In Helsinki, a small but general set of manipulative operations for boundary models of solid objects has been used to construct a comprehensive solid modeling system.

GWB: A Solid Modeler with Euler Operators

Martti Mantyla and Reijo Sulonen
Helsinki University of Technology

Any software system making use of three-dimensional computer graphics must include some means for representing three-dimensional objects. Conventional generative graphics uses rather straightforward wireframe models consisting of lines and points. While these models allow easy construction of line figures, they are not adequate for more general operations. For instance, a wireframe model is not generally sufficient for calculating the volume of an object or for eliminating hidden lines.

These problems are emphasized in computer-aided design systems. A CAD system for mechanical engineering, for instance, should include a variety of advanced operations, such as engineering analysis, drafting, and computer-aided manufacturing. All these operations should be based on a single representation or on multiple but consistent representations created by the user of the system.

Object representations in CAD must be capable of generating all data needed during the design cycle of an object. Since the geometric properties of the object form the basis for engineering analysis and CAM operations, it is natural to separate nongeometric data from data dealing with the geometric shape of the object—the *geometric model.*

To ensure that the information stored in a geometric model is potentially sufficient for calculating any geometric properties, the model should be an unambiguous representation of the object.[1] This means that each model designates a unique physical solid. The geometric models created by a *solid modeler* satisfy this requirement.

There are two main approaches to solid modeling: constructive solid geometry[2] and boundary representation. In a CSG model, solids are described by Boolean combinations of basic solids; in a BR model, solids are described by a collection of faces, which in turn are represented by their bounding edges and vertices.

The choice between these alternatives depends on the type of operations the solid modeler is expected to support. Boundary modelers tend to support stepwise construction of the models more easily than CSG modelers but require greater data storage.

The Geometric Workbench, an experimental solid modeling system we have developed at the Helsinki University of Technology, uses a variation of the BR approach. Our building blocks in GWB are a set of "atomic" functions called the *Euler operators.*[3,4] These functions allow the incremental manipulation of BR models, while restoring the "well-formedness" of the underlying data structures. We describe several algorithms based on the use of Euler operators, with special emphasis on an inversion algorithm that parses a given well-formed model into a sequence of Euler operators.

Boundary representations of solids

Mathematically speaking, boundary models describe an object by its topological boundary. The boundary is represented as a collection of faces, which in turn are represented by their bounding edges and vertices. We use the convention that faces need not be simply-connected but may include cavities; the components of the boundary of a face are referred to as *loops.*

Usually faces are chosen so that the geometric form of each face can be represented as a single (piecewise) parametric function. The modeling space[1] of a BR model is determined by the face geometries allowed. The modeling space considered in this article is the set of *basic solids,* which may be defined as three-dimensional rectilinear polyhedra whose boundaries are connected manifolds. In this case each face is a planar polygon whose shape is determined by the coordinates of the vertices of its loops.

One of the loops represents the "outer" boundary of the face, while the others describe its cavities. We shall use the term *ring* to denote an interior loop; for instance, a face with one cavity is said to have one ring.

A boundary model might be represented simply as a set of multiconnected polygons. Each loop of each polygon would then be represented as a list of vertices. To speed up various operations on the model, boundary modelers frequently store other explicit data on the connections between faces, loops, edges, and vertices of the solid. These data are commonly referred to as the *topology* of the model, while data such as face equations or vertex coordinates are referred to as the *geometry* of the model.

With regard to explicit topological information, a large number of variants of boundary representations exist. Baer, Eastman, and Henrion[5] enumerate nine distinct topological relations which may be included in a particular representation. The choice among alternatives depends on the mixture of operations desired for the model.

Not all collections of faces define a valid physical solid. The *topological integrity* of a BR model imposes restrictions on face collections to ensure the validity of the model.[6] For instance, each edge must belong to exactly two (not necessarily distinct) loops, and the ordering of edges in loops must be consistent throughout the model. To satisfy the latter condition, outer loops can be specified to be oriented clockwise (as viewed from outside the solid), and rings can be oriented counterclockwise. *Geometric integrity* is satisfied when the shape assigned to faces is consistent with the topological information; for instance, faces may intersect only at common edges or vertices, if at all.

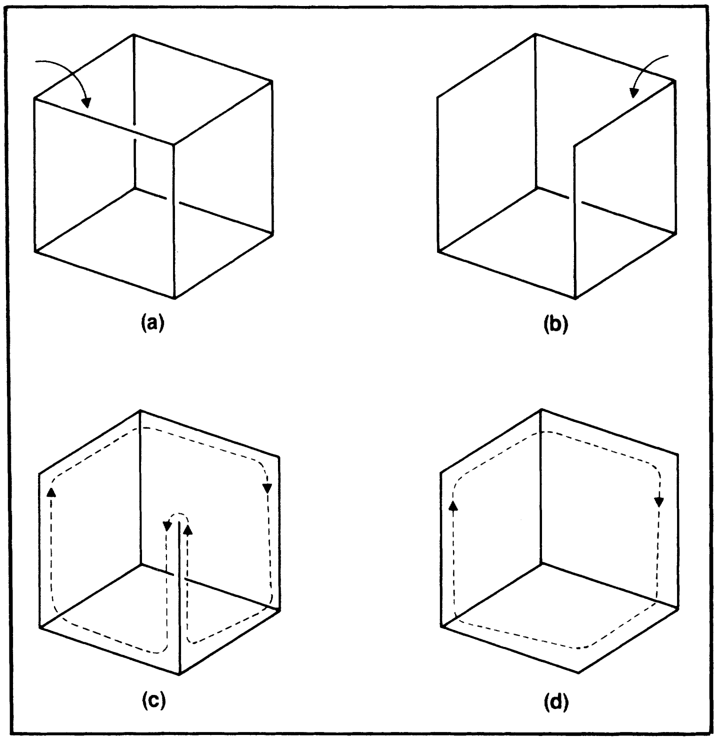


Figure 1. A cube (a), after removal of one edge (b), a second edge (c), and a vertex and third edge (d). Note that each of the operators restores the validity of the generalized Euler formula.

We do not assume any particular boundary representation in this article. Furthermore, since any topological relation may be derived from an unambiguous BR model, we freely use various topological relations. Given a solid, we assume that its faces and edges can be accessed. For each loop of each face, we assume that the vertices and edges occurring in the loop can be accessed in consistent order. Finally, we shall assume that the two loops of an edge can be obtained. For an edge E, we denote these data by E.loop1 and E.loop2. Similarly, we assume that the two vertices of E, denoted by E.vertex1 and E.vertex2, and the two faces of E, denoted by E.face1 and E.face2, can be accessed.

Atomic solid-modification operations

As indicated above, any boundary representation is a data structure consisting of faces, loops, edges, and vertices. Since these data structures are often very complex, creating them is laborious and error-prone. For instance, the object depicted later in Figure 9 includes 79 faces, 231 edges, and 154 vertices.

Fortunately, the modifications of these models can be divided into simple atomic operations. Let us consider the cube represented by six faces, twelve edges, and eight vertices as shown in Figure 1(a). Consider for a moment the effect of removing the edge marked by an arrow. In the resulting object, Figure 1(b), the two faces which meet at the edge are united and the remaining collection of five faces, eleven edges, and eight vertices is generated—a simpler model. Of course, the geometry of the face created by the uniting operation is no longer planar.

Performing the same operation on the edge marked in Figure 1(b) creates the object in Figure 1(c). It has an edge belonging twice to the loop indicated by the dashes. The "upper" vertex of the edge is adjacent to no other edges. Another operation removes such a combination of an edge and a vertex and creates the model in Figure 1(d). These two operations can remove all edges and all vertices except one. A third operation can remove the remaining stripped-down, one-vertex model.

