CAD/CAM applications need quick and easy access to topological information about objects. Here, four structures for representing this information are evaluated for sufficiency, efficiency, and ease of implementation.

Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments

Kevin Weiler
General Electric

Geometric modeling techniques have become increasingly sophisticated over the last 20 years, embodying more and more information about the physical shape of the objects they model. One of the earliest geometric modeling techniques, *wireframe modeling*, represents objects by the edge curves and endpoints on their surfaces. *Surface modeling* techniques go one step further by also providing mathematical descriptions of the surface shapes of the object. The most recently developed technique, *solid modeling*, represents even more information about the closure and connectivity of shapes and is becoming an increasingly important part of the process of computer-aided modeling of solid physical objects for design, analysis, manufacturing, simulation, and other applications.

Solid modeling techniques were developed starting in the early 1970's and today are beginning to penetrate the commercial industrial market. Solid modeling offers a number of advantages over previous modeling techniques, including an ability to calculate mass properties such as weight, volume, and center of gravity, as well as a greater ability to guarantee the integrity and physical realizability of the model.

A variety of representational forms have been developed for solid models. Each form has its own strengths and weaknesses in the context of various applications. Solid modeling representations can be differentiated on at least three independent criteria: whether they are boundary based or volume based, whether they are object based or spatially based, and whether they are evaluated or unevaluated (see Weiler[1]). A representation is boundary based if the solid volume is specified by its surface boundary; if the solid is specified directly by its volume, it is volume based. A representation is object based if it is fundamentally organized according to the characteristics of the actual solid shape; a representation is spatially based when it is organized around the characteristics of the spatial coordinate system it uses. The evaluated/unevaluated characterization is roughly a measure of the amount of work necessary to obtain information about the objects being represented.

This article deals with the boundary-based, object-based, evaluated form of representation of solid models, including solid models of objects with curved surfaces. This form has been popular in later-phase design and in manufacturing applications, where easy access to complete information on the model is important for speed and simplicity—for example, during the modification, analysis, and use of the constructed solid models. A typical application is the numerical-control machining of mechanical parts, an application in which quick access to local surface information speeds computation and verification.

We focus in this article on four different data structures for the representation of topological information for curved-surface solid modeling. The four data structures are distinguished by the use of different groupings of information related to the topological adjacencies of the elements surrounding edges in the embedded-graph representation of the surfaces of solid objects.

Three of the four edge-based data structures presented here are new, and proofs of their sufficiency are outlined. A detailed proof of the sufficiency of the popular winged-edge structure in a generalized curved-surface environment is also described.

0272-1716/85/0100-0021$01.00 © 1985 IEEE

Two of the new data structures, the V-E and F-E structures, feature simplified access procedures and are organized according to the principle of representing the *use* of edges in topological adjacencies rather than representing the edge as a single structure. This avoids the need for processing additional data to determine which side or end of the edge is intended by a reference to an edge structure. The resulting simpler accessing procedures are valid for both planar polyhedral environments and curved-surface environments.

Graph theory, topology, and adjacency relationships

Solid modeling representations contain many kinds of information. Boundary-based, object-based representations contain information about the surfaces of individual objects. The surface of an object is broken into one or more separate pieces, and each piece is fully described along with its own boundaries, which take the form of curves and points.

This type of solid model must have the ability to describe how each surface piece fits together with each adjacent surface piece in the final model, so that a single, fully enclosed volume is formed. The adjacencies of these components can be derived by numerical techniques to analyze the geometric proximity of the surface pieces, though this approach often involves considerable computational expense and numerical accuracy problems. In an evaluated representational form, however, such information is available explicitly.

This adjacency information is often informally referred to as the *topology* of the solid model. The actual geometric surface descriptions, curve descriptions, and point locations are then referred to as the *geometry* of the solid model. The topology information can serve as a framework into which the geometric information is placed. The topology therefore serves as the "glue" holding all the component information together.

The purpose of the data structures described in this article is the representation of this topological information—with enough generality that a wide variety of geometric surface and curve definitions can be utilized for representing general curved-surface solid models. These structures therefore do not embody particular geometric curved-surface representation techniques but rather concentrate on the topological framework. Almost any geometric surface representation techniques can be used in conjunction with this framework, as long as they abide by the domain restrictions specified in the section on topological sufficiency. This generality supports curved- and planar-surface representation types, or mixtures of different geometric surface description techniques.

Topological information is stored in a graph format in each of the data structures presented. Since we will be discussing embedded graphs of boundaries and the topological relationships they represent, we will review some definitions and concepts from the fields of graph theory and both algebraic and point set topology. This review will also help to define the range of solids in which we are interested and to prove the correctness of the representations defined. Of particular interest is terminology related to adjacency topology, as used in boundary-based representations for solid modeling.

Graph theory concepts. We will be using graphs to represent the edges and vertices of both planar- and curved-surface polyhedral solids. A *vertex* is a unique point. In modeling discussions we will assume it is associated with a unique three-dimensional point in modeling space. An *edge* is an unordered set of two vertices. Strictly speaking, these vertices must be distinct, meaning that each edge has two different vertices and at most one edge exists between any two particular vertices; however, we will relax this definition below. Edges are associated with boundaries of surface areas in modeling representations and may be curved or straight. A *graph* is a set of vertices and a set of distinct edges that utilize the vertices.

A *self-loop* is a graph configuration in which an edge joins a vertex to itself; in other words, the two vertices associated with the edge are not distinct. A *multigraph* is a graph configuration where multiple edges are allowed to join the same two vertices; the vertex set of such an edge is therefore not unique as in the usual definition of a graph. While ordinary graphs, by strict definition, do not allow these conditions, we will allow both. Graphs that allow both self-loops and multiple edges are called *pseudographs*. When we refer to graphs here, we are referring to pseudographs.

A *labeled* graph is a graph where each vertex and edge is uniquely identified by some means independent of the graph. (For further information on graph theory, see Harary's textbook.[2])

Topological concepts. Some ideas from topological theory also are necessary to characterize the domain of the shapes of interest in the context of geometric solid modeling. An intuitive approach to topology can be found in Arnold's introductory book.[3] A more formal approach is taken by Agoston.[4] The following definitions, while not completely rigorous, will be helpful in later discussions.

A *homeomorphism* is a one-to-one, onto, topological transformation that is continuous and has a continuous inverse. Topology is the study of properties invariant under homeomorphisms; such properties determine topological equivalence. Intuitively, homeomorphisms can be thought of as elastic transformations that preserve adjacency as well as other topological properties.

An *open disk* is that portion of a two-dimensional space that lies within some circle of positive radius centered at a given point, excluding the circle itself. An *open ball* or *open sphere* is the three-dimensional analog of the the open disk; it is a set of points inside a sphere centered at a point and with a radius greater than zero, and it excludes the sphere itself.

A subset of a topological space is *arcwise-connected* if between any two points in the subset of the space there is a continuous path that is entirely contained within that subset of space. A *surface,* for our purposes, is an arcwise-connected, two-dimensional space. Note that although a surface is locally two-dimensional, it may exist geometrically in a three-dimensional space and may be curved.

A surface is *bounded* if the entire surface can be contained in some open ball. A *boundary* on a surface may be a closed or open curve or a single point on the surface. A closed curve boundary separates a piece of the surface from the rest of the surface. A surface is *closed* if it is bounded and has no boundary. For example, a plane has no boundary but is unbounded, while a sphere is a closed surface.

A *manifold,* in particular a two-manifold, is a two-dimensional, connected surface where each point on the surface has a neighborhood topologically equivalent to an open disk. A manifold may or may not be a closed surface. A manifold is *orientable* if it is two-sided, that is, if it is not a surface like a Moebius strip or Klein bottle. The surfaces of a solid volume are required to be oriented as well as closed, so that there is a clear distinction between the inside and outside of the volume.

A graph can be *embedded* (or mapped) onto a surface if it is drawn on the surface so that no two edges intersect, except at their incident vertices.

Faces are the regions of a surface defined by a graph embedded on a surface. Each face is a connected component of the set obtained by subtracting the vertices and edges of the embedded graph from the surface. The boundary of a face consists of those edges and vertices of the embedded graph whose every part touches upon the face. Note that a face does not contain its boundary. When a graph is embedded on an orientable, two-manifold surface, each edge of the graph is used exactly twice in the traversal of the edges around each face, once in each direction. The traversal can be made by moving along each of the edges and vertices in sequence around each face, such that the area of the face is always to one side, say the right, and the end vertex of each edge is the beginning vertex of the next edge in the traversal.

An edge in an embedded graph which has the same face on both sides is called a *strut* (or a wire).

A *simply connected* face has a single, connected boundary. A *multiply connected* face has a boundary that consists of two or more disconnected components, as in a face with a hole in it.

A *handle* on an object can be formed by cutting two holes in the surface of the object and then constructing a tube to join the two holes. A doughnut shape or torus, for example, is topologically equivalent to a sphere with one handle. The *genus* of a graph is the minimum number of handles that must be added to a sphere so that the graph can be embedded on the resulting surface without edges crossing at places other than their common vertices.

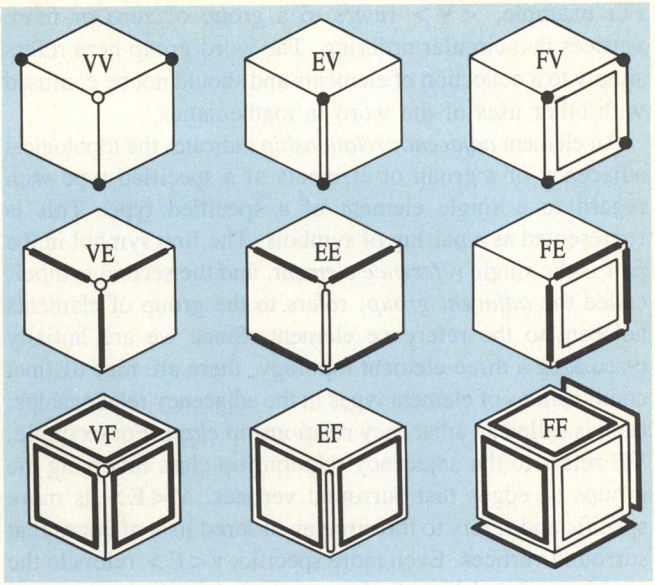


Figure 1. The nine element adjacency relationships in an adjacency topology of faces, edges, and vertices.

