Chapter 1

INTRODUCTION

1.1 AIMS OF MODELING

In general, a *model* is an artificially constructed object that makes the observation of another object easier. To make the obvervation possible, *physical models* of three-dimensional physical things such as buildings, ships, and cars usually share the relative dimensions and general appearance of their physical counterpart, but not the size. *Molecule models* used in chemistry share the relative arrangement of the various atoms of the molecule with respect to each other, but very few of its other properties. *Mathematical models*, widely used in various fields of science and engineering, represent some of the behavioral aspects of the phenomenon modeled in terms of numerical data and equations.

Engineering drawings, widely used in design and engineering, can also be viewed as models. Through a collection of two-dimensional images, aided by conventional symbolism for representing dimensions, tolerances, surface characteristics, materials, and so on, they are capable of conveying true three-dimensional information to an experienced reader.

Models are useful because often one can study certain characteristics of an object more easily based on the model than its physical counterpart. This may be so because the object does not yet exist (as often is the case for physical models), or because it is not directly observable (as is the case for molecules), or because it cannot be examined in a controlled fashion (as often is the case for mathematical models). Note, however, that physical and mathematical models are limited in the scope of their usefulness: a problem of a new type often requires a new model.

As models, engineering drawings try to avoid these problems. They are relatively *general-purpose*: a drawing can be used to extract information for

many tasks, including the production of physical and mathematical models when they are needed. From this universality it follows that they can act as a *medium of communication* between the people that participate in the design of an object. Unlike physical and mathematical models, drawings also support the iterative, "sketchy" nature of design. They can be fuzzy and inaccurate when appropriate, whereas physical and mathematical models are tools for representing or analyzing a relatively finished object.

1.2 COMPUTER MODELS

Computer models consist of data stored in computer files that can be used to perform similar tasks as the other kinds of models discussed above, with the aim of sharing their merits and avoiding their problems. In particular, we aim at general-purpose models that can support a wide variety of applications, just like engineering drawings.

1.2.1 Geometric Modeling

The totality of data that would be stored in a computer model depends on the scope of questions one wants to be able to answer, and *a priori* it does not seem possible to limit the amount of potentially interesting data. At first sight, this may appear to make computer models unattractive.

The key observation that motivates this book is that many problems that we try to solve through models are inherently *geometric*. For instance, the problem of calculating a shaded image of an object includes geometric problems such as the following:

1. Which parts of the object are visible to the viewer?
2. What color should be assigned to each element of the image?

Similarly, the problem of generating control sequences for a numerically controlled machine tool is basically geometric: which sequence of movements of the tool will produce the shape of the object?

If we can represent the geometric shape of the object adequately for these questions, we would be able to provide answers to many others as well. In fact, it appears that the geometry of an object is the most useful part of the bulk of potential information. Moreover, techniques for storing and processing geometric data are relatively independent of particular applications: essentially identical methods can be used to construct models of machines, ships, chemical plants, and baby food containers.

Accordingly, it makes sense to separate the data dealing with the geometric shape of an object from other, nongeometric data. In this approach,

the totality of data needed for a particular scope of problems is called the *object model*, while the purely geometric part of it constitutes a *geometric model*. A geometric model, of course, is a subset of the object model.

1.2.2 Solid Modeling

Solid modeling is a branch of geometric modeling that emphasizes the general applicability of models, and insists on creating only "complete" representations of physical solid objects, i.e. representations that are adequate for answering arbitrary geometric questions *algorithmically* (without the help of interaction with a human user).

The goal of relatively general applicability separates solid modeling from other types of geometric models which are biased towards some special purposes. *Graphical models* are intended for describing a *drawing* of an object rather than the object itself. *Shape models* represent an *image* of an object. They may be unstructured collections of image elements, or may have some internal structure to aid, say, image processing operations. *Surface models* give detailed information on a curved surface, but do not always give sufficient information for determining all geometric properties of the object bounded by the surface.

Because of our interest in general-purpose modeling, we exclude these other types of computer models from the scope of this book. Nevertheless, as we shall see, some methods developed for the construction of these models have found their way to solid modeling as well.

1.3 PROBLEMS OF SOLID MODELING

The requirement of general applicability demands much of the completeness and accuracy of solid models. In the following, we discuss on an intuitive level some of the problems we shall deal with in later parts of this book; in Chapter 3 we shall take a more systematic look into them.

1.3.1 Completeness

The simplest kinds of computer models of physical objects transfer the methods used in engineering drawings directly to a computer to create *two-dimensional graphical models*. These models consist of two-dimensional graphical objects such as lines, arcs, text, and other notation needed in the figure.

Graphical models are perfectly appropriate to make the generation of technical drawings more efficient and to enhance their quality. Unfortunately, whereas human beings (or at least engineers) are capable of inter-

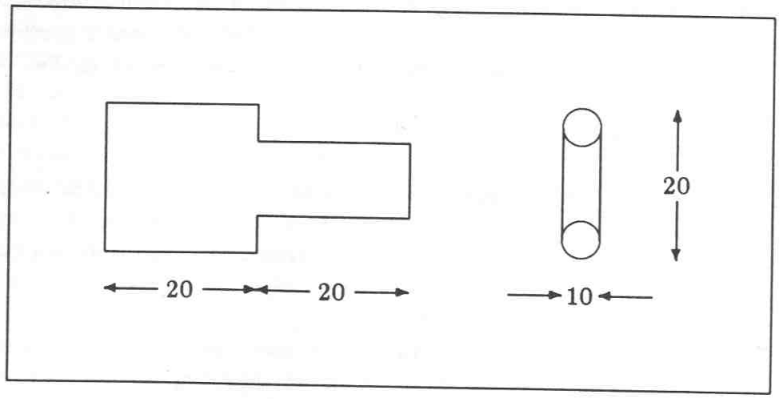


Figure 1.1 A nonsense drawing.

preting engineering drawings properly, computers in general lag far behind them in that skill. Thus graphical models in general cannot serve as solid models.

One of the problems of graphical models is that it is perfectly possible to make drawings that represent no actual shape; see, for instance, Figure 1.1 [105]. Still worse, the problem of algorithmically inferring correct three-dimensional information from a collection of two-dimensional figures remains unsolved for practical purposes, despite some interesting results gained with simplified problems.

With the aid of some computer power, two-dimensional graphical models can be upgraded to their three-dimensional counterparts by adding the information of the third coordinate. This results in a solid representation commonly referred to as the *wire frame* model. With wire frames, it becomes possible to store only a single three-dimensional model and generate all needed two-dimensional views from it. Hence one of the problems of two-dimensional graphical models can be overcome, because it is less easy to make drawings whose views are inconsistent.

Unfortunately, even a collection of three-dimensional lines is not sufficient for representing a shape, because some collections of lines may have several interpretations in terms of solid objects. The standard example is given in Figure 1.2.

The problem of graphical models is that they do not offer *complete* geometric information of the shape of the object—information that would be sufficient for calculating answers to arbitrary geometric questions. Nev-

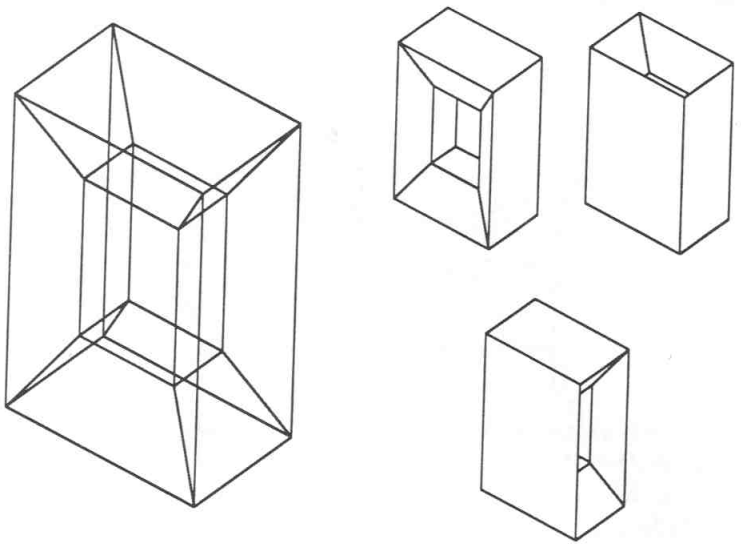


Figure 1.2 An ambiguous graphical model.

ertheless, graphical models are perfectly all right for the task they were originally developed for, namely generation and representation of drawings. We shall study them further in Chapter 2.

1.3.2 Integrity

To solve the hidden line and hidden surface removal problems, the normal step taken in computer graphics methodology is to replace graphical models by *polyhedral models* that give sufficient information for determining the hidden parts of objects. These models are constructed from two-dimensional primitives, polygonal faces, rather than just lines.

Unfortunately, new problems arise. Usually, hidden line removal algorithms assume that the polygons do not intersect each other except at common edges or vertices. Obviously, a properly constructed polyhedral model of a physical shape cannot include intersecting polygons, because the surface of the object would otherwise intersect itself; hence, it is natural to consider only non-selfintersecting polyhedral models valid. But how to ensure that models satisfy criteria for correctness such as this?

The *integrity* of solid models is one of the central issues of this book. Solid models are of little value if the construction of correct models is

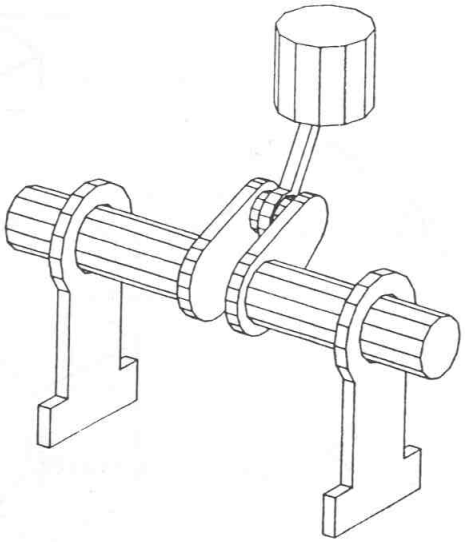


Figure 1.3 A polyhedral model.

excessively complicated. Note that mere integrity *checking* (say, detection of illegal intersections in the hidden line removal example) is only a partial solution to the problem of integrity, because the user must figure out the corrective actions to be taken himself, and a "trial-and-error" sequence is likely to emerge.

Integrity *enforcement* would avoid this problem by making the generation of incorrect models impossible. The new problem of providing sufficient integrity checks without penalizing in the ease of use and the flexibility of the modeling system now arises, however.

1.3.3 Complexity and Geometric Coverage

The problem of integrity is related to another problem, the *complexity* of generating a polyhedral model. Even a relatively simple object such as the one of Figure 1.3 requires more than one hundred polygons. It is a complicated, boresome, and error-prone task to generate such information by hand.

The geometric coverage of a polyhedral model is not sufficient for tasks that require accurate modeling of curved shapes—say, automobile body design. Industries that need such models developed very early methods

1. *What does the object look like?*
2. *What is the weight, surface area, etc. of the object?*
3. *Does this object hit this other one as they move?*
4. *Is the object strong enough to carry this load?*
5. *How can the object be manufactured with certain available manufacturing processes?*

Table 1.1 Geometric questions.

for dealing with quite complex shapes. Unfortunately, the difficulty of dealing with solid geometry in a computer rises sharply with the complexity of mathematical formulations used. Not very many solid modelers offer complete functionality for quadric surfaces and tori, and probably none for freeform surfaces.

1.3.4 Nature of Geometric Computation

According to our desire of general applicability, we expect solid models to be capable of providing *algorithmically* answers to typical geometric questions arising in engineering applications. Examples of such questions are listed in Table 1.1.

Observe that the result of a geometric question may be an image, a single number, or a Boolean constant (*true*, *false*). In fact, it can be another solid model that models the result of the calculation, as with the question "*what is the effect of this manufacturing process applied to this object?*" Clearly, it is important that a geometric modeler includes facilities to model not only physical objects, but also effects of *physical processes* applied to them. Of course, the modeler should be capable of repeated application of these operations to the results of previous similar operations. In other words, operations available in the modeler should constitute a *closed system* where all operations are guaranteed to maintain the correctness of the underlying models.

1.4 SOLID MODELING SYSTEM

One of the underlying ideas of geometric modeling is that it makes sense to separate modeling from the applications, and to seek modeling techniques

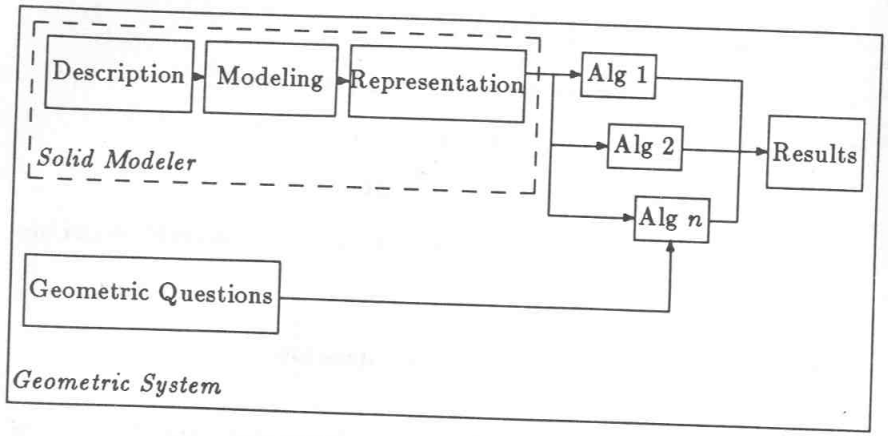


Figure 1.4 Functional components of solid modeler.

that are relatively independent of the particular objects modeled, and of the intended use of the models. Hence, is it possible to functionally distinguish from the application proper a piece of software that deals with solid models and provides answers to geometric questions such as those of Table 1.1—the *solid modeling system* or just *solid modeler* for short.

In addition to the problems of completeness, integrity, complexity, and geometric coverage discussed in the preceding section, there are many practical points that make the construction of a solid modeler a nontrivial task. To see why this is so, let us examine the functional components of a solid modeler and its role in a "geometric system" as depicted in Figure 1.4 [95].

Initially, objects are described for the modeler in terms of a *description language* based on the *modeling concepts* available in the solid modeler. The user may enter the description simply as written text, or preferably through a *user interface* with help of graphical interaction.

Once entered, object descriptions are translated to create the actual *internal representations* stored by the modeler. The relationship between the description language and the internal representations need not be a direct one: internal representations can employ modeling concepts different to the original description. Moreover, a solid modeler may well include several description languages intended for different kinds of users and applications.

The modeler must provide interfaces for communicating with other systems. These interfaces are used to transmit information for various algorithms, or perhaps complete solid models for other design systems. The

modeler must also include facilities for storing object descriptions and other data stored by the modeler into permanent data bases.

A solid modeler must necessarily handle geometric data in different, coexisting representations. The description language, of course, forms a solid model as such. For purposes of calculating answers to geometric questions, a transformation from the external description to an internal representation is necessary. In fact, for efficiency a solid modeler should support multiple internal representations of objects; hence, it must include *conversion algorithms* that can modify data from one representation to another.

PROBLEMS

1.1. List some other kinds of models you are familiar with. Can these models be represented in a computer?

1.2. In Mozart's opera *The Magic Flute*, prince Tamino fells in love with Pamina just by looking at her picture. To which extent is Tamino using a model in a reasonable fashion?

1.3. The extent to which the questions of Table 1.1 can really be considered "geometric" is actually debatable. List the actual information required for answering each of the questions. What information can be expected to be available in a generic geometric modeling system?

BIBLIOGRAPHIC NOTES

The separation of various kinds of geometric models is not strict, and the identical data structures and algorithms may be applicable in several kinds of models. For instance, picture files of graphics systems may consist of exactly identical primitives as geometric models of printed circuit boards, and many algorithms are of equal interest in both cases. Here, the distinction is more a statement of the intent than of the content of the model.

The separation of "modeling" and "graphics" is nevertheless one of the fundamental ideas behind the current standard device-independent graphics software packages such as GKS [57]. The roots of this distinction can be found from a conference held in Seillac, France in 1976 [47]; see also [89].

The design and implementation of "geometric systems" in the sense of Figure 1.4 is beyond the scope of this book. For further discussion on the architectures of computer-aided design systems, see e.g. the book by Encarnação and Schlechtendahl [37] and the proceedings [36].

Chapter 2

GRAPHICAL MODELS

Graphical models represent geometric information in terms of points and lines. As many of the techniques for processing graphical models are of interest also to solid modeling, it is appropriate to study them more closely.

2.1 BASIC MODELING PRIMITIVES

The construction of a graphical model is based on a selection of *modeling primitives*, and a collection of *modeling procedures* for their instantiation and manipulation. Let us start by examining the very basic modeling primitives of graphical models: points and lines.

2.1.1 Points

The simplest geometric object that we must be able to represent is a location in space, a single point. We shall use a 3-dimensional right-handed Cartesian coordinate system with axes x, y, and z to indicate a location in space, although other coordinate systems (such as cylindrical coordinates) can be of interest in special cases.

To represent and process points in a computer, we shall use the so-called *homogeneous coordinate representation* that adds a fourth coordinate w, as explained in Appendix A. Homogeneous coordinates are convenient because many operations of interest can be performed simply by multiplying the vector $[x\ y\ z\ w]$ with a 4-by-4 matrix. In particular, arbitrary sequences of *translation*, *rotation*, and *scaling* operations on x, y, and z can be combined by multiplying together appropriate 4-by-4 matrices explained in the Appendix.

```
typedef struct
{
        float       vx, vy, vz, vw;
        /* other information can be added here */
} point;

ex1()
{
        point       *p;

        p = newpoint(0.0, 0.0, 0.0);
}
```

Program 2.1 Type *point*.

To represent the point $(x\ y\ z)$, we need a data type *point*, having components *vx*, *vy*, *vz*, and *vw*. Program 2.1 gives a definition in C of this information. The procedure **newpoint** is assumed to allocate storage for a new *point* node and store the coordinate values into it. Note that points may be associated with other, problem-dependent data. In Program 2.1, we have represented only the essential geometric data explicitly.

2.1.2 Lines

Points alone would be quite inconvenient for modeling interesting figures. As they have just a location in space, we would not be able to create and process geometric objects having extent. The usefulness of points comes from the observation that straight line segments (that consist of infinitely many points) can be represented indirectly but completely in terms of their end points. A straightforward approach is to represent line segments in terms of two points. In this approach points can be created only as a part of a line segment. Program 2.2 gives the C definition of a type *line* that implements this, using the definitions of Program 2.1.

2.1.3 Independent Points and Lines

When dealing with multiple line segments, the approach reflected in Program 2.2 suffers from poor storage utilization because end points shared by several line segments are duplicated in each. Furthermore, that two line segments share an end point can only be detected by comparing the coordinate values, a slow and numerically risky operation. Another way


```
typedef struct
{
        point       p1;
        point       p2;
} line;

ex2()
{
        line        *l1;

        l1 = newline(0.0, 0.0, 0.0, 1.0, 0.0, 0.0);
}
```

Program 2.2 Type *line*.

of structuring the information that avoids this replaces multiple copies of a point by references to a single copy. In C, a natural way to implement this is to use separate records for points and line segments, and link them together with pointers. An implementation of types *point* and *line* that follows this approach is given in Program 2.3.

2.1.4 Design Issues

The difference between the two representations may seem to be small. Nevertheless, it is important and illustrates the nature of design decisions that must be done during the design of geometric systems. Some problems and issues are discussed below.

Description of Figures

Both data representations allow the implementation of similar procedures for the description of simple line figures, such as boxes, stroke-precision characters, and symbols. However, the "flat" design of Program 2.2 makes the implementation of certain kinds of operations inconvenient.

Consider, for instance, an operation where a box (four-sided rectilinear object whose sides are parallel to coordinate axes) is created by indicating its two opposite corners. In the first approach, the natural implementation reads two locations of a pointing device, and goes on to create the box. In the second approach it is natural to let the user indicate two existing points, and create the box using their stored coordinates. Obviously, it is much easier to create boxes that "match" in the second alternative.

```
typedef struct
{
        float       vx, vy, vz, vw;
} point;
typedef struct
{
        point       *p1;
        point       *p2;
} line;

ex3()
{
        point       *pnt1, *pnt2;
        line        *l1;

        pnt1 = newpoint(0.0, 0.0, 0.0);
        pnt2 = newpoint(1.0, 1.0, 1.0);
        l1 = newline(pnt1, pnt2);
}
```

Program 2.3 Another definition of *point* and *line*.

Manipulation of Figures

The differences of the two representations are accentuated as for the manipulation of figures. Consider a simple figure, such as the cottage in Figure 2.1.

If we modify the endpoints of the line segments so as to "stretch" the cottage, the first representation forces us to repeat the same modification for all line segments that meet at a point. In the second approach, we only need to modify the coordinates once, and all line segments would "follow" the change. The approach of storing the points separately also enhances the viewing of models, because each point needs to be projected only once.

As the second approach provides independent modeling primitives for points and lines, it makes various more advanced construction techniques for figures possible. We shall discuss these additional construction techniques further in Section 2.3.

Integrity

Small numerical errors can cause problems in the first approach. For instance, if line segments that seem to meet a point do not quite do so, an

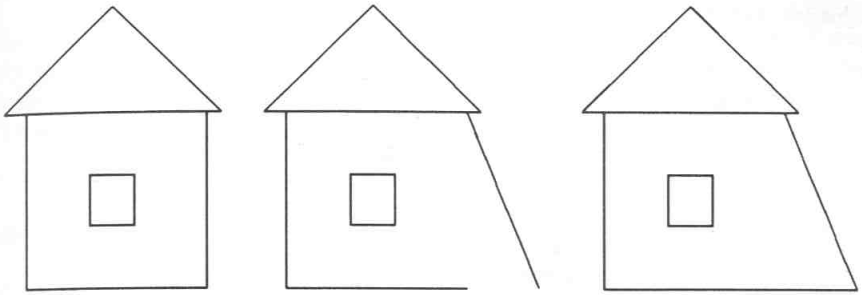


Figure 2.1 A house.

operation that attempts to move that point could erroneously leave a line segment behind.

On the other hand, the introduction of pointers has the drawback that we have also introduced the possibility of having "dangling" pointers that refer to incorrect information. For instance, suppose that in addition to procedures **newline** and **newpoint**, the corresponding *deletion* procedures **delline** and **delpoint** are needed for the interactive manipulation of figures. Obviously, **delpoint** should be applicable only if the point to be deleted does not appear in any line segments. In the data structure of Program 2.3, testing this condition is expensive because of the necessity to scan all line segments.

A straightforward way to improve the solution is to include a new attribute *usagecount* into *point*. Initially zero, its value is incremented each time **newline** creates a line segment emanating from the point. Similarly, **delpoint** can decrement *usagecount* for the end points of the line segment it deletes. Clearly, the procedure **delpoint** can work correctly by examining **usagecount**: if it equals zero, the point does not appear in any line segments, and can safely be deleted.

Another mechanism for handling the problem is based on including an access path from each point to the all line segments adjacent to it. One way to implement this is to associate with each point a pointer to a linked list of adjacent line segments. As each line segment will appear in exactly two such lists, they can be realized by including two new pointers to each line segment node. In this approach it becomes possible to implement a more powerful point deletion operation that removes a point and all its adjacent lines. We leave the details of this and the specification of the resulting modeling procedures as an exercise to the reader.

```
typedef struct
{
        point       *allpoints;     /* list of all points */
        line        *alllines;      /* list of all lines */
} object;

typedef struct
{
        float       vx, vy, vz, vw;
        point       *nextpoint;     /* next point of the list */
} point;

typedef struct
{
        point       *p1;
        point       *p2;
        line        *nextline;      /* next line of the list */
} line;
```

Program 2.4 Type *object*.

We choose the second representation for the purposes of this chapter; see [15] for additional discussion of the pros and cons of the two approaches.

2.2 GROUPING

The concepts of Program 2.3 give us the basic tools for creating graphical models of objects. In order to employ our models, we still need the capability of collecting logically interrelated points and line segments together into what might be called *groups*. This would allow our algorithms to process groups as a whole.

2.2.1 Objects

A new modeling primitive, *object* is needed to accomplish the grouping of related line segments and points. Objects consist of a set **allpoints** of points, and a set **alllines** of line segments. One way to implement this is to gather points and line segments belonging to the same set into linked lists, and let **allpoints** and **alllines** point to their heads. Definitions of *point* and *line* must then be modified to include a pointer to the next element in the list. These changes are implemented in Program 2.4.

Modeling procedures must be modified accordingly. One possibility is to include a pointer to the parent object as the first argument of procedures **newpoint** and **newline**, and let them add the new *point* or *line* to the correct list. Program 2.5 gives an example of the modeling procedures after these modifications.

2.2.2 Graphical Symbols

The grouping facility presented above is the simplest possible. A more interesting grouping mechanism allows the definition of *graphical symbols* that can be constructed through an *instantiation procedure* that assigns position and orientation information for a new *instance* of an object. In this extension, objects may consist of symbols, in addition to points and line segments.

A natural solution to this assigns a 4-by-4 transformation matrix that describes the position and orientation of each object instance relative to its parent object. (See Appendix A for information on 4-by-4 transformations.) Program 2.6 illustrates some possibilities offered by such a mechanism.

In the program, procedure **instanceof** creates a new instance of the object given as the first argument, transformed with the matrix given as the second argument. Such an instance can be added to an object with procedure **newinstance** analogously to procedures **newpoint** and **newline** of Program 2.5. Observe how the procedure can create a staircase pyramid from three instances of a single box.

The symbol data structure forms a directed acyclic graph which must be traversed when an object including symbols is "evaluated." In this process, the standard technique is to maintain a transformation stack during the traversal. Initially, an identity transformation is pushed onto the stack. When advancing to a child symbol, the current top of the stack is multiplied by the transformation of the child, and pushed onto the stack. This gives the correct transformation for positioning the child properly with respect to its parent. When the processing of a child is finished (perhaps after recursive evaluations of its children), the top of the transformation stack can be removed.

