*, intersection*, and set difference*, denoted by* $\cup^*$, $\cap^*$ *and* $\setminus^*$ *are defined as*

$$\begin{aligned} A \cup^* B &= c(i(A \cup B)) \\ A \cap^* B &= c(i(A \cap B)) \\ A \setminus^* B &= c(i(A \setminus B)) \end{aligned}$$

where $\cup$, $\cap$, *and* $\setminus$ *denote the usual set operations.*

If CSG primitives are chosen to be bounded regular sets, regularized set operations have the desirable property of being *algebraically closed* in the class of bounded regular sets. That is, every CSG tree is guaranteed to define a bounded regular set.

5.2.2 Algorithms for CSG Models

The CSG tree can be viewed as an implicit description of the geometry of the solid modeled that must be "evaluated" in some fashion in order to create graphical output or perform calculations.

The very structure of the CSG tree suggests the use of the very powerful and general "divide-and-conquer" approach to the design of algorithms for CSG trees. The basic idea of divide and conquer is to divide the problem in some fashion into two parts, recursively solve each part, and join the partial solutions to get the total solution. The recursion terminates when the problem has been subdivided into its "primitive" parts that allow a direct solution.

When applied to CSG trees, the natural way to subdivide a problem is to process the two subtrees of each interior node denoting a Boolean set operation of the tree separately. The recursion ends at leaves of the tree, where the problem is solved for one primitive. Solutions of subproblems are combined while taking the set operation at the node into respect.

Divide and conquer leads to efficient algorithms if the combination of subsolutions is simple and the problem can always be split into parts of approximately same size. Unfortunately, CSG trees are usually far from being balanced. Program 5.1 gives a generic divide and conquer algorithm for processing CSG trees. The following sections give specific instances of this schema.

Set Membership Classification

So-called *set membership classification* [121] algorithms are a particularly useful class of algorithms based on the divide and conquer approach. In general, a set membership classification algorithm works on two point sets, namely the *candidate set* C and the *reference set* R. The algorithm is expected to *classify* C against R by forming three sets $CinR$, $ConR$, and $CoutR$ representing the parts of C inside, on the boundary, and outside of R.

Specification of the "meaning" and the representations of the sets involved (C, R, $CinR$, $ConR$, $CoutR$) defines a particular set membership classification algorithm. For instance, the algorithm for classifying a finite line segment ("edge") against a CSG tree is cast into the general schema by choosing the sets as follows:

- C: An edge E given in terms of an ordered pair of coordinate triples;
- R: A CSG tree S;
- $CinR$, $ConR$, $CoutR$: Sets $EinS$, $EonS$, and $EoutS$, each being a set of nonempty subsegments of E such that $EinS \cup EonS \cup EoutS = E$, and their elements are inside, on the boundary, or outside S, respectively.

```
/* evaluate property P of a CSG tree */
P *Tree_P(S, args)
CSG_Tree *S;
{
        if(S->op == <primitive>)
                return(Primitive_P(S, args));
        else    return(Combine_P(Tree_P(S->Left, args),
                                 Tree_P(S->Right, args),
                                 S->Op));
}

/* evaluate P for a primitive */
P *Primitive_P(S, args)
{
        ...
}

/* combine two evaluations of P with set operation Op */
P *Combine_P(Left_P, Right_P, Op)
{
        ...
}
```

Program 5.1 Divide and conquer for CSG trees.


```
/* set membership classification of an edge vs. a CSG tree */
M *Tree_M(S, E)
CSG_Tree *S;
Edge *E;
{
        if(S->Op == <primitive>)
                return(Prim_M(S, E));
        else    return(Combine_M(Tree_M(S->Left, E),
                                 Tree_M(S->Right, E),
                                 S->Op));
}

/* classify E against a primitive */
M *Prim_M(S, E)
{
        ...
}

/* combine two classifications of E */
M *Combine_M(Left_M, Right_M, Op)
{
        ...
}
```

Program 5.2 Edge-solid classification.

The resulting algorithm of Program 5.2 nicely fits in the general approach of Program 5.1. The "property" evaluated is now the triple $M = \langle EinS, EonS, EoutS \rangle$. All we need to supply is an algorithm for classifying an edge against a primitive (`Prim_M` in Program 5.2), and an algorithm for combining two classifications (`Combine_M`).