Obviously, each "removal" operation has an inverse "creation" counterpart. While the operations described above can be used to *destroy* models of solids, these positive operations can be used for the perhaps more interesting purpose of *creating* models. So the inverse of the third removal operation creates an "initial object." The inverses of the two other operations add new vertices, edges, and faces to create models of any complexity. Together the destructive and the creative operations allow us to perform arbitrary modifications necessary in BR models.

Euler operators

The operations informally described in the previous section are examples of Euler operators.[3,4] We call the three destructive operators "kef," "kev," and "kvsf," for "kill edge and face," "kill edge and vertex," and "kill vertex, solid, and face." Similarly, their inverses are

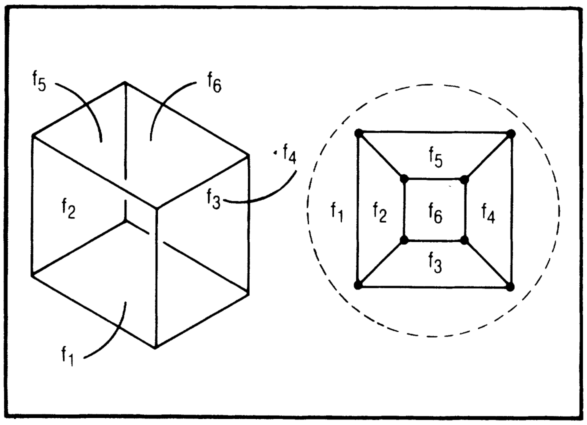


Figure 2. Cube topology: a plane model.

Table 1.
Euler operators.

OPERATOR	EXPLANATION
mvsf(f,v)	MAKE VERTEX, SOLID, FACE
kvsf()	KILL VERTEX, SOLID, FACE
mev(v_1,v_2,e)	MAKE EDGE, VERTEX
kev(e,v)	KILL EDGE, VERTEX
mef(v_1,v_2,f_1,f_2,e)	MAKE EDGE, FACE
kef(e)	KILL EDGE, FACE
kemr(e)	KILL EDGE, MAKE RING
mekr(v_1,v_2,e)	MAKE EDGE, KILL RING
kfmrh(f_1,f_2)	KILL FACE, MAKE RING, HOLE
mfkrh(f_1,f_2)	MAKE FACE, KILL RING, HOLE
semv(e_1,v,e_2)	SPLIT EDGE, MAKE VERTEX
jekv(e_1,e_2)	JOIN EDGES, KILL VERTEX

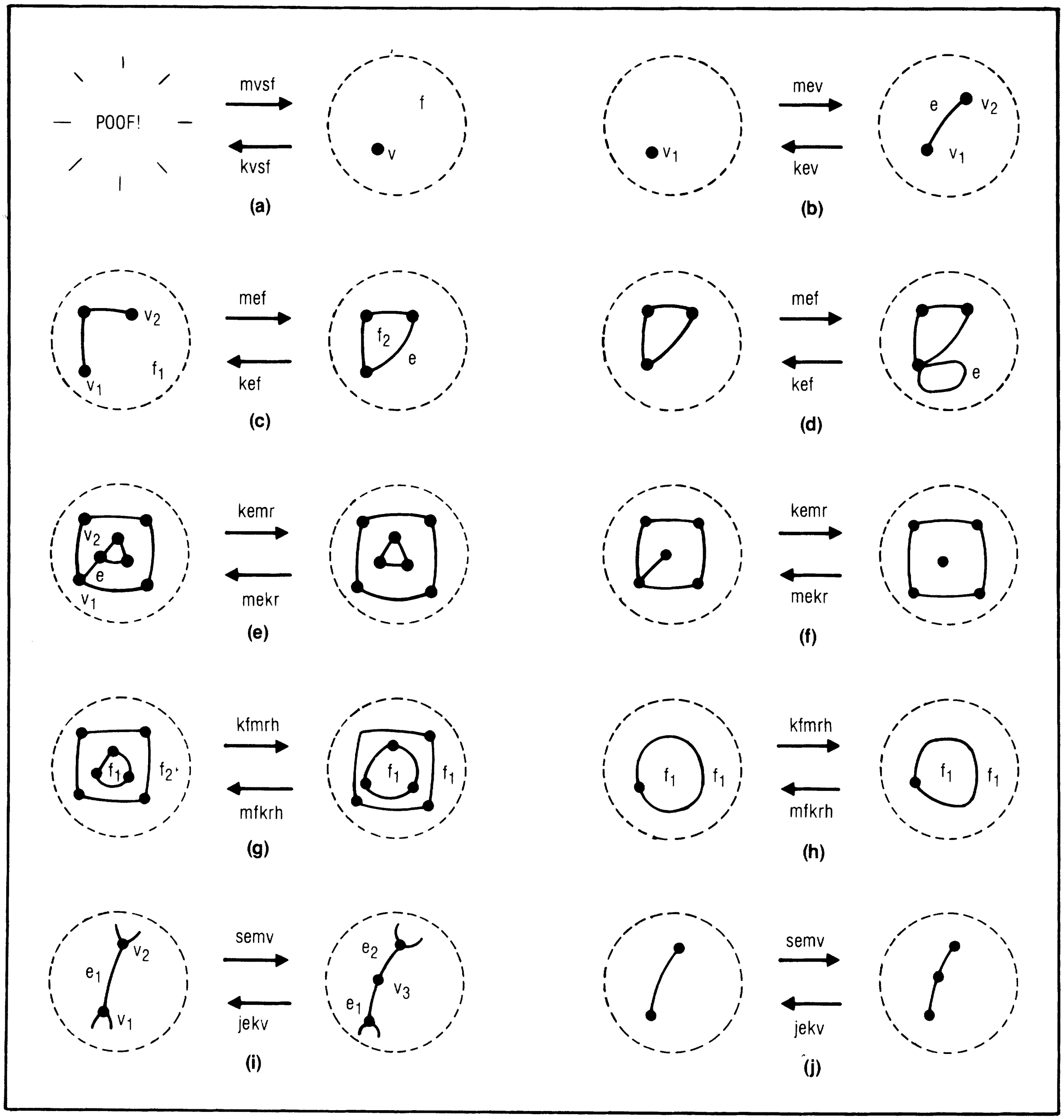


Figure 3. Euler operators: plane models.

called "mef," "mev," and "mvsf," where "m" stands for "make."

Euler operators derive their name from the well-known Euler's Law: in any simple polyhedron, the numbers of faces (f), edges (e) and vertices (v) must satisfy the equation

$$v - e + f = 2.$$

The formula may be generalized to arbitrary solids by introducing three other parameters, namely (1) the total number of rings (cavities in faces, r) in the solid, (2) the total number of holes (h) through the solid, and (3) the number of disconnected components (s) in a collection of solids. The general formula is

$$v - e + f = 2(s-h) + r.$$

As noted by Braid, Hillyard, and Stroud,[7] just five operators (with their inverses) are sufficient for describing all objects satisfying the Euler formula. While these five may be chosen in several ways, considerations of modularity and mutual independence have led to only small variations in the collections appearing in the literature.[4,7,8]

We list the operators chosen for this project in Table 1. The collection includes the three operators already discussed and two other operator-inverse pairs related to creation of rings (kemr, mekr) and holes (kfmrh, mfkrh). To aid geometric operations, an auxiliary "split-edge" operator is included (semv, jekv).

Operators creating faces, edges, and vertices assign identifiers for them; in the table these output parameters are in italics. Although not shown in the table, vertex coordinates are stated when a vertex is created. Additional orientation parameters must be given when the parameters listed are not sufficient to specify the desired effect unambiguously.