Symbol schemes are useful when pictures are formed from many instances of a small number of basic objects; consider, for instance, circuit diagrams. Correspondingly, techniques for symbol manipulation belong to the core of computer graphics methodology. We shall not delve deeper into graphical symbols; for a good exposition on the subject, the reader is referred to [39].

```
object  *newbox(dx, dy, dz)
float   dx, dy, dz;
{
        object      *box;
        point       *p1, *p2, *p3, *p4, *p5, *p6, *p7, *p8;

        /* make box an empty object */
        box = newobject();

        /* create points is the box */
        p1 = newpoint(box, 0.0, 0.0, 0.0);
        p2 = newpoint(box, dx, 0.0, 0.0);
        p3 = newpoint(box, dx, dy, 0.0);
        p4 = newpoint(box, 0.0, dy, 0.0);
        p5 = newpoint(box, 0.0, 0.0, dz);
        p6 = newpoint(box, dx, 0.0, dz);
        p7 = newpoint(box, dx, dy, dz);
        p8 = newpoint(box, 0.0, dy, dz);

        /* create lines of the box */
        newline(box, p1, p2);
        newline(box, p2, p3);
        newline(box, p3, p4);
        newline(box, p4, p1);
        newline(box, p5, p6);
        newline(box, p6, p7);
        newline(box, p7, p8);
        newline(box, p8, p5);
        newline(box, p1, p5);
        newline(box, p2, p6);
        newline(box, p3, p7);
        newline(box, p4, p8);

        return(box);
}
```

Program 2.5 Definition of a box.


```
typedef float matrix[4][4];

object  *pyramid()
{
        object      *b, *pyr1, *pyr2, *pyr3;
        instance    *s1, *s2, *s3, *s4;
        matrix      m1, m2;

        b = newbox(1.0, 1.0, 0.25);

        ident(m1);
        trans(m1, -0.5, -0.5, 0.0);
        ident(m2);
        scale(m2, 1.2, 1.2, 1.0);
        trans(m2, 0.0, 0.0, -0.25);

        pyr1 = newobject();
        s1 = instanceof(b, m1);
        newinstance(pyr1, s1);

        pyr2 = newobject();
        s2 = instanceof(pyr1, m2);
        newinstance(pyr2, s1);
        newinstance(pyr2, s2);

        pyr3 = newobject();
        s3 = instanceof(pyr2, m2);
        newinstance(pyr3, s1);
        newinstance(pyr3, s3);
}
```

Program 2.6 Pyramid.

2.3 EXTENSIONS

The very basic functionality developed so far is adequate only for illustrating the problems of, and techniques for graphical models. A practical system should include various extensions both for increased functionality and ease of use. Some of these extensions are discussed below.

2.3.1 Construction Techniques

For the time being, we can construct graphical models from two modeling primitives, *point* and *line* that can be grouped to form *objects*. Our modeling system is capable of

1. creating new *points* at desired locations,
2. creating *lines* between two existing *points*, and
3. scanning through *lines* and *points* of an *object*.

Clearly, by just these constructs we can model all possible line figures, so as for expressive power, we are done! However, for practical purposes many other construction methods for points and line segments would be useful as well. These additional constructions may include the following:

1. In plane, construct a line segment parallel to a given line, of given length, and starting from a given point. Length is measured along the direction of the argument line.
2. Construct a line segment perpendicular to a given line, of given length, and starting from a given point. The direction of the line segment is chosen to be 90 degrees counterclockwise from the direction of the argument line.
3. Construct a point as the intersection of two lines.
4. Construct a point as the mirrored image of a point with respect to a line.

Note that some of these operations may create many points and line segments in one shot.

Commercial drafting systems include all these and many other construction methods as well, including functionality for creating circles and arcs in terms of existing points, lines, and arcs. See [105] for a compact account of various construction methods and their implementations.

2.3.2 Systems With Constraints

Suppose the additional construction operations described in Section 2.3.1 above are implemented on top of the data structure of Program 2.4 (perhaps extended as suggested in Section 2.1.4). Let us consider the implementation of a new procedure `modifypoint(p, newx, newy, newz)` that assigns new coordinates to point.

Suppose further that a line segment l_1 starting from point p and of given length is created to be parallel to a line l_0. What happens if we update the coordinates of p? After the operation, l_1 (and all other line segments emanating from p) has been modified, and in all likelihood l_1 is not anymore parallel to l_0.

This effect can be perfectly acceptable in some applications. Sometimes, however, we would like to see the line segment l_1 move with p while remaining parallel to l_0. Hence, the other end point of l_1 must move as well, and any line segments potentially emanating from it move according to how they were constructed. This facility is useful, say, when dealing with *rigid* objects whose shape is invariant under movement.

Representations for Constraints

To implement such a scheme we must be able to associate constraints with each line segment and point. For simplicity, let us discuss how we could store the constraints in the case of just two construction methods of line segments, namely

1. construct a line segment between two points
2. construct a line segment parallel to another line, starting at a point, and of given length.

Observe that the latter construction creates both a line segment and a point. It can be implemented as first constructing the other end point of the line segment as determined by the arguments, and then constructing the resulting line segment between the two end points.

Constrained graphical models generally break the representation of each entity into two parts. Data that are independent of the construction method by which the entity was created are referred to as the *canonical* representation of the entity, whereas the remaining data forms the *reference* representation of the entity.

Let us use the representation of Program 2.4 as the canonical representation of our line segments and points. To include the reference information into line segments and points, we add a new attribute *refs* for storing a pointer to a node that records the construction method used for the

```
typedef struct
{
        int         code;          /* = 0 */
        float       x, y, z;
} point_from_coordinates;

typedef struct
{
        int         code;          /* = 1 */
        line        *l;
        point       *p;
        float       *l;
} point_at_distance_from_point_along_line;
```

Program 2.7 Points with constraints.

line segment or the point. These nodes include a code that identifies the particular construction method they represent, and a list of arguments of the construction procedure. The arguments may be pointers to geometric entities or numerical values.

As our line segments are always bounded by two points, it turns out that only points need to be constrained. In our example, we need the two constraint nodes depicted in Program 2.7 for the two different kinds of points: (1) those that simply are created from a triple of coordinates (like in `newpoint`), and (2) those that are created in the construction above as the other end point of the parallel line segment.

Regeneration

Let us now return to the implementation of `modifypoint` with the constrained data structures of Program 2.7. Obviously, the operation is meaningful only if the point is of type (1) above (why?). After checking this, the algorithm should *regenerate* all line segments and points whose canonical representation depends on the modified point either directly or through a sequence of any constructions.

Observe that the construction information defines a partial order of line segments and points[1]. Let us write $a < b$ to denote that a is defined in terms of b. To perform the regeneration operation correctly, we must process all objects $e < p$, the modified point. Moreover, we must assure

[1]This is so because the constrained model was created by a linear sequence of modeling operations in the first place.

that if $a < b$, b is regenerated before a.

One way to accomplish this is to link all points and line segments into yet another linked list in the order they were created. The regeneration algorithm can then consider points and line segments in the order they appear in the list. However, this solution has the drawback that lots of unnecessary work is performed if only a few entities must be updated. At the cost of additional stored information, a more efficient approach is to perform a search in the graph defined by constrain relationships while taking the relation $<$ into account.

Unbounded Graphical Entities

Some graphical modelers make a distinction of *bounded* graphical entities (such as our line segments) and *unbounded* graphical entities. In these systems lines are the truly unbounded lines of mathematics, whereas a separate entity is used for representing bounded line segments.

While increasing the complexity of the modeler, this separation allows the inclusion of additional construction methods and simplifies the specification of many constructions. For instance, a parallel line could be constructed just by giving a distance from the argument line, whereas a starting point and a length would be additionally required for the creation of a bounded line segment.

Observe that if unbounded graphical entities are used, both lines and points would be constrained. For instance, a parallel line constructed as above would depend on the argument line. Points created on the line would depend on it, and line segments adjacent to those points on them.

2.3.3 Parametric Design

Constrained graphical data structures bring us to data structures useful for a *drafting system*. A drafting system allows the user to interactively construct figures from simple geometric entities. A full-scale system would include all facilities discussed in this chapter, including grouping (perhaps in several ways), symbols, many construction methods, and constrained data structures. It would probably include additional modeling primitives, such as arcs and character strings. It would also include functional components not discussed here at all, such as a graphical user interface and mechanisms for storing models into secondary storage.

One interesting characteristic of some drafting systems is the capability of supporting *parametric design*. It can be understood as a generalization of constrained data structures. Consider again the example of Program 2.7. Going a step further, let us replace the number *length* by a pointer to a

new entity *number*. This gives us the capability of storing *all* dimensions of an object as *numbers*, and referring to them in other constructions.

In fact, we can include construction methods for numbers in our modeling system. These constructions are likely to include simple mathematical expressions and geometric measurements such as distance. Like other entities, each number has a canonical format that simply gives its value, and a constrained format that represents the expression or measurement from which the number was generated. Extending the regeneration algorithm to numbers gives the capability of modifying the (few) numbers that form the roots of the whole construction, and hence generating new versions of the whole design easily.

PROBLEMS

2.1. Write a C definition of points and lines as suggested in Section 2.1.4. That is, associate with each point the collection of lines it is adjacent to.

2.2. Write modified versions of procedures `newline` and `newpoint` that use the modified data structures of Problem 1.

2.3. Specify and write procedures `delline` and `delpoint` based on the data structures of Problem 1.

2.4. Based on the data structures of Program 2.5, write procedures `newobject`, `instanceof`, and `newinstance` discussed in Section 2.2.

2.5. Based on the data structures of Program 2.7, specify and write the procedures needed for drawings with constraints.

BIBLIOGRAPHIC NOTES

For this book, graphical models are only of secondary interest. A reader interested on them should therefore consult additional literature.

Graphical models are the major ones currently in use in Computer-Aided Design and Drafting systems. Consequently, books on CADD [15,11,43] contain material on graphical models much beyond this chapter.

The texts of Faux and Pratt [38] and Mortenson [85] contain (in addition to a lot of other material) good overviews of curve and surface presentations. The book by Woodwark [131] gives a compact account on not only two-dimensional techniques, but also on many aspects on three-dimensional models as well. The books of Chasen [24], Rogers [103,102], and Bowyer and Woodwark [16] contain readily useful techniques for various geometric constructions (primarirly) for simple geometric objects. Commercially available CADD systems are the subject of [32].

Parametric design techniques are in use not only in drafting systems but also in the user interfaces of solid modelers. For instance, the user interface of MEDUSA [87] has a parametric design facility that includes also circular arcs and is capable of storing relationships involving tangency constraints. As a more advanced example, Lin *et al.* [71] give an algorithm that can check whether a set of such constraints fully determines a two-dimensional figure consisting of line segments and arcs. The algorithm can also flag overconstrained figures.

Chapter 3

GEOMETRICALLY COMPLETE MODELS

Graphical models do not satisfy our requirements for completeness and general applicability. We now start the search for other solutions by characterizing more closely the desired properties of solid models.

3.1 PROBLEMS OF GRAPHICAL MODELS

Chapter 2 gave us an intuitive understanding on how to construct graphical models from lines and points. While there are many practically important aspects that we have not touched at all (for instance the representation of graphical models in secondary storage, or their generalization to represent and process curves), we can now assess the representational issues of graphical models.

So, which of the questions of Table 1.1 on page 9 can be answered based on the information available in graphical models?

It is easy to generate simple graphical output based on the information in a graphical model (as the term implies). Unfortunately, points and lines alone do not, in general, convey sufficient information to create more sophisticated output involving removal of hidden lines or shaded surfaces. The least that is needed is a polyhedral model. (Of course, for a mere wire frame consisting of just points and lines, surface properties such as visibility and shading are not even defined.)

No fully automatic conversion from a graphical model to a polyhedral model can exist, as the ambiguous object of Figure 1.2 on page 7 shows. Remarkably, an algorithm capable of enumerating *all* polyhedral models

corresponding to a given graphical model has been reported by Markowsky and Wesley [80]. But human guidance is still needed to select the correct one.

For restricted classes of objects, even an automatic conversion is possible. For instance, Hanrahan [50] reports an algorithm that can construct a polyhedral model from a wire frame whose points and lines form a planar graph, i.e., a wire frame which is known to correspond with a polyhedron with no "holes."

For some problems graphical models are adequate. If we need a collision testing algorithm for finding free paths for a moving object in a scene of static objects with lots of free space around them, we may find that collision testing between the convex hulls of the objects is adequate. Then graphical models are perfectly all right, because their information is sufficient for convex hull calculation. However, if the collision testing problem involves the search of narrow free paths inside a partially finished assembly, convex hulls will always intersect and provide little help.

With graphical models, we face thus the problems of incompleteness and potential ambiguity. If we are willing to seriously limit the scope of problems we want to deal with automatically, or if we are willing to limit the class of objects to be modeled, these problems can be avoided. But such limitations are in contradiction with the aim for general-purpose models set forth in Chapter 1, and motivate our search for other, geometrically more rigorous models.

3.2 A THREE-LEVEL VIEW OF MODELING

In the preceeding, we used terms such as "completeness" and "ambiguity" to speak of the merits and the problems of graphical models without bothering too much about their meaning. Some examples of the previous section seem to blur the intuitive image we have of these terms, however. Are graphical models "complete" as far as convex hulls are concerned? Are "ambiguities" resolved if we wisely stick to sufficiently simple objects?

This confusion can be avoided if we adopt a more rigorous view of modeling [95]. This view is based on distuingishing between three separate levels of modeling (see Figure 3.1):

1. *Physical objects:* By means of models, our aim is to speak and argue about some real things of our three-dimensional real world. Unfortunately, assuming a Platonician view, we cannot even perceive a real-world object in its full complexity and sub-microscopic detail, much less represent all aspects of it in a computer.

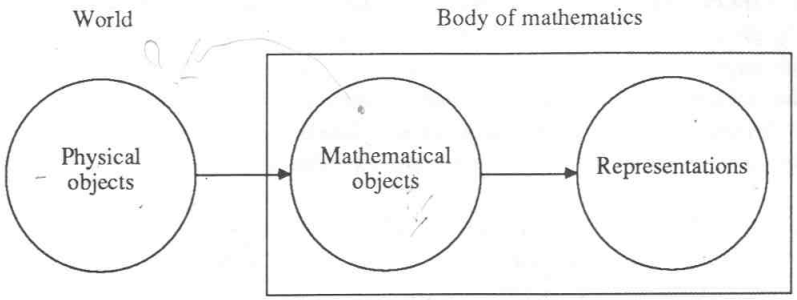


Figure 3.1 A three-level view of modeling.

2. *Mathematical objects:* In order to have any hope of modeling objects in a computer, we must therefore adopt a suitable idealization of the real three-dimensional physical objects we are ultimately interested in. These idealized objects should have an intuitively clear connection with the real world, while being so simple that we can assign computerized representations to them.

   To achieve this, we shall characterize rigorously a class of mathematical objects (the *modeling space*) we would like to speak about. As we shall see later, such a characterization can be achieved by using basic concepts from the mathematical theories of point-set and algebraic topology.

3. *Representations:* After establishing rigorously the class of mathematical objects we are interested in, the final step of the modeling activity is to assign to the mathematical object a representation which is suitable for computer manipulation. As we shall see later, the three-level view of modeling allows us to characterize the desirable properties of solid modeling methods rigorously by analyzing the relationships of mathematical objects and their computer representations.

According to the three-level view, we shall not attempt to model actual physical properties of things. Instead, we shall try to represent some idealized objects in a manner that allows us to analyze their idealized properties, and hope that the results gained in this fashion permit us to gain information on the actual real objects hiding in the background.[1]

[1] Of course, sometimes the aim of our modeling is *not* related to physical things, but precisely to idealized things. For instance, the specification of a mechanical part defines an idealized nominal object (or a class of mechanically equivalent objects). In this case,

3.3 MATHEMATICAL MODELS OF SOLIDS

Let us now concentrate on sharpening our intuitive view of "solidity," i.e., on the proper definition of a class of mathematical objects that forms the subject matter of solid modeling, while leaving the exact nature of the particular geometric problems we want to solve with models aside (as for now).

It turns out that we can arrive at a characterization based on two approaches, one stressing the "three-dimensional solidity" of things, and the other concentrating on the fact that we are primarily interested on the bounding surfaces of things. These points of view lead us to using the languages of point-set topology and algebraic topology, respectively.

The next sections are aimed at introducing a rigorous characterization of the objects we shall be interested in. By necessity, they are somewhat technical; some of the required terminology is introduced in Appendix B.

3.4 POINT-SET MODELS

Starting from the very basics, we shall consider the three-dimensional Euclidean space E^3 a suitable idealization of the real space our real objects lie in. Consequently, the most general mathematical abstraction of a real solid object is a subset of E^3, i.e., a set of points of E^3.

The advantage of the point-set idealization of real objects is that we can use concepts of *point-set topology* to characterize rigorously the desired properties of three-dimensional objects. Because we shall be interested usually only in finite objects, we can start from the following working definition:

Definition 3.1 *A* solid *is a bounded, closed subset of* E^3.

The restriction to closed sets (as an alternative of open sets) is a matter of convention only.

While Definition 3.1 certainly captures certain aspects of our notion of a solid, the class of objects permitted by it is far too large to make it really useful for our purposes. It turns out that to be considered "solid," a bounded, closed point set must satisfy a collection of additional requirements. The next sections will elaborate these.

it is the real world (the manufacturing engineer) who must imitate the idealized world, and not vice versa.

3.4.1 Rigidity

It is natural to expect that a solid object should remain the same if it is moved from one location to another. This can be expressed more rigorously by requesting that solids must remain invariant under *rigid transformations*, i.e., translations and rotations. The following definition captures the essence of this characterization:

Definition 3.2 *A* rigid object *is an equivalence class of point sets of* E^3 *spanned by the following equivalence relation* $\circ$*: Let* A, B *be subsets of* E^3. *Then* $A \circ B$ *holds iff* A *can be mapped to* B *with a rigid transformation.*

We might now add rigidity as one additional requirement to Definition 3.1. However, the usefulness of rigidity as defined above is somewhat diminished by the fact that most solid representations do not allow an efficient test for whether the relation $\circ$ is true for two models.

3.4.2 Regularity

Whether rigid or not, many closed and bounded point sets do not satisfy our notion of solidity. For instance, a set consisting of distinct points or lines is not solid.

We expect that a solid is "all material"; it is not possible that a single point, or a line, or even a two-dimensional area would be missing from a solid. Likewise, a "solid" point set must not contain "isolated points," "isolated lines," or "isolated faces" that do not have any material around them.

Again, these informal requirements can be compactly captured in the point-set topology language:

Definition 3.3 *The* regularization *of a point set* A, $r(A)$, *is defined by*

$$r(A) = c(i(A))$$

where $c(A)$ *and* $i(A)$ *denote the closure and interior of* A. *Sets that satisfy* $r(A) \equiv A$ *are said to be* regular.

Informally speaking, the regularization tears off all "isolated" parts of a point set, covers it completely with a tight "skin," and fills the result up with material (see Figure 3.2).

Regularity is so widely used as a characterization of reasonable solids that the following definition (by Requicha [95]) is appropriate:

Definition 3.4 *A bounded regular set is termed an* r-set.

Observe that regular sets need not be connected; for instance, the rigid combination of two regular sets is itself a regular set.

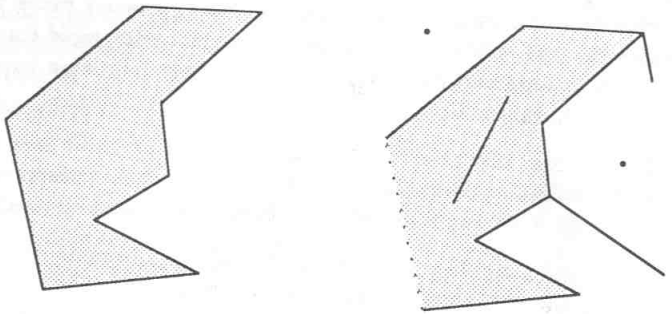


Figure 3.2 A regular and a nonregular set.

3.4.3 Representational Finiteness

We have no direct means for representing continuous point sets in a computer. To model objects with dimensionality (such as a straight line), we must always employ indirect methods (such as representing the line in terms of its endpoints). Even if it is physically naive, we must therefore assume that solids have some indirect finite representation.

Usually we must require that the surface of a solid is "smooth"[2] and can adequately be modeled in terms of reasonably simple mathematical functions. There are certain subtleties involved in the "reasonably simple" of above. For instance, the set of Figure 3.3 [94] is defined by the expression

$$x, y, z : \; x_0 \leq x \leq x_1, \; y = \sin(1/x) + h, \; z_0 \leq z \leq z_1$$

that includes only (seemingly) simple functions. Yet the set does not satisfy the ordinary notion of a solid if 0 is contained in the range $[x_0 x_1]$.

To get rid of such anomalous situations, we should restrict ourselves to objects whose surfaces are algebraic (or analytic at least). That is, the surfaces must be representable by means of (finite) polynomials of x, y, and z.

[2]The so-called *fractal surfaces* [72,41] form a remarkable exception in this respect: they are nowhere smooth but still have an indirect, recursive representation.

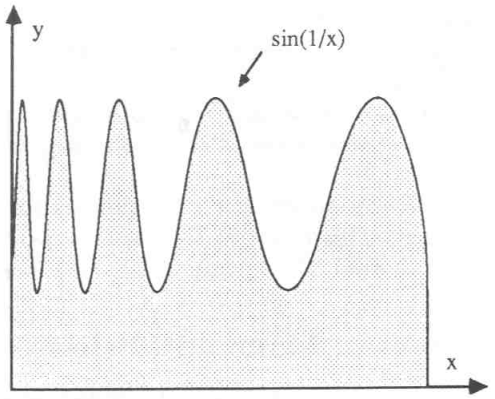


Figure 3.3 A nonphysical solid.

3.5 SURFACE-BASED MODELS

As seen in the previous section, we had to refer to properties of the bounding surfaces of a solid to avoid certain anomalies not captured by the point-set view. Indeed, it is perfectly possible to characterize modeling spaces just based on properties of surfaces.

The surface-based characterization of solids looks at the boundary of a solid object. The boundary is considered to consist of a collection of "faces" which are "glued" together so that they form a complete, closing "skin" around the object. In order to be able to rigorously speak about this informal procedure, and state explicitly under which conditions we get an object satisfying our notion of "solidity," we shall use concepts developed for another branch of topology, the so-called *algebraic topology* [52,81]. Again, the following development is somewhat technical, and some concepts introduced here will be used only in Part Two of this text.

3.5.1 2-Manifolds

Intuitively, a surface may be regarded as a subset of E^3 which is essentially "two-dimensional": every point of the surface (except points on the edge of an "open" surface patch) is surrounded by a "two-dimensional" region of points belonging to the surface.

The inherent two-dimensionality of a surface means that we can study its properties through a two-dimensional model. First, we shall define a

more abstract (and precise) notion of a "surface," and then construct a simple two-dimensional model for it in terms of a special topology defined on the two-dimensional Euclidean space E^2.

The more abstract counterpart of a "closed surface" is defined as follows:

Definition 3.5 *A* 2-manifold M *is a topological space where every point has a neighborhood topologically equivalent to an open disk of* E^2.

Intuitively, a bug living on M sees a continuous simple region of the surface all around itself. This intuitive view applies perfectly to ourselves and the planet Earth; indeed, the surface of a sphere is a 2-manifold.

3.5.2 Limitations of Manifold Models

Let us now fix our attention to an r-set A. If a 2-manifold M and the boundary of A, $b(A)$, are topologically equivalent, we say that A is a *realization* of M in E^3. As we shall see, the class of 2-manifolds that have at least one realization can be characterized precisely; we shall call such 2-manifolds *realizable*.

Unfortunately, not all r-sets are realizations of some 2-manifold. Figure 3.4 depicts some such sets. The problem with all these objects is that the surface "touches" itself in a point or at a curve segment. The neighborhoods of such points are not simple disks, as required by Definition 3.5. For instance, the neighborhood of the problem point in Figure 3.4(a) consists of two disks.

Hence, there is an inherent theoretical mismatch between r-set models and manifold models. For practical purposes, objects such as those of Figure 3.4 can be represented indirectly as 2-manifolds by "ignoring" the exceptional points and line segments; hence the object of Figure 3.4(a) would be represented as the rigid combination of two components that just happen to touch each other at a point.

3.5.3 Plane Models of 2-Manifolds

To establish sufficient conditions for the realizability of a 2-manifold, we need a mechanism that can represent all 2-manifolds in a way that allows us to reason about their properties. For this purpose, we use the *plane models*.

To illustrate the basic idea of the representation, let us consider first a simple example. Figure 3.5(a) represents a four-sided polygon, while (b) represents the surface (cylinder) obtained by gluing the two edges labeled with α in (a) together.