Classification Against Primitive To do its task properly, the procedure `Prim_M` must calculate the intersections (if any) between E and the primitive, and subdivide E accordingly. For instance, in the case of Figure 5.5 the desired result of the primitive classification is the triple

$$\begin{aligned} M = <&\{(p_2, p_3)\}, && (EinS)\\ &\{\}, && (EonS)\\ &\{(p_1, p_2), (p_3, p_4)\} >. && (EoutS) \end{aligned}$$

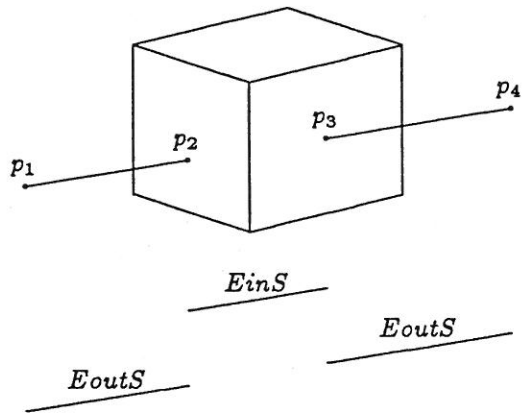


Figure 5.5 Edge vs. block primitive.

As all CSG primitives actually are combinations of half-spaces, the basic task needed is the classification of E against a half-space. This is a relatively straightforward operation.

For instance, suppose that we need to classify an edge E against a cylinder half-space. By transforming E appropriately, we can arrange things so that the cylinder is of the form

$$x^2 + y^2 - r^2 = 0,$$

and that E is represented by the parametric equation

$$E(t) = p_1 + t(p_2 - p_1),\ t = [0, 1]$$

where $p_i = (x_i\ y_i\ z_i)$ are the end points of E, and t is the line parameter. This gives us three simultaneous equations

$$\begin{aligned} x^2 + y^2 &= r^2 \\ x &= x_1 + t(x_2 - x_1) \\ y &= y_1 + t(y_2 - y_1). \end{aligned}$$

By substituting the second and third equation into the first, we get a second-order equation in t. If the equation has no real solutions, E does not intersect the cylinder at all; if a double root occurs, E is tangent to the cylinder; otherwise, the parameter values of two intersection points are given. What remains is to check whether the intersections occur within the

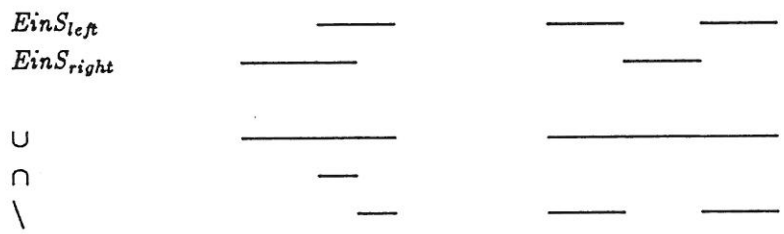


Figure 5.6 Combination rules.

edge, i.e., whether the parameter values are within 0 and 1. In the most complicated case of tori, a similar procedure yields a fourth-order polynomial in t, i.e., still something that can be solved with standard techniques.

Combination of Classifications Each of the results of edge-primitive classification (*EinS*, *EonS*, *EoutS*) consists of a sequence of edge segments. By arranging these sequences so that the segments appear in sorted order along E, the combination of classifications can be reduced to the merging of sets of line segments. The primitive task of this operation is the combination of just two edge segments by the Boolean set operation *Op*. For *EinS* and *EoutS* this leads to the combination rules illustrated in Figure 5.6.

Unfortunately, the rules for combining segments of the "left" *EonS* with the "right" *EonS* are not quite so simple. As illustrated in Figure 5.7, the combination rules for this "on-on"-case must take also the orientations of the respective surfaces into account.

Wire Frame Generation As an example on the use of the edge-solid classification, let us consider the *wire frame problem*:

> *Given a CSG tree S, determine a set* WireFrame *of line segments that form the "wire-frame" figure of S.*

Based on the edge-solid classification, the wire frame generation algorithm can be combined from the following steps:

1. *Generate*: Generate the set of all "tentative" edges by calculating the set of pairwise intersection curves of all half-spaces appearing in S.
2. *Classify*: Classify each tentative edge against S, and append the component *EonS* to the result.

```
WireFrameGen(S)
CSGTree S;
{
    /* Generation step: */
    Tentative_list = empty;
    for each half-space G of S
    {
        for each half-space H of S
        {
            Tentative_list = union(Tentative_list, Intersect(G, H));
        }
    }

    /* Classification step: */
    WireFrame = empty;
    for each edge E of Tentative_list
    {
        <EinS, EonS, EoutS> = Tree_M(S, E);
        WireFrame = union(WireFrame, EonS);
    }
}
```

Program 5.3 Wire frame generation algorithm.