Adjacency relationship terminology. Adjacency topology concerns the physical adjacencies of the edges and vertices on the surface of an object. There are other forms of topology, but this particular kind of topological information so far has proven most useful in object-based, boundary-based solid modeling representations. It is also valuable as a framework around which the rest of a solid modeling implementation can be built.[1]

Given the three primitive topological elements presented so far (faces, edges, and vertices), nine element adjacency relationships are possible (see Figure 1). A terminology for identifying the nine basic kinds of topological adjacencies was originally developed by Baer, Eastman, and Henrion.[5] To discuss these adjacency relationships in greater detail and to include the key concepts of element ordering and direction of ordering, a more comprehensive terminology, which is briefly reviewed here, was developed by the author.[6]

An *element* is a primitive topological object such as a vertex, edge, or face. The *type* of an element is represented by a single character symbol, such as V, E, or F, indicating the vertex, edge, and face types, respectively. The *plurality* of a symbol is indicated by the case of a character; upper case refers to a group consisting of zero or more elements, while lower case refers to a single specific element. A lower-case element symbol may be subscripted to differentiate it from others, as in v_i. The *ordering* of a group of elements indicates its organizational structure. This organization can be indicated in the terminology by bracketing symbols:

{group} indicates an unordered set of elements;
(group) indicates a linear-ordered list of elements;
<group> indicates a circular-ordered list of elements.

For example, <V> refers to a group of zero or more vertices in a circular ordering. The word *group* here refers simply to a collection of elements and should not be confused with other uses of the word in mathematics.

An element *adjacency relationship* indicates the topological adjacency of a group of elements of a specified type with regard to a single element of a specified type. This is represented as a pairing of symbols. The first symbol in the pair is the single *reference element,* and the second symbol, called the *adjacent group,* refers to the group of elements adjacent to the reference element. Since we are initially discussing a three-element topology, there are nine distinct combinations of element types in the adjacency relationships; each is called an adjacency relationship *class.* For example, VE refers to the adjacency relationship class involving the groups of edges that surround vertices. V<E> is more specific and refers to the circular-ordered lists of edges that surround vertices. Even more specific, *v*<E> refers to the circular-ordered list of edges around a particular vertex.

An adjacency relationship carries two kinds of information: the class of the adjacency relationship and the ordering information of the adjacent group. Adjacency relationships that have unordered adjacent groups are called *unordered* adjacency relationships; relationships with linearly or circularly ordered adjacent groups are called *ordered* adjacency relationships. The distinction is a vital one in terms of the informational sufficiency of the adjacency relationship, as will be discussed later.

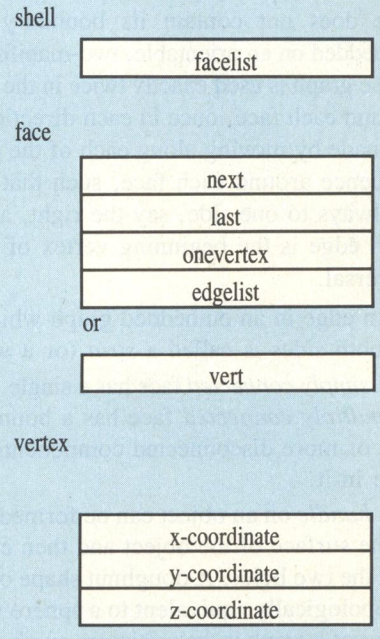
Correspondence is the ability to make adjacency associations between different adjacency relationship classes that utilize the same element type in either their reference element or adjacent group. A particularly strong form of correspondence occurs when two adjacency relationships have the same reference element type. Correspondence is symbolized as two adjacency relationships connected by a dash. The order of appearance of the adjacency relationships in the correspondence is not significant. For example, it is possible to have the V<E> and V<F> adjacency relationships in correspondence, symbolized as V<F>-V<E>. In this case correspondence means that information about face-then-edge-then-face . . . around-a-vertex is available in addition to the expected information about edge-then-edge-around-a-vertex (V<E>) and face-then-face-around-a-vertex (V<F>).

Edge-based graph data structures

Four different data structures for representing embedded graph topologies useful in solid modeling are presented. One, the winged-edge structure, will be familiar to many. Another is a slight modification of the winged-edge structure. This and the remaining two structures are new and not previously published. All four are based on use of the edge element as the reference element from which adjacencies with other elements are determined. They are edge based in the sense that all the information required to reproduce the entire embedded graph topology is contained in the edge-related data structures. Two of the structures, the winged-edge and the modified winged-edge structures, keep the edge

```
type    vptr = ^vertex;
        eptr = ^edge;                    {used for W-E and modified W-E edges}
        ehptr = ^edgehalf;               {used for F-E and V-E edges}
        fptr = ^face;

        shell = record
                facelist: fptr           {heads circular double-linked list}
                end;

        face = record
               next,last: fptr;     {double-linked list of faces}
               case onevertex: boolean of      {false if any edges on boundary}
                     True:  (vert: vptr);
                     False: (edgelist: eptr {or ehptr})
               end;

        vertex = record
                 x,y,z: coordinate               {geometric location of vertex}
                 end;
```

(a) Pascal declarations



(b) Storage allocation description

Figure 2. Pascal types for data support structures.

information as a single unit, while the other two split the information related to each edge into two parts based on the specific usage of the edge in the adjacency relationships. These last two structures are identical in the form of their storage format but differ greatly in the semantic interpretation of that format. We will look at these data structures in the context of a computer implementation.

Support data structures. Most of the topological information for the structures described here is embodied in the edge structures. Before the edge-based data structures are described, however, we must consider the other data structure elements in the embedded graph representation. The representation of these other elements is essentially the same regardless of the edge data structure used.

Data structures for two of the three element types, faces and vertices, and a structure, called the *shell*, that pulls together all the elements found on a single surface are described here. These support structures are shown in Figure 2. This figure and the following figures describing the four edge-based data structures show Pascal record declarations of the structures, a graphic depiction of the storage fields required, and, in the case of the edge structures, a graphic depiction of the adjacency relationships embodied.

There is some bias in the design of these support structures in that, together with the edge structures, they form a hierarchical description of the graph from higher levels of dimensionality (shell) to lower levels (vertex). This is not the only way to organize a graph representation. For example, one could use the vertex type as the root of the data structure. But information organized top-down hierarchically allows increased time efficiency in many modeling applications because objects can often be processed at higher levels of abstraction that roughly correspond to grosser geometric features. Thus, if a solid modeling system utilizing such boundary topological representations provides a top-down hierarchical topological description of an object, then we can consult more abstract (and more concise) levels of the structure before deciding if detailed information is needed for a given application. For example, if bounding box or sphere information is associated with higher level topological elements, interference analysis tasks need only check the higher level shell extents to eliminate many possible object overlaps, without referring to lower level and more numerous face elements. The support structures given here follow this principle.

Since we are primarily concerned with addressing the topological issues, geometry has been excluded from the structures for clarity, with the exception of three-dimensional coordinate values for the vertex element. In a typical complete solid modeling representation, a face might include plane equation or patch geometric information, and edges might contain spline or other curve information as well as nontopological and nongeometric data.

Strictly speaking, the boundary of a face is the ordered, alternating sequence of edges and vertices that surround the face. In most cases, a sequence of edges, with orientation information, can be used in place of this list of edges and vertices, and the vertex information can be derived when needed. It is possible, however, for a boundary of a face to consist of only a single vertex. A Pascal record variant is shown in the face structure record in Figure 2 to handle this unusual situation. For connected graphs this situation usually occurs as only an initial condition, where the entire graph is the graph consisting of only a single vertex and the face surrounding it. Normally the face representation structure points to an edge on its boundary, but in this case there are no edges and the face points to the single vertex on its boundary. Dealing with this special case at the face level is a more general solution than others which treat the situation as a special case of the shell, as we will see when the structures are extended to handle disconnected graphs in a later section of this article.

When pointers in the structures are not pointing to structures of their own type, they usually point from higher dimension elements to lower dimension elements, such as from edges to vertices, as might be expected in a top-down hierarchical arrangement. Backpointers, pointing from lower to higher levels of dimensionality, are generally not included here for clarity, though an actual implementation often uses them for increased efficiency, trading space for time by eliminating search. Backpointers typically included are edge-to-face, face-to-shell, and sometimes vertex-to edge. Often, linear lists of vertices, edges, and faces associated with a shell are also maintained for applications requiring fast enumeration of single element types, such as graphic display of edges.

The winged-edge structure. The winged-edge structure represents the edge adjacency information as a single unified structure. As is true for all four edge structures presented here, it features a fixed number and length of data fields.

Originally developed by Baumgart[7,8] at Stanford in the early seventies, the winged-edge structure served to model environments of planar polyhedral solids in computer vision research in robotics. Two results of that work were adopted in the solid modeling field. First, the winged-edge structure is often used to represent the boundary graph of the topological adjacencies of faces, edges, and vertices embedded in the surface of planar-faced polyhedral solid models. Second, the Euler operators introduced by Baumgart provide a relatively high level way of constructing such adjacency topology graphs without getting into the details of the underlying data structure. In general these operators provide a way to create and manipulate the model of the embedded graph on an edge-by-edge basis independent of the actual data structure. As this article is primarily concerned with the actual data structures and their informational sufficiency, we will not discuss the Euler operators further, although they are of great interest to implementations using any of the four edge-based structures discussed here. For discussions of the Euler operators, see articles by Eastman and Weiler,[9] Braid et al.,[10] and Mantyla and Sulonen.[11] A

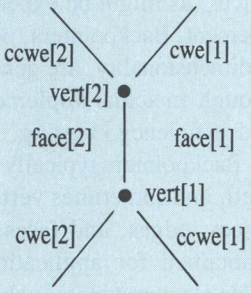
```
side = 1..2;

edge = record
        vert: array [side] of vptr;
        cwe,ccwe: array [side] of eptr;
        face: array [side] of fptr
        end;
```

(a) Pascal description

edge

vert[1]	vert[2]
cwe[1]	cwe[2]
ccwe[1]	ccwe[2]
face[1]	face[2]

(b) Storage allocation description



(c) Diagram

Figure 3. The winged-edge data structure.

```
side = 1..2;

edge = record
        vert: array [side] of vptr;
        cwe,ccwe: array [side] of eptr;
        cwehalf,ccwehalf: array [side] of side;
        face: array [side] of fptr
        end;
```

(a) Pascal description

edge

vert[1]		vert[2]	
cwe[1]	cwehalf[1]	cwe[2]	cwehalf[2]
ccwe[1]	ccwehalf[1]	ccwe[2]	ccwehalf[2]
face[1]		face[2]	

(b) Storage allocation description

Figure 4. The modified winged-edge data structure.

discussion of the validity of the Euler operators can be found in an article by Mantyla.[12]

Groups of researchers at Carnegie-Mellon and Cambridge universities independently enhanced the winged-edge representation to allow disconnected graphs by making additions to the supporting structures,[9,10] an example of which is discussed in a later section. These enhanced representations and Euler operators were incorporated into these groups' respective solid modeling systems, Glide and Build 2.