Euler operators work on the topology of a boundary model—that is, on the relative arrangement of its faces, edges, and vertices. To describe the effects of the operators, we can represent the topologies they create by planar graphs that we call *plane models*. The nodes and arcs of the graph represent the vertices and edges, and the planar areas bounded by the arcs of the graph represent faces. For instance, the topology of a cube is represented by the graph of Figure 2. Plane models of each of the Euler operators are shown in Figure 3. Note that in Figure 3(f), kemr creates a special case of a loop that has no edges at all. A similar loop is also created by the mvsf operator in Figure 3(a).

The effects of the kfmrh/mfkrh operators (Figures 3(g) and (h)) are not easily represented by plane models, since they affect only the labeling of the areas. Effectively, kfmrh transforms the boundary of one face (f_2) into a ring of another (f_1) and ordinarily creates a hole through the object. This is pictured in Figure 4(a). Operators kfmrh/mfkrh modify the *topological genus* of the solid—the number of holes through it. The same operators can also modify the *connectivity* of the solid— the number of components in it. For instance, the "gluing" of two touching cubes may be accomplished by joining their adjacent faces by the kfmrh operator, as depicted in Figure 4(b). Thus kfmrh really decreases the value of the expressions ($s-h$) by unity either by removing a solid or by adding a hole, while mfkrh increases it by the reverse operation. We make use of kfmrh/mfkrh as in Figure 4(b) when developing solid-manipulation algorithms in the next section.

For further clarification, we represent in Figure 5 the creation of a cube with a hole. The "exterior" cube is created with operations (a) through (f). The "bottom" of the hole to be drilled is created with operations (g) through (i), while its side faces are created by operations (j) through (k). The last operation (kfmrh) is also shown in Figure 4(a). For brevity, the parameter lists of the operators have been excluded.

Analysis of the plane models shows that Euler operators guarantee the topological validity of BR models.[6] For instance, each model is consistently oriented. On the other hand, in the algorithms using the operators, geometric validity must be verified, since the intermediate geometries generated can be invalid.

In a formal sense, Euler operators form a sufficient set of solid definition and manipulation operations. Even so, a solid modeler must include more user-oriented ways to create and handle solid models. Euler operators themselves are rather unintuitive and primitive and should not be directly visible to the end user. Nevertheless, it is advantageous to use them as primitives of higher-level operations to ensure that the models created are well formed.

In the following section we describe in detail various operations and tools for solid modeling which can be constructed from the primitive Euler operators. To give some practical flavor, we first discuss the operations needed in a solid modeler based on our experience with the GWB system.

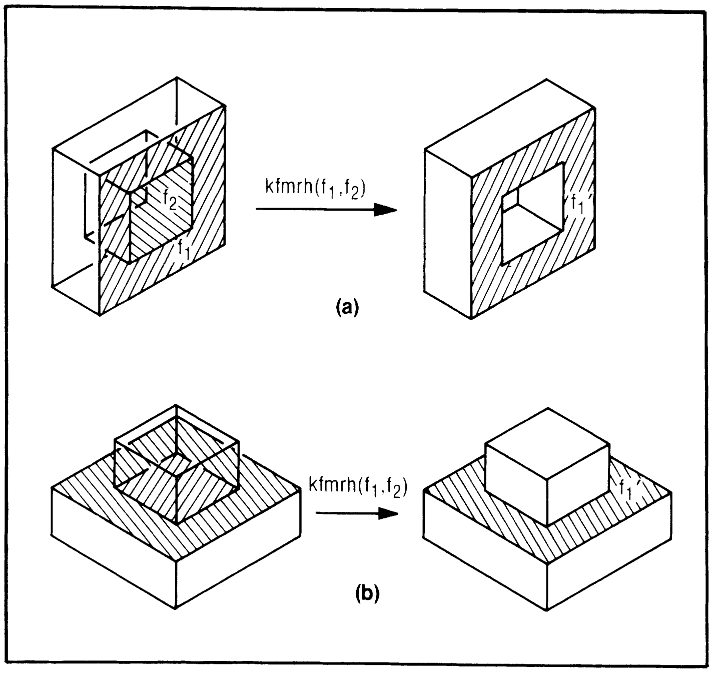


Figure 4. The kfmrh operator: (a) creating a hole and (b) gluing two touching solids.

GWB: structure of a solid modeler

The Geometric Workbench is designed to be an open-ended solid modeler that can be tailored for particular applications. Its structure is diagrammed in Figure 6. GWB provides the basic object manipulation operations in the form of Euler operators, and it provides query functions for scanning the underlying data representation. These primitives allow the creation of extendable tools for solid definition and manipulation.

Solid-definition operations create solid models by means of parameterized functions or drafting-like commands. These models can be manipulated and combined by *solid-manipulation* operations such as the Boolean set operations. Thus definition operations operate on one solid at a time, while manipulation operations operate on collections of solids.

The internal data structures created by these operations are stored in a workspace, from which they can be retrieved and used to create graphic displays or to calculate geometric properties. The objects created by solid-definition operations can be represented in secondary storage simply by the sequence of the definition commands used. When needed, the internal data structure can easily be reconstructed from this external data format.

Unfortunately, the opposite is true for the solid-manipulation operations, which effectively destroy the straightforward correspondence between the external representation of a solid (for instance its Euler-operator sequence) and the internal data structure. It would clearly be beneficial to be able to reconstruct a list of Euler operators that conveys the information of the internal data structure. This brings us to what we call *solid inversion,* which is covered in the last part of this section.