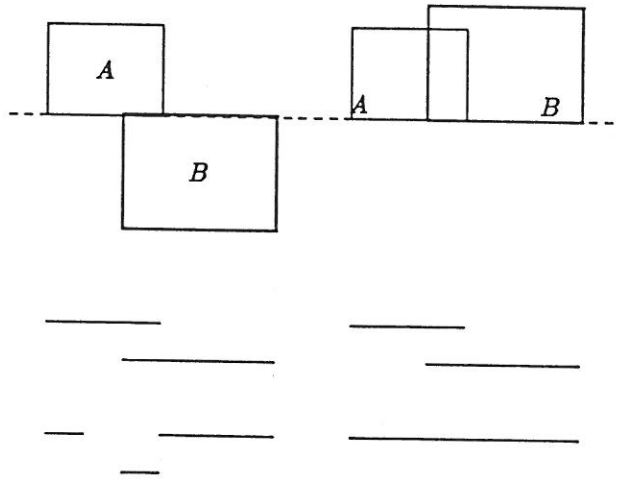


Figure 5.7 Combination rules for "on-on"-cases.

Program 5.3 outlines the resulting algorithm. The use of the complete three-way edge-solid classification may of course appear an overkill here, as only the component *EonS* really is needed. For more practical implementations of the algorithm, the reader is referred to Tilove [119].

Boundary Evaluation The generation of images with hidden lines removed requires a much more complicated algorithm that calculates the "faces" of the solid modeled via the CSG tree. This process is termed the *boundary evaluation*. Ordinarily, boundary evaluation is based on set membership classification between "primitive faces" (faces derived from CSG primitives) and CSG trees [96]. Of course, the capability of calculating intersections of quadratic surfaces is required [109].

In CSG modelers PADL-1 [126,97], PADL-2 [21], and GMSOLID [17] boundary evaluation is used to construct a complete boundary model based in the CSG tree. In particular, PADL-2 pursues so-called "incremental boundary evaluation" [119] that updates the boundary model so as to reflect changes in the CSG tree. These and other CSG modelers that can construct a boundary model from a CSG model can, of course, utilize algorithms for boundary models whenever they seem more appropriate than the self-contained methods for CSG models. Boundary models and their algorithms are discussed in Chapter 6.

INTRO TO SOLID MODELLING

MARTTI MANTYLA

COMP. SCI. PRESS

Chapter 6

BOUNDARY MODELS

Decomposition models and constructive models both view solids as point sets, and seek representations for the point set either by discretizing it or by constructing it from simpler point sets. In contrast to these models, *boundary models* represent a solid indirectly through a representation of its bounding surface.

6.1 BASIC CONCEPTS

Historically, boundary models emerged from the polyhedral models used in computer graphics for representing objects and scenes for hidden line and surface removal. They can be viewed as "enhanced" graphical models that attempt to overcome the problems of graphical models by including a complete description of the bounding surfaces of the object.

Boundary models are based on the surface-oriented view to solid modeling represented in Section 3.5. That is, they represent a solid object by dividing its surface into a collection of *faces* in some convenient fashion. Usually, the division is performed so that the shape of each face has a compact mathematical representation, e.g., that the face lies on a single planar, quadratic, toroidal, or parametric surface. In order to guarantee that the subdivision corresponds with a legal plane model as discussed in Section 3.5, it is usually required to satisfy certain "topological" criteria to be discussed in the sequel.

The portion of the underlying surface that forms the face is "chalked out" in terms of a closed curve that lie on the surface. A face may well have several bounding curves, provided that they define a connected object. That is, faces are like "continents" that may have "lakes"; "isles" on lakes do not belong to faces, however.