The topological information stored in the winged-edge structure for each edge is composed of the adjacencies of the given edge with other edges, vertices, and faces. The name "winged edge" results from the graphical appearance of the adjacent edges drawn in relation to the reference edge (see Figure 3). Note that this implies a labeled-graph environment, where all three basic elements are uniquely labeled or named.

As seen in Figure 3, the winged-edge structure maintains the adjacency information with pointers to the two faces, the two vertices, and some of the edges adjacent to the reference edge. This last set of adjacencies is divided into two sections, each associated with the use of one of the sides of the reference edge in the circuit of edges around a face. The only edge adjacencies represented, therefore, are the four edges that directly follow or precede the reference edge in the edge cycles surrounding the two faces adjacent to the reference edge. Thus, the *cwe* and *ccwe* field names used here refer to their use in determining the cycle of edges surrounding a face, as viewed from just outside the solid volume looking toward the surface. This is different from the original Baumgart field names, which used *cwe* and *ccwe* to refer to the positioning of the adjacent edges around a vertex of the edge.

The information in the winged-edge structure can be described in adjacency relationship terminology as the E(V)-E((E)(E))-E(F) adjacency relationships in correspondence, where the adjacent edge information, being only partial EE information, is represented as two ordered lists of length two, one for each endpoint of the edge. The other edge-based data structures also embody this information, though with subtle but important differences.

We will refer to the winged-edge structure as the W-E structure for brevity.

The modified winged-edge structure. The modified winged-edge structure is a slight but important variation on the W-E structure. Like the W-E structure, it represents the edge adjacency information as a single unified structure. In fact, it is identical to the W-E structure except that it contains additional data. The difference is that each of the *cwe* and *ccwe* pointers are accompanied by edge side fields, which indicate exactly which side of the unified edge pointed at is intended. As will be seen later, this reduces algorithm complexity, which is particularly troublesome in curved-surface domains. The structure is shown in Figure 4; its diagrammatic description is similar to the W-E structure.

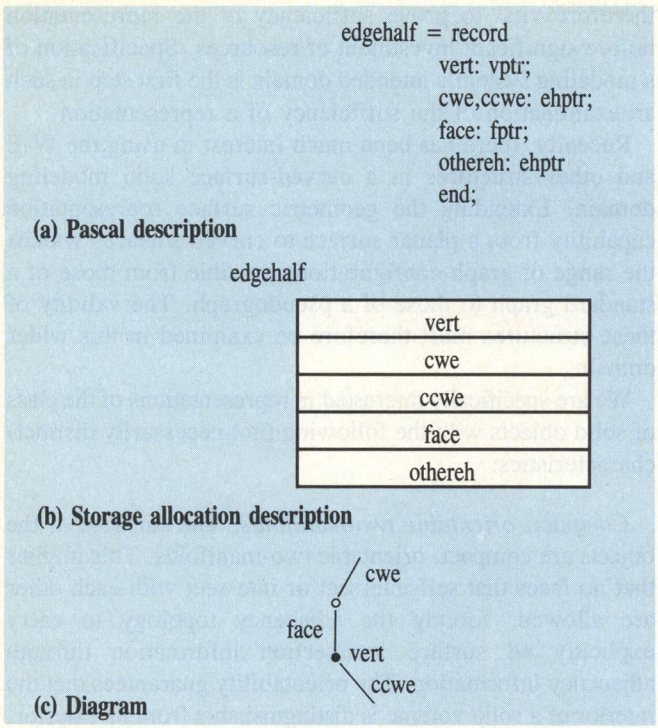


Figure 5. The vertex-edge data structure.

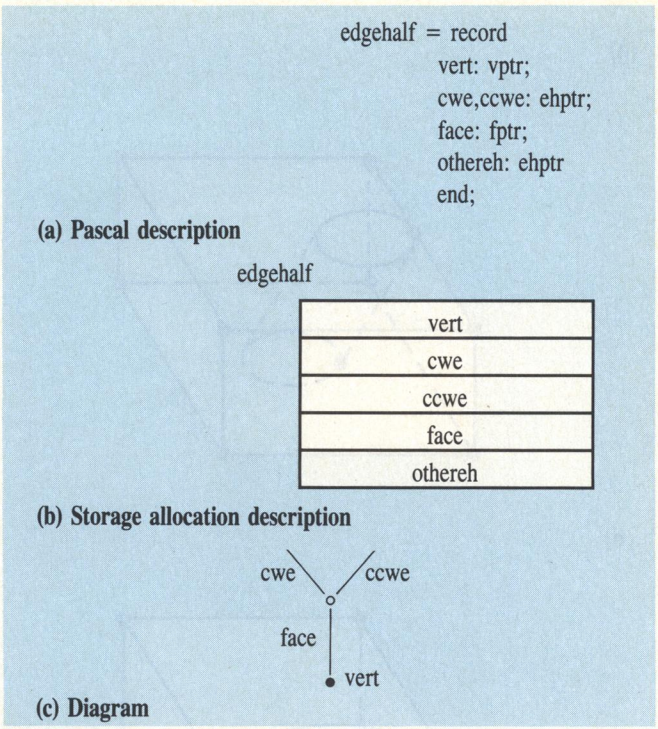


Figure 6. The face-edge data structure

The vertex-edge structure. The vertex-edge structure represents the adjacency information of the edge by splitting it into two structures, each related to one of the two edge ends which is found adjacent to other edge ends around a vertex.

The structure is shown in Figure 5. The adjacency of edges around a vertex forms a circular-ordered list and is represented with the *cwe* fields. The oppposite vertex, one of the adjacent faces, and the other end of the edge are also available through pointers. The *ccwe* field is optional, but is usually included for access time efficiency. We will refer to the vertex-edge structure as the V-E structure.

The face-edge structure. The face-edge structure represents the adjacency information of the edge by splitting it into two structures, each of which is related to one of the two edge sides as found around the periphery of faces. The structure is shown in Figure 6. The adjacency of edges around a face represents a circular-ordered list and is represented using the *cwe* fields. Access to one vertex, the opposite adjacent face, and the other side of the edge is also available through pointers. The *ccwe* field is optional, but is usually included for access time efficiency.

Each side of an edge is used only once as a boundary of a face, and the side implies an orientation toward that face. This orientation is specified here as the area to the right of the edge side when traveling along the edge side from the vertex specified in the *vert* field to the other vertex of the edge. Each edge is therefore used twice and in opposite directions by face boundaries, when we consider the entire embedded graph.

The Cripl-edge representation structure, described by Stoker,[13] was an early edge-based representation with an edge structure similar to the face-edge structure but intended for the planar-faced domain. It utilized *vertex, cwe,* and *face* fields, but not an *othereh* field. As will be seen, the missing *othereh* field is critical for curved-surface applications. The Cripl-edge representation had some unusual initialization conditions and other limitations because of design decisions unrelated to the edge structure chosen. The representation was used in the Carnegie-Mellon solid modeling effort (see Eastman and Henrion[14]) until it was replaced by an enhanced winged-edge representation (see Eastman and Weiler[9]).

It is interesting to note the structural similarity between the face-edge and vertex-edge structures; their semantics, however, are quite different.

We will refer to the face-edge structure as the F-E structure for brevity.

Variations. Minor variations are possible with all four of the edge structures presented: more backpointers can be included; with the exception of the W-E structure, the *ccwe* pointers could be removed even with curved surfaces; the face field of the V-E and F-E structures could point to the other face; and the vertex field of the V-E and F-E structures could point to the other vertex. Most of these variations are compute-vs.-store issues, which require statistical usage data to support rational preferences.

Many major variations in the form of the data structures are also possible, particularly if more information is moved away from the edge and into other element types such as the vertex or loop (introduced later) elements. In this and other

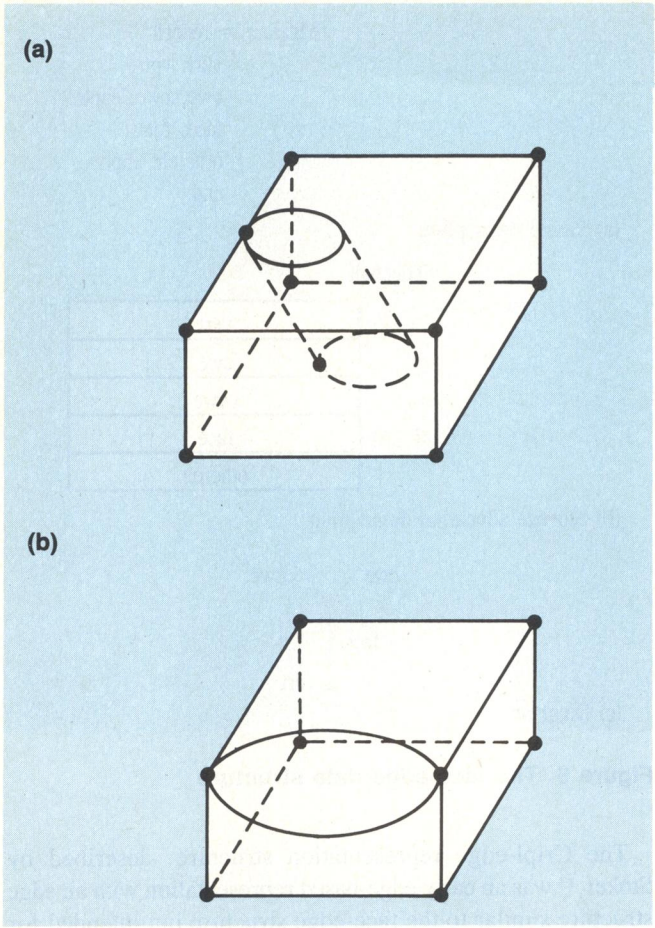


Figure 7. Self-loops created by subtraction of a cylinder from a rectangular solid (a); multigraph created by subtraction of sphere from a rectangular solid (b).

cases, other element types can be used as reference elements. Such alternatives are not discussed here.