Not all 2-manifolds are realizable

Not all r-sets are realization of 2 manifold

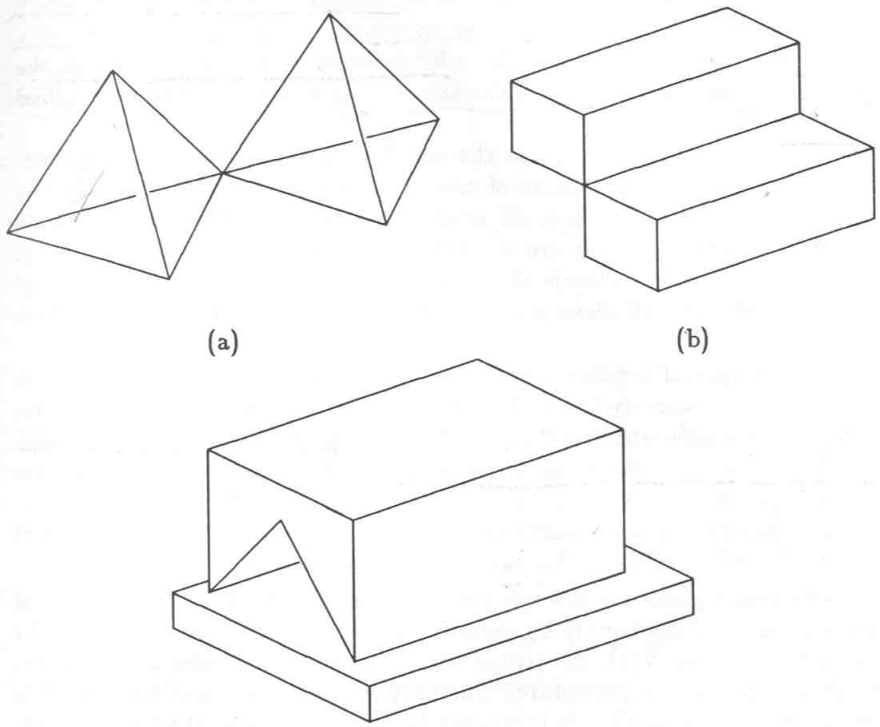


Figure 3.4 Solids with nonmanifold surfaces.

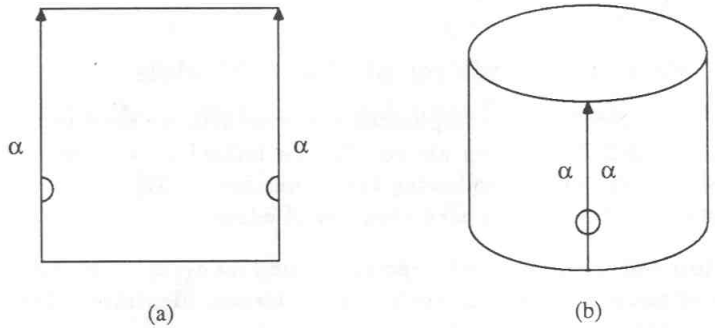


Figure 3.5 A cylinder surface.

The basic idea of what follows is to employ a (labeled) plane figure (such as the rectangle (a)) in order to reason about the properties of a surface (the cylinder). This is done by defining a *special topology* on the plane figure that pretends that the labeled edges have already been glued together as in (b).

As noted in Appendix B, in the *natural topology* of E^2, all neighborhoods of points are open disks of some radius r around the point. In the special topology we shall use, all neighborhoods except the neighborhoods of points on labeled edges are the same as those of the natural topology. However, the neighborhoods of those points are considered to consist of the union of two half-disks around symmetrical points on the edges (see Figure 3.5(a)).

By this special topology, symmetrical points on edges have the *same* neighborhoods, namely the disk neighborhoods they would have if the labeled edges were glued together. Such points are said to be *(topologically) identified*, and are treated as a single point. Observe that now all points of the cylinder, except those on the unlabeled edges of (a), have neighborhoods topologically equivalent to a disk of E^2. Because of these exceptional points the cylinder is called a *surface with boundary*.

Additional plane models are given in Figure 3.6. The plane model of a sphere is obtained simply by identifying the corresponding points on the two labeled edges of (a); the (American) football is an almost perfect practical example of this procedure. The natural continuation of Figure 3.5 is the model of a torus (b). It is created by identifying also the top and bottom edges of the cylinder model in a pairwise manner. (Think of bending a cylinder until its ends meet.) Observe that now all point neighborhoods are full disks, and these surfaces are *closed*.

Case (c) gives another variation of Figure 3.5: here the orientation of the identified edges have been reversed to yield the model of the well-known *Möbius strip*[3].

3.5.4 Formal Definition of Plane Models

In order to deal with more complicated plane models, we need to formalize the intuitive definition given above. In the following, we give a formal definition of plane models following the exposition of [52].

We start by defining the *identification* of edges:

Definition 3.6 *Let P be a set of polygons, and let $a_1, a_2, \ldots$ be a collection of edges of these polygons. These edges are termed* identified *when a new topology is defined on P as follows:*

[3]The use of plane models was invented by Möbius.

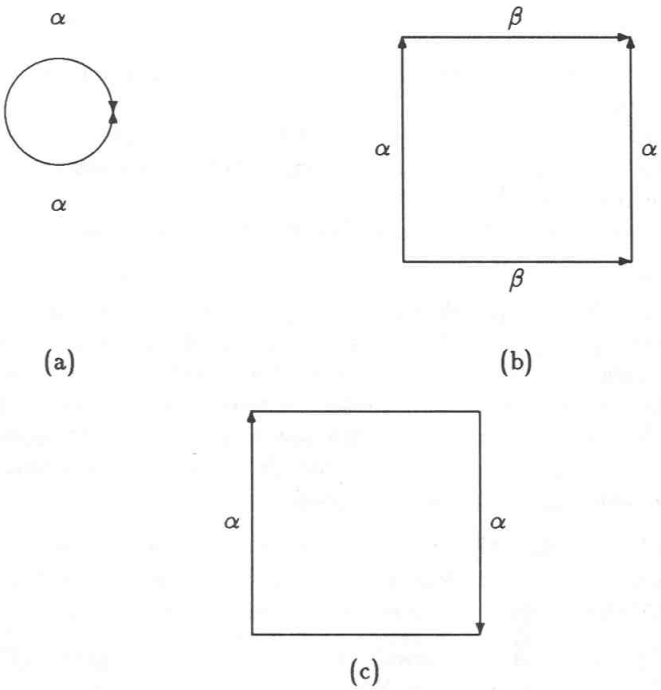


Figure 3.6 Other surfaces: sphere (a), torus (b), and Möbius strip (c).

1. *Each edge is assigned an orientation from one endpoint to the other, and placed in topological correspondence with the unit interval in such a way that the initial points of all edges correspond to 0 and final points correspond to 1.*
2. *The points on the edges $a_1, a_2, \ldots$ that all correspond to the same value from the unit interval are treated as a single point.*
3. *The neighborhoods of the new topology on P are the disks entirely contained in a single polygon plus the unions of half-disks whose diameters are matching intervals around corresponding points on the edges $a_1, a_2, \ldots$.*

In other words, in the new topology the identified edges $a_1, a_2, \ldots$ are treated as a single edge.

In the preceding examples, identification of edges was represented by labeling identified edges with the same label; arrows were used to indicate the orientation of the edges.

Next we need to define the identification of vertices:

Definition 3.7 *Let P be a set of polygons, and let $p_1, p_2, \ldots$ be a collection of vertices from these polygons. These vertices are said to be* identified *when a new topology is defined on P in which this collection of vertices is treated as a single point and the neighborhoods are defined to be disks completely contained in a single polygon plus the unions of portions of disks around each of the points $p_1, p_2, \ldots$. In case any of the edges meeting at one of these vertices is also identified, the sectors forming a neighborhood at $p_1, p_2, \ldots$ must contain matching intervals from these edges.*

If a collection of vertices $p_1, p_2, \ldots$ are identified, we shall say the respective polygons $r_1, r_2, \ldots$ are identified at that collection.

The plane models can now be defined more rigorously:

Definition 3.8 *A* plane model *is a planar directed graph $\{N, A, R\}$ with a finite number of* vertices $N = \{n_1, n_2, \ldots\}$, edges $A = \{a_1, a_2, \ldots\}$, *and* polygons $R = \{r_1, r_2, \ldots\}$ *bounded by edges and vertices. Each polygon of the graph has a certain orientation around its edges and vertices. Polygons, edges, and vertices of the graph are labeled; if a collection of edges or vertices has the same label, they are considered to be identified according to definitions 3.6 and 3.7 above.*

Figure 3.7(a) illustrates the definition above. In the figure, edges and vertices with corresponding labels are identified. Hence, for instance, the neighborhoods of each point on the edge with the label e_1 consist of two half-disks, and the neighborhood of the vertex v_1 consists of four disk sectors.

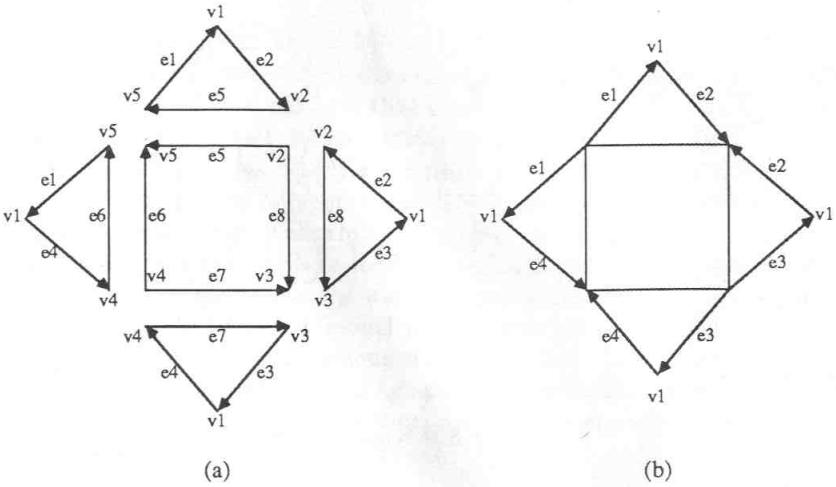


Figure 3.7 Plane models of a pyramid.

For simplicity, we shall often draw plane models in a condensed form such as in (b), where some identified edges are drawn just once, and their labels are not included. Similarly, we can draw a collection of identified vertices as a single point. Note that Figure 3.7(a) and (b) both represent the same structure $\{N, A, R\}$, and hence the same plane model. We shall also include the infinite "surrounding" polygon in a drawing of a plane model, when convenient (see, for instance, Figure 3.9 on page 44).

3.5.5 Realizable Plane Models

We can finally turn our attention to the necessary conditions for the realizability of a plane model.

Surface Subdivisions

Topological identification as defined above can produce neighborhoods that are not disks and do not, hence, satisfy the definition of a 2-manifold. To exclude these cases, we restrict the topological identification of plane models:

Definition 3.9 *A plane model is a* (surface) subdivision *if the following conditions are obeyed in the topological identification of its edges and vertices:*

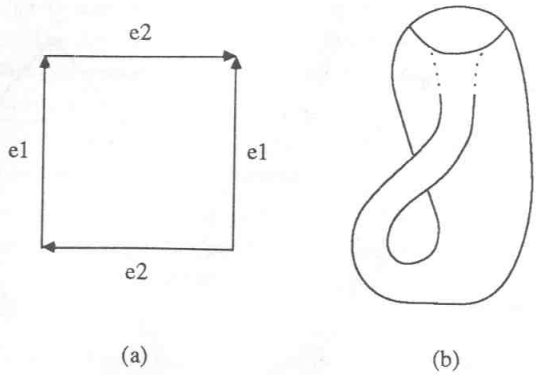


Figure 3.8 Klein bottle.

1. *Every edge is identified with exactly one other edge.*
2. *For each collection of identified vertices, the polygons identified at that collection can be arranged in a cycle such that each consecutive pair of polygons in the cycle is identified at an edge adjacent to a vertex from the collection.*

Observe that the edge $v2$–$v5$ in Figure 3.7(b) actually represents the pair of edges labeled $e5$ in (a); hence, $v2$–$v5$ satisfies the first condition. The condition disallows objects such as that depicted in Figure 3.4(b); in this case, four edges are identified.

The second condition guarantees that the combined neighborhood of each identified group of vertices is a disk; for instance, the six polygons identified in Figure 3.4(a) cannot be arranged in a single cycle.

Orientability

There are 2-manifolds that do not have physical counterparts in E^3, i.e., that cannot be constructed in three-dimensional space at all, and are hence not the boundary of any r-set. For instance, the *Klein bottle* represented by the plane model of Figure 3.8 is such a surface. These nonrealizable 2-manifolds can be distinguished from realizable ones by the concept of *orientability*:

Definition 3.10 *A plane model is* orientable *if the directions of its polygons can be chosen so that for each pair of identified edges, one edge occurs in its positive orientation in the direction chosen for its polygon, and the other one in its negative orientation.*

This condition is known as the *Möbius' rule.*

In the following, we shall be interested in orientable plane models only as they correspond with surfaces that can be realized in E^3, i.e., can form the boundary of an r-set. As seen from the definition above, that all polygons are *consistently oriented* (i.e., all polygons are oriented, say, clockwise except the surrounding polygon which is oriented counterclockwise) is sufficient to guarantee the realizability of a plane model.

Intuitively, realizable plane models can be drawn on an r-set without crossing edges. A more rigorous way to express this is to note that points and open sets of the plane model can be mapped on points and open sets of the r-sets with a continuous transformation. In practice, we prefer mappings that can be represented by assigning geometric information (such as coordinate values, curve equations, and surface equations) to vertices, edges, and polygons of the plane model.

3.5.6 The Euler Characteristic

Consider the cube depicted in Figure 3.9(a). The cube has a total of $f = 6$ faces, $e = 12$ edges, and $v = 8$ vertices (where identified edges and vertices count as one). Observe that the plane model includes the surrounding exterior polygon, and that all edges are identified.

Clearly, the same surface could be modeled in terms of the modified model (b), obtained by replacing one square face of (a) by two triangles. The new model has one additional face and edge, or a new total of $f' = 7$ faces and $e' = 13$ edges.

Both plane models represent the same surface—namely, a surface topologically equivalent to the sphere. A property of fundamental importance is that the plane models are directly able to tell us this, as shown by the following theorem:

Theorem 3.11 *Let the surface S be given as a plane model and let v, e, and f denote the numbers of vertices, edges, and faces in the plane model. Then the sum $v - e + f$ is a constant independent of the manner in which S is divided up to form the plane model. This constant is called the* Euler characteristic *of the surface and is denoted $\chi(S)$.*

For our purposes, this *invariance theorem* is one of the central results of algebraic topology. Its formal proof would bring us too far from the scope of this book; see e.g. [52]. To get some reliance on it, it is easy to verify for the original cube that

$$\chi = v - e + f = 8 - 12 + 6 = 2.$$

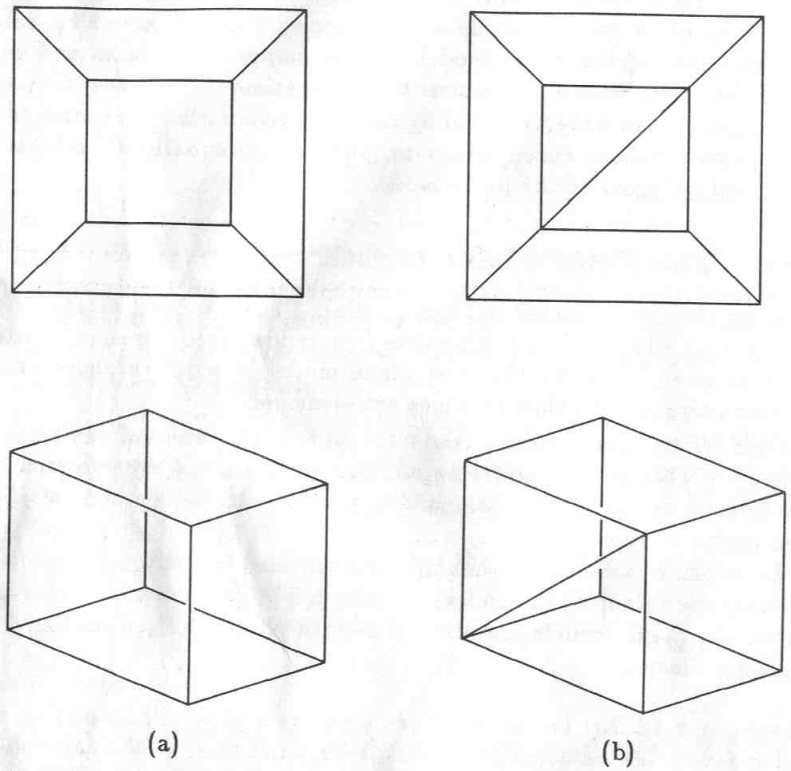


Figure 3.9 Models of a cube.

After the modification we have

$$\chi' = v - e' + f' = 8 - 13 + 7 = 2,$$

as expected. No matter how we alter the plane model to yield a new plane model, the Euler characteristic remains the same.

Betti Numbers

The theory of *homology* tells us that the Euler characteristic can be expressed as

$$\chi = h_0 - h_1 + h_2 \tag{3.1}$$

where h_0, h_1, and h_2 are called the *Betti numbers* of the plane model. Formula 3.1 is called the *Euler-Poincaré formula.*

The Betti numbers can be calculated through group-theoretic techniques from the plane model; however, this is beyond the scope of our interest. Instead, let us take a look at the topological significance of the Betti numbers. The second Betti number h_2 reveals the orientability of the surface, and equals h_0 for the kinds of surfaces we are interested in.

An arbitrary surface is always the union of a number of connected pieces called *components.* The zeroth Betti number equals the number of components. Hence, for the cube $h_0 = 1$, while for a set of six tennis balls $h_0 = 6$.

The first Betti number h_1 is called the *connectivity number* of the surface. It tells the largest number of closed curves that can be drawn on the surface without dividing it into two or more separate pieces. For the cube (or any surface topologically equivalent to it) $h_1 = 0$, because every closed curve would cut a part of the surface away. More intuitively, h_1 equals twice the number of "holes" in the object. A doughnut has one hole, and its $h_1 = 2$. Hence the Euler characteristic of a doughnut (or a coffee cup) is

$$\chi = 1 - 2 + 1 = 0.$$

The invariance theorem can be generalized for the Betti numbers as well. That is, no matter how a surface is divided up to make a plane model, the Betti numbers (and all topological characteristics conveyed by them) will remain invariant. This fact will be of interest to us in Chapter 9; for reference, let us state it as a theorem:

Theorem 3.12 *Let the surface S be given as a plane model. Then the Betti numbers h_0, h_1, and h_2 of the plane model are constants independent of the manner in which S is divided up to form the plane model.*

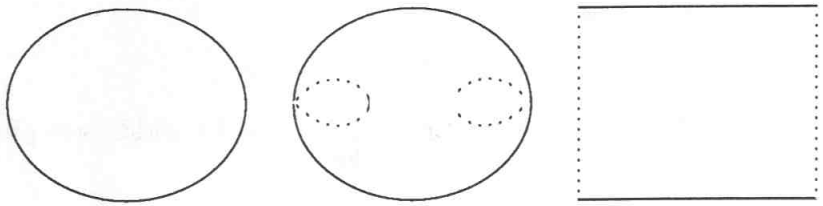


Figure 3.10 The making of a cylinder.

In the following, we shall use a more mnemonic notation for the Euler-Poincaré formula 3.1. We shall write s, the number of connected surfaces or "shells", to denote h_0. Similarly, h, the *genus* of the surface (or the sum of the genera of all components of a multicomponent model), is used to denote the number of holes and is defined $h_1/2$. With this notation, the Euler-Poincaré formula may be written in more familiar terms as

$$v - e + f = 2(s - h). \tag{3.2}$$

3.5.7 Surfaces With Boundary

As seen in previous sections, orientable 2-manifolds coincide with the informal concept of "closed physical surfaces." Sometimes, however, we would like to handle also surfaces that do not necessarily bound a solid.

One such case, the cylinder surface, was already depicted in Figure 3.5. Its plane model was characterized by some edges that were not identified to some other edge. These edges form the boundaries of the surface; hence the term *surfaces with boundary*.

Surfaces with boundary can be thought of as created from their closed counterparts by cutting some polygons away. For instance, a cylinder is created by removing two disks from a sphere, as depicted in Figure 3.10. In the figure, we start from a plane model of a sphere (a). The plane model is modified by removing two polygons (b) which leads to the plane model of the cylinder (c).

Using this mental tool, the theory of closed surfaces can be carried over to surfaces with boundary easily. For instance, to calculate the Euler characteristic of a surface with b boundary components, we proceed by adding b new polygons to its plane model that "mend" it to form the corresponding

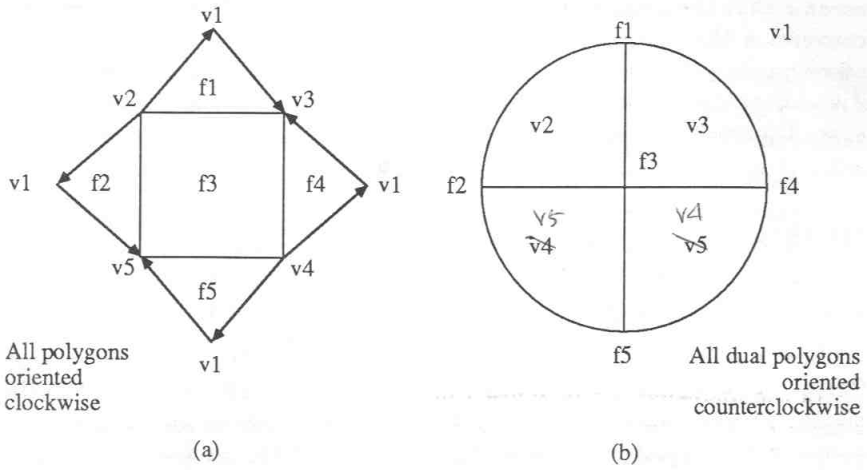


Figure 3.11 A plane model and its dual.

closed surface. Clearly, the Euler characteristic of the mended surface is

$$\chi = v - e + f + b.$$

Hence for a surface with b boundary components, the Euler-Poincaré formula 3.2 can be written as

$$v - e + f = 2(s - h) - b. \tag{3.3}$$

3.5.8 Duality

The *dual* of a planar graph is constructed by assigning a dual vertex to each polygon of the original graph, and joining a pair of dual vertices with a dual edge, if the corresponding original polygons share an edge (possibly through identification). In this fashion, dual polygons correspond with original vertices.

Using this procedure, we can convert a plane model of a surface to its dual plane model. An example of a plane model and its dual is given in Figure 3.11. Observe that the dual model (b) includes the infinite "surrounding" polygon v_1.

Note that polygons and edges of the dual plane model are oriented. To explain how the orientation of the dual plane model is chosen, let us

assume that the original model is consistently oriented. First, we use the convention that each dual edge is oriented towards the dual vertex that corresponds with the original polygon in which the edge occurs in its positive orientation. This determines uniquely the orientation of every dual edge. Directions of dual polygons are chosen by the following rule:

> *Let v, e, v^*, e^* denote an original vertex, an original edge adjacent to v, the dual polygon corresponding with v, and the dual of e. If v is the final point of e, the orientation of v^* is chosen so that e^* is traversed in its positive orientation, otherwise the opposite orientation is chosen.*

Under this rule, the orientation of dual polygons will be consistent. Furthermore, the dual of a dual plane model will be identical with the original model. We actually define the cyclical ordering of the edges around a vertex to be equal to the ordering of the corresponding dual edges in the dual polygon of the vertex. The dual is a surface subdivision because all its edges are identified with exactly one other edge.

We are interested on duality primarily because of the following:

Lemma 3.13 *All topological properties of a plane model are preserved in its dual.*

In particular, the Betti numbers and the Euler characteristic χ of a plane model and its dual are equal.

Duals of plane models will be of use when arguing about the manipulation of plane models in Section 9.1.1.

3.5.9 Wrap-Up of Plane Models

We may now conclude the somewhat lengthy development as follows:

> The surface of a solid can be rigorously modeled in terms of a planar graph with a special topology. Based on the graph, we can decude the overall topological properties of the surface, such as its orientability, connectivity, and the number of "holes" through it. Such properties are compactly expressed in the form of the Euler-Poincaré formula.

Of course, for the purposes of solid modeling, we shall be primarily interested in realizable plane models—that is, plane models that can be drawn on the boundary of an r-set. As we shall see later, the theory of plane models will allow the derivation of manipulation operations of plane models that always will yield realizable plane models while being general enough to model all plane models of interest.

3.6 REPRESENTATION SCHEMES

Having now tackled the proper definition of a mathematical modeling space for solid modeling, we can now consider the problems of assigning computerized representations of the objects of the modeling space. As mentioned in Section 3.2, the three-level view of modeling allows us to precisely define the relationship of objects and their representations.

Let M denote a mathematical modeling space of objects. In this section, let "solid" denote an entity of M.

Definition 3.14 *A* solid representation *is a finite collection of symbols (of a finite alphabet) that designates a solid of M.*

Definition 3.15 *The representation techniques of a given solid modeler define the* representation space R *of the modeler. Those representations that actually can be constructed by the solid modeler according to its syntax rules are termed* admissible. *In this section, we shall call entities of R "representations."*

Observe that any solid representation $r \in R$ is meaningful only when related to an entity $m \in M$. Hence the significance of solid representations is expressed by a mapping from R to M:

Definition 3.16 *A* representation scheme *is a relation $s : M \rightarrow R$. The domain of s (i.e., those $s \in M$ that have an image under s) is denoted by D and the image of D under s by V. The inverse relation of s is denoted by s^{-1}. That $r \in R$ is the image of $m \in M$ under s will be denoted by $\{m, r\} \in s$.*

The validity of a representation can now be expressed more rigorously:

Definition 3.17 *Any representation $r \in V$ is termed* valid.

Note that each valid representation is hence the image of at least one entity of M. However, we neither assume that all solids of M are representable (i.e., D need not equal M) nor that all representations of R are valid (i.e., V need not equal R). See Figure 3.12 for an illustration of this.

With the concepts developed so far, we can characterize ambiguity as a property of a representation scheme:

Definition 3.18 *If any valid representation models exactly one solid under s (i.e., it is the image of exactly one solid of M), s is called* unambiguous *or* informationally complete. *In other words, this property requires that for each valid representation $r \in V$, it holds that*

$$\{m, r\} \in s \wedge \{m', r\} \in s \Rightarrow m = m'.$$

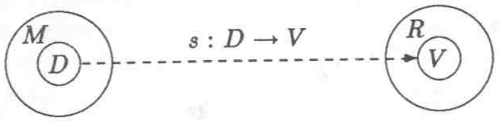


Figure 3.12 Definition of a representation scheme.