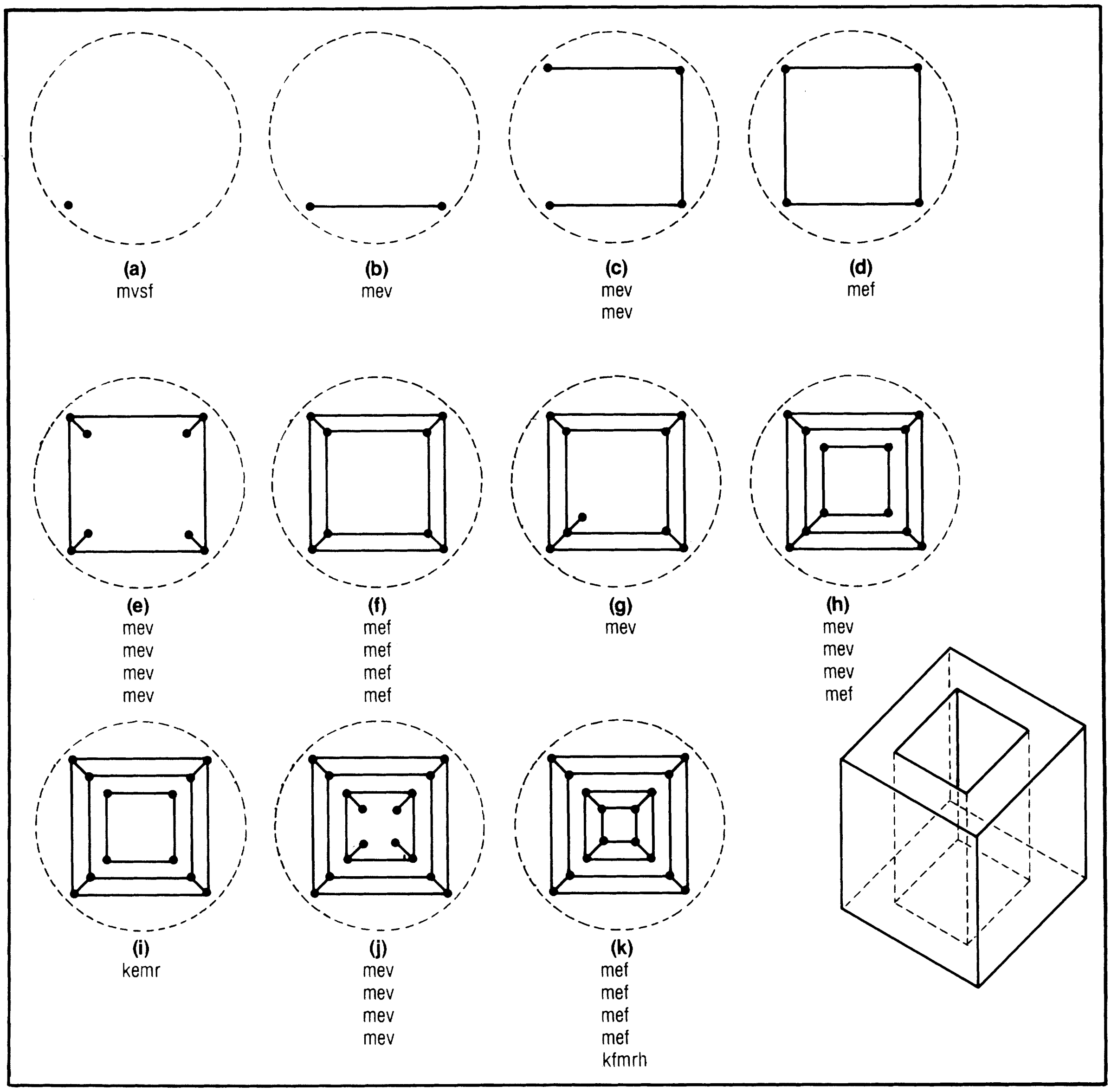


Figure 5. A sample definition.

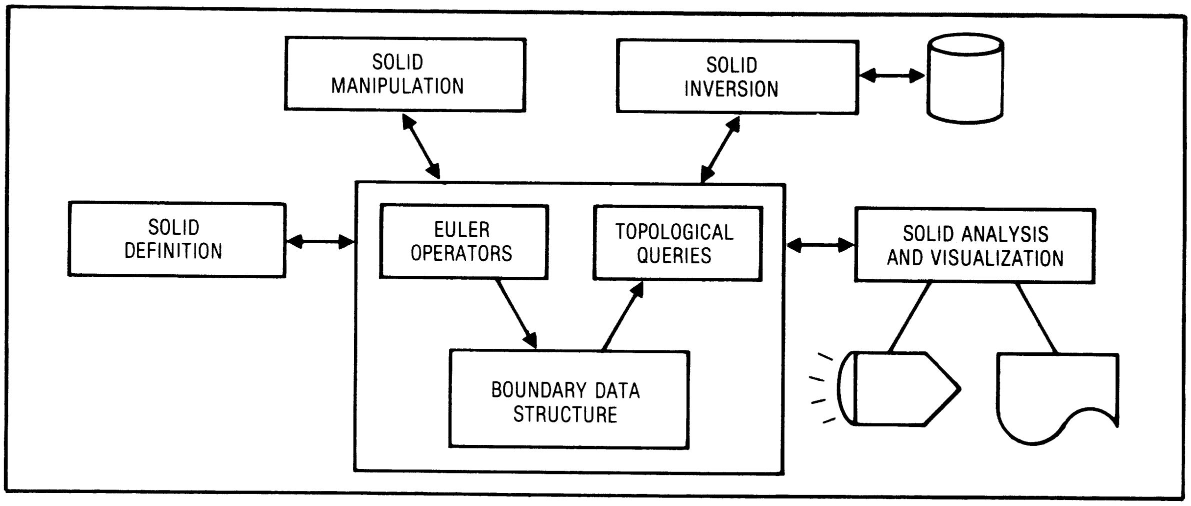


Figure 6. Components of GWB.

```
block(): solid
begin
  /* create bottom face */
  mvsf($f_1$,$v_1$);
  mev($v_1$,$v_2$,$e_1$);
  mev($v_2$,$v_3$,$e_2$);
  mev($v_3$,$v_4$,$e_3$);
  /* create top face*/
  mef($v_1$,$v_4$,$f_1$,$f_6$,$e_4$);
  /* create sides */
  mev($v_1$,$v_5$,$e_5$);
  mev($v_2$,$v_6$,$e_6$);
  mev($v_3$,$v_7$,$e_7$);
  mev($v_4$,$v_8$,$e_8$);
  mef($v_5$,$v_6$,$f_6$,$f_2$,$e_9$);
  mef($v_6$,$v_7$,$f_6$,$f_3$,$e_{10}$);
  mef($v_7$,$v_8$,$f_6$,$f_4$,$e_{11}$);
  mef($v_8$,$v_5$,$f_6$,$f_5$,$e_{12}$)
end
```

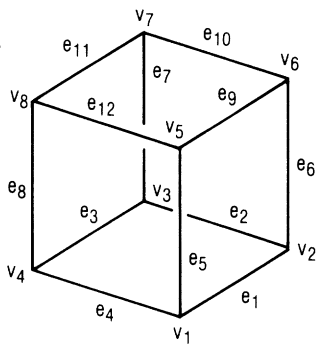


Figure 7. The block creation function.

```
sweep(S:solid, F:face)
begin
  for(all loops L of F) do
  begin
    firstv = first vertex of L;
    mev(firstv, firstup, newe);
    prevup = firstup;
    nextv = next vertex of L;
    while(nextv not equal to firstv) do
    begin
      mev(nextv, up, newe);
      mef(prevup, up, F, newf, newe);
      prevup = up;
      nextv = next vertex of L
    end;
    mef(prevup, firstup, F, newf, newe)
  end
end;
```

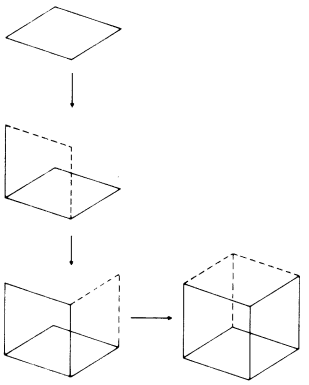


Figure 8. The translational-sweep operator.

Solid-definition operations. Many designs can be constructed from a limited set of basic solids, each of which can be described by a few parameters. For such applications, Euler operators can be packaged into functions that create often needed solids such as rectilinear blocks (Figure 7) and cylinders.

Another useful class of solids can be described by their two-dimensional cross sections. The cross section is constructed from its bounding lines. It is not hard to construct a "drafting" interface that enables the user to construct cross sections interactively with "drawing" commands, which actually call the appropriate Euler operators. In order to construct a true three-dimensional solid, areas of the cross section are "lifted" or "pushed" through with a *translational-sweep* operator. The translational-sweep operator transforms a face of a model into a three-dimensional object by "sweeping" the face a certain distance in a given direction. The sweep operator, also constructed from Euler operators, is given in Figure 8. A *rotational-sweep* operator can be similarly constructed.

With the drafting operations and the sweep operator, it is easy to construct solids such as the one in Figure 9, which serves as an example of the drafting interface implemented in GWB.

Solid-manipulation operations. The class of solids that can be constructed by the solid-definition operations discussed so far is quite limited. The so-called Boolean set operations, which take set union, set intersection, or set difference between their argument solids, make the construction of a wider class of solids quite natural. Unfortunately, algorithms for these purposes are generally very complex.