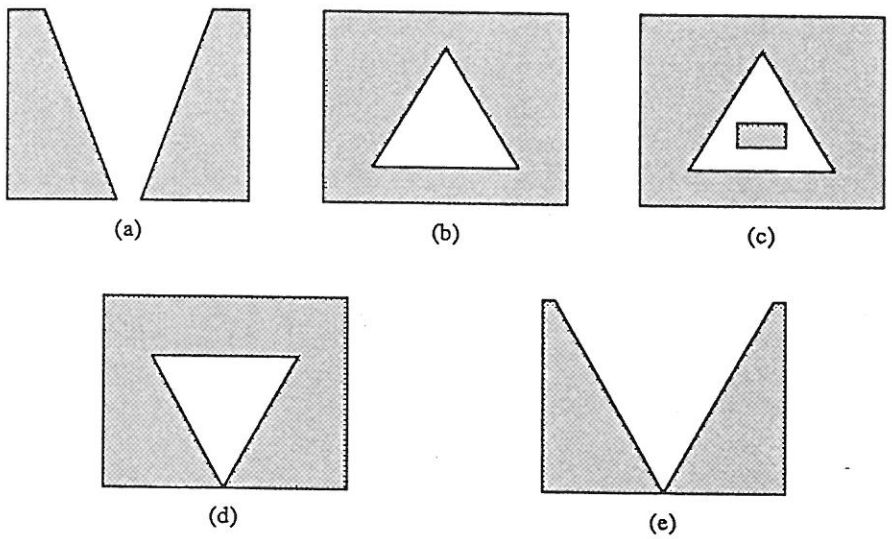


Figure 6.1 Definition of a face.

The definition of a face is depicted in Figure 6.1. In the figure, cases (a) and (c) are disconnected 2-dimensional sets and would be represented as two faces. Cases (d) and (e) are somewhat exceptional faces whose boundaries "touch" themselves. Usually case (d) would be considered a good face, whereas (e) would be represented as two faces. Observe that the interior of (d) is connected, and of (e) not. Case (c) is a face with two boundaries.

In turn, the bounding curves of faces are represented through a division into *edges*. Analogously to the above, edges are chosen so as to have a convenient representation, say, a parametric equation. The portion of the curve that forms the edge is chalked out in terms of two *vertices*.

Figure 6.2 illustrates the basic components of a boundary model. In the figure, the surface of the object is divided into an enclosing set of faces (a), each of which is represented in terms of its bounding polygon (b), in turn represented in terms of edges and vertices (c).

6.2 BOUNDARY DATA STRUCTURES

The three object types *face*, *edge*, and *vertex*, and the geometric information attached to them form the basic constituents of boundary models. In addition to geometric information such as face and curve equations and vertex coordinates, a boundary model must also represent how the faces, edges, and vertices are related to each other. It is customary to bundle all information of the geometry of the entities under the term *geometry* of a boundary model, and similarly information of their interconnections under the term *topology*.

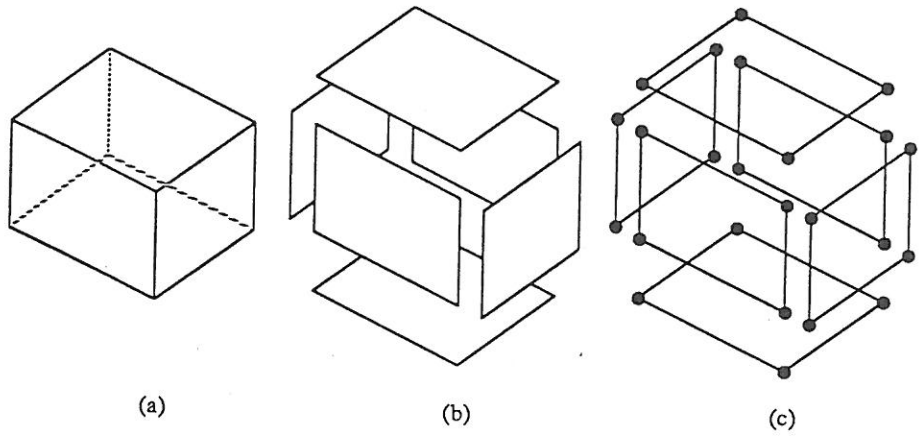


Figure 6.2 Basic constituents of boundary models.

All boundary models represent faces in terms of explicit nodes of a boundary data structure. After that, many alternatives for representing the geometry and the topology of a boundary model are possible, some of which are described below. For simplicity and clarity, we shall illustrate them in terms of alternative representations for the rectilinear block shown in Figure 6.3. Reference [8] discusses further alternatives and provides information on the data structures used in a number of modelers.

6.2.1 Polygon-Based Boundary Models

A boundary model that has only planar faces is called a *polyhedral model*. Because all edges of a polyhedron are straight line segments, it is possible to shrink the boundary data structure considerably in this important special case.