Unambiguity assures that a representation always designates a unique solid. Hence unambiguous representations in principle contain complete information of the solid. Note that a solid may still have many representations under an unambiguous representation scheme. A symmetric variation of unambiguity requires that this may not happen:

Definition 3.19 *A representation scheme s is termed* unique *if all solids $m \in D$ have exactly one representation, i.e., if*

$$\{m, r\} \in s \wedge \{m, r'\} \in s \Rightarrow r = r'.$$

If interpreted literally, uniqueness requires that a solid has just a single representation independent of its orientation. As this is unfortunately rarely the case, we shall also use a weaker form of uniqueness as defined below:

Definition 3.20 *A representation scheme s is* weakly unique *if every oriented instance of every solid $m \in D$ has only one representation.*

In the following, the term "uniqueness" will refer to weak uniqueness, unless explicitly mentioned otherwise.

3.6.1 Properties of Representation Schemes

Alternative major approaches to solid modeling differ in the modeling spaces, representation spaces, and representation schemes they can support. In addition, the definition of representation schemes allows us to discuss their desired properties on a fairly general level. Some properties were already described in the preceding section. For clarity, the following list rephrases those properties as well.

1. *Expressive power:* What objects are included in the domain D covered by the representation scheme? Is it possible to extend the domain? One aspect of the expressive power is the *precision* of the representation scheme: how accurately can complicated objects be modeled?

2. *Validity:* Are all admissible representations valid, i.e., do they designate some solids of M? A scheme with this desirable property is termed *syntactically valid* as it enforces validity of all solid descriptions that obey its syntax rules.

3. *Unambiguity and uniqueness:* Do all valid representations model exactly one solid? Do some solids have more than one valid representation?

4. *Description languages:* What kinds of solid description languages can be supported in a modeling system based on the representation scheme? Are they *self-contained*, i.e., directly based on the representations of the scheme, or are they based on a *conversion* from other representations?

5. *Conciseness:* How large (in terms of computer storage) do representations of practically interesting solids become? (This property is often in contradiction with the precision of the representation.)

6. *Closure of operations:* Do solid description and manipulation operations preserve the validity of solid representations? How general are the manipulations that can be supported?

7. *Computational ease and applicability:* What kinds of algorithms can be written for the representations of the scheme for, say, computations of Table 1.1? What kinds of computational complexities are involved? What kinds of applications are the representations of the scheme suited for? Of course, we are particularly interested on "self-contained" solutions that do not involve a conversion into some entirely different representation.

Some of these properties have a formal meaning in terms of the theory of representation schemes, while others have a practical flavor which is important in implementing a particular modeling system.

3.7 PRIMITIVE INSTANCING

Let us now clarify the theory of the preceding section by looking at a sample scheme for representing solid objects. This scheme will be of interest also in later chapters.

3.7.1 Concepts of Primitive Instancing

The simplest way of describing geometric objects is the *Primitive Instancing* scheme. In this approach, we restrict ourselves to modeling objects belonging to one of predefined primitive object types. For instance, we might model mechanical components by generating a sufficiently large collection of "library" parts. Models are simply instances of them, created with an instantiation operation. The instantiation requires a few descriptive parameters (such as, say, component type and size) that together form the model. Algorithms for this scheme operate by examining a tuple of descriptive parameters, and performing operations tailored for each component type.

Consider, for instance, an instancing scheme consisting of the single primitive type "t-brick." Its descriptive parameters would consist of an object type code, and five numerical values l, h_1, h_2, w_1, and w_2 indicating the dimensions of the object as depicted in Figure 3.13. Additionally, the orientation of the brick would be specified, say, in terms of a rigid transformation.

The domain D of this instancing scheme is the set of all bricks, and the representation space R consists of all 6-tuples (records) of the form

$$(tbrick, l, h_1, h_2, w_1, w_2).$$

To be a valid representation, i.e., to designate a physically meaningful object, the parameters should satisfy the following inequalities:

$$\begin{aligned} 0 &< l \\ 0 &< h_1 \\ 0 &< h_2 \leq h_1 \\ 0 &< w_1 \\ 0 &< w_2 \leq w_1. \end{aligned} \tag{3.4}$$

Obviously, primitive instancing schemes are unambiguous: a valid tuple fully determines a brick. Their uniqueness is somewhat more problematic, however. Too see why, let us introduce another primitive into our sample instancing scheme, the primitive type "block," described with a type code and numerical values l, h, and w (Figure 3.14). Observe that two different

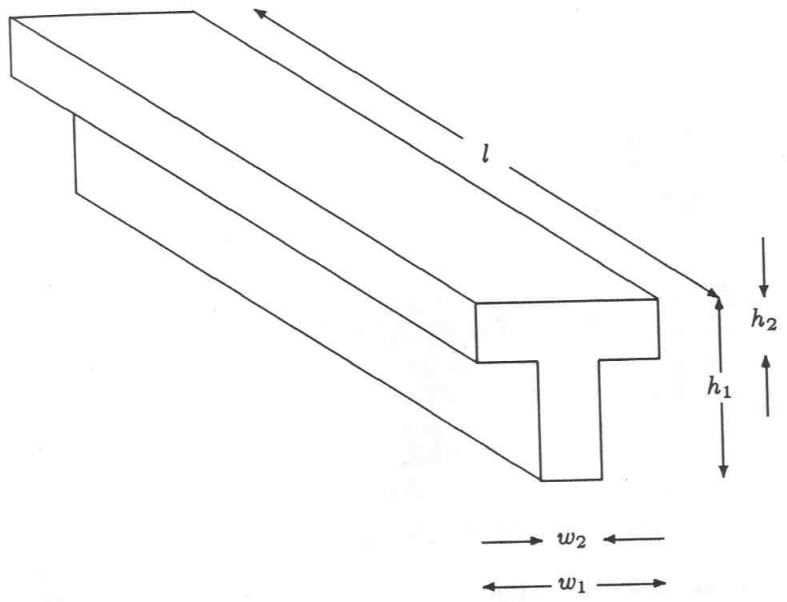


Figure 3.13 T-brick.

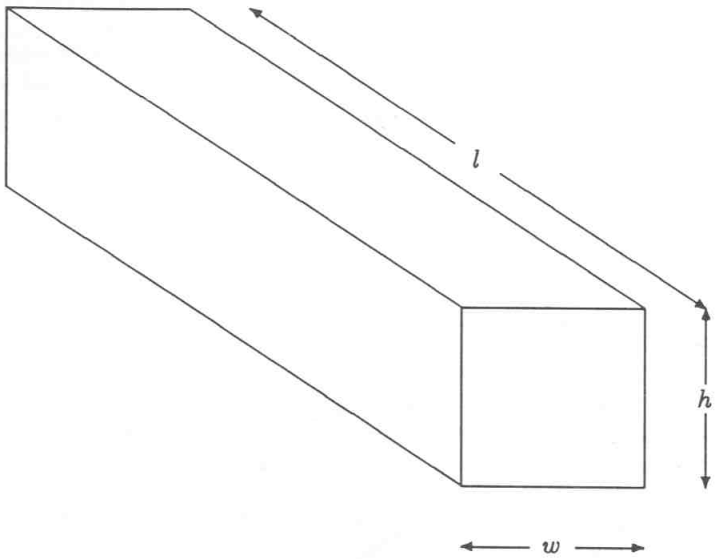


Figure 3.14 Block.

tuples

$$(tbrick, l, h, h, w, w)$$

and

$$(block, l, h, w)$$

describe the same object (Equations 3.4 allow $w_1 = w_2$). We may conclude that primitive instancing schemes are not necessarily unique.

3.7.2 Properties of Primitive Instancing

Let us discuss the properties of the primitive instancing scheme according to the framework given in Section 3.6.1.

Expressive power: Clearly, the modeling space of a primitive instancing scheme is determined completely by (1) the collection of primitive object types and (2) the nature of the instantiation operation. Obviously, it is painful to incorporate a large modeling space in this approach. A more involved problem is that it is not easy to find natural parameterizations for certain parts.

Validity: It is easy to build validity enforcement checks into the instancing operation. For instance, in the example above this amounts to checking whether the Equations 3.4 are satisfied.

Unambiguity and uniqueness: Instances always determine uniquely an object; hence they are unambiguous. As the example shows, they are not unique.

Description languages: The essential task describing an object in the primitive instancing scheme is the *selection* of the proper solid type from a fixed collection. The selection can be performed in many ways, including a menu driven, graphical operation. These procedures are obviously self-contained.

Conciseness: Properly structured primitive instancing schemes are concise.

Closure of operations: Primitive instancing does not support object modification operations in the general sense: the only possibility is the modification of instantiation parameters. As this operation can be subjected to similar checks as the original instantiation, the operations of the scheme are closed, i.e., yield valid models.

Computational ease and applicability: Algorithms for the primitive instancing scheme take the form of a (huge) case analysis, that finally breaks into tailored code for each object type. In the example, the volume of an object would be calculated by examining the type code and by performing a different routine for each object type. The resulting computational power can be good, because each case can be separately optimized.

As we shall see in the following chapters, primitive instancing is often used as an *auxiliary* scheme in modelers based on other representations in order to ease the description of often-needed parts. In this role it is indispensable, for instance, in problem areas such as design of electrical instrumentation or piping.

3.8 A TAXONOMY OF SOLID MODELS

As set forth in the preceeding sections, we may view solid objects as point sets of the Euclidean three-dimensional space satisfying restrictions that catch our idea of "solidity." To such sets, solid modeling aims to assign finite representations suitable for generating data for algorithms.

Primitive instancing can be viewed as a method of describing certain point sets indirectly through a tuple of their attributes. Unfortunately, as we saw in the previous section, either the modeling space of the modeler must be severely limited or a excessively large collection of primitives must be supported to make this approach generally applicable.

Therefore, we need a representation that encodes the infinite point set in a finite amount of computer storage in a more generic fashion. Representations achieving this may be divided in three large classes as follows:

1. *Decomposition models*, that represent a point set as a collection of simple objects from a fixed collection of primitive object types, combined with a single "gluing" operation. (Primitive instancing can be considered a special case of this approach.)

2. *Constructive models*, that represent a point set as a combination of primitive point sets. Each of the primitives is represented as an instance of a *primitive solid type*. Constructive models include more general construction operations than mere gluing. As we shall see, their theory is based on the point-set formulation of solidity presented in Section 3.4.

3. *Boundary models*, that represent a point set in terms of its boundary. The boundary of a three-dimensional "solid" point set is a two-dimensional surface that is usually represented as the collection of *faces*. Faces, again, are often represented in terms of their boundary being a one-dimensional curve. Hence boundary models may be viewed as a hierarchy of models. Boundary models are based on the surface-oriented formulation of solid objects presented in Section 3.5.

These major approaches to solid modeling will be described in the following three chapters. We shall also be interested on *multirepresentational* and *hybrid* modelers that employ several representations simultaneously. These models shall be dealt with in Chapter 7.

PROBLEMS

3.1. The footnote on page 32 points out that sometimes we are not modeling physical reality, but an idealized world. Can you think of other examples of this besides engineering design?

3.2. Discuss the physical meaningfulness of nonmanifold objects such as those of Figure 3.4 on page 37. Are some more "meaningful" than others? Can you think of criteria that would allow us to make a distinction between "meaningful" and "nonmeaningful" nonmanifold objects?

3.3. The nonuniqueness of primitive instancing schemes can be demonstrated without introducing the "block" primitive in addition to the "t-brick." Give two different "t-brick" 6-tuples that correspond with the same object.

BIBLIOGRAPHIC NOTES

Much of the material of this chapter is based on the works of Aristides A.G. Requicha and his colleagues with the PADL-project at the University of Rochester. The technical report series of the PADL-project is strongly suggested reading to all persons interested on the fundamentals of solid modeling.

The three-level modeling approach is originally presented in report [94] that also discusses in detail point-set and manifold models. The report goes into much more detail as for the group-theoretic techniques for dealing with plane models, although Requicha uses a different notation.

Report [97] can be considered a basic reference to point-set models and constructive models which are the subject matter of Chapter 5.

The article [95] draws some of the material of the technical reports cited above in a more streamlined form, and presents the theory of representation schemes. The list of desired properties of solid presentations of Section 3.6.1 is based on the list presented in this article.

Plane models for rigorous discussion on solid models were introduced in articles by Hanrahan [50] (briefly) and this author [75] (extensively). In their article [49], Guibas and Stolfi present another variation of plane models, and modeling primitives that can handle even nonorientable objects such as the Klein bottle.

Texts on topology and particularly in algebraic topology are usually rather heavy for a non-mathematician reader. The (in this chapter often-cited) book of Henle [52] is an exception of this rule. Other standard texts include [3,1,30]. A standard reference in graph theory is [51].

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

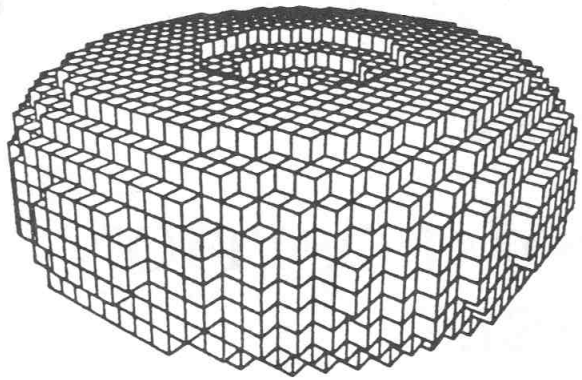


Figure 4.1 Exhaustive enumeration.

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.

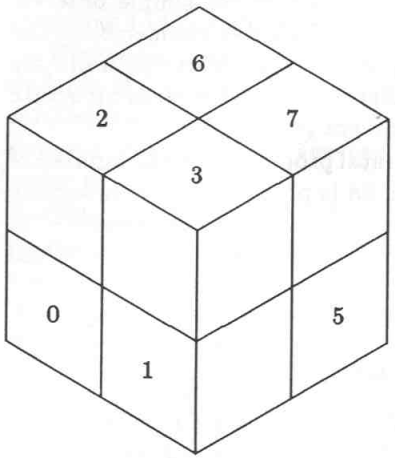
The octree representation uses a recursive subdivision of the space of interest into eight *octants* that are arranged into an 8-ary tree (hence the name). Figure 4.2(a) depicts the octant subdivision.

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.

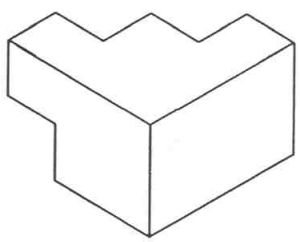
Each node of an octree consists of a *code* and eight pointers towards eight sons, numbered 0 through 7. If *code* = *black*, the part of the space represented by the node is all material and the pointers 0 through 7 are nil, i.e., the node is a leaf. If *code* = *white*, the part of the space is empty 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.

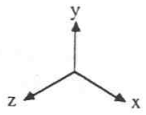
It is easy to show that 7/8 of the nodes of an octree are leaves. Because of this, also leaves are usually represented with a special node to avoid allocating storage for the eight nil pointers.

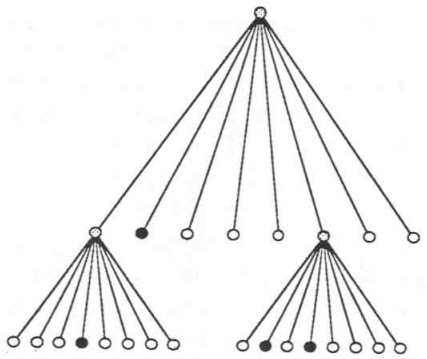


(a)



(b)





(c)

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.

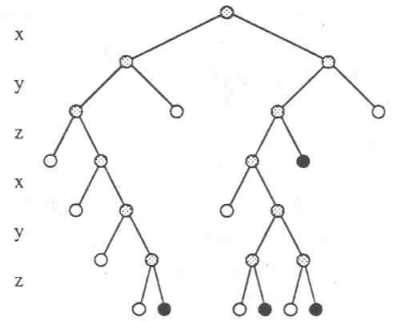


Figure 4.3 Binary subdivision tree.

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].

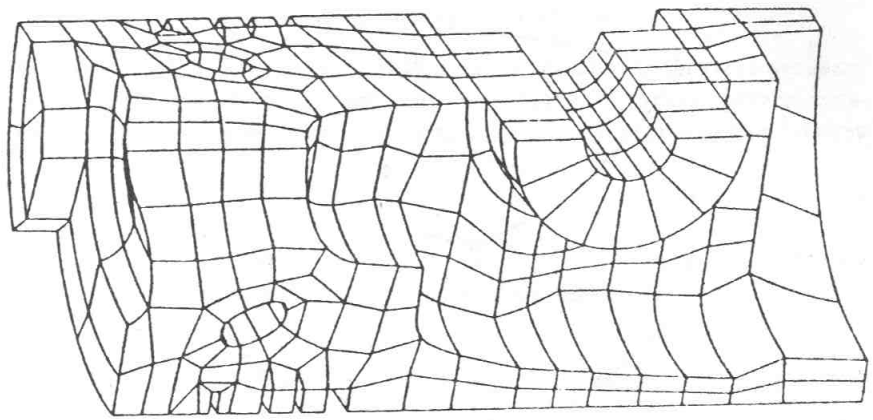


Figure 4.4 A cell decomposition.

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.

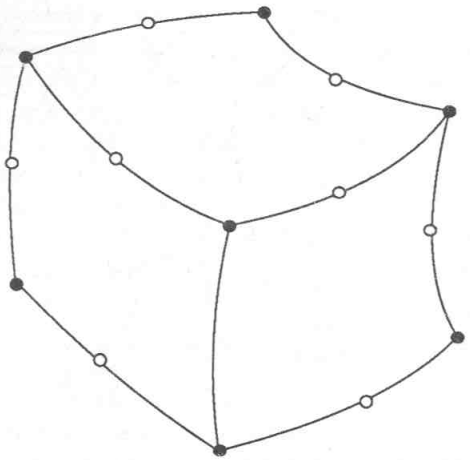


Figure 4.5 A quadratic cell.

Validity: The validity of a cell decomposition is hard to establish. While exhaustive enumeration and octree representation have structural properties that assure validity, a general cell decomposition is just an unordered set of cells. To check its validity, an intersection test for *every* pair of cells is necessary.

Unambiguity and uniqueness: A valid cell decomposition completely determines a solid. Cell decompositions are not unique.

Description languages: In general, it is very hard to create cell decompositions of interesting objects by direct mechanisms. Usually cell decompositions are created by conversion from other, more convenient representations.

Conciseness: Cell decompositions are relatively concise. The usual representation consists of a list of *nodes*, each being a named three-dimensional point, followed by a list of *elements*, each having a type code and a list of point numbers, whose interpretation is determined by the type code.

Closure of operations: Usually cell decompositions are used only as input to analysis algorithms that passively examine the model without modifying it. Nontrivial closed algorithms for cell decompositions (such as Boolean set operations) are either unknown or computationally unattractive.

Computational ease and applicability: Cell decompositions are indispensable as the input representation for computational algorithms. Finite element models (FEM) give a prime example of this.

The role of cell decompositions is clear from the above: they are auxiliary representations used for specific computations rather than as general independent solid models in their own right. We shall return to this later in Section 5.2.

PROBLEMS

4.1. Show that 7/8 of the nodes of an octree are leaves.

Hint: Use induction on the depth of the tree.

4.2. How would you classify an octree node against straight cylinder primitive?

4.3. Describe an algorithm for rotating an octree by a integer multiple of 90 degrees.

4.4. Describe algorithms for calculating (a) the set union and (b) the set difference of two octrees.

4.5. Describe an algorithm that can calculate Boolean set operations of two linear quadtrees represented by means of the DF-representation.

4.6. Design a data structure that can be used to represent a cell decomposition consisting of quadratic cells such as the one shown in Figure 4.5.

BIBLIOGRAPHIC NOTES

Octrees seem to have been invented independently by several workers. The basic works are by Hunter [59], Jackings and Tanimoto [60], Meagher [83,82], and Reddy and Rubin [93].

The major source of information on quadtrees and related data structures is the comprehensive survey [106] of Samet that also includes a large bibliography.

The survey of Shirari [112] offers lots of information on various representations of three-dimensional digital images primarirly from the point of view of image processing applications.

While accurate octree models of curved solids would be very large, it seems possible to use octrees for cases where somewhat rough models are appropriate. In addition to graphics, Yamaguchi *et al.* [135] describe the use of octree models for rough cutting of mechanical parts.

Cell decompositions are the primary data representation for the Finite Element Method. Standard references to FEM are [12,54].

FEM systems typically include a preprocessor that aids the generation of proper FEM meshes. The more powerful preprocessors are actually solid modelers of their own right that can be useful even outside the FEM context proper; for instance, see [23].

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 "$\cap$":

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

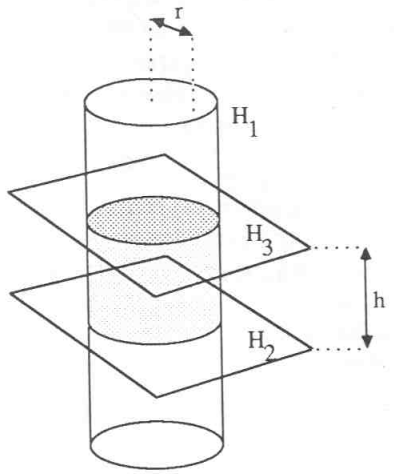


Figure 5.1 Half-space model of a finite cylinder.

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

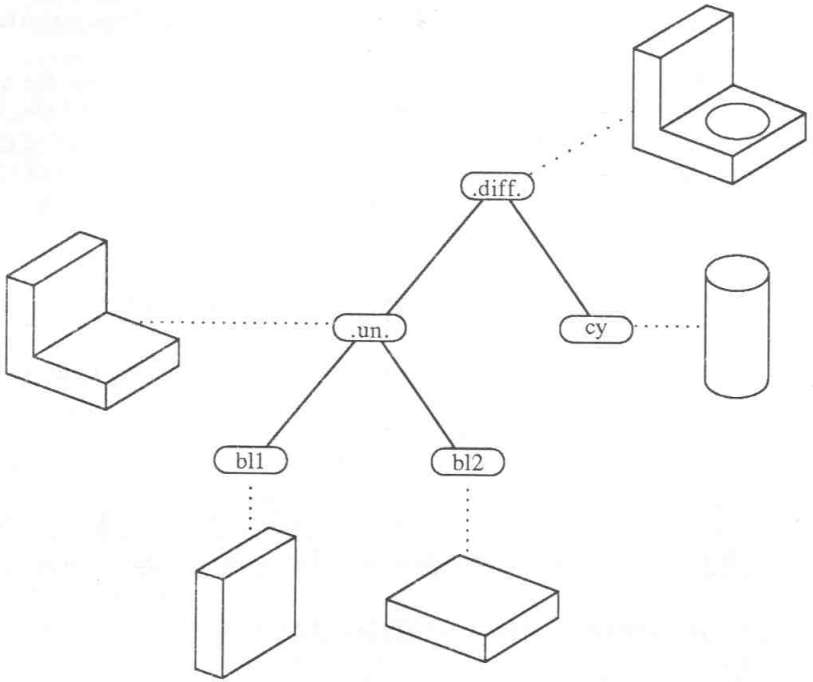


Figure 5.2 A CSG tree.

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.

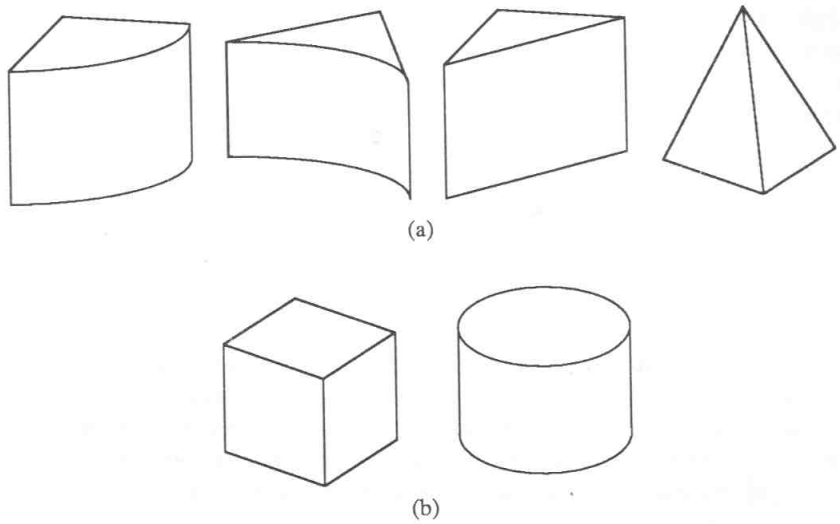


Figure 5.3 Two collections of CSG primitives.

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,

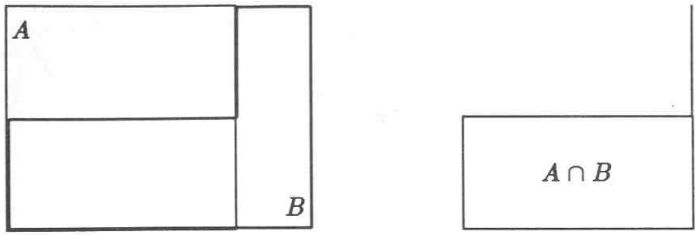


Figure 5.4 A nonregular set operation.