The set-operation algorithms could be written to manipulate the low-level data structures directly. We believe, however, that the use of Euler operators in these manipulations offers important advantages:

(1) Since Euler operators guarantee the topological validity of the resulting model, an additional "safety" level is reached in algorithms based on them.

```
moveto(0.0, 0.0);
lineto(300.0, 0.0);
arcto(300.0, 64.0, 64.0 -90.0, 90.0);
lineto(214.0, 128.0);
lineto(214.0, 93.0);
lineto(189.0, 93.0);
arcto(189.0, 64.0, 29.0, 90.0, 180.0);
lineto(160.0, 35.0);
lineto(90.0, 35.0);
lineto(90.0, 128.0);
lineto(0.0, 128.0);
lineto(0.0, 0.0);
moveto(255.0, 64.0);
arcto(300.0, 64.0, 45.0, -180.0, 180.0);
moveto(189.0, 86.0);
arcto(189.0, 64.0, 22.0, 90.0, 270.0);
lineto(201.0, 42.0);
arcto(201.0, 64.0, 22.0 -90.0, 90.0);
lineto(189.0,86.0);
sweep(100.0);
```

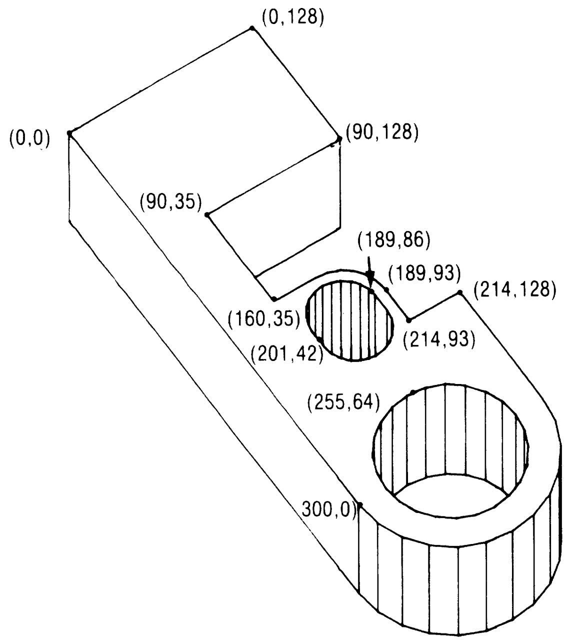


Figure 9. A solid created by drafting operations. The *lineto* operation adds a straight line from the current point to a new point, while *arcto* creates a circular arc defined by its center coordinates (*cx,cy*), radius, and start and end angles in degrees. The *moveto* operation can be used to establish a new CP.

(2) Euler operators effectively hide the actual data structure used. Thus algorithms using them are less dependent on the particular features of the data representation chosen.

(3) The use of Euler operators is economical, since the resulting algorithms share the code of Euler operators and interrogation functions; also, the corresponding program texts become shorter.

To clarify these points, we shall describe an algorithm for a sample geometric task in detail.

The Split Algorithm. Let us define the split problem as follows: given a solid A and a plane P, give the two solids A.above and A.below corresponding to the parts of A above and below P, respectively. In other words, a split algorithm is expected to "saw" a solid with a plane. Such an algorithm is useful for generating section views needed in technical documents. It is natural to require a split algorithm to be *closed* in the class of solids—that is, capable of repeated application to its own products. An algorithm having this property is not restricted to mere visualization. Note that in general, the two resulting solids (Figure 10) may not be connected but may be collections of solids. The split algorithm generally takes a collection of solids and gives two collections of solids. Since the algorithms for solid collections are straightforward versions of the classification algorithms of Tilove,[9] we concentrate on the splitting of one solid only.

For simplicity, we assume that no vertex of the solid lies on the splitting plane. This assumption relieves us of certain types of special cases a general algorithm should consider. Splitting can be performed only by the mfkrh operator, since it is the only operator that can transform a solid into two solids. To apply it, the topology of the solid must first be modified appropriately.

Let us consider the example of Figure 11. The two small cubes can be "glued" to the larger part with two calls of the kfmrh operator by first positioning their plane models onto areas of the plane model of the large solid. Likewise, the "splitting" could be performed by creating the two faces marked in the plane model of Figure 11(b) and applying the mfkrh operator twice.

The idea of our algorithm is to create, by means of other Euler operators, *split faces* similar to the faces marked in the plane model of Figure 11(b), thus allowing the use of the mfkrh operator for each of them. This can be accomplished as follows:

(1) Scan through all edges of the solid and split them by two new vertices to create things we call *null edges* in the intersection points between edges and the plane. This can be carried out by two calls of semv(). See Figures 12(a,b).

(2) Combine null edges pairwise to create *null faces* in the intersection areas. In this step, the "lower" and the "upper" vertices of each pair of neighbor null edges are combined by a new edge using the mef operator. See Figures 12(b,c).

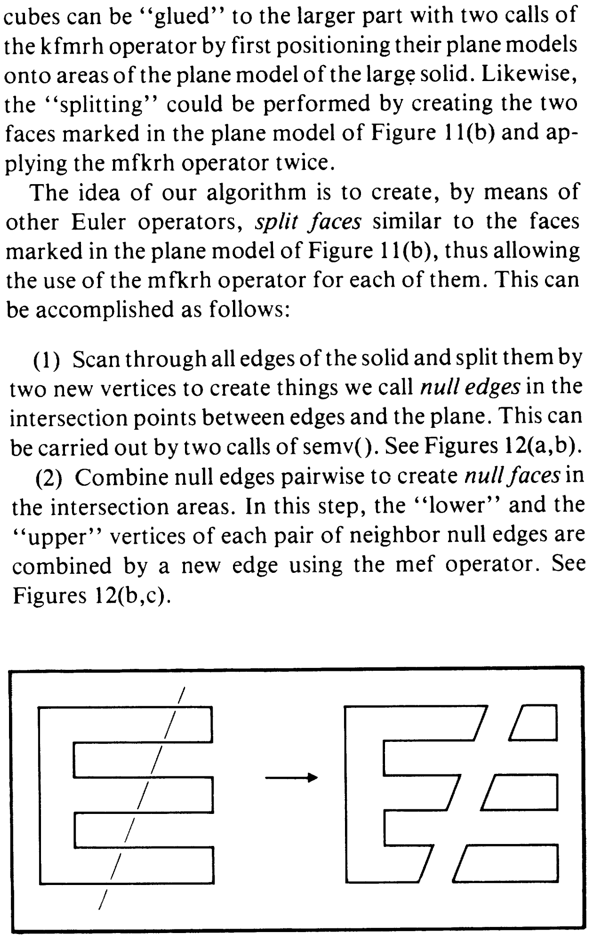


Figure 10. Results of the split need not be connected.

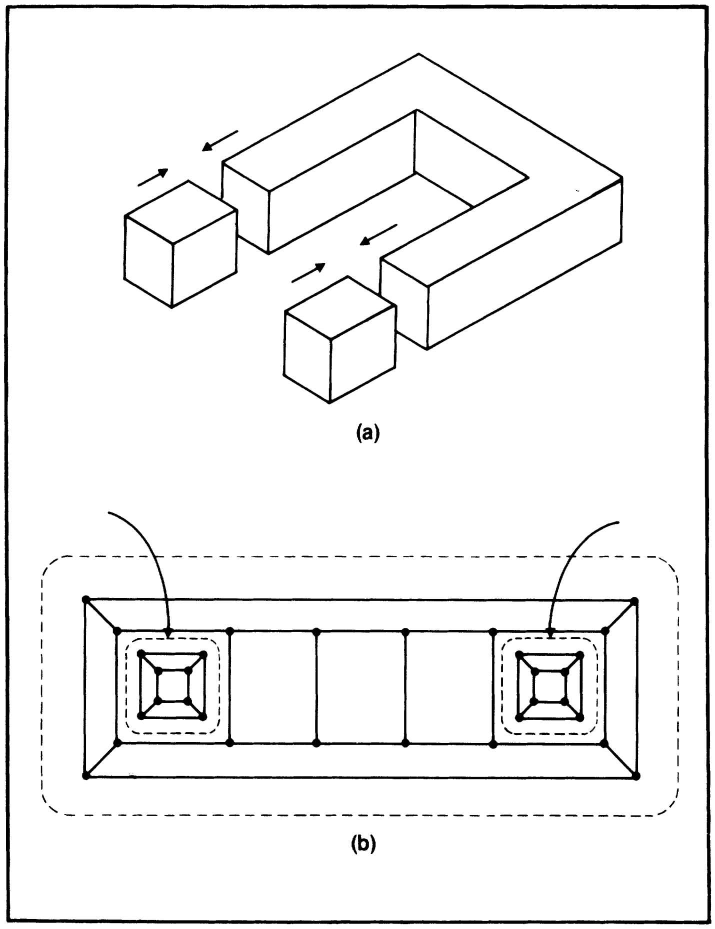


Figure 11. Gluing performed by the kfmrh operator: (a) solid model, (b) plane model.