In the simplest variation faces are represented as *polygons*, each polygon consisting of a sequence of coordinate triples. A solid consists of a collection of faces grouped together in terms of, say, a table of face identifiers or a linked list of face nodes. Sometimes even the grouping information is elim-

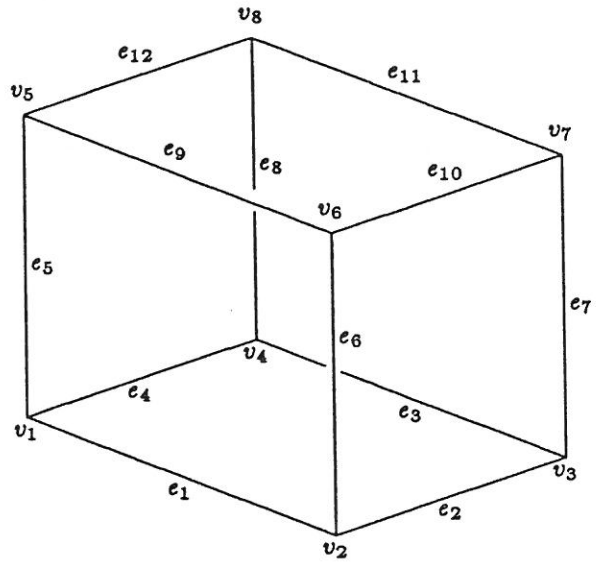


Figure 6.3 A sample object.

inated and face relationships are completely implicit. This representation is usually used in *graphical metafiles* of graphics systems.

6.2.2 Vertex-Based Boundary Models

In a polygon-based boundary model vertex coordinates appear as often as the vertex appears in a face. This redundancy can be eliminated by introducing vertices as independent entities of the boundary data structure. In this case, vertex identifiers (or something equivalent, such as pointers to vertex nodes) are associated with faces. This approach leads us to various *vertex-based boundary models.*

Note that the sample representation of Figure 6.4 lists vertices of each face in a *consistent order*, namely clockwise as seen from the outside of the cube. This consistent orientation is useful in many algorithms. For instance, in hidden line or surface removal, it allows the elimination of *back faces* on the basis of face normal vectors pointing consistently away from the material. In the case of Figure 6.4, faces f_1, f_4, and f_5 would immediately be discarded by the hidden line or surface removal algorithm.

The representation of Figure 6.4 does not include any surface information at all; as all faces are planar, their geometries are completely defined through the coordinates of their vertices. On the other hand, if face equations would be needed often for numerical calculations (say, for shading), they could be associated with faces.

vertex	*coordinates*
v_1	$x_1\ y_1\ z_1$
v_2	$x_2\ y_2\ z_2$
v_3	$x_3\ y_3\ z_3$
v_4	$x_4\ y_4\ z_4$
v_5	$x_5\ y_5\ z_5$
v_6	$x_6\ y_6\ z_6$
v_7	$x_7\ y_7\ z_7$
v_8	$x_8\ y_8\ z_8$

face	*vertices*
f_1	$v_1\ v_2\ v_3\ v_4$
f_2	$v_6\ v_2\ v_1\ v_5$
f_3	$v_7\ v_3\ v_2\ v_6$
f_4	$v_8\ v_4\ v_3\ v_7$
f_5	$v_5\ v_1\ v_4\ v_8$
f_6	$v_8\ v_7\ v_6\ v_5$

Figure 6.4 A vertex-based boundary model.

In general, many choices as to what information is stored explicitly and what information is left implicit (i.e., to be computed as needed) must be made when implementing a boundary model. In Figure 6.4, the explicit inclusion of vertex coordinates implicitly gives us also face equations. (The opposite approach of storing face equations explicitly and leaving vertex coordinates implicit leads to a rather peculiar half-space model appropriate for convex objects only.)

If some redundant geometric information is stored (like the face equations in the example above), subtle numerical problems may arise. Suppose that due to small inaccuracies in floating point calculations, vertices of a face do not lie exactly on the plane defined by its face equation. Which information is considered "correct"?

6.2.3 Edge-Based Boundary Models

If curved surfaces are present in a boundary model, it becomes useful to include also edge nodes explicitly in the boundary data structure to be able to store information of intersection curves of faces. (Edge nodes are also useful for recording topological relationships as discussed in the next section.)