Topological sufficiency

A representation consists not only of the static data and data structure but also of the operators and procedures applied against them. The validity of a representation depends on

- the complete specification of the domain over which it is intended to be useful;
- proof of its sufficiency over that entire domain;
- operators that cover the entire domain yet cannot create or manipulate the data into a state outside the intended domain of the representation.

Domain. Traditionally, the domain of solid modeling representations, especially boundary-based representations, is rarely specified—often leaving open the question of their validity for various applications. Considering the amount of effort required to construct a significant, robust solid modeling system, implementors can ill afford to base an implementation on a representation structure that is insufficient over the domain it is intended to support. It is therefore vital to prove sufficiency of the representation before significant investment of resources. Specification of a modeling system's intended domain is the first step in such an examination of the sufficiency of a representation.

Recently, there has been much interest in using the W-E and other structures in a curved-surface solid modeling domain. Extending the geometric surface representation capability from a planar surface to curved surfaces widens the range of graph configurations possible from those of a standard graph to those of a pseudograph. The validity of these structures must therefore be examined in this wider domain.

We are specifically interested in representations of the class of solid objects with the following (not necessarily distinct) characteristics:

Compact, orientable two-manifolds. The surfaces of the objects are compact, orientable two-manifolds. This implies that no faces that self-intersect or intersect with each other are allowed, forcing the adjacency topology to carry explicitly all surface intersection information through adjacency information. The orientability guarantees that the interior of a solid volume is distinguishable from its exterior.

Embedded-graph adjacency topology. Their topologies are represented by embeddings of graphs into a surface. In other words, the graph is totally contained in the surface, without any edges crossing except at mutual endpoints. Every face in the embedded graph must have a boundary of at least one vertex.

Pseudographs. Their graphs are pseudographs; they may be multigraphs and may contain self-loops. This allows curved edges with little constraint on geometry, other than the embedded-graph constraint that edges must not intersect except at endpoints. This freedom is very desirable because multigraphs and self-loops occur naturally during typical modeling operations involving curved surfaces, particularly those involving the Boolean set operations (see Figure 7). While these situations can be simulated by dividing each multiple and self-loop edge into several edges, detecting and dealing with these situations require additional intelligence on the part of the modeler.

Labeled graphs. Their graphs are labeled (at least for those element types involved in the adjacency relationships being used to represent their topology). Our interest in maintaining the labels of graph elements is explained below.

Faces containing no handles. This insures that an arbitrary number of handles cannot be added to the surface of a solid without changing its boundary graph structure, forcing the adjacency topology to carry all genus information (and maintain the validity of the Euler-Poincare formula). It is important to note that a face does not include its boundary; otherwise, faces of objects like the one in Figure 11 would have to contain a handle. Intuitively this can also be described as the condition that the face must be mappable to a plane without cutting the face or changing its boundary. For example, a sphere must have a boundary of at least one vertex

to be mappable to a plane. Note that the no-handle-on-faces restriction is not implied by the two-manifold condition.

Genus. There is no restriction on the genus of the total object being represented.

Connected graph. For now, we will assume the graphs of the objects are connected graphs, and their individual faces are simply connected. This restriction will be lifted in a later section.

While polyhedra are normally thought of as having straight edges and planar faces, topologically it makes no difference if the edges and surfaces are curved. Therefore, in general, graph-based, solid-boundary representational techniques can be equally valid for both planar- and nonplanar-faced solid objects with curved or straight-line edges. However, a much wider variety of embedded graph configurations is possible if the underlying surface is curved, as indicated by the pseudograph condition. This condition is not needed for domains involving only planar surfaces, since self-loops and multigraphs cannot occur in these more restricted environments.

There are several reasons for using labeled graphs. First, it is desirable to have the ability to associate nontopological and possibly nonunique attributes with the topological elements for application purposes (including associating geometric coordinate values with a vertex). Second, adjacency relationship information, even if sufficient, in general does not uniquely identify an element. Third, all three element types generally will be required in a solid modeling representation since we want the relationships of all three element types to be derivable and associated with each other by label. This means that any representation for which reproducible labels are desired must allow the derivation of at least two adjacency relationships involving all three element types.

Holes in faces and internal cavities in solids can be represented with disconnected graphs. These situations are not directly allowed by the connected-graph condition, but this restriction will be removed in the section on disconnected graphs.

Sufficiency. Within the context of an adjacency topology consisting of the three primitive element types (faces, edges, and vertices), sufficiency is the ability to recreate all of the nine topological-element adjacency relationships without error or ambiguity. It is not necessary to store information on all nine adjacency relationships to achieve sufficiency. In fact, there are single adjacency relationships and combinations of single insufficient adjacency relationships that can be used to achieve topological sufficiency over the specified domain.[6]

All elements in an embedded-graph solid modeling structure must be bound together in some fashion to produce a single cohesive representation of an object. It should again be noted that in a labeled-graph environment at least two or more adjacency relationships are necessary to bind all of the different element types together, since each individual

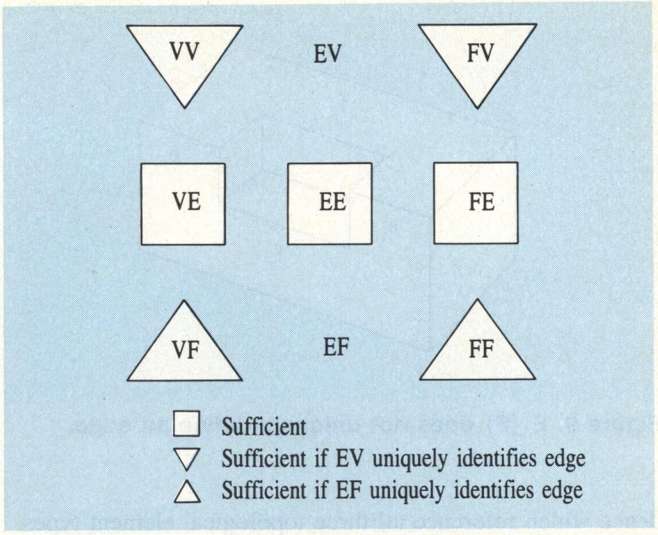


Figure 8. Sufficiency of the nine individual element adjacency relationships.

relationship can refer only to two element types at most. Thus sufficient combinations of two individually insufficient adjacency relationships are just as interesting for solid modeling representations as individually sufficient relationships as long as they involve all three element types.

Informational sufficiency of the adjacency relationships. To examine the sufficiency of data structures, we first need to find what information is sufficient—in other words, which set of adjacency relationships is sufficient. We briefly summarize here relevant portions of previous work,[6] which explores the sufficiency of adjacency relationship information.

Of the nine adjacency relationships, three of the ordered adjacency relationships are individually sufficient over the domain specified. The three are the V<E>, the F<E>, and some forms of the EE relationship (see Figure 8). Under more restricted environments than the domain specified here, other adjacency relationships are also individually sufficient. If the two vertices of edges, E{V}, uniquely define edge identity (as in a planar-faced polyhedral environment), then V<V> and F<V> are also sufficient. If the two adjacent faces of edges, E{F}, uniquely define edge identity, then V<F> and F<F> are also sufficient.

Much of this work is based on a theorem derived by Edmonds,[15-17] which states that the directed circular orderings of the edges around the vertices in an embedded graph (essentially the V<E> relationships) are sufficient information to completely and uniquely describe polyhedron topologies. The proof involves a permutation theorem.

It can be seen from the duality principle in planar-graph environments that the F<E> adjacency relationship is also sufficient by itself. The proof involves a topological identification procedure.[6]

It turns out that there are five possible pairs of individually insufficient element adjacency relationships in correspon-

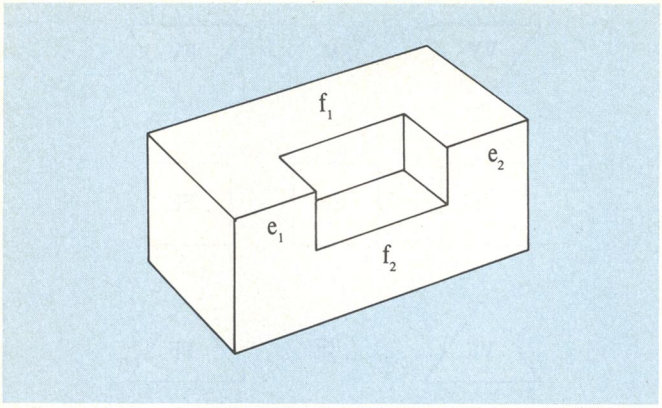


Figure 9. E {F} does not uniquely define an edge.

dence which reference all three topological element types. Unfortunately, none are sufficient. Thus, although a minimum of two adjacency relationships is usually required to tie a graph-based representation together, at least one of the two adjacency relationships must be individually sufficient (must be V<E>, F<E>, or a sufficient form of EE) for the representation to be informationally sufficient.

Sufficiency of the edge-based structures. It's worth noting that the discussion here concerns the informational sufficiency of data structures. While information about a sufficient adjacency relationship must be available from the data structures, the use of particular adjacency relationship information does not imply a particular format for the data structure. Many data structure formats are possible, even with the information of the same sufficient adjacency relationship, if we distribute and partition the information in different ways across different elements. A data structure is sufficient itself as long as its information content allows for the derivation of some sufficient adjacency relationship.