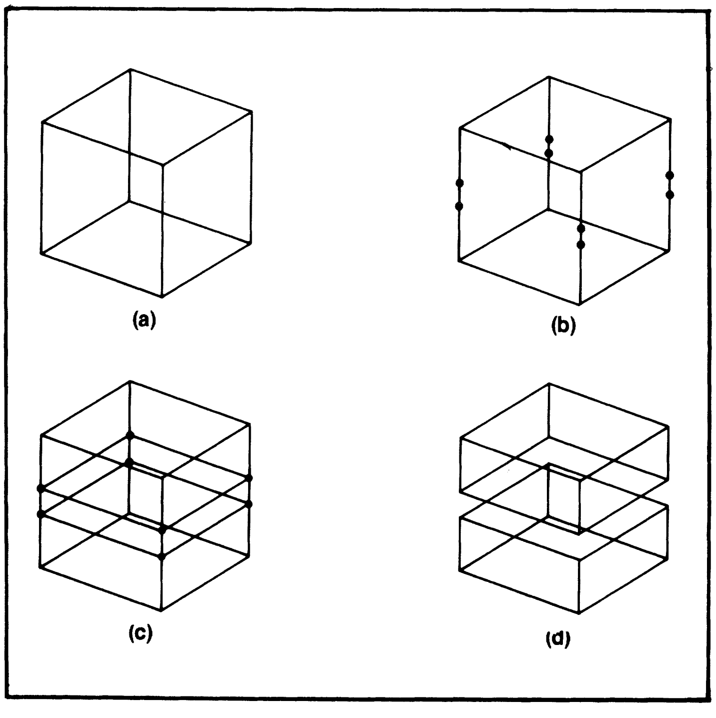


Figure 12. Steps of the split algorithm: (a) the initial solid, (b) null edges and new vertices, (c) null faces, (d) split faces.

(3) Join null faces by removing all null edges. This is accomplished by kef's and a kemr (which removes the last null edge). In this step, *split faces* consisting of two loops with geometrically identical vertices are created. See Figures 12(c,d).

(4) Steps (1) through (3) have now created the desired situation. Next, the solid is split by transforming each split face into two ordinary faces with a mfkrh—i.e., by the exact inverse of the operation depicted in Figure 4(b).

(5) The model is decomposed into its components, each of which represents a solid. This is accomplished by finding the maximal connected components in the graph formed by faces and their adjacencies over edges. In certain special cases (for instance, with solids having "pockets"), additional gluing operations are needed. The resulting components are arranged into two collections of solids according to their position with regard to the splitting plane.

A draft of the main algorithm is given in Figure 13. The "analyze()" algorithm is intended to manage step (5) above.

```
split(S: solid, P: plane): pair of collections of solids
begin
    /* step 1: split edges */
    l_o_n_e = empty list; /* list-of-null-edges */
    for(all edges E of S) do
    begin
        /* if E intersects P, generate a null edge */
        if(E.vertex1 and E.vertex2 lie on different sides of P) then
        begin
            calculate intersection point;
            semv(E, V, E');
            semv(E', V', E");
            add E' to l_o_n_e
        end
    end;

    /* step 2: combine null edges */
    combine_edges(l_o_n_e); /* see Fig. 14 */

    /* step 3: create null faces */
    l_o_n_f = empty list; /* list-of-null-faces */
    while(l_o_n_e not empty) do
    begin
        E = any edge of l_o_n_e;
        if(E.face1 = E.face2) then
        begin
            /* this must be the last null edge of this split curve */
            add E.face1 to l_o_n_f;
            kemr(E)
        end
        else
            kef(E)
    end;

    /* step 4: apply mfkrh */
    for(all members of l_o_n_f) do
    begin
        mfkrh(F, F')
    end;

    /* step 5: construct results */
    return(analyze(S))
end;
```

Figure 13. The split algorithm.

The "combine_edges()" algorithm managing step (2) above can be performed by a *bug algorithm.* Conceptually, the algorithm sends two "bugs" that traverse through the null edges and join their new endpoints by new edges. To accomplish this, bugs consider null edges not yet traversed in a sorted order by x, y and z of their endpoints. The sorting effectively combines a "line-polygon classification" algorithm[9,10] with a bug algorithm. The idea of the bug algorithm is depicted in Figure 14.

Set operations. The ideas of the split algorithm extend to general set operations. For simplicity, let us restrict ourselves to calculating a binary set operation (union, intersection, set difference) between two solids A and B, whose boundaries are known not to "touch" each other in a two-dimensional area. In this restricted case, set operations can all be constructed from a fourfold relative boundary classification of the two solids. We transform A and B into four temporary objects AinB, AoutB, BinA, and BoutA. AinB consists of the parts of A's boundary inside B, whereas AoutB is outside B. BoutA and BinA are defined similarly.

Note that having these objects available makes it easy to form the results of the set operations as follows:

union	← AoutB ⊕ BoutA
intersection	← AinB ⊕ BinA
difference	← AoutB ⊕ BinA

Above ⊕ denotes "gluing" of the components along the intersection polygon of A and B.

To form the classification, we compare each of the faces of A with the edges of B. If the edge intersects the face in a point, we (1) create a null edge as a ring into the face and (2) split the edge by two vertices, again forming a null edge. The process is then repeated for faces of B and edges of A. To construct the intersection polygons, the null edges created are combined by a bug algorithm similar to that in the split algorithm. Since the intersection polygons on both solids are similar, it is advantageous to use a "dual" bug algorithm and construct both polygons simultaneously. After formation of the intersection polygons, the remaining steps are very similar to those of the split algorithm. Null edges are removed to construct intersection faces along the intersection polygons. A and B are divided by applying the mfkrh operator to form the desired classification. Finally, the result is formed by gluing the appropriate components of the classification. The resulting algorithm is remarkable in that its geometric steps and its topological steps are cleanly separated from each other; thus it is more easily manageable altogether.

Solid inversion. One can easily imagine that the results of set operations potentially lead to quite voluminous data

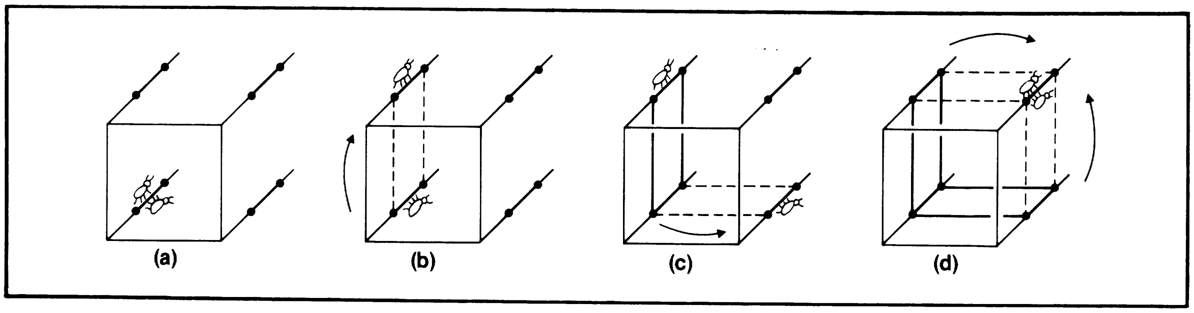


Figure 14. The bug algorithm sends two bugs (a) to crawl along null edges. Bugs move from a null edge to the *nearest* null edge bounding the same face and join their endpoints by mefs (b and c) or mekrs. When the bugs reach the same edge again (d), a split curve has been closed. If null edges not yet traversed remain, a pair of new bugs is sent forth. Two bugs are necessary to ensure the correct termination of the algorithm.