An *edge-based boundary model* represents a face boundary in terms of a closing sequence of edges, or *loop* for short. Vertices of the face are represented only through edges. This approach leads us to the model of Figure 6.5.

edge	*vertices*
e_1	$v_1\ v_2$
e_2	$v_2\ v_3$
e_3	$v_3\ v_4$
e_4	$v_4\ v_1$
e_5	$v_1\ v_5$
e_6	$v_2\ v_6$
e_7	$v_3\ v_7$
e_8	$v_4\ v_8$
e_9	$v_5\ v_6$
e_{10}	$v_6\ v_7$
e_{11}	$v_7\ v_8$
e_{12}	$v_8\ v_5$

vertex	*coordinates*
v_1	$x_1\ y_1\ z_1$
v_2	$x_2\ y_2\ z_2$
v_3	$x_3\ y_3\ z_3$
v_4	$x_4\ y_4\ z_4$
v_5	$x_5\ y_5\ z_5$
v_6	$x_6\ y_6\ z_6$
v_7	$x_7\ y_7\ z_7$
v_8	$x_8\ y_8\ z_8$

face	*edges*
f_1	$e_1\ e_2\ e_3\ e_4$
f_2	$e_9\ e_6\ e_1\ e_5$
f_3	$e_{10}\ e_7\ e_2\ e_6$
f_4	$e_{11}\ e_8\ e_3\ e_7$
f_5	$e_{12}\ e_5\ e_4\ e_8$
f_6	$e_{12}\ e_{11}\ e_{10}\ e_9$

Figure 6.5 An edge-based model.

The data structure indicates an *orientation* for each edge; say, edge e_1 is considered to be (positively) oriented from vertex v_1 to vertex v_2. Again, faces are consistently oriented, i.e., their edges are listed clockwise as viewed from the outside of the block. Note that each edge occurs in exactly two faces, once in its positive orientation, and once in the opposite (negative) orientation.

Winged-Edge Data Structure The inclusion of explicit nodes for each of the basic object types (face, edge, and vertex) opens the door for elaborating an edge-based boundary model further. For instance, to aid algorithms such as hidden surface removal and shading, explicit face-face neighborhood information can be added to the data structure of Figure 6.5 by associating edges with the identifiers of the two faces they separate.

The so-called *winged-edge data structure*, first introduced by Baumgart [13,14] gives a primary example of such elaborated edge-based boundary models. It goes one step further by representing also the loop information in edge nodes.

Because each edge e appears in exactly two faces, exactly two other edges e' and e'' appear after e in these faces. Moreover, because of the consistent orientation of faces, e occurs exactly once in its positive orientation, and exactly once in the opposite orientation.

The winged-edge data structure takes advantage of these structural properties by associating the identifiers of the two "next" edges with an

edge	*vstart*	*vend*	*ncw*	*nccw*
e_1	v_1	v_2	e_2	e_5
e_2	v_2	v_3	e_3	e_6
e_3	v_3	v_4	e_4	e_7
e_4	v_4	v_1	e_1	e_8
e_5	v_1	v_5	e_9	e_4
e_6	v_2	v_6	e_{10}	e_1
e_7	v_3	v_7	e_{11}	e_2
e_8	v_4	v_8	e_{12}	e_3
e_9	v_5	v_6	e_6	e_{12}
e_{10}	v_6	v_7	e_7	e_9
e_{11}	v_7	v_8	e_8	e_{10}
e_{12}	v_8	v_5	e_5	e_{11}

vertex	*coordinates*
v_1	$x_1\ y_1\ z_1$
v_2	$x_2\ y_2\ z_2$
v_3	$x_3\ y_3\ z_3$
v_4	$x_4\ y_4\ z_4$
v_5	$x_5\ y_5\ z_5$
v_6	$x_6\ y_6\ z_6$
v_7	$x_7\ y_7\ z_7$
v_8	$x_8\ y_8\ z_8$

face	*first edge*	*sign*
f_1	e_1	+
f_2	e_9	+
f_3	e_6	+
f_4	e_7	+
f_5	e_{12}	+
f_6	e_9	−

Figure 6.6 The winged-edge data structure.

edge node. By convention, these data are denoted by *ncw* and *nccw* for "next clockwise" and "next counterclockwise"; in particular, *ncw* identifies the next edge in the face where the edge occurs in its positive orientation, and *nccw* the next edge in the other face.

By virtue of this edge representation, faces only need to include the identifier of an arbitrary edge and a bit that indicates its orientation. Figure 6.6 gives an example of the winged-edge data structure. Signs + and − denote the orientations of the edge.

Starting from the edge directly associated with a loop, all other edges may be retrieved by following the *ncw* and *nccw* pointers. For instance, in the case of Figure 6.6, the boundary of face f_5 is extracted by starting from e_{12}. The next edge is given by $ncw(e_{12}) = e_5$. Because e_5 was traversed in its negative orientation (which can be seen by examining vertex identifiers), the next edge is now given by $nccw(e_5) = e_4$. Similarly we get

$nccw(e_4) = e_8$ as the next edge. It is traversed in its positive orientation, so the next edge is $ncw(e_8) = e_{12}$ again, and we know that all edges have been extracted.