It is important to distinguish between the occurrence of edge element identity information in adjacency relationships and a representation of the edges themselves. The structures we are dealing with here are not primarily intended to model the topological elements themselves but rather to model their *use* or occurrence in adjacencies.

All four data structures directly store all three adjacency relationships with the edge as the reference element in correspondence: E(V)-E((E)(E))-E(F). Unfortunately, none of these adjacency relationships are individually sufficient.[6] There are subtle differences in the edge-based data structures concerning the existence and form of information beyond that of E(V)-E((E)(E))-E(F). The W-E structure stands out as having no additional information. The modified W-E structure contains explicit side information associated with its element pointers, as do the V-E and F-E data structures. These relatively small differences in information content make large differences in the ease with which their sufficiency can be proven.

Proofs of the sufficiency of the F-E and V-E structures are fairly direct. Proof of sufficiency of the W-E structure is more complex and demonstrates the need for additional storage (but not information) along with clever accessing algorithms when it is used in a nonplanar environment. Proof of sufficiency of the modified W-E structure follows directly from the sufficiency of the W-E structure. The following are outlines of proofs of sufficiency for the four edge-based data structures:

(1) F-E structure. The F-E structure is sufficient if it can be used to correctly and unambiguously derive the singly sufficient F<E> adjacency relationship. In the case of a planar-faced domain, the F-E structure can easily generate F<E> using only its *vertex* and *cwe* fields. We proceed by traversing all of the edge half structures that represent the edge sides surrounding each face, following the *cwe* fields of the edge half structures until we arrive back at the starting-edge half structure. The *vertex* field is necessary for determining the adjacency of the faces, as explained below. The *vertex* and *cwe* fields alone are therefore sufficient information for the planar-faced domain.

Finding the other side of the edge is possible without the *othereh* field because the other vertex of the edge can be found as the vertex of the next edge; in a planar-faced domain E{V} uniquely determines the identity of an edge. Edge sides with the same two vertices therefore belong to the same edge. This allows the total surface mesh of faces to be assembled into the whole closed surface with a topological identification procedure. The identification procedure matches edge halves by using the vertex information to find the identity of the full edges to which the halves belong. Identifying the edge sides brings the whole surface together, much as a puzzle is put together by matching up patterns on the edges of the puzzle pieces.

In a curved-surface domain, however, access to the other half of the edge must be explicit in order to handle self-loops and multigraphs unambiguously. Since all edge-related pointers are to edge halves, specifically edge sides, which side of the edge is intended in the adjacency representation is explicit. Each edge side can be used in only one direction, and this direction is unambiguous due to the convention that an edge side is a boundary of the face area we find to its right when we travel from its specified vertex to its second vertex. Access to the other side of the edge is explicitly required in order for the individual faces to be assembled into a complete, closed-surface mesh, since in a curved-surface domain E{V} generally does not unambiguously identify a specific edge.

Thus, with access to the other side of the edge given by the pointer in the *othereh* field, the F-E structure is topologically sufficient over the specified curved-surface domain.

(2) V-E structure. Access to the other half of the edge is mandatory in the V-E structure even in a general planar-faced domain, since E{F} does not uniquely determine edge identity without some additional connectivity restrictions (see Figure 9).

Sufficiency for the V-E structure is most easily shown by deriving the V<E> adjacency, which proves its sufficiency by the Edmonds theorem. This can be obtained directly from the *cwe* pointers of edge end structures. First, we can find one edge adjacent to each vertex by using the *othereh* field of an edge whose *v* field matches the vertex in question. Then, for each vertex, we follow the *cwe* fields of each edge in sequence until the cycle of edge ends around each vertex is complete. This is not ambiguous even in the presence of self-loops because edge ends are pointed to instead of entire edges.

This is what we meant earlier by the difference between use of an edge by an adjacency relationship and the edge itself. The primary purpose of the last two edge structures presented is to represent the adjacency relationships of the edges, not the edges themselves. In this case, for the V-E structure, we are referring to a particular *end* of an edge around a vertex, and an edge half is the end of the edge immediately adjacent to the vertex. In the case of the F-E structure, we were referring to a *side* of the edge used to bound a face and the edge half is the side rather than the end of the edge. Since, by the definition of the V-E and F-E structures, the end information and side information are coordinated, references (in context) to either edge sides or ends are unambiguous for either structure. When edges are described in a single structure rather than two edge halves, however, and simple pointers to the full edges are used, confusion can result because the two possible uses of a single edge in a given situation cannot always be easily distinguished without additional processing (in one case, which of the two ends should be used, and in the other case, which of the two sides should be used). This potential confusion becomes more apparent when the sufficiency of the W-E structure is examined, since the edge adjacency information is represented by the W-E structure in a single unified structure. This particular weakness of the W-E structure is addressed by the modified W-E structure.

(3) W-E structure. Proving sufficiency for a curved-surface domain in the case of the W-E structure is more difficult than for the F-E or V-E structures because this structure represents an edge as a single structure rather than separately representing each of its two uses in adjacencies. This results in a situation where one must use pointers to full edges rather than to uses of edges (sides or ends), leaving the burden of determining which half was intended to the algorithms that manipulate the structure.

As stated before, the W-E representation is essentially the E(V)-E((E)(E))-E(F) adjacencies in correspondence, utilizing all the adjacency relationships that use the edge as the reference element. All three are individually insufficient (since the E((E)(E)) form of the EE adjacency relationship is individually insufficient.[6] As mentioned by Hanrahan,[18] the winged-edge structure can be shown to be equivalent with the Edmonds representation involving V<E> information. This is clearly true for the case of planar surfaces where self-loops and multigraphs are disallowed. We will demonstrate here in detail, however, that specifically the adjacency relationship pair E(V)-E((E)(E)) in correspondence is sufficient to generate unique topological embeddings for curved surfaces, but only if some additional mechanisms (but not additional information) are available.

Theorem: *The pair of E(V)-E((E)(E)) adjacency relationships in correspondence with an additional form of "global" memory is sufficient to unambiguously represent adjacency topologies of general polyhedra.*

The proof involves examination of whether ambiguity can result during an attempt to construct unique V<E> information from the E(V)-E((E)(E)) adjacency relationship pair in correspondence. If at any point during the derivation of the next edge clockwise (looking toward the surface from just above it) around a given vertex, say v_1, starting from a given edge incident to that vertex, say e_1, there is more than one choice of which edge should be the next edge, and that choice will result in a topologically different embedding, then the E(V)-E((E)(E)) adjacency relationship pair is insufficient.

Given access to E(V)-E((E)(E)), v_1, and e_1, the *identity* of the next edge clockwise about the vertex, say e_2, is always known, but which *use* is intended (in this case, which end) is not explicit. Confusion would be possible only if the next edge e_2 were a self-loop. In that case, we would not know which end of e_2 should be included next; choosing the wrong end might cause a misordering of the V<E> adjacent group being constructed or even cause some edges to be missed, unless we examined all edges to detect errors and backtracked as necessary. But we can look at each end of e_2 and use only the end where the counterclockwise edge from e_2 is e_1 (note that this makes the *ccwe* fields mandatory in the W-E structure). Then confusion would be possible only if e_1 were also a self-loop, for only then would it be possible for e_1 to be counterclockwise from e_2 about v_1 at two places.

So far we have a possible situation where e_1 and e_2 are self-loops, where e_2 is clockwise from e_1 about v_1 in at least one place, and where e_1 is counterclockwise from e_2 about v_1 in two places (see Figure 10a). But if e_1 is counterclockwise from e_2 in two places, then e_2 must be clockwise from e_1 in two places also (see Figure 10b).

Given this configuration, any additional structure in the graph can exist only in sequential positions A and/or B (see Figure 10b). No matter how these additional structures are configured, the V<E> relationship for v_1 will be of the form v_1<E> = <e_1 e_2 edges-of-A e_1 e_2 edges-of-B>. The key feature of the v_1<E> adjacent group is the repeating sequence (e_1 e_2 something). Because the sequence repeats, a choice of either end of e_2 following e_1 (or a choice of either end of e_1 following a structure, or a choice of either additional structure following e_2) will generate identical topological results when one considers that the sequence in the V<E> adjacent group forms a circular list.

Another way of looking at it is that the E((E)(E)) adjacencies are the same for both positions A and B, and,

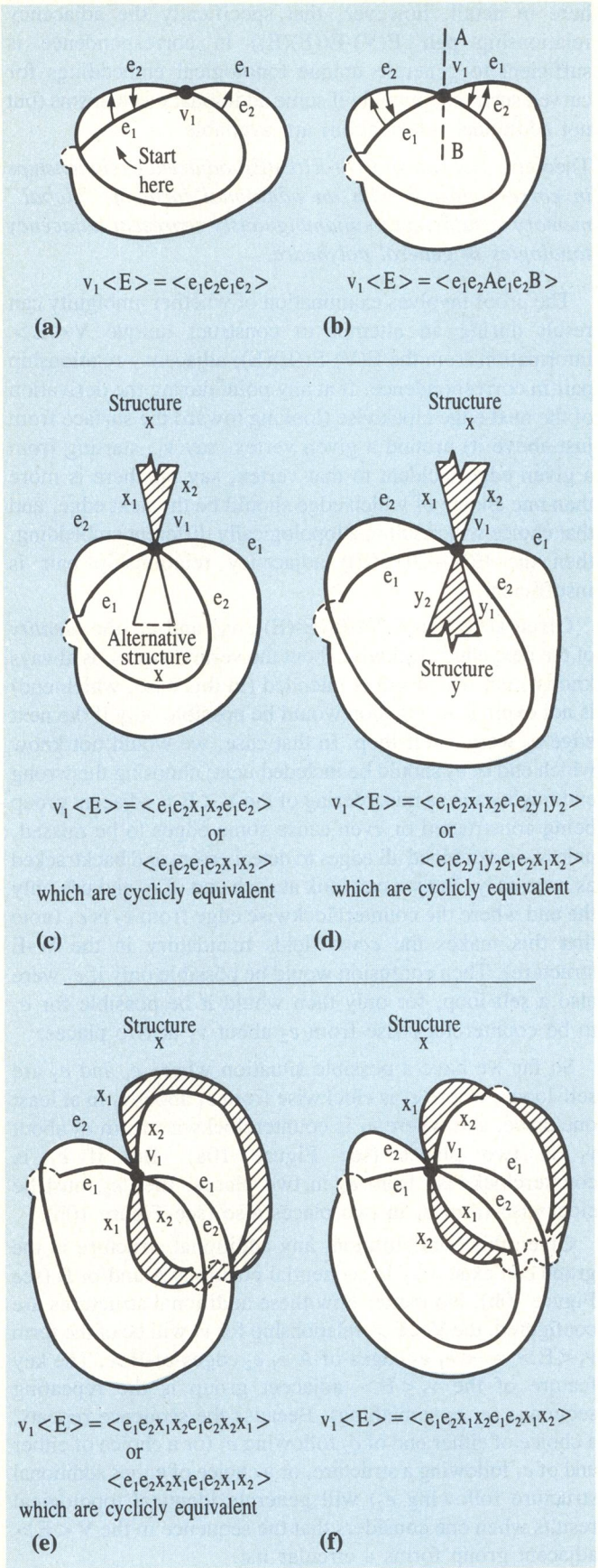


Figure 10. Sufficiency of E(V)—E((E)(E)) adjacency relationship pair in correspondence.

given E(V)-E((E)(E)), there is no basis for distinguishing between them in terms of topological adjacencies.