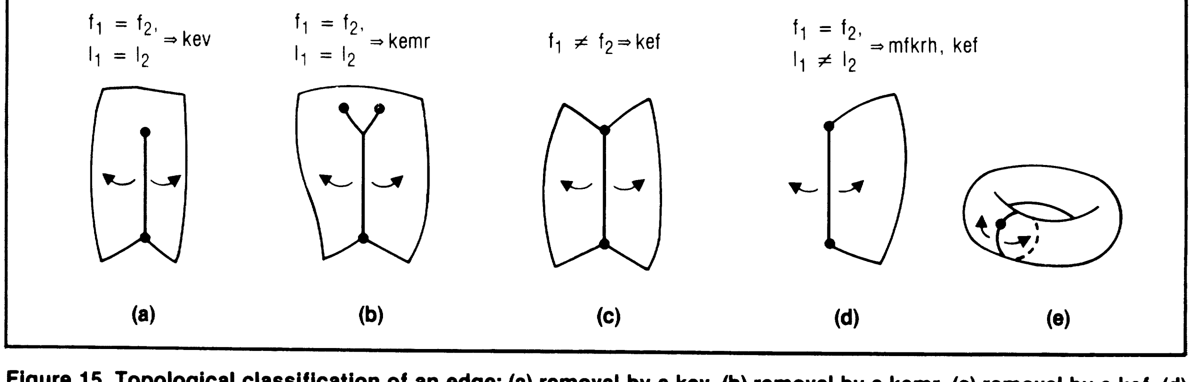

$f_1 = f_2, l_1 = l_2 \Rightarrow$ kev
$f_1 = f_2, l_1 = l_2 \Rightarrow$ kemr
$f_1 \neq f_2 \Rightarrow$ kef
$f_1 = f_2, l_1 \neq l_2 \Rightarrow$ mfkrh, kef
(a)
(b)
(c)
(d)
(e)

Figure 15. Topological classification of an edge: (a) removal by a kev, (b) removal by a kemr, (c) removal by a kef, (d) removal by a kef after transformation by an mfkrh because of a torus situation (e).

structures. This would be true even with the split() algorithm; for instance, the result of the split of Figure 12 is twice as large as the original data structure. In most cases the geometric model created by evaluating a user definition should be put into external storage for later use. This raises the problem of how to store the result obtained by evaluating solid definitions, including set operations.

Two obvious alternatives manifest themselves immediately. The first is to store only the original definition of the solid (and the set operations), but this means that the computationally expensive set operations must be reevaluated each time the data structure is needed for solid analysis. The second possibility is to map the BR data structure into a database format; this leads to well-known data-structure-relocation problems.

When a complex data structure is to be stored in secondary storage, a general solution is to store a sequence of calls to data-structure constructors instead of the data structure itself. From the discussion in preceding sections, it is evident that any solid can be constructed by a list of Euler operators. This raises an interesting problem: is it possible straightforwardly to construct a sequence of Euler operators capable of creating any given solid? If this "inversion problem" could be solved, it would give an elegant solution to the secondary-storage-format problem. The sequence of operators can easily be stored in secondary storage and efficiently reevaluated, since no expensive operations are needed. Note that the Euler operator format is also useful as a transfer format between modeling systems, since implementation-specific data representation details are effectively hidden.

One of the most interesting properties of Euler operators is that, with them, the inversion problem *is* solvable. The algorithm is based on the idea of transforming the solid to a null object by Euler operators. (This is exactly what was done in Figure 1.) In the first step, all edges of the solid are removed by an appropriate Euler operator. In the second step, all but one of the remaining vertices are removed; the last vertex is removed in the final step. The desired sequence of operators to construct the solid, then, consists of the inverses of the operators in these steps applied in reverse order.

The first step may be accomplished by scanning through all edges of the solid and classifying each edge topologically. The classification is very simple: if the two loops of the edge are equal, it may be removed either by a kev or by a kemr. If the loops are not equal, the faces of the loops are compared. If the faces are not equal, the edge bounds two distinct faces and may be removed by a kef. These cases are illustrated in Figures 15(a-c). The remaining case, where the faces of an edge are equal but the loops are not, is the key issue for the algorithm. In this case none of the edge-removing operators is directly applicable. This particular topology characterizes a torus-like situation as depicted in Figures 15(d,e). In order to be able to remove the edge also in this case, the hole must first be transformed into a new face by mfkrh before the edge is removed by kef. This approach is actually able to detect all holes in the solid. The algorithm is clever in that it does not attempt to manage holes until they manifest themselves in their "skeletal" form as in Figure 15(d).

After the first step there exists in general one vertex for each ring originally present or generated by kemr's during the removal of edges. All but one of these vertices may be removed by a mekr followed by a kev. In the final step, the last remaining vertex may be removed by a kvsf. The whole algorithm is represented by Figure 16.

To be useful, an inversion algorithm should lead to a data representation which is concise in comparison with the original data structure. The length of the Euler-operator list generated is therefore an interesting characteristic of an inversion algorithm. For the algorithm of Figure 16, the length is determined by the ordering of edges during the first step. The algorithm generates new rings by kemr's in the first step only to remove them by mekr's in the second step. Let v, e, f, r, and h denote the numbers of vertices, edges, faces, rings, and holes. In the worst case (Figure 17), a total of *floor* $(v/2)$ new rings and

$$\begin{aligned} h + f + r + v + 2\,\mathit{floor}(v/2) - 1 &= \\ h + f + 2v + r - 1 &\text{ (v even)} \\ h + f + 2v + r - 2 &\text{ (v odd)} \end{aligned}$$

operators are generated by the algorithm.[11] The 2 *floor* $(v/2)$ factor can easily be removed by avoiding kemr's in the first step. This may be accomplished by applying kev's after

```
invert(S: solid): operator sequence
begin
  /* step 1: remove edges */
  for(all edges E of S)
    if(E.face1 = E.face2) then
      if(E.loop1 = E.loop2) then
        if(E.vertex1 is adjacent only to E) then
          /* case Fig. 15(a) */
          kev(E, E. vertex1)
        else
        if(E.vertex2 is adjacent only to E) then
          kev(E, E.vertex2)
        else
          /* case Fig. 15(b) */
          kemr(E)
      else
      begin
        /* case Fig. 15(d) */
        mfkrh(E.face1, Ftemp);
        kef(E)
      end
    else
      /* case Fig. 15(c) */
      kef(E)
  end;

  /* step 2: remove rings */
  V = any vertex of S;
  for(all other vertices V' of S) do
  begin
    mekr (V, V', E);
    kev(E, V')
  end;

  /* step 3: remove the final vertex */
  kvsf()
end;
```

Figure 16. The inversion algorithm.