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

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

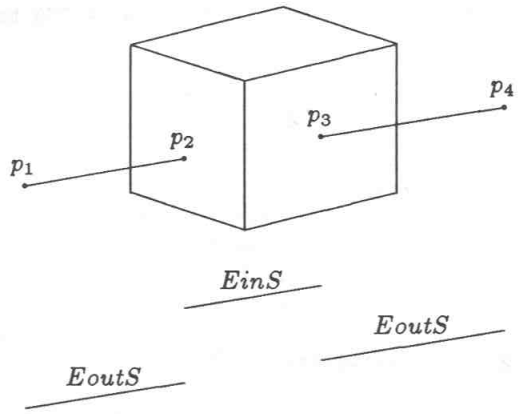


Figure 5.5 Edge vs. block primitive.

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

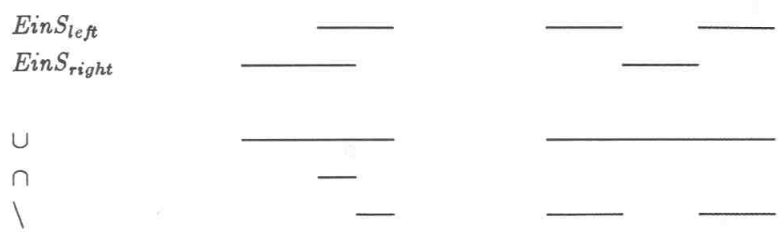


Figure 5.6 Combination rules.

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.

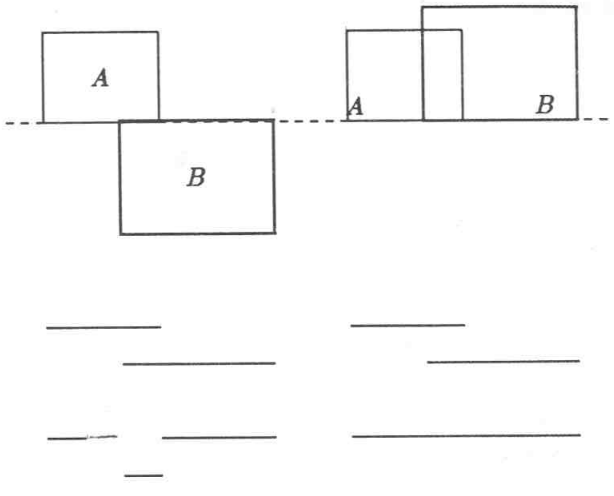


Figure 5.7 Combination rules for "on-on"-cases.

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.

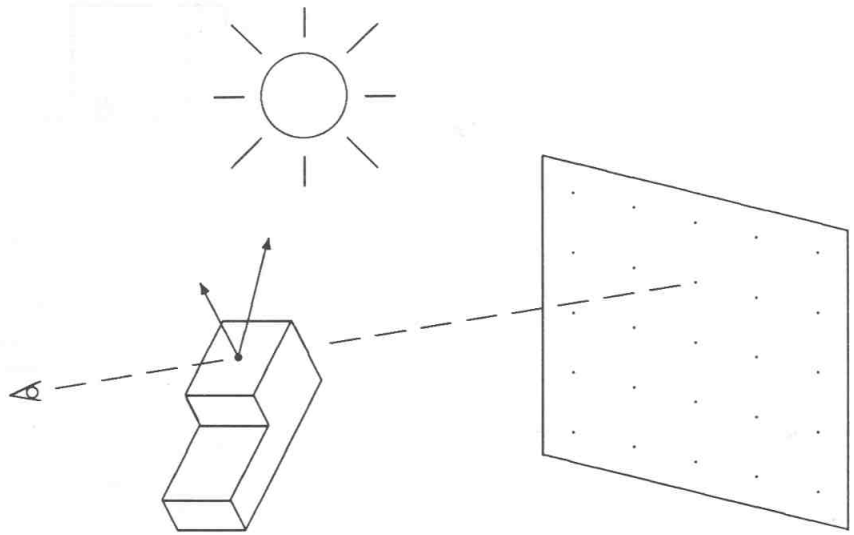


Figure 5.8 The ray casting paradigm.

Ray Casting

Another general approach to algorithm design for CSG models is the *ray casting* approach [104]. Ray casting is by its very nature approximative: instead of "evaluating" a CSG tree exactly, the tree is "sampled" along a finite number of lines, hence reducing the calculation of three-dimensional set operations to that of one-dimensional ones.

The term arises from the problem of generating an image of a CSG model for a color raster scan terminal (see Figure 5.8). In the ray casting paradigm, a "view ray" is sent from the location of the viewer through each pixel of the image. Based on the location where the ray first hits the object viewed, the shade and the intensity of the pixel are calculated. By sending secondary rays from the hit location, special visual effects such as shadowing, transparency, and mirroring can be created. This process is termed *ray tracing*.

The ray casting approach is actually a variation of the edge-solid classification, where instead of a finite edge, a semi-infinite ray is classified by a *ray classification* procedure. As a two-way "in-out" classification is sufficient, and as it can be assumed that the location of the "eye" is outside the solid, the ray classification procedure can represent the results by

means of a sequence of intersection points between the ray and the solid. An adequate representation is to have two arrays

$$\begin{array}{cccccc} t(1) & t(2) & t(3) & t(4) & \ldots \\ s(1) & s(2) & s(3) & s(4) & \ldots \end{array}$$

where $t(i)$'s give the parameter values of intersection points in a sorted order, and $s(i)$'s identify the half-spaces intersected at these points. The ray segments $[0, t(1)]$, $[t(2), t(3)]$, and so on are considered to occur outside the solid, and segments $[t(1), t(2)]$, $[t(3), t(4)]$ inside of it.

The implementation of the ray-classification procedure follows similar lines as the edge-solid classification, being only simpler. For clarity, the outline of the algorithm given in Program 5.4 indicates the intersection tests needed between the ray and the half-spaces of the primitives.

The image generation algorithm can now simply pick the first intersection of the sequence, and calculate the color based on the information it has on normal vector and color of the surface intersected and the locations of light sources.

Ray casting is useful not only for color image generation, but also for calculating line drawings and integral properties of solids; see, e.g., [44]. The problem is that the amount of computation needed is large: to calculate an image at 1000^2 resolution, one million ray classifications are needed. As each classification is a recursive algorithm potentially involving numerical solutions of fourth order polynomials, the speed of ray casting algorithms is very far from interactive.

Integral Properties

The approximative evaluation of integral properties (such as volume) of CSG models is ordinarily based on conversion algorithms that (implicitly) construct a decomposition model approximating the solid. The evaluation of the property then reduces to calculating the contribution of each cell of the decomposition to the total result.

The conversion algorithms are generally based on the set membership classification and ray casting approaches. Several alternative arrangements of cells are possible, each corresponding to a particular set membership classification algorithm. Various alternatives are shown in Figure 5.9 [98,69,68].

For instance, the case of "column decomposition" corresponds with ordinary ray casting. A column is determined by sending a ray along its center; where the ray is inside the solid, a solid column is created. This algorithm is used in solid modelers employing the ray casting approach, e.g., in the SYNTHAVISION^{TM} solid modeler [45].

```
Classification
RayCast(S, R)
CSGTree *S;
Ray *R;
{
    if(S->Op == <set operation>)
    {
        LeftClassification = RayCast(S->Left, R);
        RightClassification = RayCast(S->Right, R);
        return(Combine(LeftClassification,
                       RightClassification,
                       S->Op);
    }
    else
    {
        switch(S->Op)
        {
            case "block":
                do 6 ray-plane intersection tests
            case "sphere":
                do 1 ray-quadratic intersection test
            case "cylinder":
                do 2 ray-plane and 1 ray-quadratic tests
            case "cone":
                do 1 ray-plane and 1 ray-quadratic test
            case "torus":
                do 1 ray-quartic intersection test
        }
        Classification = results of tests;
        return(Classification);
    }
}
```

Program 5.4 Ray classification algorithm.

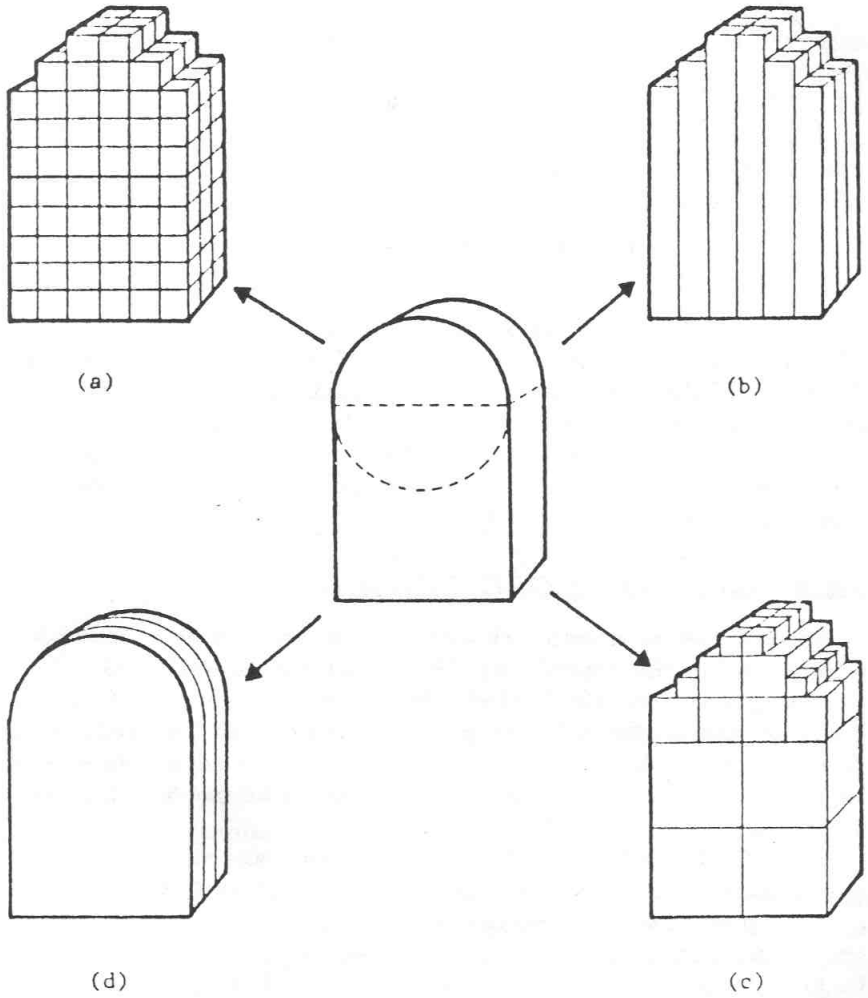


Figure 5.9 CSG-to-cell conversions.

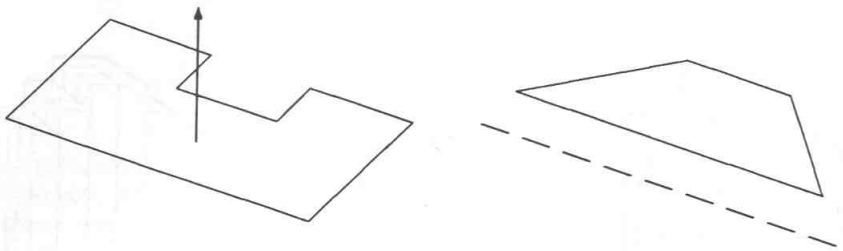


Figure 5.10 Sweeping primitives.

The simplest case, exhaustive enumeration, is based on *point-solid classification*: given a point P (the center point of the cell), determine whether P is inside, on the boundary, or outside the solid. CSG is directly useful for answering this kind of a question as classification with respect to primitives amounts to substituting P to the defining function of each half-space. For the two-way classification, the combination rules are simple and left to the reader. This method is used in TIPS.

5.2.3 Generalized CSG Primitives

To apply the set membership classification and ray casting approaches to CSG algorithms for a particular CSG model, the essential task reduces to writing procedures that perform the underlying operations of these algorithms. Correspondingly, *any* primitive for which such procedures can be written can potentially be included into a CSG modeler. In particular, there is no need to restrict CSG primitives to be simple collections of half-spaces.

Many CSG-based systems follow this approach, and allow the inclusion of *generalized CSG primitives* of various types into their CSG models. For instance, *polyhedral primitives* can be added into a ray casting modeler by offering sufficient facilities for their creation, and a ray casting algorithm for arbitrary polyhedra. This approach leads to a hybrid of the CSG and boundary modeling approaches.

Another potentially useful addition is the inclusion of *sweeping primitives*, modeling primitives of the *sweeping model* approach to solid modeling. A *translational sweeping primitive* is defined by a *base set*, a bounded subset of a plane, and a *sweeping direction*; informally, the primitive consists of the set of points "swept" by the base set as it moves along the sweeping direction. A *rotational sweeping primitive* is similarly defined by a base set and a rotation axis, and consists of the set of points swept by the

base set as it rotates around the axis. Of course, restrictions on admissible base sets must be posed so as to guarantee that the resulting objects are valid solids.

In this context, the essential fact is that there are algorithms, for example, for the ray casting of translational and rotational sweeping primitives directly (without conversion to some other representation); see [62,125,124]. Hence they can be included into the domain of a ray casting modeler. Moreover, "two-and-half-dimensional" models generated by sweeping simple outlines consisting of straight lines and arcs along a normal axis can be converted directly to CSG models; see [128].

5.2.4 Properties of CSG Models

The properties of CSG models can be summarized similarly as those of other models we have discussed.

Expressive power: Depends on the class of half-spaces available. Cannot easily be extended to cover surface patches.

Validity: Every CSG tree is guaranteed to model a valid solid object, provided that the primitives are valid (i.e., bounded regular sets).

Unambiguity and uniqueness: Every CSG tree unambiguously models a solid. They are not unique.

Description languages: Usually textual languages. It is possible to include a graphical interface into a CSG modeler.

Conciseness: CSG trees are in principle relatively concise. In practical modelers, they tend to grow as other information aiming at efficient graphical operations is attached to the basic CSG tree.

Closure of operations: Set operations are algebraically closed for CSG trees.

Computational ease and applicability: The computational power some important CSG algorithms (such as boundary evaluation) is poor. However, as their basic steps are very simple, they are prime candidates for VLSI implementation. In cases such as the point classification for conversion to a decomposition model, the divide and conquer approach generally leads to efficient algorithms, although their efficiency deteriorates if the CSG tree is ill-balanced.

CSG models are based on rigorous mathematical background, and the families of algorithms available for them are well understood and applicable to many problems. Because of these reasons, many commercial CSG modelers are available and in wide use.

PROBLEMS

5.1. Give a Boolean expression of half-spaces for each of the CSG primitives of Figure 5.3.

5.2. Specify and write a ray-sphere intersection procedure based on the general outline of Program 5.4. In addition to the ray parameter value of the intersection point, the procedure should also calculate the surface normal of the sphere at the intersection point for use of the ray tracing program of Problem 5.3.

5.3. Based on the ray-sphere intersection procedure of Problem 5.2, specify and write a ray tracing program that can display images (possibly intersecting) spheres on a color raster display.

Hint: You will need to apply a suitable shading method based on the surface normal of the intersection of the ray with a sphere and the location(s) of the light source(s). See, e.g., [39,88] for additional information.

5.4. Generalize your ray tracing program of Problem 5.3 by including additional primitive types, such as blocks, cylinders, and cones. Try not be become too hooked on pretty pictures!

BIBLIOGRAPHIC NOTES

The basic ideas of constructive modeling can be traced back to late 60's and early 70's. Ray casting was probably introduced by Appel [5]. In this context, many aspects of CSG modeling were quite early developed. Probably the first practical modeler was the original SYNTHAVISION system [46].

As a general technique for CSG modeling, ray casting was introduced by Roth [104].

The rigorous theory of CSG modeling was developed by the workers of the PADL-project at the University of Rochester [126,97,94]. The project constructed two successive CSG modelers, PADL-1 and PADL-2, which became widely known in the industry and the academia. They were eventually used as a basis for commercial development.

Advanced algorithmic techniques for CSG diminish the amount of processing by using several techniques. The thesis of Tilove [119] is a basic reference to many of them. A good example of these techniques is the *null object detection* algorithm described in [120]. A null object detection algorithm is expected to test whether a given CSG tree models the empty set or not.

The *pruning* of a CSG tree tries to diminish the amount of processing by modifying a CSG tree into a smaller equivalent tree. Woodwark uses a spatial subdivision and pruning technique to speed up the generation of images from a CSG representation [133,132]. Samet and Tamminen [107] use a similar technique for the conversion of CSG models to bintrees.

Figure 5.9, page 95 has been previously published as Figure 1 of the article "Algorithms for computing the volume and other integral properties of solids" by Y. T. Lee and A. A. G. Requicha in *Communications of the ACM*, Volume 25, Number 9.

CSG in/out test paper!

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.

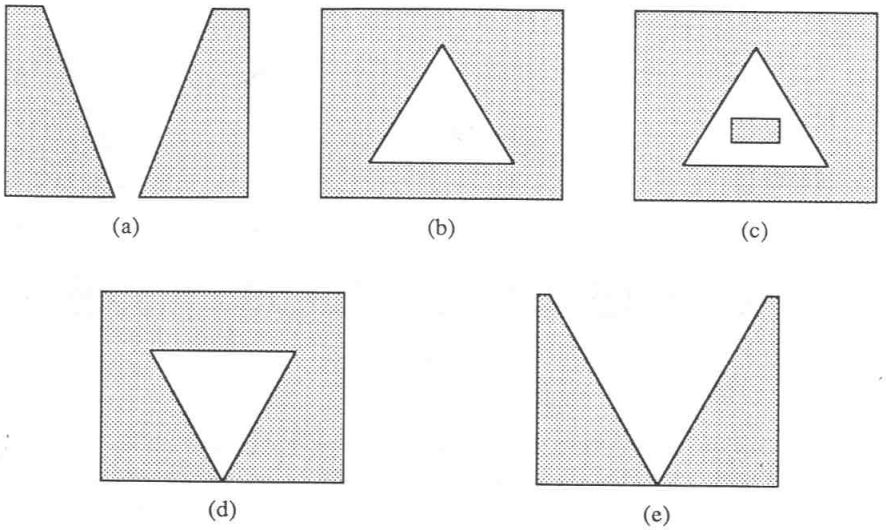


Figure 6.1 Definition of a face.

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*.

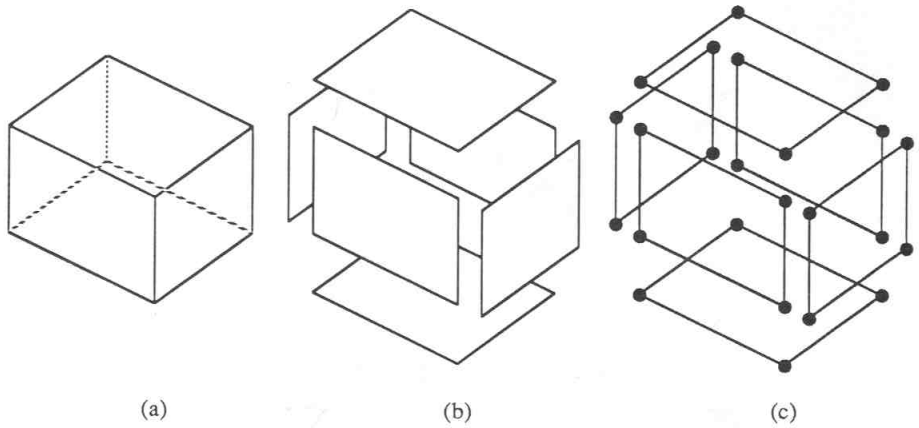


Figure 6.2 Basic constituents of boundary models.

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-

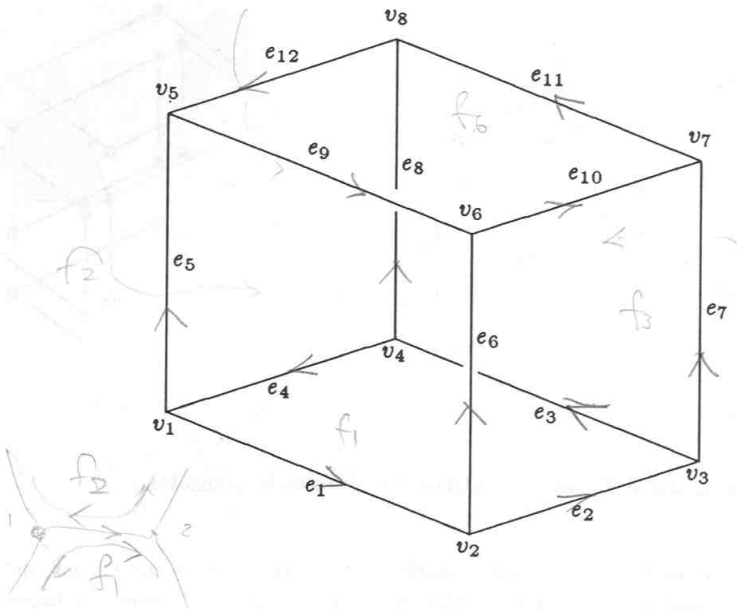


Figure 6.3 A sample object.

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 informa-

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.

tion 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.

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)).

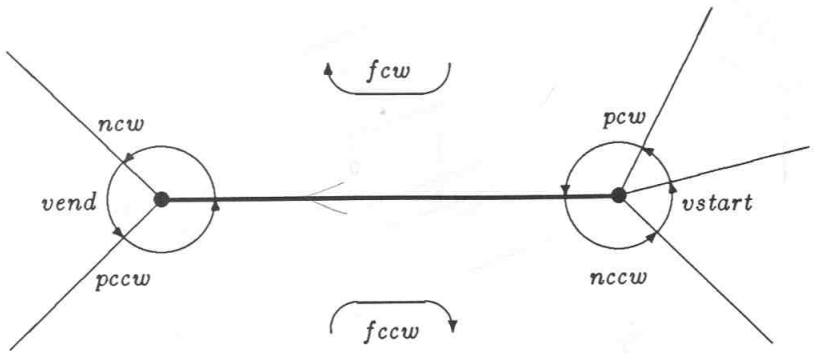
In fact, the data structure treats faces and vertices in a completely symmetric fashion. If the nodes *vstart* and *vend*, and the nodes *fcw* and *fccw* are exchanged, we end up with the *dual* of the original polyhedron.

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 6.8.

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.

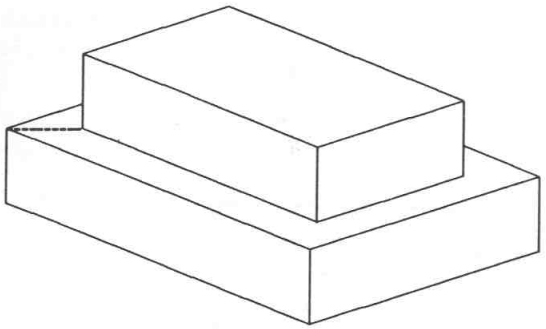


Figure 6.8 Object with non-simple faces.

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

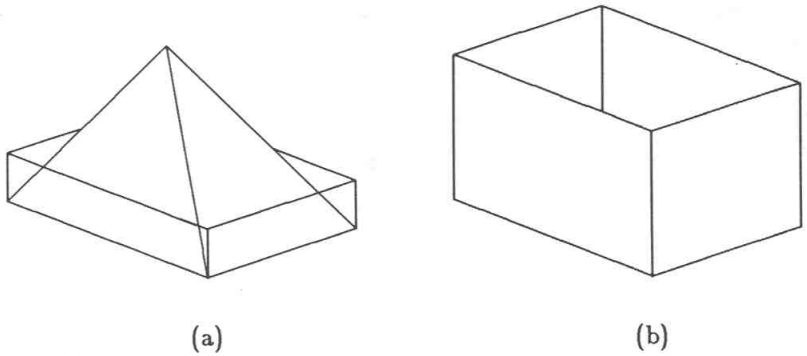


Figure 6.9 Invalid boundary models.

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-

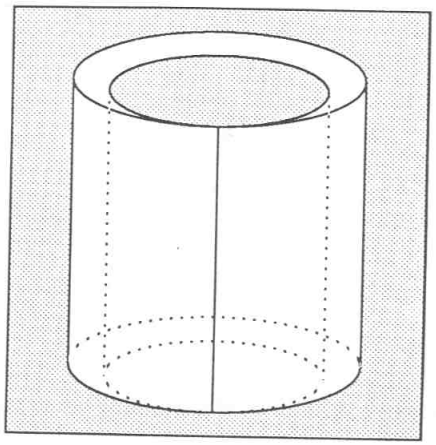


Figure 6.10 An object with no convenient BR.

scription facilities. On the other hand, he must take the integrity problems of boundary models discussed in the previous section into account, and seek for solid description techniques that either guarantee the integrity criteria, or (al least) make it difficult to construct invalid models.

Some frequently used description mechanisms are discussed briefly below.

6.4.1 Conversion from CSG

A common solution is to give a CSG-based solid description language to the user and construct boundary models through conversion from CSG. In this approach, the boundary modeler must include algorithms for creating boundary models from the CSG primitives, and an algorithm for Boolean set operations on boundary models.

Unfortunately, set operations for boundary models are computationally expensive and sensitive for numerical problems. Even if the numerical problems of calculating surface intersections can be solved, the problem of the mismatch between the modeling spaces of CSG and conventional boundary models remains. That is, some objects that can be represented in CSG have no convenient representations as boundary models.