Note, however, that for this adjacency relationship pair to be sufficient, the ability to recognize (remember) which of the ends of an edge have been used already must be present; otherwise, spurious infinite sequences could be generated for $v_1<E>$. This is the "global" memory we previously referred to—an ability to mark an edge end and return to it later to determine its status. We assume this capability since both adjacent groups of E(V)-E((E)(E)) are ordered as part of correspondence, and a marker field would be easy to add to the W-E structure. It is also possible to embed this memory in the local states of procedures that operate on the W-E structure.

To further demonstrate that identical embeddings will be generated regardless of which ends of the edges are used in the only ambiguous situations, Figure 10 shows all three possible configurations of additional structures: no additional structure (10b), one additional structure at A or B (10c), two additional structures at A and B, which may be disconnected (10d) or connected (10e and 10f). In any of the cases, the repeating sequence in the circular group controlled by the original adjacencies of e_1 and e_2 about v_1 guarantees that either end of an edge will generate an identical embedding in the only situation where confusion could arise, as long as the edge is used only twice (this limitation can be guaranteed by use of a marker field).

It should be noted that objects with multiple self-loops are not necessarily esoteric or unimportant; Figure 11 shows a fairly familiar example of such an object.

Since $F<E>$ is also a sufficient adjacency relationship, the configurations of v_1, e_1, and e_2 will also create repeating or identical sequences in the adjacent groups of the $F<E>$ information. We would need marker fields or procedural-state memory to remove any confusion, but in this case the sequence would be associated with each of the two edge sides instead of ends. The purpose of using these additional mechanisms of extended pointers and mark bits is to easily disambiguate which use of the edge is intended in a given adjacency relationship (which end or side).

Since the derivation of $V<E>$ information is therefore unique, E(V)-E((E)(E)) is sufficient to represent polyhedral topologies by way of the Edmonds theorem. Perhaps more important, this verifies in detail that the winged-edge structure E(V)-E((E)(E))-E(F), used in several solid modeling systems, is indeed sufficient for the representation of polyhedral topologies for planar-faced polyhedra since it is a superset of the information in the sufficient adjacency relationship pair E(V)-E((E)(E)). For curved-surface polyhedra, however, some additional mechanisms, but not additional information, are necessary to consider the winged-edge structure sufficient.

While the W-E structure is informationally sufficient, additional marker field space and comparatively intelligent algorithms (which check the mandatory backpointers) are required to determine the next edge around a vertex or the next edge around a face in curved-surface domains. Even

in the planar-face environment, however, simple adjacent edge field access is not sufficient to handle traversals of the edges around faces, especially in cases involving struts. Traversal and adjacency relationship access algorithms for the W-E structure must perform conditional testing to derive the proper adjacencies.

(4) Modified W-E structure. Being a superset of the information contained in the W-E structure, the modified W-E structure is also sufficient. In fact, a full proof of the sufficiency of the modified W-E structure is similar to the proof of sufficiency of the F-E structure, since the sufficient F<E> adjacency relationship can be trivially derived from it even under curved-surface conditions. The proof is simpler than the proof of the F-E structure, however, since the other side of the edge is already known.

The modified W-E structure avoids the computational complexity of algorithms for the W-E structure by explicitly including information concerning which of two possible uses of an edge element is intended, via the *cwehalf* and *ccwehalf* fields. Compared with the W-E structure, this structure simplifies access algorithms, especially in curved-surface environments. The additional fields do cause more complexity in accessing than do the V-E and F-E structures, however, as will be seen in the next section.

A comparison of time, space, and complexity requirements

A comprehensive and complete comparison of alternative data structures ultimately involves extensive data gathering and statistical analysis over a wide variety of user applications. Even then, issues will remain regarding the comparative optimality of each implementation and each application of the various alternatives. The analysis here does not intend to be complete or rigorous in this fashion, but it does attempt to reflect an approximate measure of the time and space requirements and the complexity of access algorithms necessary to use the four data structures in a large model environment over the domain for which they are intended to provide information.

Time requirements. An issue that arises when we compare timing performance of alternative data structures over a minimal set of functions is the selection of which functions are representative of actual use and are valid as predictors of performance in actual applications. Two criteria were used in selecting the functions for this performance evaluation:

(1) The operations should be as primitive as possible in the sense that all the information from the data structure can be extracted by one or more applications of the functions.

(2) The operations must not be at too low a level (involved in an extended sequence of field manipulations, for example, as would be required in an actual implementation of the Euler operators), or their value in comparison will be lost since they would be dealing with situations unique to each data structure.

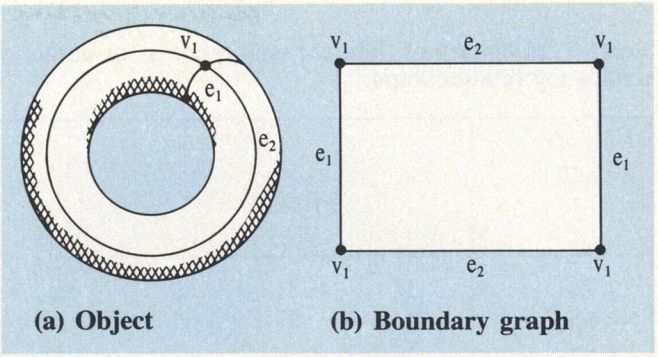


Figure 11. An object with multiple self-loops.

The functions chosen as meeting these criteria and still providing some insight into the four alternative data structures are the functions for obtaining the element adjacency relationships of the embedded graph from the data structures. This choice ensures usage of all information available over the domain of the structures while still remaining at a level reasonably close to, but independent of, the data structures themselves.

At least three criteria are relevant in determining overall time requirements. First, the number of accesses to fields of the data structure records can be considered. Second, the number of record accesses necessary can be considered for database implementations that access data one record at a time. Third, processing time or overall instruction counts can be considered.

The number of field accesses necessary to obtain an adjacency relationship from each data structure is one of the most important criteria in virtual-memory environments; any I/O overhead due to page faults is related to field rather than record accesses because records are never accessed directly as single units. A field access is the reading of a particular field of a particular topological element data structure, such as obtaining the pointer value of the *cwe* field of an edge structure record. The number of field accesses can be an important predictor of the time to obtain the information from the data structure because the access of field information from a record in a computer implementation requires access to main memory at best (usually considerably slower than access to internal registers on today's sequential machines) and can cause a page fault in virtual-memory systems at worst, producing considerable I/O overhead. While an exact prediction would also be based on field sizes, page sizes, memory available, and the amount of the representation already in memory, the number of field accesses can still be considered a reasonable approximate measure of speed in this respect. The access costs in terms of the number of field accesses required to generate the elements of the adjacent group of the nine adjacency relationships with respect to a given reference element are summarized in Table 1.

Table 1. Summary of field accessing costs for deriving adjacency relationships.

Adjacency Relationship	Representation			
	W—E	Modified W—E	V—E	F—E
V<V>	3	3	2[1]	3
V<E>	2	2	1	2
V<F>	3	3	2	3
E(V)	3	2	3	3
E((E)(E))	5	4	5	5
E(F)	3	2	3	3
F<V>	3	3	3	2
F<E>	3	2	2	1
F<F>	4	3	3	2[2]

[1]Would be 3 if other vertex used in edge end representation.
[2]Would be 3 if other face used in edge side representation.

Table 2. Summary of record accessing costs for deriving adjacency relationships.

Adjacency Relationship	Representation			
	W—E	Modified W—E	V—E	F—E
V<V>	1	1	1	2
V<E>	1	1	1	2
V<F>	1	1	1	2
E(V)	1	1	2	2
E((E)(E))	1	1	2	2
E(F)	1	1	2	2
F<V>	1	1	2	1
F<E>	1	1	2	1
F<F>	1	1	2	1

Table 3. Representation storage requirements per edge.

Representation	Number of Pointer Fields
W—E	8
Modified W—E	8
V—E	10
V—E	10

Record access costs are particularly relevant in database implementations where each topological element data structure is stored as a separate database record, and the entire record must be retrieved individually before any field access is permitted. In this case the cost of retrieving a record is high, involving disk accesses, while the cost of accesses to fields of a retrieved record are by comparison low and therefore less significant. The record access costs for each of the four data structures are shown in Table 2.

Processing time can probably be considered the least important, though not insignificant, factor in evaluating overall timing costs, since the time penalty for a few additional instructions is relatively small compared to the possible delays from record or field accesses or the delay due to I/O overhead.

The accessing procedures used for determining the number of field and record accesses necessary are given in detail in the appendix. The first section of the appendix describes the actual query to be answered by each adjacency relationship access procedure through operations on the data structures presented. The second section describes initial conditions assumed to be in effect by the accessing procedures, including initialization of local variables and loop termination conditions. The remaining sections describe the actual accessing procedures for the four structures.