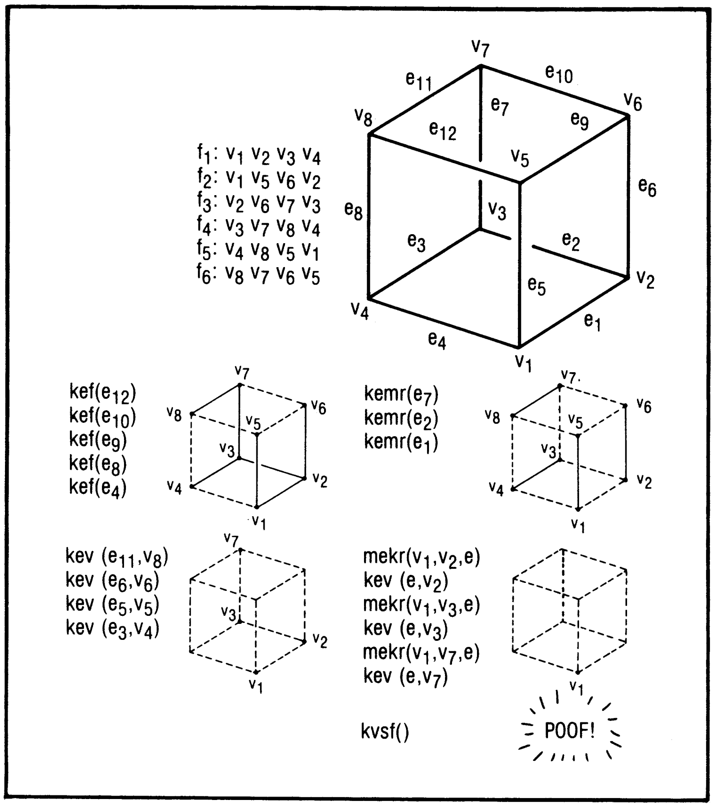


Figure 17. A worst case example.

each mef operation as far as possible. It can be shown that the modified algorithm generates no more than

$$3h + f + r + v - 1$$

operators.[11] The enhanced algorithm is optimal in that it always generates the shortest sequence of Euler operators in our collection that is capable of creating the solid.

Discussion and conclusions

Euler operators are currently used in a number of geometric modeling systems[12,13] as the basic definition operations. Our work—in particular, the inversion algorithm—shows that Euler operators may be used as a basis for a comprehensive solid modeler also in a more general sense. By the inversion algorithm, we are able to generate sequences of Euler operators corresponding to the results of set operations or other solid definition and manipulation operations. These lists form the main external representations of the system.

Certain algorithms can operate directly on these lists; for instance, any linear transformation (such as rotation or translation) may be applied to a solid by scanning its inversion and transforming all vertex coordinates. As another example, a line figure of a solid can be generated just by scanning through the sequence of operators and storing the coordinates of the vertices created. Also, in those contexts where a BR data structure must be evaluated, the result may be tailored for the particular use—quite a different data structure for hidden-line removal, as compared with set operations.

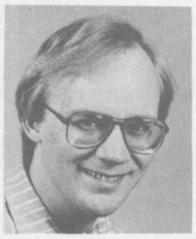
In the future, other algorithms based on Euler operator sequences may also be developed. One problem of great importance is the object equivalence problem: given two operator sequences, do they model the same solid? ■

Acknowledgments

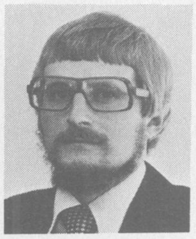
The work described has been supported by the Finnish Academy and the Helsinki University of Technology Foundation.

References

1. A. A. G. Requicha, "Representations of Rigid Solids—Theory, Methods, and Systems," *ACM Computing Surveys,* Vol. 12, No. 4, Dec. 1980, pp. 437-464.
2. H. B. Voelcker and A. A. G. Requicha, "Geometric Modeling of Mechanical Parts and Processes," *Computer,* Vol. 10, No. 2, Dec. 1977, pp. 48-57.
3. B. G. Baumgart, "Geometric Modeling for Computer Vision," Report No. AIM-249, STAN-CS-74-463, Stanford Artificial Intelligence Laboratory, Stanford University, Oct. 1974.
4. B. G. Baumgart, "A Polyhedron Representation for Computer Vision," *AFIPS Conf. Proc.,* Vol. 44, 1975 NCC, pp. 589-596.
5. A. Baer, C. M. Eastman, and M. Henrion, "Geometric Modeling: A Survey," *Computer-Aided Design,* Vol. 11, No. 5, Sept. 1979, pp. 253-272.
6. M. Mantyla, "Methodological Background of the Geometric Workbench," Report No. HTKK-TKO-B30, Laboratory of Information Processing Science, Helsinki University of Technology, 1981.
7. I. C. Braid, R. C. Hillyard, and I. A. Stroud, "Stepwise Construction of Polyhedra in Geometric Modeling," in K. W. Brodlie, ed., *Mathematical Methods in Computer Graphics and Design,* Academic Press, London, 1980, pp. 123-141.
8. C. M. Eastman and K. Weiler, "Geometric Modeling Using the Euler Operators," *Proc. First Ann. Conf. Computer Graphics CAD/CAM Systems,* MIT, Apr. 1979, pp. 248-254.
9. R. B. Tilove, "Set Membership Classification: A Unified Approach to Geometric Intersection Problems," *IEEE Trans. Computers,* Vol. C-29, No. 10, Oct. 1980, pp. 874-883.
10. R. B. Tilove, "Line/Polygon Classification: A Study of the Complexity of Geometric Computation," *IEEE Computer Graphics and Applications,* Vol. 1, No. 2, Apr. 1981, pp. 75-88.
11. M. Mantyla, "An Inversion Algorithm for Geometric Models," *Computer Graphics* (Siggraph '82 Conf. Proc.), Vol. 16, No. 3, July 1982, pp. 51-59.
12. C. M. Eastman and M. Henrion, "GLIDE: A Language for Design Information Systems," *Computer Graphics* (Proc. Siggraph 77), Vol. 11, No. 2, 1977, pp. 24-33.
13. "Romulus: Extracts from User's and Programmer's Guide," Shape Data Ltd., Cambridge, UK.



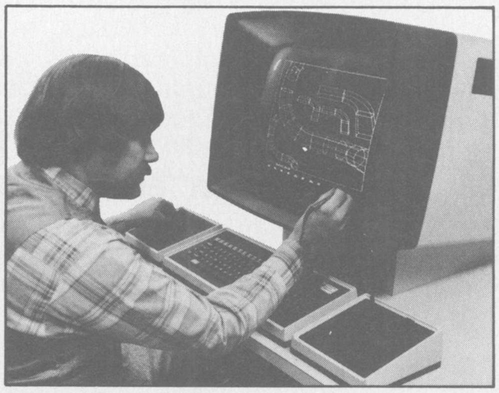
Martti J. Mantyla is a research associate with the Laboratory of Information Processing Science at the Helsinki University of Technology, Finland. Since 1980, he has worked in the CAD project headed by Prof. Sulonen (below) and financed by the Finnish Academy. His research interests include geometric modeling, computer-aided design, computer graphics and database management. He received the MSc in computer science and mathematics in 1979 and the Licenciate of Technology in computer science in 1981, both from the Helsinki University of Technology. He is currently writing his doctoral dissertation on topological manipulations and interrogations in geometric modeling. He is a student member of the IEEE and the ACM.



Reijo Sulonen is a professor of computer science at the Helsinki University of Technology and currently also leads the CAD project of the Finnish Academy. He received the PhD in computer science from the Helsinki University of Technology in 1975. Besides various activities in Finland, he has worked as a visiting research associate at Brown University in 1972-1973 and as a visiting associate professor at Stanford University in 1977-1978. His current research activities include computer graphics, computer-aided design, and database technology.

Sulonen is a member of the IEEE and an associate member of the ACM. He is also active in the Eurographics Association as a member of its executive committee and an editor for its Computer Graphics Forum.

Superior Interactive Design 3-D Graphics Workstation



If you need high-performance calligraphic design capability with 4K x 4K resolution and true high speed stroke graphics, your best choice is the new IMI-500 Intelligent Graphics Design Workstation. In fact, the cost-effective and user-friendly IMI-500 is priced about 30% *below* well-known competitive systems which offer much less performance.

If the IMI-500 sounds like it could benefit your graphics design projects, we'd welcome the chance to show you why the IMI-500 is perfect for stand-alone or large system applications. Why not call or write for complete details, today!

IMI **INTERACTIVE MACHINES INCORPORATED**
2500 B Townsgate Road
Westlake Village, CA 91361 **(213) 707-1880**