In the most general variation, edge nodes of the winged-edge data structure also include the identifiers *fcw* and *fccw* of its neighbor faces, and analogously to *ncw* and *nccw*, the identifiers *pcw*, *pccw* of the previous edges in these faces. The orientation indicators of Figure 6.6 now become redundant. The resulting "full" winged-edge data structure is shown in Figure 6.7. The schematic diagram (a) indicates once more the meaning of the various data items.

Because the full winged-edge data structure includes the identifier of an adjacent edge into each vertex node, all edges meeting at a vertex can be extracted by an algorithm similar to the loop extraction algorithm (see Figure 6.7(a)).

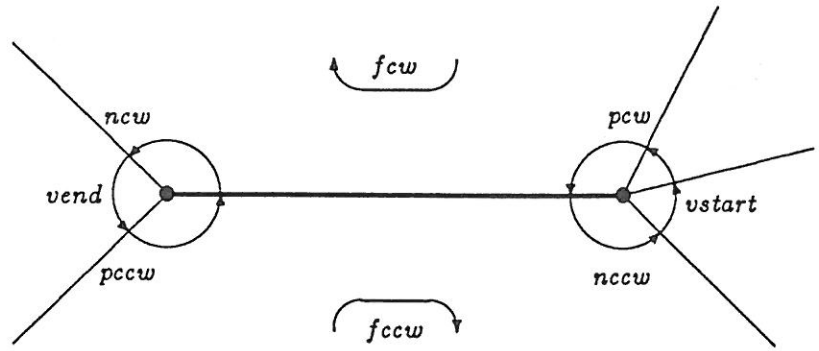
In fact, the data structure treats faces and vertices in a completely symmetric fashion. If the nodes *vstart* and *vend*, and the nodes *fcw* and *fccw* are exchanged, we end up with the *dual* of the original polyhedron.

6.2.4 Faces With Several Boundaries

The data structures described so far assume that each face is *simply-connected*, i.e., that it has just one boundary curve. However, sometimes it seems natural to include faces that have several boundary curves; see, for instance, the object depicted in Figure 6.7.

Two alternatives present themselves. One approach is to model faces with nonsimple boundaries by connecting the boundary segments with auxiliary edges. (In [134], this representation was termed the "bridge edge representation".) In the case of Figure 6.8, this could take place by adding the dashed edge. Unfortunately, these special edges should be marked somehow to avoid their inclusion in line drawings. However, in the winged-edge data structure this is not necessary because auxiliary edges are considered to occur twice in the same face, and can be detected by comparing the *fcw* and *fccw* data items.

The other approach is to add a separate loop node that models a simple boundary into the boundary model, and associate each face with a list of loops. References to faces would be replaced by references to the appropriate loops; for instance, face identifiers *fcw* and *fccw* in edge nodes of the full winged-edge data structure would be replaced with the analogous loop identifiers *lcw* and *lccw*. All data structures discussed can be readily generalized in this fashion, and we will not elaborate on this topic further.



(a)

edge	*vstart*	*vend*	*fcw*	*fccw*	*ncw*	*pcw*	*nccw*	*pccw*
e_1	v_1	v_2	f_1	f_2	e_2	e_4	e_5	e_6
e_2	v_2	v_3	f_1	f_3	e_3	e_1	e_6	e_7
e_3	v_3	v_4	f_1	f_4	e_4	e_2	e_7	e_8
e_4	v_4	v_1	f_1	f_5	e_1	e_3	e_8	e_5
e_5	v_1	v_5	f_2	f_5	e_9	e_1	e_4	e_{12}
e_6	v_2	v_6	f_3	f_2	e_{10}	e_2	e_1	e_9
e_7	v_3	v_7	f_4	f_3	e_{11}	e_3	e_2	e_{10}
e_8	v_4	v_8	f_5	f_4	e_{12}	e_4	e_3	e_{11}
e_9	v_5	v_6	f_2	f_6	e_6	e_5	e_{12}	e_{10}
e_{10}	v_6	v_7	f_3	f_6	e_7	e_6	e_9	e_{11}
e_{11}	v_7	v_8	f_4	f_6	e_8	e_7	e_{10}	e_{12}
e_{12}	v_8	v_5	f_5	f_6	e_5	e_8	e_{11}	e_9

vertex	*first edge*	*coordinates*
v_1	e_1	$x_1\ y_1\ z_1$
v_2	e_2	$x_2\ y_2\ z_2$
v_3	e_3	$x_3\ y_3\ z_3$
v_4	e_4	$x_4\ y_4\ z_4$
v_5	e_9	$x_5\ y_5\ z_5$
v_6	e_{10}	$x_6\ y_6\ z_6$
v_7	e_{11}	$x_7\ y_7\ z_7$
v_8	e_{12}	$x_8\ y_8\ z_8$

face	*first edge*
f_1	e_1
f_2	e_9
f_3	e_6
f_4	e_7
f_5	e_{12}
f_6	e_9

(b)

Figure 6.7 The full winged-edge data structure.