These procedures utilize the edge data structures shown in Figures 3 through 6. Where necessary, the accessing procedures presented perform extra assignments to reduce the number of field accesses necessary, utilizing temporary variables. The list of procedures in the appendix can be used to examine the number of temporary variable assignments performed to obtain the figures in the tables.

Since adjacent groups of elements in ordered adjacency relationships are usually circular lists of elements, the accessing procedures for finding an adjacent element are required to perform all setup necessary to continue the operation of finding the following element in the circular list. Thus each accessing procedure given is essentially the body of a loop that enumerates all elements in the adjacent group of the adjacency relationship. This ensures that the total accessing cost is uniformly distributed over all accesses and is included in the comparisons. The field and record access counts given are for each iteration of the loop—that is, for each individual member of the adjacent group found during the adjacency relationship access. For record access counts, it is assumed that the last record accessed by the immediately preceding iteration is available.

Several comments are in order. The W-E and modified W-E structures are superior with respect to record access requirements. As seen in Table 2, the W-E structure and modified W-E structure accessing procedures require only one record access per adjacency relationship access. For six out of the nine adjacency relationship accesses, the split edges of the V-E and F-E structures cause two record accesses. Superiority in terms of database record accesses, however, may not be as important in virtual-memory implementations.

In terms of overall field-accessing costs, the modified W-E structure is superior to the W-E structure, even in planar-surface environments. The V-E and F-E structures, however, offer performance equal to or better than the modified W-E structure for adjacency relationship accesses where the edge is not the reference element. Efficiency for this situation is often more important in practice, since the field-accessing costs given in Table 1 must be multiplied by the number of elements found during traversal of the adjacent group to obtain the total adjacency relationship field access costs.

Also of note is the symmetry of the field- and record-accessing costs of the V-E and F-E structures. Overall, the two structures have essentially identical accessing costs, with the V-E structure biased for more efficient access of adjacency relationships using the vertex as the reference element, and the F-E structure biased for more efficient access of adjacency relationships using the face as the reference element. Both perform equally in obtaining adjacency relationships using the edge as the reference element.

Space requirements. The number of pointers required for complete edge adjacency information in each of the four data structures is shown in Table 3. In general, the V-E and F-E structures are slightly larger than the W-E and modified W-E structures due to the need for additional pointers to coordinate their split edge structures.

A comparison of the space requirements for the full topological data structure of prismatic and approximated spherical polyhedral objects is shown in Table 4. Assumptions made to allow comparison are that pointers are 32 bits in length and that an eight-bit byte is the minimum-size storage unit. Sizes of face (13 bytes) and vertex (12 bytes) records were derived from the support record structures defined earlier. Individual vertex coordinate values were assumed to be 32 bits in length, but no other geometry was assumed. These choices have a tendency to maximize the apparent difference between the alternative structures; the figures presented are therefore worst-case differences. In a more realistic situation additional information storage (such as larger coordinate values) would reduce the percentage of space attributable to differences in the edge structures.

The additional single-bit side fields of the modified W-E structure are not considered here since it is a small amount of space compared to full pointer values. The extra marker bit fields for the regular W-E structure in curved-surface environments are also not included, biasing the comparison slightly in its favor. Space for these fields is necessary in the W-E structure either explicitly in the edge structure or implicitly in the state of the accessing procedures used (such as state information present in recursion stacks). When present explicitly, these marker fields make the storage costs of the W-E and modified W-E structures equivalent.

In general, for the types of objects considered, the V-E and F-E structures required about 18 percent more storage than the W-E and modified W-E structures.

As we see in the tables, none of the structures provides a minimal-size representation of the objects considered, in contrast to many procedural or other conceptualizations of the objects. In general, the four representation structures are not intended to be minimal-size storage formats but to provide quick access to adjacent elements during the manipulation and creation of the solid model of an object. An equally important feature of all the representations is their ability to maintain their validity without requiring a change of representational form regardless of the number of manipulations made to the model.

Accessing-algorithm complexity. The complexity of the algorithms necessary to manipulate the data structures is sometimes a more important evaluation criterion for implementations than speed. Reduction of the resources for code creation and maintenance, as well as greater reliability, hinges on the simplicity of the manipulation algorithms.

As the appendix shows, the W-E structure involves the most complex accessing strategy, necessitated by its representation of the edge adjacencies in a single unified structure. The W-E structure must determine continually which edge side or end was intended every time an edge pointer appears in a field used to find an adjacent element.

Table 4. Typical storage requirements for some solid objects.

Object Type	Number of Faces	Number of Edges	Number of Verts	Total Size in Bytes		% Increase
				W—E and Modified W—E	V—E or F—E	
Prism						
4-sided	6	12	8	558	654	17.20
4000-sided	4002	12000	8000	532026	628026	18.04
Approximated sphere						
32 quad facets	32	56	26	2520	2968	17.28
32K quad facets	32768	65280	32514	2905112	3427352	17.98

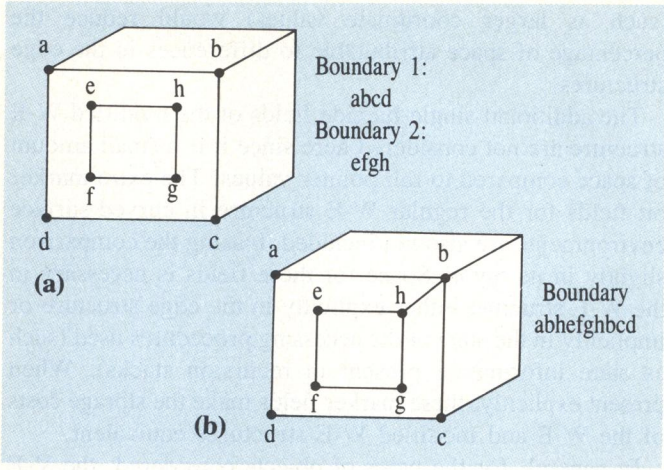


Figure 12. Artifact edge technique: hole in face (a); artifact edge (b).

The planar polyhedral environment implementation described in the third section of the appendix, makes this determination by carrying along a vertex as well as an edge pointer during traversals of adjacent groups, using the vertex pointer information to determine which side is indicated. Note the effect of this scheme on the field-accessing costs of the F<F> relationship (Table 1). In most implementations of the W-E structure, this determination involves either carrying along additional information, as done in the procedures presented, or using marker bit fields; both techniques require additional processing during traversals. The accessing procedures described for the W-E structure are valid only for planar polyhedral environments; curved-surface environments require more complex accessing procedures and/or additional marker field space, and usually a larger number of field accesses, as demonstrated in the proof of sufficiency of the W-E structure.

The modified W-E structure avoids the accessing-algorithm complexity of the W-E structure by the use of explicit side fields. The extra fields do require extra field accesses to process, however, and the resulting algorithms, while much simpler than for the W-E structure, especially in curved-surface environments, are still more complex than those of the V-E and F-E structures.

The algorithms of the V-E and F-E structures are nearly identical in a symmetrical fashion, though the actual semantics of the algorithms vary.

Extensions for disconnected graphs

Up to this section we have assumed that the embedded-boundary graphs being used were connected graphs. We now dispense with this restriction in favor of extensions of the previously given support data structures and topological elements to allow direct representation of faces with multiple boundaries and objects with multiple shells.

Multiply connected faces. There are several ways to handle multiply connected faces (faces with more than one boundary contour yet still possessing a single connected surface area—such as a face with a hole in it) in boundary-graph-based solid modeling representations. Unrestricted use of multiply connected faces can lead to disconnected-graph conditions, although presence of a multiply connected face does not necessarily mean the graph is disconnected.

Multiply connected faces can be simulated by the artifact edge technique in which an additional edge joins each boundary contour to some other contour on the same face (see Figure 12). The artifact edge therefore has the same face adjacent to both of its sides. This technique is not as desirable as the direct approach, however, since it not only demands that the modeling system determine exactly how to connect the contours with artifact edges, but also increases the number of edges in a model—a situation that becomes worse when a model is subjected to many modeling operations that further subdivide the artifact edges (such as Boolean operations or section cuts).

Changes of the graph data structures to directly support multiply connected faces without use of the artifact edge technique do not affect the data structures at the edge level but rather at the face level.

There are several ways to modify the face structures already described to handle multiply connected faces directly. One technique is an additional fixed-length structure called a loop, which is required for each boundary contour associated with a face. An alternative implementation might be simply to have a variable-length face record with a pointer for an edge (or the single vertex) in every contour associated with the face. While the two approaches are informationally equivalent, we prefer the fixed-length loop structure approach, since most current popular programming languages are not adept at providing data objects with dynamically variable length.

As shown in Figure 13, the loop structure simply provides the face structure with a mechanism to maintain a linked list of pointers to boundary contours. Thus each contour adjacent to the face has, in the linked list, a loop structure that has a pointer to one of the edge structures (or the vertex) associated with it.

Of note in the loop structure presented is the Pascal record variant to handle the case in which the contour consists of a single vertex rather than a series of edges. The situation was handled in the face record structure definition given earlier. This is the case, previously mentioned, where an initialization step in the creation of an object allows the object to consist of only one shell, one vertex, one face, and no edges. This general situation can also happen, though, when a face contains many separate vertices on the interior of its surface, perhaps as a transitional state. For this reason the record variant approach, associated with the loop level in the data structure, is preferable to treating the situation as a special initialization condition at the shell level.

Naturally, if backpointers are used in the edge records, as is true with the four edge structures presented, what were previously edge-to-face pointers become edge-to-loop pointers when the loop structure is added to the scheme.

Since the loop structure explicitly stores the boundary contour relationships, it is biased toward the top-down hierarchical approach of maintaining relationships between higher and lower level dimensionality elements. The use of this structure seems more natural with the W-E and F-E structures for this reason. A representation using the V-E edge structure would probably be vertex centered rather than face centered in organization. In this case loops may not be considered an important concept; equivalent information can be derived if the edge-to-face pointers are preserved. The top-down hierarchical approach is often preferable, however, for reasons already discussed.