An example of this is shown in Figure 6.10. The object can be represented in CSG as the difference of two cylinders. On the other hand, ordinary boundary data structures (such as the winged-edge data structure)

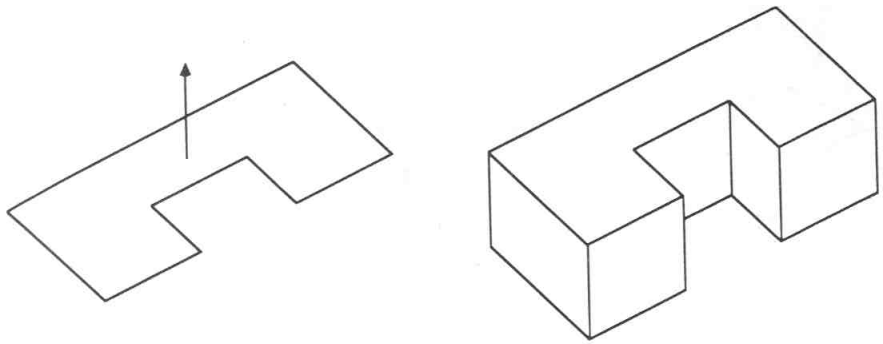


Figure 6.11 Solid description by drawing and sweeping.

cannot represent properly the region where the "inner" cylinder touches the "outer" one. Because an edge cannot have four neighbor faces, a pair of coinciding edges must be used. This, however, breaks down the geometric integrity criteria, and problems may arise if additional Boolean operations are performed on the boudary model.

Reference [96] outlines a number of set operations algorithms for boundary models; we shall study the problems of set operations further in Chapter 15.

6.4.2 Two-and-Half-Dimensional Drawing

Many objects needed in practice exhibit symmetry that can be exploited in their description. Quite often the object can be described in terms of a two-dimensional cross section, and information of material "thickness." Rotationally symmetric objects give another example of how the solid can be described just by a profile and a rotation axis.

Sometimes both of these types of objects could be described in terms of basic objects and Boolean set operations. Nevertheless, many people find it far more natural to "draw" an outline, and "sweep" it to give it a thickness or "swing" it to make it a rotational solid. Because of the relatively large computations needed for evaluating Boolean operations of boundary models, these operations can be executed much more efficiently than the sequence of set operations that creates the same model.

These *two-and-half-dimensional* solid description mechanisms can naturally be incorporated to boundary modelers. The term "two-and-half-dimensional" emphasizes that most description operations take place in

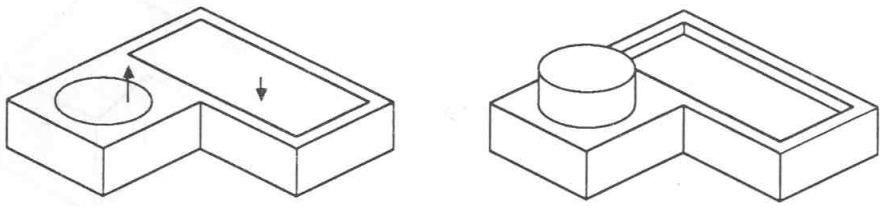


Figure 6.12 Face sweeping.

two dimensions, and can be visualized as operations on a two-dimensional graphical model.

One possibility is to include separate sweeping primitives in the boundary modeler, and to provide a conversion algorithm from the sweeping model to a boundary model. This approach is completely analogous to the inclusion of sweeping primitives to CSG models described in Section 5.2.

A more general approach is to consider sweeping an operation that can be performed on *any* planar face of a boundary model. This "face lifting" operation allows, for instance, the construction sequence of the object depicted in Figure 6.12. This efficient and economical input style is heavily used in boundary modelers belonging to the so-called BUILD group of solid modelers [53].

Engineering drawings generally display an object from several orthogonal views. This technique can be used for describing boundary models in the form of the so-called *profile set operations*. This operation combines two (or several) closed profiles, representing the outlines of an object from two views, by sweeping the outlines by an appropriate amount and evaluating the Boolean intersection of the swept objects. Surprisingly many objects can be modeled quite satisfactorily with this operation only; see, for instance, Figure 6.13.

Unfortunately, the sweeping operations are not completely "safe" either; is is easy to create self-intersecting objects with them, for example. See [18] for discussion on conditions under which sweeping and similar operations can be kept integrity-preserving.

6.4.3 Local Modification

A third natural solid description facility is based on *local modifications* of a boundary model. Actually, even the face lifting operation of the preceding section can be understood as a local modification that replaces a face by a

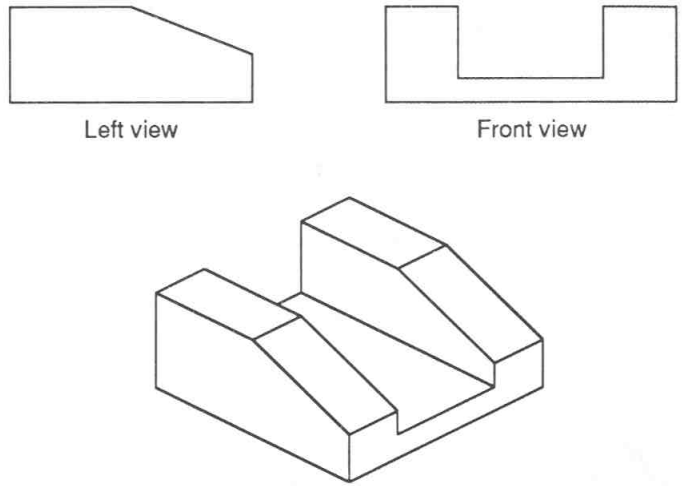


Figure 6.13 A sample profile set operation.

collection of new faces modeling the lifted region.

The boundary modelers BUILD-2 [18] and ROMULUS include a very rich collection of local modification operations. A typical example is the "rounding" of a sharp edge as depicted in Figure 6.14(a). A more general case of this is the "blending" of several adjacent edges depicted in (b); note how a vertex is replaced with a triangular blend surface.

The boundary modeler MODIF [25] includes another kind of a local modification operation. In MODIF, a solid is first modeled as a polyhedron. The polyhedron is then converted into a curved surface model by introducing control points on each edge. The user is then allowed to modify the coordinates of control points in order to create the desired shape. Special techniques are employed to keep the surface smooth.

6.5 ALGORITHMS FOR BOUNDARY MODELS

6.5.1 Visualization

The design of visualization algorithms for boundary models is greatly simplified by the availability of the faces, edges, and vertices of a solid. In this respect, they may be considered "explicit" models in comparison to CSG models that must be "evaluated" for visualization purposes.

All techniques for generating graphical output of graphical models are directly applicable to boundary models. In addition, for a polyhedral model

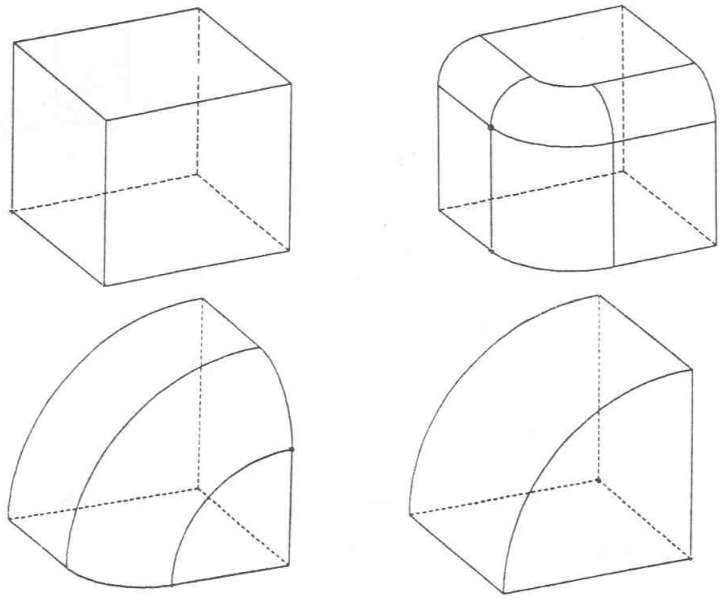


Figure 6.14 Local modifications of boundary models.

all well-known techniques of hidden line and surface removal and shading can readily be applied, including "exact" object-space methods. Furthermore, advanced graphical display devices often include hardware support for the rapid processing of polyhedral models.

Unfortunately, the presence of curved surfaces makes things far more complicated especially if hidden line output is required. There are specialized hidden line removal algorithms that can operate also on natural quadratic surfaces (see [70]), but in general it is more efficient to convert curved-surface models into polyhedral ones in order to create hidden line output.

Hidden surface removal is (surprisingly) simpler, because standard techniques such as *scan-line algorithms*, the *z-buffer algorithm*, and ray casting are applicable. These methods allow the generation of very high quality images at medium to high computational cost.

6.5.2 Integral Properties

Two general techniques are available for the calculation of basic engineering properties based on boundary models, namely the method of *direct integration* and the use of the *divergence theorem* of calculus [69].

Direct integration is the standard technique discussed in calculus textbooks. This technique is based on evaluating a volume integral over a solid as the sum of the appropriately signed contributions of its faces; see Figure 6.15(a).

For instance, the volume integral of a function $f(x, y, z)$ over a solid S can be evaluated by

$$\int\int\int_S f(x,y,z)dzdydx = \pm\sum_i \int\int_{F_i'} \left(\int_0^{z_i(x,y)} f(x,y,z)dz \right) dydx \tag{6.1}$$

where F_i' is the projection of face F_i on xy-plane, and $z_i(x, y)$ is obtained by solving the equation of F_i for z. The sign is determined by looking at the surface normal of the face: if the normal is facing the xy-plane, the minus is selected, otherwise the plus.

The resulting double integral can be evaluated by applying a similar technique to evaluating the contribution of each edge of F_i; see Figure 6.15(b). That is, the area integral of a function $g(x, y)$ over the face can be evaluated by

$$\int\int_{F_i'} g(x,y)dydx = \pm\sum_j \int_{E_j'} \left(\int_0^{y_j(x)} g(x,y)dy \right) dx \tag{6.2}$$

where E_j' is the projection of E_j on x-axis, and $y_j(x)$ the equation of E_j solved for y. This will finally lead us to single integrals of the form

$$\int_{x_{j,1}}^{x_{j,0}} h(x)dx. \tag{6.3}$$

where $x_{j,0}$ and $x_{j,1}$ are the (projected) endpoints of E_j.

Observe that both $z_i(x, y)$ and $y_j(x)$ are assumed to be single-valued in x and y. Hence curved faces and edges may have to be subdivided. It may still be hard to solve the surface and curve equations for z and y even numerically, and the equations of the projected edges E_j' of F_i' may not even be available. However, for polyhedral models the direct integration is quite attractive, because face equations are either single-valued or can be ignored, and straight lines project to straight lines.

The divergence theorem provides an alternative method for the evaluation of integral properties [122]. From the observation that it is always possible to find at least one vector function $g(x, y, z)$ such that $\mathrm{div} g = f$ for any given continuous function $f(x, y, z)$, it follows that

$$\int_S f\, dV = \int_S \mathrm{div} g\, dV = \sum_i \int_{F_i} g \cdot n_i\, dF_i$$

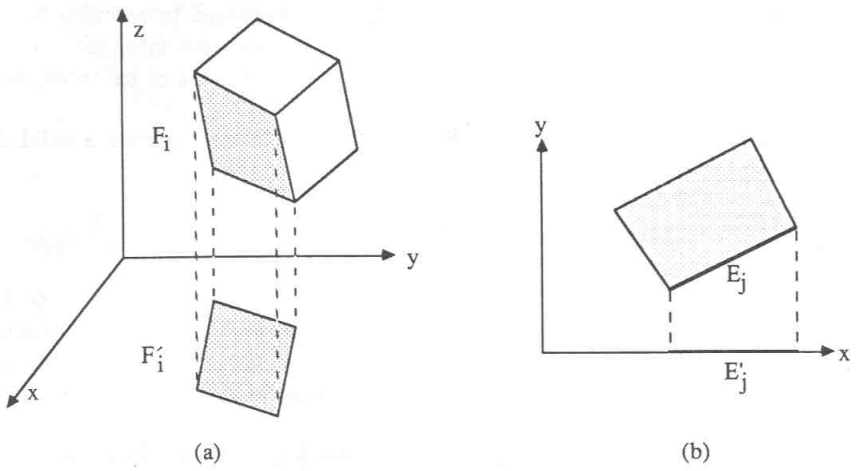


Figure 6.15 The method of direct integration.

where F_i is a face of the solid, S, n_i is the unit normal vector to F_i, and dF_i is the surface differential. For instance, if

$$f(x, y, z) = y^2 + z^2,$$

then, for example, the choices

$$g(x, y, z) = [x(y^2 + z^2) \;\; 0 \;\; 0]$$

and

$$g(x, y, z) = [0 \;\; y^3/3 \;\; z^3/3]$$

are both possible. If all F_i are planar polygons, integrals on the right side can be readily evaluated. In the general case, approximation techniques are needed; see [122] for additional information and practical results.

In addition to the general methods discussed above, very straightforward direct techniques for evaluating integral properties of consistently oriented polyhedral objects are available. For instance, the volume of a polyhedron can be reduced to calculating to the sum of the signed volumes of tetrahedra [129].

Assume the tetrahedron has one corner at origin, and the other corners at

$$P_i = [x_i \; y_i \; z_i],$$

for $i = 1, 2, 3$. Then the signed volume of the tetrahedron is given by

$$V_{tetra} = \frac{1}{6} \; P_1 \times P_2 \cdot P_3$$

where $\times$ and $\cdot$ stand for the vector and scalar product of vectors. Observe that V_{tetra} is positive or negative depending on whether the orientation of the triangle $P_1 \; P_2 \; P_3$ as viewed from the origin is clockwise or counterclockwise. The volume of a polyhedron can calculated by taking all tetrahedra formed by three consecutive vertices in faces of the solid, and summing their signed volumes. The reference [129] reports very similar algorithms for other integral properties.

6.6 PROPERTIES OF BOUNDARY MODELS

We can again formulate the summary of this chapter by discussing the properties of boundary models within the general framework of Section 3.6.1.

Expressive power: The modeling space of boundary models depends on the selection of surfaces that can be used. There is no inherent reason to limit this collection to mere half-spaces; hence boundary models can be used to represent objects from a more general modeling space than that possible for CSG.

On the other hand, as noted in Section 6.4.1, some objects that can be represented in CSG have no convenient representations as boundary models. While knowledgeable users can avoid such objects when working in direct interaction with a boundary modeler, this deficiency may cause problems e.g. if conversion from a CSG model is attempted.

Validity: Validity of boundary models is in general quite difficult to establish. Validity criteria split into topological constraints and geometric constraints as discussed above. While it is possible to manage topological validity without large overhead, it is hard to enforce geometric correctness without penalizing the speed in interactive design.

Unambiguity and uniqueness: Valid boundary models are unambiguous. They are not unique.

Description languages: Boundary models are tedious to describe directly. Fortunately, it is possible to build description languages that are based on graphical "drawing" and "sweeping" operations or on a CSG-like input on top of a boundary model.

Conciseness: Boundary models of useful objects may become large, especially if curved objects are approximated with polyhedral models.

Closure of operations: As discussed in Chapter 3, boundary models are usually *not* closed under set operations (regularized or not). In this theoretical sense boundary model description mechanisms based on CSG

conversion or set operations are always vulnerable. The natural closed operations for boundary models are the *Euler operators* to be studied in Chapter 9.

Computational ease and applicability: Boundary models are useful for generating graphical output, because they readily include the data needed for driving a graphical display. Analysis algorithms based directly on boundary models become quite difficult if representations of objects from more general modeling spaces than polyhedra must be processed.

We shall return to boundary models in much more detail in Part Two of this book.

PROBLEMS

6.1. Describe an algorithm for extracting the edges around a vertex from the full winged-edge data structure of Figure 6.6.

6.2. Form the dual of the full winged-edge data structure of Figure 6.6 as indicated in the last paragraph of Section 6.2.3. Which object do you get?

6.3. Present (in the style of Figure 6.6) a modified winged-edge data structure of the "pyramid" of Figure 6.7 that can represent faces with several boundaries.

6.4. Based on the general Equations 6.1–3, outline an algorithm for evaluating the following integral properties of a polyhedral solid S:

(a) the volume V, which is expressed as the triple integral

$$V = \int\int\int_S dzdydx$$

(b) the moment of inertia about z,

$$I_z = \int\int\int_S x^2 + y^2 \, dzdydx$$

BIBLIOGRAPHIC NOTES

An overview of boundary data structures and the representations used in a number of modelers is given in the survey by Baer, Eastman, and Henrion [8], which still is recommended reading for a student of solid modeling.

The solid modelers BUILD-1 and BUILD-2 written by the former BUILD-group at the University of Cambridge under the leadership of Ian Braid belong to the most influential works to the solid modeling technology. The report [18] provides an overview of the design and implementation of the version of BUILD-2 of that

time, and is a must to anybody contemplating the design of a boundary modeler. Work on BUILD modelers still continues at the Cranfield Institute for Technology.

A good reference to scan-line techniques is [102].

Section 6.5.2 is based on the survey article on integral properties by Lee and Requicha [69].

Chapter 7

HYBRID MODELERS

None of the three major approaches to solid modeling we have described so far is superior to the others in all respects. This motivates the use of multiple simultaneous representations in a serious modeling system. These *hybrid modelers* are the topic of this chapter.

7.1 WHY MULTIPLE REPRESENTATIONS?

In the three previous chapters, the three major approaches to solid modeling were introduced, and their inherent properties discussed. Even at the cost of oversimplification, let us review some key points of this discussion:

1. The simplest of the three approaches, *decomposition models* are superior to the others as sources of data for numerical algorithms, even when they are approximative. Except when the original source of information is a binary image, their creation is complicated without conversion from other representations. They are also voluminous.

2. *Constructive models* are the most concise of the three major approaches to solid modeling. Because the directly available methods for the generation of graphical output or the creation of data for numerical algorithms are slow, these tasks are performed through conversion to other representations.

3. *Boundary models* are directly useful for graphical applications. Especially polyhedral models are useful for numerical applications as well, if the limited modeling space captured by polyhedra can be tolerated. Without the help of other representations and conversions, their creation is difficult, however.

Clearly, a *combination* of the three major approaches to solid modeling would be superior to any of them alone. This has led designers of solid modeling systems to investigate possibilities of combining the basic approaches.

A *hybrid* solid modeler is capable of supporting several coexisting solid representations, and tries to pick the most suitable of them for each task. These representations are guaranteed to be *consistent* up to the level made possible by differences in the inherent modeling spaces. Consistency enforcement is materialized in *conversion algorithms* that can go from one representation to another.

Hence, the representations stored by a hybrid modeler are a combination of several different basic representations. A graphical model can well appear as one of these representations. Observe that a hybrid modeler can well include, say, several types of boundary models. For instance, it may store a winged-edge representation with curved surfaces for modeling procedures, and a distinct polyhedral model for display purposes. Solid representations for secondary storage could also form still another distinct boundary model.

7.2 PROBLEMS OF HYBRID MODELERS

The coexistence of several solid representations in one modeler raises a number of new kinds of problems.

7.2.1 Conversions

Clearly, the conversion algorithms form a fundamental part of a hybrid modeler. Unfortunately, there are inherent limitations on what kinds of conversions a hybrid modeler can expect to support.

Constructive models and CSG models in particular have the virtue that they can be converted to any other representation. The feasibility of the conversion to a boundary model, the *boundary evaluation* has been well demonstrated in many research and commercial CSG modelers. General approaches for converting from a CSG representation to an enumerative representation were outlined in Chapter 5.

Unfortunately, the inverse conversions are not available. In particular, the *inverse boundary evaluation*, i.e., the conversion from a boundary representation to a CSG model, still remains an unsolved problem for practical purposes in the general 3-dimensional case. However, boundary models can be converted to decomposition models approximately as easily as CSG.

The conversions from a decomposition model to a boundary model or a CSG model would be valuable in the case that information received as

a binary image must be mapped against information stored as a boundary model or a CSG model. Robot vision offers an example of great practical interest. Unfortunately, these conversions are still more a research subject than a technology that could be exploited in practical solid modelers.

Because conversions from boundary modes or CSG models to decomposition models generally are approximative, it may seem that the inverse conversion makes little sense: if the exact information is already available, why attempt its reconstruction? Solid modelers based on boundary models or CSG models usually assume this view, and treat decomposition models as *transient* representations, i.e., decomposition models are created "on the fly" to solve certain problems, and discarded afterwards.

In fact, *all* modelers that create graphical output during an interactive design session can be considered hybrid modelers, because the display is a kind of geometric model of its own.

7.2.2 Consistency

If a conversion algorithm between two representations included in a hybrid modeler is available, it will be possible to keep the representations consistent. Unfortunately, this usually happens at the cost of limiting the functionality available in the modeler.

For instance, CSG models cannot ordinarily include parametric surfaces. This means that if a boundary model created by conversion from CSG is modified further to include blends represented as parametric surfaces, the original CSG representation of the object modeled cannot be kept consistent with it.

In general, a hybrid modeler that aims at total consistency between the representations can only support such operations on solids that can be mapped to each of the representations. Boolean set operations have their prominent position in solid modeling partly because they can be mapped to all the major solid representations. Unfortunately, set operations alone are not sufficient for constructing a good user interface to a solid modeler, and it is much more difficult to support "drawing" operations on several representations.

7.2.3 Modeling Transactions

All modifications to solid representations take place in a sequential fashion. Therefore, a total consistency between the various representations cannot exist at all time instances.

This leads us to define the concept of a *modeling transaction* as an indivisible sequence of modeling operations that commences from a consis-

tent state and ends in a consistent state. For instance, in a hybrid modeler consisting of a CSG modeler and a polyhedral modeler, the modeling transaction of performing a set operation on two primitives would consist of the creation of a CSG tree consisting of the primitives, and computing a polyhedral model of the result of the set operation. In an interactive environment, the display can be considered a representation in its own right; in this case, the modeling transaction should include the updating of the displayed image. Solid representations in secondary storage add another dimension to modeling transactions: an interactive modeling session is complete only after the relevant results are stored.

The functions available to the user must be designed with care in any solid modeler. In a hybrid modeler the careful design becomes still more important.

7.3 HYBRID ARCHITECTURES

The solution usually adopted in hybrid modelers to the problems of lacking conversions and consistency is to treat one of the representations supported as the *primary* representation, and the others as *secondary* representations.

In this respect, most solid modelers available today seem to fall into one of two major architectures. The modelers in the first category (Figure 7.1(a)) ordinarily use CSG trees as their primary solid representations. From CSG trees, boundary models may be created through boundary evaluation, and decomposition models through classification algorithms in the style discussed in Section 5.2.

The user, however, has no direct access to the secondary representations. In other words, although boundary models might be included in the modeler (and perhaps continuously updated through incremental boundary evaluation to reflect changes in the CSG tree), the user of the modeler cannot perform modeling operations which are specific to a boundary model, say, local modifications. Modelers such as PADL-1 [127], PADL-2 [21], and GMSOLID [17] belong to this group.

Modelers that use boundary models as their primary representation form the second group (Figure 7.1(b)). These modelers usually include CSG as one solid description facility, and many of them also store a variation of the CSG tree (see e.g. [18]). In addition to CSG, however, these modelers have also other solid description facilities that directly modify boundary data structures. These facilities may include the "drawing" and "sweeping" operations or local solid modifications as discussed in Chapter 6. Hence, the role of CSG in these systems is really that of an auxiliary representation only. Many industrially used modelers, such as ROMULUS,

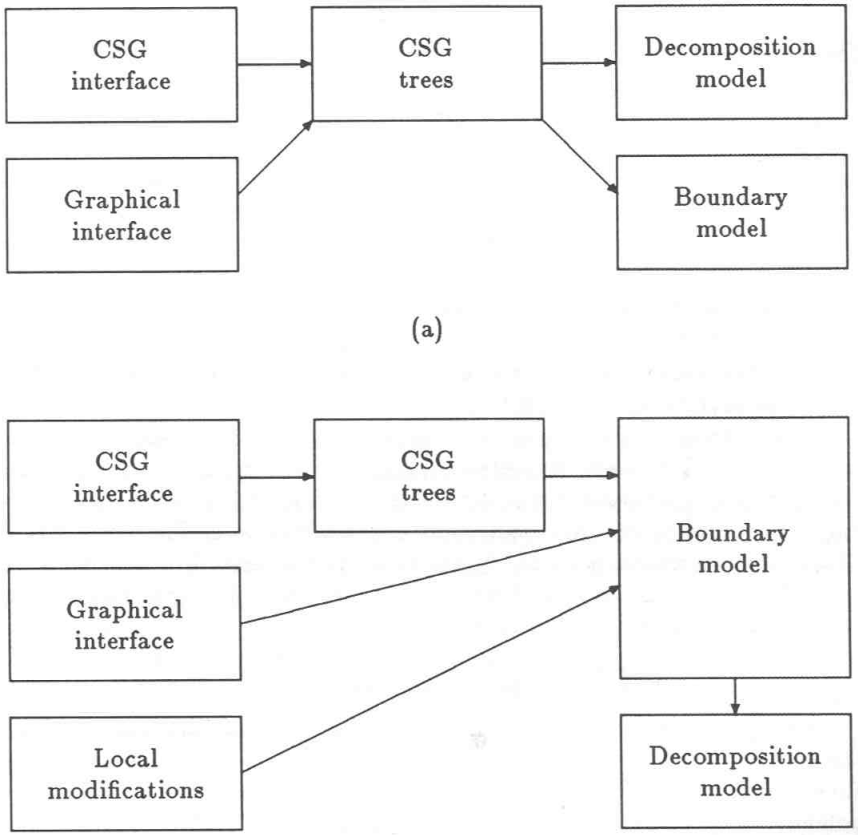


Figure 7.1 Architectures for hybrid modelers.

GEOMOD [118], and MEDUSA [87] belong to the second major group.

Various architectures for hybrid modelers are discussed in greater detail in references [99,100], including information of numerous solid modelers.

7.4 DISTRIBUTED MODELERS

The current trend towards the use of *distributed* graphics systems consisting of a host computer and graphics workstations has significance to hybrid solid modelers. As current graphics workstations have considerable processing power of their own, and actually are usually small computers themselves, the problem of dividing the labor sensibly between the host and the workstations becomes urgent. The analogous problem also arises in top-of-the-line workstations that have a special graphics processor for the rapid display of polygonal models.