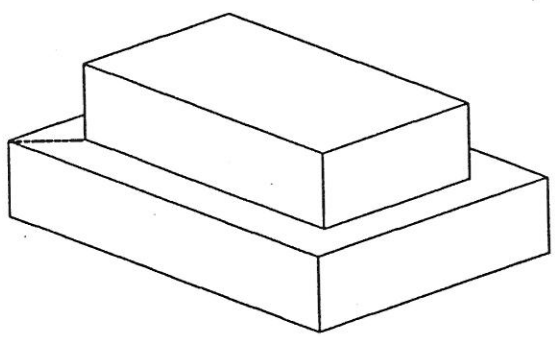


Figure 6.8 Object with non-simple faces.

6.3 VALIDITY OF BOUNDARY MODELS

A boundary model is *valid* if it defines the boundary of a "reasonable" solid object. According to the theory of Section 3.5, we are usually only interested on objects bounded by closed, orientable surfaces. In that case, the validity criteria of a boundary model includes the following conditions:

1. The set of faces of the boundary model "closes," i.e., forms the complete "skin" of the solid with no missing parts.

2. Faces of the model do not intersect each other except at common vertices or edges.

3. The boundaries of faces are simple polygons that do not intersect themselves.

The first and second conditions exclude self-intersecting objects such as the object of Figure 6.9(a). Observe that the pyramid-shaped bottom surface of the object intersects the top surface. The third condition disallows objects such as the "open box" (b). (To represent "open" objects with a solid boundary model, the best approach is to model "openings" as specially marked faces, instead of leaving them out.)

The first condition relates to the *topological integrity* of a boundary model. The theory of plane models of Section 3.5 tells us that all topological integrity criteria can be enforced just by structural means. In particular, according to Definition 3.8, the first condition can be enforced by demanding that each edge occurs in exactly two faces; hence, no edge can

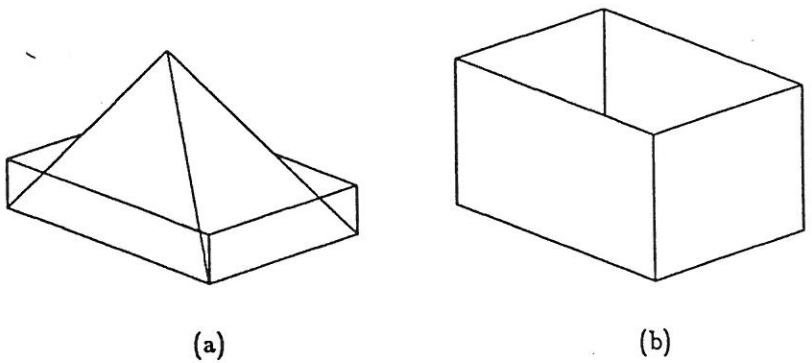


Figure 6.9 Invalid boundary models.

be the boundary of a missing part of the surface. Observe that the winged-edge data structure "automatically" satisfies this criterion, because edges occurring in just one face cannot be represented.

The winged-edge data structure also enforces that the Möbius' rule of Definition 3.9 is satisfied; in fact, the faces will always be consistently oriented. Hence the surface represented by it is guaranteed to be orientable. However, nonorientable objects are disallowed also by the second condition above, because they would always intersect themselves in the three-dimensional space.

Unfortunately, the *geometric integrity* of a boundary model defined by the second and third conditions cannot be enforced just by structural means: by assigning inappropriate geometric information to a completely reasonable topological entities, invalid models can be created. To guarantee the geometric integrity, one must either resort to a computationally expensive test that involves a comparison of each pair of faces in the solid, or limit the user's freedom by giving him only validity-enforcing solid description mechanisms.

6.4 DESCRIPTION OF BOUNDARY MODELS

Examination of Figures 6.6 and 6.7 soon reveals one of the major problems of boundary models, namely the complexity of their construction. It is beyond the capabilities of a normal human being to build correct boundary models directly, e.g., by typing information such as that of Figure 6.7.

The designer of a boundary modeler is therefore faced with the need of providing a sufficient collection of more convenient and efficient solid de-