A solution to this dilemma is the introduction of a *distributed modeler*, part of which resides in a workstation and part in the host. Because the modeler maintains solid models in both parts, distributed modelers are a special case of hybrid modelers.

As an example of the division of labor in a distributed modeler in the context of CSG models, Atherton [6] suggests the separation of a *visual modeler* from a *analytical modeler*. The visual model is intended to support rapid interactive processing, and resides locally in the workstation. The analytical model is stored in the (still quintessential) host computer; for CSG, the analytical model is the exact CSG tree. In particular, Atherton suggests that CSG primitives are approximated by polyhedra in the workstation, and proposes an efficient algorithm for rapidly displaying CSG trees consisting of such polyhedral primitives.

A similar division of labor is also possible in boundary models. So-called *faceting modelers* are a class of boundary modelers that "know" about curved surfaces, but do not represent them explicitly. Rather, they store polyhedral approximation of the surface, and it is this representation that is used for, say, generation of graphical output. The distinction between faceting modelers and what we have called merely polyhedral modelers is that polyhedral modelers do not know where the planar faces came from, even if they can create polyhedral approximations of curved objects.

Many graphics workstations store a structured display list with polygonal graphical primitives, which can be considered a simple polyhedral model. The display list is constructed by a boundary modeler residing in the host computer, and it must be updated to reflect operations performed on the solid model. Unfortunately, this architecture still poses heavy requirements for host-workstation communication in the distributed

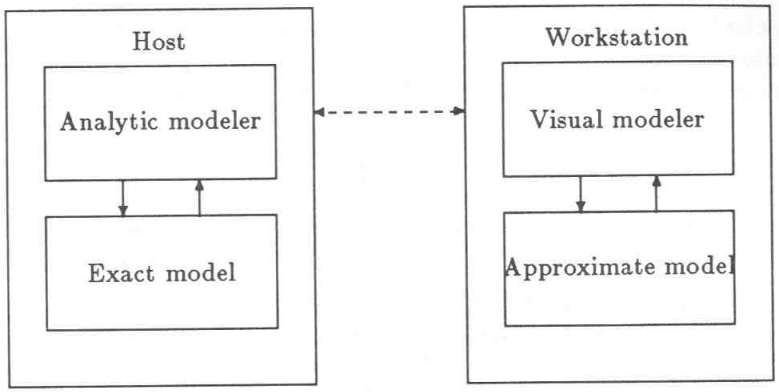


Figure 7.2 A distributed modeler.

environment.

As a step forward, a faceting modeler could be used in the role of a visual modeler, i.e., to provide rapid interactive processing locally in a workstation. This approach is strongly supported by that many graphics workstations have special hardware for rapid processing of polyhedral models. For numerical applications requiring high accuracy, the implicit surface information of the faceted modeler can be transmitted to the host computer, and elaborated by the analytic modeler.

7.5 OTHER HYBRID APPROACHES

Hybrid modelers discussed in the previous sections use several coexisting independent representations to be able to use the most appropriate ones for each task.

To avoid the inherent difficulties caused by several represenrations, modelers that pursue the combination of several basic approaches in a single integrated solid representation scheme have been proposed.

Typically, these modelers combine a decomposition model (exhaustive enumeration or an octree) with either a CSG model or a boundary model, and use the decomposition model as a geometric access method to CSG or boundary information.

For instance, the *extended octree* of Ayala *et al.* [7,86] is based on storing a boundary representation of a polyhedral object within the cells of an octree. The octant subdivision is continued until each cell contains at

most one vertex, one edge, one face, or is homogeneously "full" or "empty." The cited references give algorithms for conversion from a boundary representation to an extended octree and vice versa, and for set operations of extended octrees. References [22,136] describe similar representations.

PROBLEMS

7.1. Are there other solid description techniques besides set operations that can easily be supported on top of all major approaches to solid modeling?

7.2. The separation between "general-purpose" geometric models and graphical data structures (e.g., structured display lists) for display purposes is debatable. Consider the pros and cons of keeping these data separate versus integrating them into a single representation.

BIBLIOGRAPHIC NOTES

An algorithm for inverse boundary evaluation is expected to convert an arbitrary boundary model into an equivalent CSG tree. Unfortunately, the problem seems to be subject to combinatorial explosion, and rather difficult to solve optimally (i.e., with the minimum number of primitives) or even satisfactorily.

The feature extraction algorithm of Woo [130] can *in principle* generate a CSG expression from a boundary model by using a convex hull technique. Unfortunately, without human help the algorithm seems to be subject to infinite looping.

Two-dimensional versions of the problem are much simpler. See, for instance, the work of Vossler [128] on the practically interesting problem of the conversion of swept planar outlines consisting of straight lines and circular arcs to a CSG model.

As for the other conversions, Kunii *et al.* [66] have reported an algorithm for converting an octree to a boundary representation. However, the algorithm makes no attempt to infer the actual surfaces of a curved object from the rasterized data.

Part Two

THE GEOMETRIC WORKBENCH

Part One covers the basic approaches and technologies of solid modeling without going deeply into system and implementation aspects. In contrast, Part Two takes a much closer look boundary modeling by means of developing the basic algorithms for the *Geometric WorkBench* (GWB), a simple polyhedral solid modeler following the boundary approach.

Chapter 9

EULER OPERATORS

In the architecture of GWB, the Euler operators have a central role. Based on the theory of plane models of Chapter 3, this chapter takes a closer look on the manipulation of boundary data structures, derives Euler operators, and presents rigorously their properties.

9.1 MANIPULATION OF PLANE MODELS

As noted in Section 6.3, the topological integrity of a boundary data structure of the winged-edge type can be enforced just by structural means. Nevertheless, the problem of manipulating a complicated data structure while making sure that all required references to nodes are maintained properly is somewhat difficult, and makes the writing of nontrivial algorithms (such as Boolean set operations) inconvenient.

Fortunately, the theory of plane models will give us help. As discussed in Section 3.5, plane models give us a useful mathematical abstraction of boundary models. In particular, the topological properties of the surface can be evaluated from the plane model. What we still need, however, is a scheme for creating plane models that have some desired properties.

Our aim here is to identify a small set of plane model manipulation operations that are powerful enough to describe *all* plane models of physical significance. While being general, the operations should be *safe* in that, for instance, nonorientable models (such as the Klein bottle of Figure 3.8 on page 42) cannot be created with them.

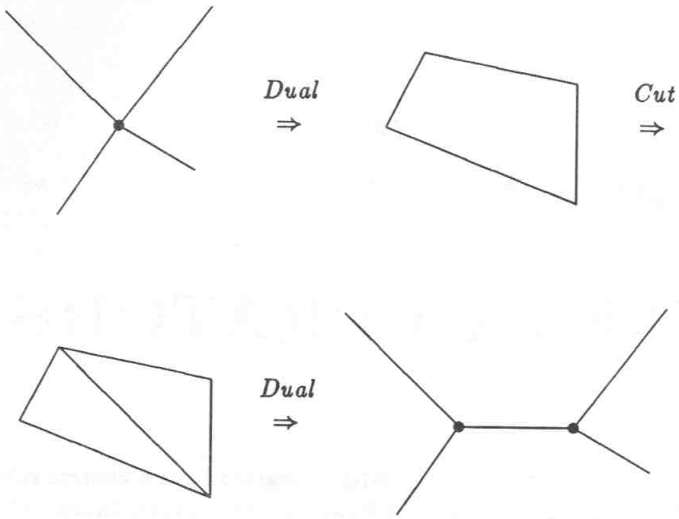


Figure 9.1 The vertex splitting operation.

9.1.1 Local Topological Operations

One of the main results of the plane model theory presented in Section 3.5 is the invariance theorem (Theorems 3.11 and 3.12, pages 43–45). The theorem states that the Betti numbers and all the topological properties conveyed by them (such as the Euler characteristic and orientability) remain invariant for all plane models that can be constructed for a surface.

A direct consequence of the invariance theorem is the following:

Lemma 9.1 *The topological properties of a plane model are not altered by either cutting a polygon in two or pasting together two polygons along a common edge.*

We interpret the cutting and pasting operations of polygons as a pair of inverse manipulation operations for plane models. That is, the cutting operation can divide a polygon into two polygons by joining two of its vertices with a new edge. Hence the sequence of edges around the polygon is divided into two sequences. Conversely, the inverse pasting operation removes an edge separating two polygons and merges their edge-cycles.

It is natural to ask whether there are other useful manipulation operations. In the search, the duality introduced in Section 3.5.8 offers help. To

see how, let us consider the effect of the cutting operation on the dual plane model. As demonstrated in Figure 9.1, a cutting operation on a polygon of the dual model has the effect of splitting the corresponding vertex of the original model in two vertices. Similarly, a pasting operation on the dual has the effect of a vertex joining operation that combines two vertices connected with an edge into one, and merges their edge-cycles. The splitting and joining operations are inverses of each other.

By Lemmas 9.1 and 3.13 of Section 3.5.8, a polygon cutting operation or its inverse on the dual plane model will preserve all topological characteristics of a plane model. Therefore, the vertex splitting operation or its inverse do not alter the topological properties of the plane model they modify, and they can be added to our set of manipulation operations.

Do we need other operations? An answer to this is provided by the following theorem:

Theorem 9.2 *Let PM be a realizable plane model of genus $= 0$ (i.e., a plane model topologically equivalent to a sphere). Then the polygon pasting and vertex joining operations are capable of reducing PM to a plane model that has just one vertex and one polygon, but no edges.*

Proof: *Suppose there is an edge E of PM such that it cannot be removed with either a polygon pasting or a vertex joining operation. Therefore, E must occur twice at the same polygon, and twice at the same vertex. Hence, E forms a closed curve on the 2-manifold M modeled by PM that does not split M into two components, and PM must be of genus > 0.*[1] *By contradiction, no such E can occur, and all edges of PM can be removed.*

So, by means of the pasting and joining operations only, *all* plane models of the sphere can be reduced to a one-vertex, one-polygon model. We call this special model the *skeletal plane model*. A sample case of the theorem is presented in Figure 9.2. In this particular case, all vertices of the object are first made to occur in a single polygon by removing edges with the pasting operations until no further faces can be merged (Figure 9.2(a,b)). Thereafter, all remaining edges are removed with the joining operation until only the skeletal plane model (Figure 9.2(c)) remains.

Of course, the skeletal plane model as such is not a particularly useful model of a sphere or a cube. Its significance for our purposes is due to the following:

Corollary 9.3 *Let PM be a realizable plane model of genus 0. Starting from a plane model consisting of a single vertex and a single polygon, PM*

[1] See discussion on the first Betti number h_1 on page 45.

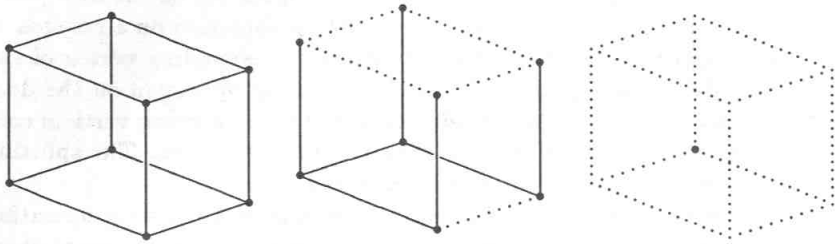


Figure 9.2 Reducing a cube to the skeletal plane model.

can be created by a sequence of e polygon cutting or vertex splitting operations, where e is the number of edges of PM.

Proof: *By Theorem 9.2, all edges of PM can be removed with a sequence S consisting of e pasting and joining operations. Because the polygon cutting and the vertex splitting operations are the exact inverses of the pasting and joining operations, the sequence S^{-1} consisting of the cutting and splitting operations corresponding with the operations of S in reverse order is a sequence satisfying the claim.*

That is, starting from the skeletal plane model, *all* plane models of genus 0 can be created by means of the cutting and splitting operations! Therefore, in order to create an arbitrary plane model of genus 0, we only need to supply the "prototype" skeletal plane model as a primitive, and work on it with the two operations.

As the two manipulation operations do not alter the global topological properties of the plane model they operate on, we call them *local topological operations.*

9.1.2 Global Topological Operations

In addition to the local topological operations, we need manipulation operations which are capable of producing objects whose genus is greater than zero. As these operations must definitely alter the global topological properties of a plane model, we call them *global topological operations.*

At this point a slight diversion back to the topology of surfaces is necessary. Given two surfaces, it is always possible to construct a new surface by cutting a small disk off from each original surface, and pasting the two

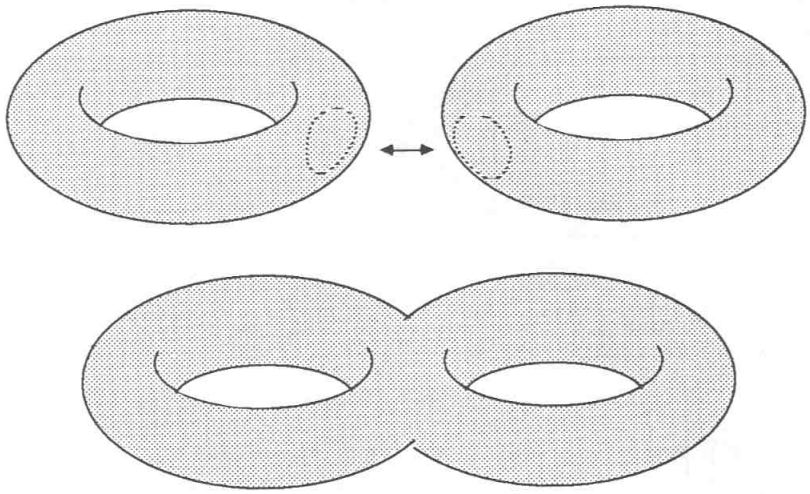


Figure 9.3 Connected sum of two tori.

surfaces together along the boundary of the cutouts. For instance, the *two-holed torus* can be created in this fashion from two tori (see Figure 9.3).

The surface resulting from the cut-and-paste procedure is called the *connected sum* of the two original surfaces. That the cut-and-paste procedure is general enough for creating all surfaces we are interested in, is shown by the so-called *classification theorem*:

Theorem 9.4 *Every orientable surface is topologically equivalent either to the sphere, or the connected sum of n tori.*

The proof of the classification theorem is beyond the scope of this text; again, see [52] for a good exposition.

The cut-and-paste procedure can be readily interpreted in the language of plane models. In the interpretation, the "cutting of disks" becomes the creation of two polygons in the two plane models at the desired positions. The "pasting" is modeled as an operation that combines those two polygons with a new edge that will occur twice at the new combined polygon.

For instance, Figure 9.4 depicts the plane models of a cube and a tetrahedron. They can be combined by joining the polygon p of the cube and the "external" polygon with a new edge. Observe that new edge occurs twice in the combined polygon p'; i.e., p' now has nine edges.

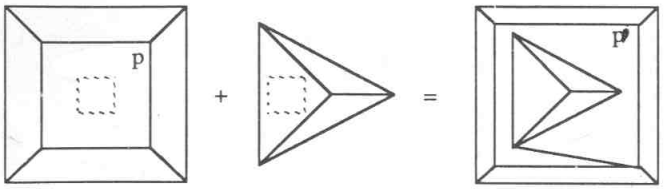


Figure 9.4 Connected sum of two plane models.

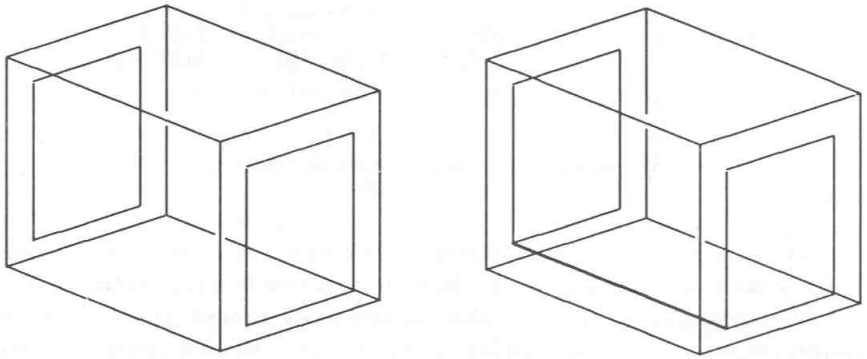


Figure 9.5 Connected sum on a single plane model.

If we interpret the two original plane models of Figure 9.4 as a single disconnected plane model, we may observe that the connected sum operation alters the connectivity of the model. By extending the use of the connected sum operation to two polygons of a connected (component of) plane model, we can also alter the genus of a plane model, as desired. For instance, we can produce a hole through the cube of Figure 9.5 by joining the two polygons at its ends with a new edge.

Again, the connected sum operation has a natural inverse operation. This *connected minus* operation can be used to remove an edge occurring twice at a polygon. As a result, the polygon is divided into two polygons.

Furnished with the connected sum operation and its inverse, we can show the following:

Theorem 9.5 *Let PM be a realizable plane model. Using the local topological operations (polygon cutting and pasting, vertex splitting and joining), and the connected sum and connected minus operations, all edges of PM can be removed, i.e., PM can be reduced to the skeletal plane model.*

Proof: *Consider an arbitrary edge E of PM. The following case analysis applies:*

1. *E separates two distinct polygons. In this case, E can be removed with the polygon pasting operation.*
2. *E occurs in a single polygon between two distinct vertices. In this case, E can be removed with the vertex joining operation.*
3. *Otherwise, E can be removed with the connected minus operation.*

Because all cases are covered, all edges can be removed.

Similarly to the corresponding results of Theorem 7.2 and Corollary 7.3, the following corollary is readily seen to be valid:

Corollary 9.6 *Let PM be a realizable plane model. Starting from the skeletal plane model, PM can be created by a finite sequence of local topological operations and connected sum or minus operations.*

Proof: *Analogous to Corollary 9.3.*

That is, *all* realizable plane models can be created with the local topological operations and the connected sum and minus operations.

9.1.3 Soundness of Plane Model Operations

Corollary 9.6 tells us that every plane model of interest to us (that is, all realizable plane models) can be generated by means of the plane model manipulation operations. Therefore, the operations are *complete*, i.e., they are powerful enough for describing all realizable plane models.

It is natural to ask whether the operations also are *sound*, i.e., whether we can be certain that no others besides the realizable plane models can be generated through their use. That this really is so is shown by the following theorem:

Theorem 9.7 *Both the local plane model manipulation operations (polygon cutting and pasting, vertex splitting and joining), and the global plane model manipulation operations (connected sum and connected minus) are* sound, *i.e., they cannot create nonrealizable plane models.*

Proof: *Let us consider an arbitrary finite sequence of plane model manipulation operations. The proof will be inductive on the length of the sequence.*

First, the criteria for realizable plane models as given by the Definitions 8.9 and 8.10 of Chapter 8 are as follows:

1. Edge identification: *Every edge of the plane model is topologically identified with exactly one other edge.*
2. Cyclical identification: *The polygons identified at each vertex can be arranged in a cycle such that each consecutive pair of polygons in the cycle is identified at an edge adjacent to the vertex.*
3. Orientability: *The polygons of the model can be oriented so that for each pair of identified edges, one edge occurs in its positive orientation in the direction of its polygon, and the other in the negative orientation.*

As the basis of the induction, the skeletal plane model trivially satisfies the three conditions. From there on, we have exactly six cases corresponding to the six possible operations to consider. In each case, we shall have to show that the operation will preserve the criteria 1–3 stated above.

Let us only elaborate two of the cases here and leave the rest as an exercise to the reader. First, the vertex splitting operation adds a pair of identified edges that occur in opposite orientations in their respective polygons in the plane model. Hence, if the plane model originally satisfies the edge identification and the orientability constraints, it will do so also after the operation. As for the cyclical identification constraint, observe that the vertex splitting operation divides the cycle of edges of a vertex into two sequences; hence also cyclical identification will remain unaltered.

As vertex splitting and face cutting are duals of each other, the reasoning above immediately has the consequence that the face splitting operation will not alter the validity criteria either. Finally, also the connected sum operation can be shown to preserve the criteria.

Because none of the cases violates the inductive hypothesis, an arbitrary sequence of the operations will create a model that satisfies the realizability criteria, and we have the theorem.

Together, Corollary 9.6 and Theorem 9.7 state that the local and global manipulations are *sound and complete* for the family of realizable plane models. Therefore, they fill exactly the desired characteristics stated in the start of this section.

9.2 MANIPULATION OF BOUNDARY MODELS

As seen in the previous section, a small collection of manipulation operations (and their inverses) are sufficient for describing all plane models of

interest. According to the theory, only three basic types of manipulative operations were needed:

1. A "prototype" primitive that creates skeletal plane models.
2. Two local topological operations that subdivide the sequences of edges of a face or of a vertex.
3. A global topological operation that implements a connected sum of two polygons (either from distinct models or one model).

The theory carries over to boundary models. The manipulation operations on plane models will have their counterparts in the form of *Euler operators* that act on the topology of a boundary model data structure. Being realizations of the theoretical plane model operations for the domain of real bounary data structures, Euler operators share their completeness and soundness stated as Corollary 9.6 and Theorem 9.7.

9.2.1 Notation and Conventions

Euler operators were originally introduced by Baumgart [13,14] in the context of the winged-edge data structure. As this data structure is the natural basis for discussing the operators, we shall assume the use of the data structure of Figure 6.6, extended with loop nodes for representing nonsimple faces. For convenience of language, the internal loops will be termed *rings*. We shall also use the term *shell* introduced in Section 3.5.6 to denote a connected component of a boundary data structure.

We shall also use the convention of having *empty loops* in the data structure, i.e., loops consisting of just a single "lone" vertex with no edges at all. Such entities are useful, for instance, for representing the boundary model counterpart of the skeletal plane model.

By historical convention, the operators are usually denoted by rather cryptic, mnemonic names. The key to the names to be used in this text is given in the below:

M — make	V — vertex	H — hole
K — kill	E — edge	R — ring
S — split	F — face	
J — join	S — solid	

For instance, the name "mev" translates as "Make Edge, Vertex".

It should be borne in mind that Euler operators can be implemented in several ways on top of widely varying data structures. We shall emphasize this by using the SMALL CAPS font when refering to an Euler operator

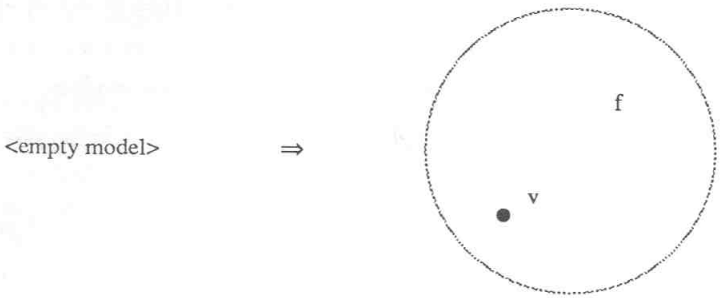


Figure 9.6 The MVFS operator.

"in general," without regard for a particular implementation. This is in contrast to our usual convention of using the `typewriter` font when refering to an actual piece of code.

9.2.2 Skeletal Primitives: MVFS and KVFS

The theory tells us that by means of local and global topological manipulations, all plane models of interest can be created from a single skeletal plane model. This result means that a single skeletal primitive is sufficient for us.

The operator MVFS takes this role in our collection. As the name suggests, it creates from scratch an instance of the data structure that has just one face and one "lone" vertex. Hence the new face has one "empty" loop with no edges at all. The effect of MVFS is depicted in Figure 9.6.

The "solid" created by MVFS may not satisfy the intuitive notion of a solid object. Nevertheless, it it useful as the initial state of creating a boundary model with a sequence of Euler operators.

Similarly to the plane model manipulation operations, all Euler operators will have corresponding inverse operators that can undo the effect of the "positive" operator. The inverse of MVFS is called KVFS, and it destroys a skeletal instance of the data structure identical to that created by MVFS.

9.2.3 Local Manipulations

In analogy to local topological operations on plane models, we need operators that can work on the local topological properties of a boundary model.

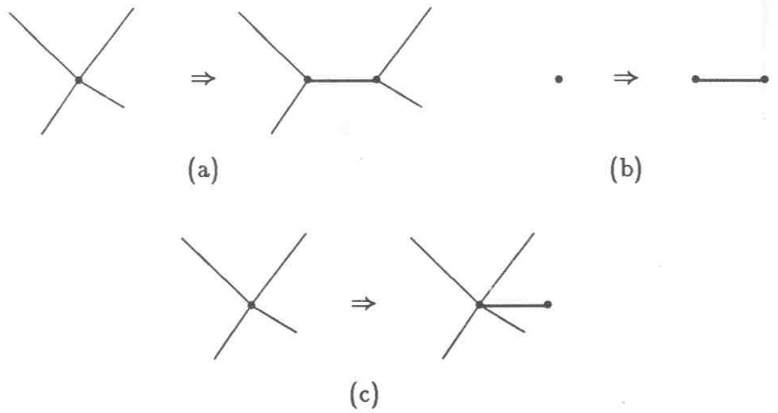


Figure 9.7 The MEV operator.

MEV, KEV

Operators MEV and MEF (to be described in the next section), with their inverses (KEV, KEF) correspond exactly with the polygon and vertex splitting operations for plane models. Hence MEV subdivides the cycle of edges of a vertex into two cycles by "splitting" a vertex into two vertices, joined with a new edge (see Figure 9.7(a)). The net effect of this is to add one vertex and one edge in the data structure, as suggested by the name of the operator.

The applicability of MEV is extended to "lone" vertices (i.e., vertices appearing in empty loops such as the one created by MVFS) by considering the result of subdividing a vertex with no edges at all as consisting of two vertices joined with an edge (see Figure 9.7(b)). We also include the case that a new vertex joined with the old vertex by means of a new edge is created as depicted in Figure 9.7(c).

The inverse operator KEV is capable of undoing any of the three cases of Figure 9.7. In other words, given an edge connecting two distinct vertices, KEV can remove the edge, collapse the vertices into one and merge their edge cycles.

MEF, KEF

The operator MEF subdivides a loop by joining two vertices with a new edge. Its net effect is to add one new edge and face into the data structure.

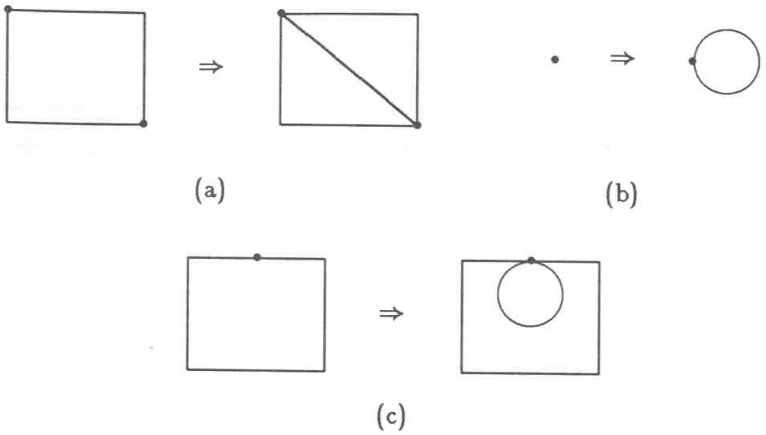


Figure 9.8 The MEF operator.

In addition to the ordinary case of Figure 9.8(a), we also extend the applicability of MEF to empty loops in strict analogy with MEV. Hence the result of subdividing a "lone" vertex consists of a "circular" edge separating two faces. Observe that the starting and final vertices of such an edge are equal; this is a case admitted by the winged-edge data structure (Figure 9.8(b)). As a more general case of this, it is always possible to "connect" a vertex to itself by means of MEF as depicted in Figure 9.8(c).

Again, the inverse operator KEF can undo the effect of MEF in each case. More accurately, given an edge adjacent to two distinct faces, KEF is capable of removing the edge, and joining the two faces into one whose bounding loop is the result of merging the two original boundaries.

The reader is urged to contemplate Figures 9.7 and 9.8 until their analogy is clear to him. Note also how the descriptions of KEV and KEF are "duals" to each other.

KEMR, MEKR

Operators MEV and MEF are "fundamental" in that their significance can be derived directly from the theory of plane models. In contrast, the remaining local manipulation is a "convenience" operator whose necessity is due to our conventions rather than the underlying theory.

In the above, "empty loops" were used to realize the skeletal plane mode in the boundary data sturcture domain. To make full use of empty loops, it is useful to introduce an operator specially tailored for their creation.

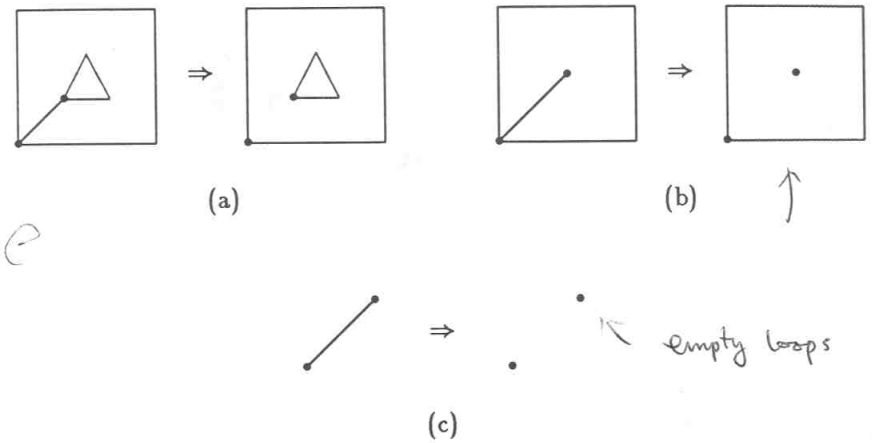


Figure 9.9 The KEMR operator.

The resulting operator KEMR splits a loop into two new ones by removing an edge that appears twice in it (see Figure 9.9(a)). Hence KEMR divides a connected bounding curve of a face into two bounding curves, and has the net effect of removing one edge and adding one ring to the data structure. The special cases that one or both of the resulting loops are empty are also included (Figure 9.9(b, c)).

The inverse operator MEKR can merge two loops of a face by joining one vertex of each with a new edge.

9.2.4 Global Manipulations: KFMRH, MFKRH

None of the operators discussed so far is capable of modifying the global topological properties of the data structure e.g. by dividing a solid into two components or by creating a "hole." This is the task of the remaining Euler operator.

Operator KFMRH is a realization of the connected sum operation discussed in Section 9.1.2. Given two faces (say, f_1 and f_2), it joins them into one face by transforming the bounding loop of f_2 to a ring of f_1. Hence its net effect is to remove one face (f_2) and add one ring. Note that KFMRH has no effect on the local arrangement of edges and vertices of its argument faces—it is truly a global manipulation. This also means that it is hard to illustrate its effect. One attempt is given in Figure 9.10(a).

KFMRH is actually a misnomer, because the operator does not neces-

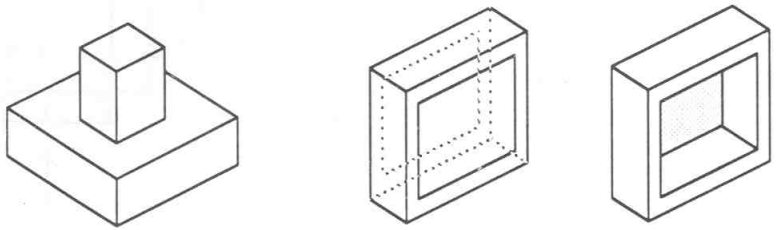


Figure 9.10 The KFMRH operator.

sarily create a "hole". Actually, KFMRH creates a hole only if the two argument faces belong to the same shell. When applied to faces belonging to two distinct shells, its effect is to combine them into one shell, and the name "KFSMR" (where S now stands for "shell") would seem a better condensed description of what really takes place (see Figure 9.10(b)). But neither of the two names is better than the other, and we shall stick to KFMRH.

Again, an inverse operator MFKRH is capable of reversing the effect of KFMRH. It modifies a ring of a face as the bounding loop of a new face.

Recall that the connected sum operation for plane models was specified to add a new edge connecting the two polygons to be joined, while KFMRH achieves the same effect by making the boundary of one face a ring in the other. Otherwise, the operations are strictly analogous.

9.3 AN EXAMPLE OF EULER OPERATORS

Let us clarify the use of Euler operators by means of a simple example, the construction of a rectilinear box with one rectilinear hole. The construction is depicted in Figure 9.11 in terms of both plane models and three-dimensional space models.

The construction starts by creating the skeletal model by means of a MVFS (a). By applying MEV three times, three edges and vertices are added so as to create (b). By joining the end vertices of the edge string (b) with a MEF, the model (c) is created. Note that the space model of (c) consists of two planar faces tightly upon each other like the two sides of a piece of paper. Such a model is called a *lamina*.

Four MEV's lead us to the model (d) that has the side edges of the emerging box. Their tips are joined with four MEF's to form the side faces

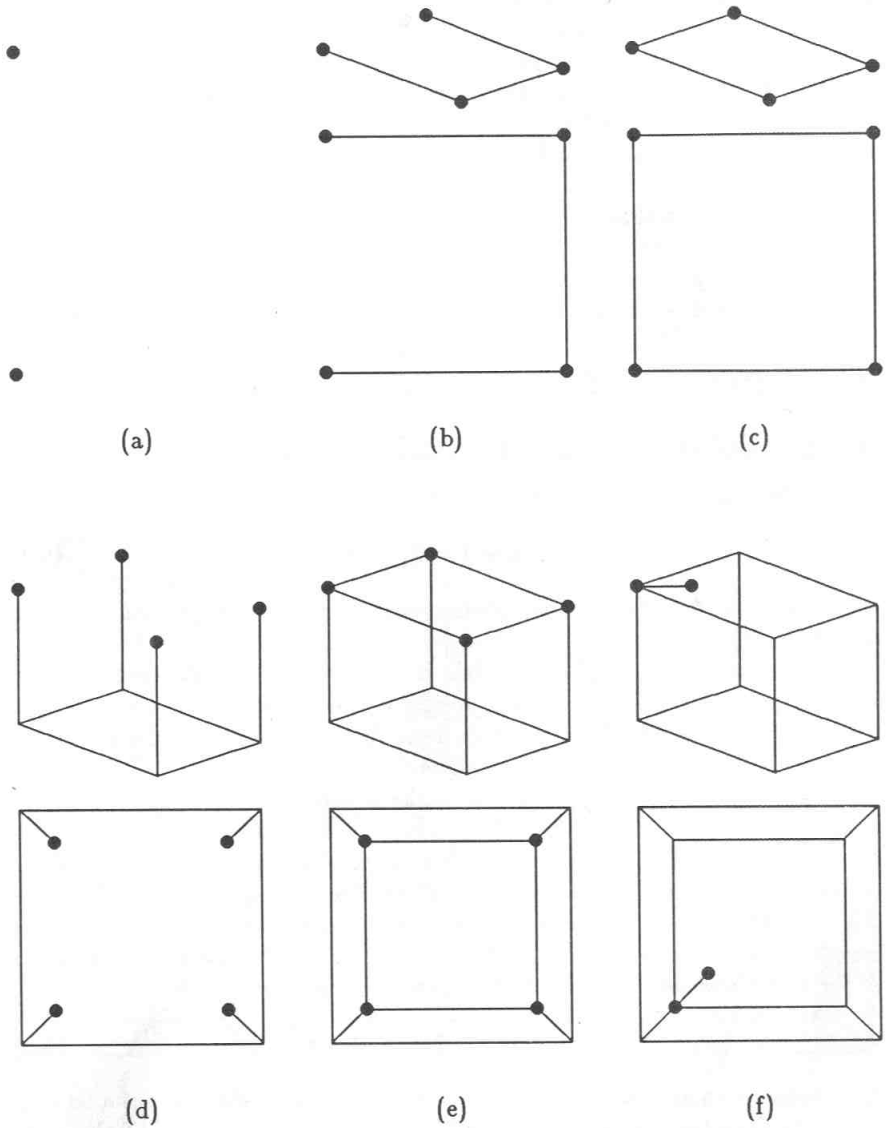


Figure 9.11 Example of Euler operators.

(e). We now have formed the model of a simple box.

To create the hole, we first need to form an interior face on the top of the box (e). We do that by adding an edge on the top face with a MEV (f), and removing the edge with a KEMR (g). This gives us an empty loop with one vertex only inside the top face. After three additional edges are drawn from that vertex with MEV's (h), the interior face can be formed with a MEF (i). Now the side faces of the hole can be formed with MEV's (j) and MEF's (k). Steps (h) through (k) are strictly analogous to (b) through (e).

We now have formed all edges and vertices of the desired result. To finish the description, the bottom face of the emerging hole is "subtracted" from the bottom face of the box by means of a KFMRH (l). This is the same operation as that depicted in Figure 9.10(a).

Of course, to actually create the box, we need to assign coordinates to vertices as they are created. This can be accomplished by operators that create vertices (i.e., MVFS and MEV).

9.4 PROPERTIES OF EULER OPERATORS

9.4.1 Euler-Poincaré Formula Revisited

Recall the Euler-Poincaré Formula 3.2 of Section 3.5.6:

$$v - e + f = 2(s - h), \tag{9.1}$$

where v, e, f, s, and h denote the numbers of vertices, edges, faces, shells, and holes in a solid.

From the theory of plane models it it clear that a collection of faces, edges, and vertices can be a valid boundary model only if the numbers of these elements satisfy Equation 9.1. Hence it can be considered a necessary integrity criterion for boundary models.

Equation 9.1 can be modified to make it consistent with our boundary data structure conventions by introducing a new parameter r, standing for the number of rings. Obviously, each ring can be removed by introducing a new edge that connects it into the external bounding loop of its face (see Figure 6.7). No new vertices or faces are added in this operation, but the number of edges after the removal of r rings will be $e' = e + r$. Substituting e' for e in Equation 9.1 gives us the desired form

$$v - e + f = 2(s - h) + r \tag{9.2}$$

that describes a necessary condition for the numbers of elements of a boundary data structure (v, e, f, r), and the global characteristics of the solid modeled (s, h).

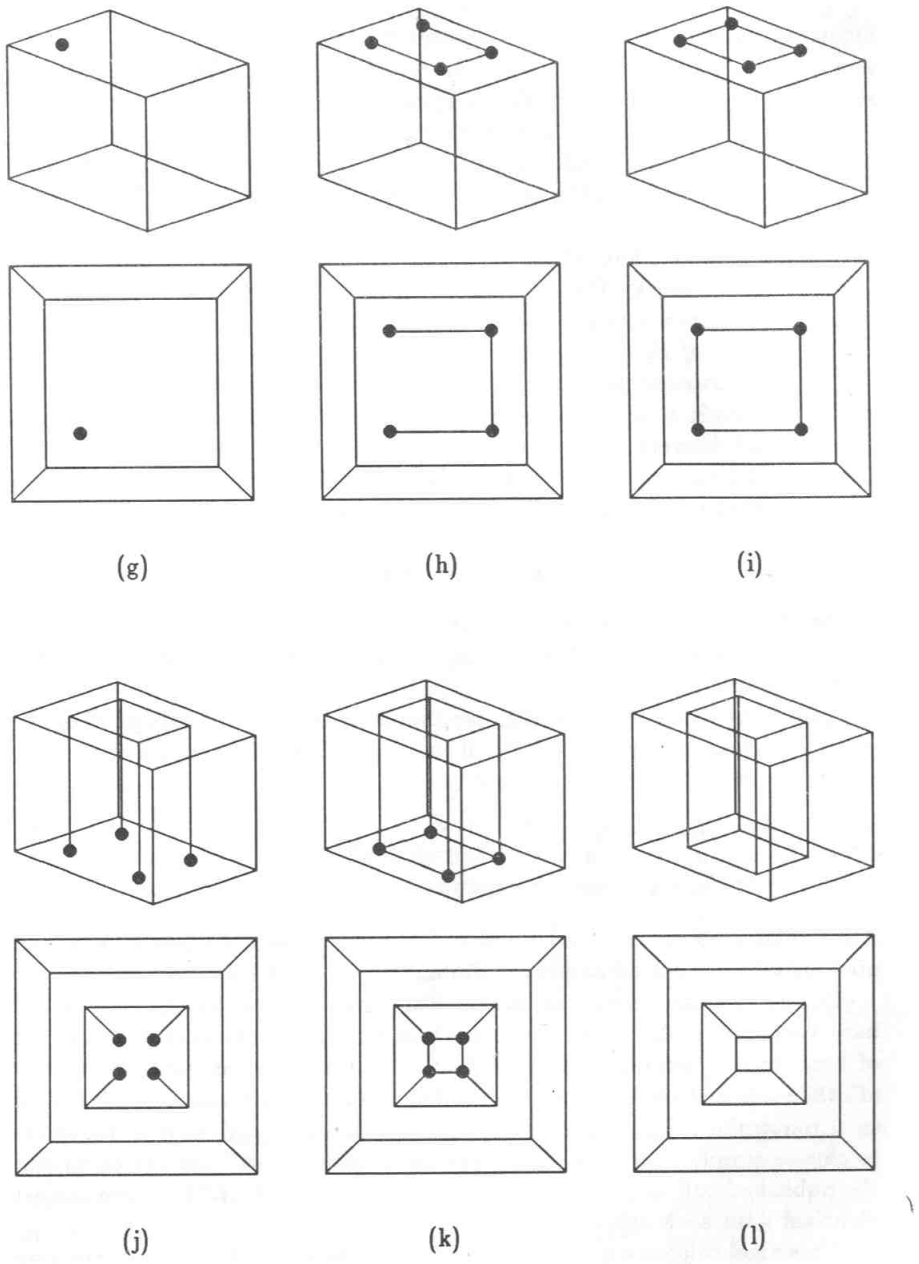


Figure 9.11 Example of Euler operators (cont.).

9.4.2 Algebraic Properties of Euler Operators

Observe that the data structure created by MVFS has $v = f = s = 1$ and $e = h = r = 0$. By substituting these numbers into Equation 9.2, we can see that they make it hold. Furthermore, all other operators will add or remove entities of the data structure in a fashion that keeps the condition expressed by Equation 9.2 satisfied. For instance, MFKRH adds one face and removes a shell and a ring. In terms of Equation 9.2, its effect is to replace f by $f' = f + 1$, r by $r' = r - 1$, and h by $h' = h - 1$. By substituting f', r', and h' into 9.2 one can readily see that the formula will remain satisfied. Hence Euler operators have the property that *the numbers of elements in each data structure created by them will satisfy the Euler formula of Equation* 9.2.

This observation suggests that Euler operators could be analyzed in purely algebraic terms. Following Braid et al. [19], let us consider a six-dimensional discrete space, and call its axes v, e, f, h, r, and s. Then Equation 9.2 can be interpreted as the equation of a five-dimensional hyperplane E of the space. In this view, the analogy of a boundary model is a point

$$P = (v\ e\ f\ h\ r\ s),$$

of the six-dimensional space and the condition of Equation 9.2 can be interpreted as stating that "interesting" points of the space are those lying on the plane E.

The hyperplane E is itself a five-dimensional discrete subspace of the six-dimensional space. Therefore, it can be spanned by five six-vectors V_i satisfying the following conditions:

1. Each vector V_i lies on E.
2. Vectors V_i are linearly independent.

After such a set of base vectors of E has been chosen, all points of E can obviously be expressed as linear combinations of the base vectors.

In the analogy suggested above, Euler operators take the role of the base vectors. In this view, our collection would be interpreted as the set of base vectors summarized in Table 9.1. Obviously, an infinite amount of other sets of base vectors are possible, and every set can be interpreted as a particular collection of Euler operators. In practice, it is desirable to choose simple operators that correspond with "short" vectors of E, and the published collections of Euler operators [14,33,34,19,4,78,75] are almost identical with each other.

This algebraic view of Euler operators can be applied to gain interesting insight into their properties. To see how, let us arrange the base vectors

Operator	*Base Vector*					
	v	e	f	h	r	s
MEV	1	1	0	0	0	0
MEF	0	1	1	0	0	0
MVFS	1	0	1	0	0	1
KEMR	0	−1	0	0	1	0
KFMRH	0	0	−1	1	1	0
KEV	−1	−1	0	0	0	0
KEF	0	−1	−1	0	0	0
KVFS	−1	0	−1	0	0	−1
MEKR	0	1	0	0	−1	0
MFKRH	0	0	1	−1	−1	0

Table 9.1 Euler operators.

corresponding with the Euler operators of our collection, together with the coefficients of Equation 9.2 into the matrix M given below:

$$M = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & -1 & 1 & 1 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 1 & -1 & 1 & 2 & -1 & -2 \end{bmatrix} \tag{9.3}$$

Observe that the coefficient vector on the last row of M acts as the normal vector of the hyperplane E. Because Euler operators form a base of E, the matrix has an inverse. The inverse matrix M^{-1} is given below.

$$M^{-1} = \frac{1}{12} \begin{bmatrix} 9 & -5 & 2 & -2 & 3 & 1 \\ -3 & 5 & -2 & 2 & -3 & -1 \\ 3 & 7 & 2 & -2 & 3 & 1 \\ -6 & 2 & 4 & 8 & -6 & 2 \\ 3 & 5 & -2 & 2 & 9 & -1 \\ -6 & -2 & 8 & 4 & -6 & -2 \end{bmatrix} \tag{9.4}$$

The inverse matrix can be used to calculate the "Euler coordinates" of any $[v\ e\ f\ h\ r\ s]$-tuple. For instance, the object constructed in Figure 9.11 has $v = 16$ vertices, $e = 24$ edges, $f = 10$ faces, $r = 2$ rings, $h = 1$ hole, and $s = 1$ shell. Hence it corresponds with the location

$$P = [16\ 24\ 10\ 1\ 2\ 1]$$

of the discrete six-dimensional space. Now the product

$$P \times M^{-1} = [15\ 10\ 1\ 1\ 1\ 0]^T$$

tells us that to model this solid, a total of 15 MEV's, 10 MEF's, 1 MVFS's, 1 KFMRH's and 1 KEMR are needed at the minimum. That the last component is zero shows that the original 6-tuple satisfies Equation 9.2. Observe that the construction of Figure 9.11 uses exactly these numbers of operators; hence, it it "optimal" in the length of the construction.

The algebraic analysis gives us hence a mental tool for discussing the complexity of Euler operator descriptions of boundary models. As another example of this, if $s = 1$, the general product

$$[v\ e\ f\ h\ r\ s] \times M^{-1}$$

yields the following operator counts:

$$\begin{array}{ll} v-1 & \text{MEV's} \\ f+h-1 & \text{MEF's} \\ 1 & \text{MVFS} \\ h & \text{KFMRH's} \\ r-h & \text{KEMR's} \end{array} \qquad (9.5)$$

Clearly, Equations 9.5 give the smallest numbers of operators needed to model an object corresponding with location $[v\ e\ f\ h\ r\ 1]$ of the six-dimensional discrete space.

Unfortunately, the inverse operators count as -1 in Equations 9.5, which makes their interpretation difficult. Nevertheless, because all models of interest have $v, f > 0$ and $f > h$, the four first counts are nonnegative. This suggests that all connected objects can be modeled without the "inverse" operators KEV, KEF, KVFS, and MFKRH, a fact that will be shown in Chapter 16. Also, from the fact that the last line can be negative, we can infer that some models cannot be created without the MEKR operator.

9.4.3 Descriptive Power of Euler Operators

Euler operators remove some part of the complexity of dealing with boundary data structures. Does this impose a penalty as far as the generality and flexibility is concerned? Is it possible to create arbitrary models under restriction that only Euler operators are used in their construction?

The algebraic view to Euler operators presented in the last section gives us some insight into these questions. Obviously, each 6-tuple $[v\ e\ f\ h\ r\ s]$ can be associated with a tuple of "Euler coordinates" through the inverse of the transition matrix M. But there is still some concern, however. As the

reader can easily verify with a counterexample, a 6-tuple does not uniquely determine a solid. Can we produce all objects that have the same tuple? Furthermore, while the discussion of the last section suggests that the number of operators needed to model a solid grows linearly with the complexity of the object, it is not completely clear whether an operator sequence of that length exists for all objects, because inverse Euler operators contribute as -1 to the counts given by the inverse transition matrix.

The theory of plane model manipulations presented in Section 9.1 resolves the first question. In particular, Theorems 9.5 and 9.7, and Corollary 9.6 can be reinterpreted in the domain of boundary data structures and Euler operators. In this new form, the theorems are as follows:

Theorem 9.8 *Let S be a valid boundary data structure (i.e., S satisfies all topological integrity constraints). Then there exists a finite sequence of Euler operators that can completely remove S.*

Corollary 9.9 *All valid boundary data structures can be created with a finite sequence of Euler operators.*

The direct proofs are analogous to those of Theorem 9.5 and Corollary 9.6. In Chapter 16, we shall give a constructive proof by developing an algorithm that actually constructs such a sequence of operators.

Theorem 9.10 *Euler operators are* sound, *i.e., they cannot create topologically invalid boundary data structures.*

Again, the direct proof is similar to that of Theorem 9.7. As an additional case, the soundness of KEMR and MEKR would have to be demonstrated.

Two notes on the soundness theorem may be needed. First, to model anything at all, a sequence of operators must satisfy certain "syntactic" validity criteria pertaining to a particular way of representing Euler operators. For instance, all entities that operators work on must exist and be of proper type. Say, the operator KEF can be applied only to an edge that appears in two distinct faces.

Second, the theorem has nothing to say on the *geometric* validity of the resulting model; by assigning inappropriate geometric information to faces, edges, and vertices, it is still perfectly possible to create invalid models that have no physical significance.

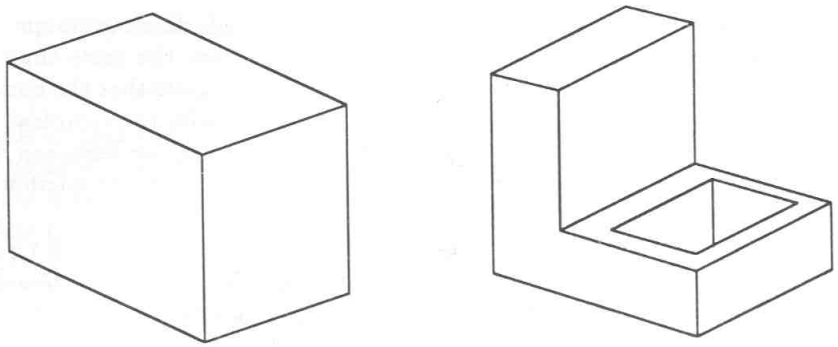


Figure 9.12 Sample solids.

PROBLEMS

9.1. Complete the proof of Theorem 9.7 on page 145.

9.2. Give an Euler operator construction in the style of Figure 9.11 for the following objects (see Figure 9.12):

(a) A rectilinear block with six faces and eight vertices.

(b) An "L-brick with hole" with 12 faces and 20 vertices.

9.3. Show that some solids cannot be described without the MEKR operator.

Hint: Equations 9.5 suggest that for such an object, $h > r$.

9.4. Provide an example of two different boundary models having the same $[v\ e\ f\ h\ r\ s]$-tuple.

9.5. Demonstrate the soundness of the Euler operators KEMR and MEKR.

Hint: You will have to show that they preserve the conditions of edge identification, cyclical identification, and orientability.

BIBLIOGRAPHIC NOTES

The idea of using plane models to argue rigorously about the formal properties of Euler operators was originally presented in the article [75] of the author. Most of the material presented in this chapter stems from this publication, but appears here in extended and clarified format.

The discussion on algebraic properties of Euler operators presented in Section 9.5.2 is based on the exposition of Braid et al. [19] with some minor changes in details.