Multiple-shell objects. Another situation encountered in solid modeling is that a solid object may contain one or more hollow cavities but still consist of a single connected volume. This case also requires the ability to handle disconnected graphs. Unlike the multiply connected face situation discussed above, however, more than one surface is necessary; thus there must be changes in the data structure above the face level. This requirement can be handled most simply by a list of separate shells in an object.

In keeping with a hierarchical approach, however, it is also possible to maintain topological information on the containment relationships of the shells. We can do this by utilizing a binary tree structure in which one branch of a node is used to list shells inside the shell associated with that node, and the other branch to list those shells outside it (see Figure 14). Maintaining this additional topological information can increase efficiency in many geometric modeling applications, such as interference detection. This approach can be used to represent not only single solid volumes with multiple voids, but also solids within the voids, voids in those solids, and so on, in what amounts to a containment classification of space.

Both the loop and shell containment techniques were utilized in a solid modeler based on the W-E structure,[9] although they are equally applicable to the other three structures presented here.

(a) Pascal declarations

```
type    sptr = ^shell;
        lptr = ^loop;

        shell = record
                  sinside, soutside: sptr;
                  facelist: fptr
                  end;

        face = record
                 next, last: fptr
                 loop: lptr
                 end;

        loop = record
                 next: lptr;
                 case onevertex: boolean of
                           True: (vert: vptr);
                           False: (edgelist: eptr {or ehptr})

                 end;
```

(b) Storage allocation description

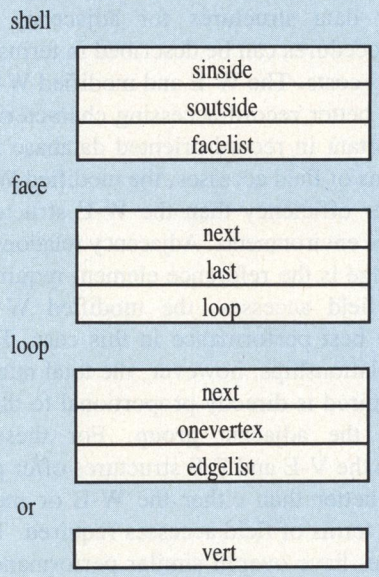


Figure 13. Data structures for disconnected graphs.

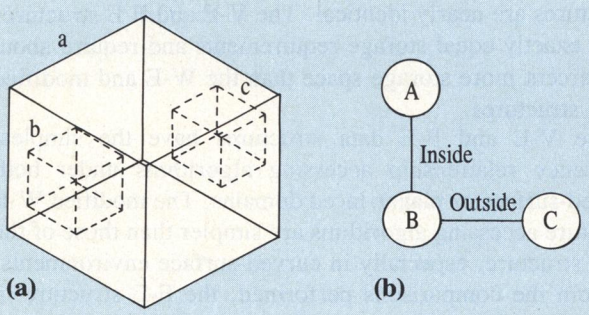


Figure 14. Shell containment tree: shell with multiple cavities (a); binary shell containment tree (b).

Conclusions

We have looked at four different data structures, the W-E (winged edge), the modified W-E, the V-E (vertex-edge), and the F-E (face-edge) structures for the representation of topological information in solid modeling. After specifying their domain, we examined their topological sufficiency and found three of the four data structures sufficient as defined. The W-E structure, however, required additional temporary

data fields and/or more complex accessing algorithms to allow sufficiency in curved-surface environments.

Two general principles of interest in solid modeling emerged from this study. First, to provide a reliable theoretical basis for a representation, the domain of a modeling representation must be carefully specified and the sufficiency of the representation over that domain must be proven. This is important because it improves the chances of avoiding major representational pitfalls during design and implementation, as well as other errors affecting reliability of the modeling system. Some kinds of representational errors may not appear until well after the implementation effort, especially since it is unlikely that the system will be exercised over the entire domain during a testing phase if the domain is not specified or is specified improperly.

Second, it is important in boundary-based solid modeling systems that representations of topological element adjacency information represent the *occurrence* of elements in the adjacencies, not the elements themselves. Recognition of this fact in the design of relevant data structures simplifies the accessing algorithms.

We explored the time, space, and complexity requirements of the four data structures. The time operating characteristics of the four data structures for adjacency relationship accessing procedures can be described in terms of field and record access costs. The W-E and modified W-E structures clearly have better record-accessing characteristics, which can be important in record-oriented database implementations. In terms of field accesses, the modified W-E structure offers greater efficiency than the W-E structure, even in planar-surface environments. Adjacency relationship accesses where the edge is the reference element require a constant number of field accesses; the modified W-E structure provides the best performance in this case. For all other adjacency relationships, however, the total number of field accesses required is directly proportional to the number of elements in the adjacent group. For these adjacency relationships the V-E and F-E structures offer performance equal to or better than either the W-E or modified W-E structures in terms of field accesses required. The V-E and F-E structures have overall similar performance, although each performs slightly better than the other on different adjacency relationship accesses in a symmetric fashion.

The space requirements for the W-E and modifed W-E structures are nearly identical. The V-E and F-E structures have exactly equal storage requirements and require about 20 percent more storage space than the W-E and modified W-E structures.

The V-E and F-E data structures have the simplest adjacency relationship accessing algorithms under both curved-surface or planar-faced domains. The modified W-E structure accessing algorithms are simpler than those of the W-E structure, especially in curved-surface environments.

From the comparisons performed, the F-E structure in particular seems quite promising for a top-down hierarchical implementation of an edge-oriented, evaluated, object-based, boundary-graph-based solid modeling system. ■

Acknowledgment

The author is currently pursuing a PhD at Rensselaer Polytechnic Institute under the direction of Michael Wozny. This article is based on part of a thesis to be submitted in partial fulfillment of the PhD degree requirements in the Department of Electrical, Computer, and Systems Engineering at RPI.

Appendix: accessing procedures and costs of obtaining adjacencies

Adjacency relationship queries

v<V>—find the clockwise-ordered circular list* of vertices surrounding v

v<E>—find the clockwise-ordered circular list of edges surrounding v

v<F>—find the clockwise-ordered circular list of faces surrounding v

e(V)—find the two vertices of e (ordered access)

e((E)(E))—find the edges adjacent to e which precede and follow e in the F<E>-adjacent groups of the two faces adjacent to e (ordered access)

e(F)—find the two faces adjacent to e (ordered access)

f<V>—find the ordered circular list of vertices surrounding f, where the area of f is found to the right when traversing the sequence of vertices

f<E>—find the ordered circular list of edges surrounding f, where the area of f is found to the right when traversing the sequence of edges

f<F>—find the ordered circular list of faces surrounding f, where the area of f is found to the right when traversing the sequence of faces

Initial conditions for queries

v<V>, v<E>, v<F>

W-E: given vertex structure v = *vfirst* and edge structure e = *efirst* for which vert[1][e] = v or *vert*[2][e] = v (*vfirst*, *efirst* is used to detect end of loop)

modified W-E: given vertex structure v = *vfirst*, edge structure e = *efirst*, and side indicator h = *hfirst* for which *vert*[h][e] = v (*efirst*, *hfirst* is used to detect end of loop)

V-E,F-E: given vertex structure v and edge half structure e = *efirst* such that *vert*[*othereh*[e]] = v (*efirst* is used to detect end of loop)

e(V),e((E)(E)),e(F)

W-E: given edge structure e and vertex structure v to determine ordering such that vert[1][e] = v or *vert*[2][e] = v

modified W-E: given edge structure e and side indicator h to determine ordering such that *vert*[h][e] = v

V-E,F-E: given edge half structure e selected to determine ordering

*Adjacent group circular lists with the vertex as the reference element type are clockwise ordered as viewed from just above the surface and just outside the volume of the solid body looking toward the surface.

f<V>, *f*<E>, *f*<F>

W-E: given face structure *f* and an edge structure *e* = *efirst* such that *face*[1][*e*] = *f* or *face*[2][*e*] = *f*, select *v* = *vfirst* such that *v* = *vert*[*n*] (*vfirst*, *efirst* is used to detect end of loop)

modified W-E: given face structure *f*, edge structure *e* = *efirst*, and side indicator *h* = *hfirst* such that *face*[*flip*(*h*)][*e*] = *f*, where *flip*(*h*) gives opposite side as *h* (*efirst*, *hfirst* is used to detect end of loop)

V-E,F-E: given face structure *f* and edge half structure *e* = *efirst* such that *face*[*othereh*[*e*]] = *f* (*efirst* is used to detect end of loop)

W-E structure adjacency relationship accessing procedures (for connected-graph planar polyhedral environments only)

V<V>—three field accesses/one record access
```
if (v = vert[1][e])
  then half ← 1; otherhalf ← 2
  else half ← 2; otherhalf ← 1
adjacentv ← vert[otherhalf][e]
e ← ccwe[half][e]
```
V<E>—two field accesses/one record access
```
if (v = vert[1][e])
  then half ← 1
  else half ← 2
e ← ccwe[half][e]
```
V<F>—three field accesses/one record access
```
if (v = vert[1][e])
  then half ← 1
  else half ← 2
f ← face[half][e]
e ← ccwe[half][e]
```
E(V)—three field accesses/one record access
```
if (v = vert[1][e])
  then half ← 1; otherhalf ← 2
  else half ← 2; otherhalf ← 1
v1 ← vertex[half][e]
v2 ← vertex[otherhalf][e]
```
(E((E)(E))—five field accesses/one record access
```
if (v = vert[1][e])
  then half ← 1; otherhalf ← 2
  else half ← 2; otherhalf ← 1
e1 ← ccwe[otherhalf][e]
e2 ← cwe[half][e]
e3 ← ccwe[half][e]
e4 ← cwe[otherhalf][e]
```
E(F)—three field accesses/one record access
```
if (v = vert[1][e])
  then half ← 1; otherhalf ← 2
  else half ← 2; otherhalf ← 1
f1 ← face[half][e]
f2 ← face[otherhalf][e]
```
F<V>—three field accesses/one record access
```
if (v = vert[1][e])
  then half ← 1; otherhalf ← 2
  else half ← 2; otherhalf ← 1
v ← vert[otherhalf][e]
e ← cwe[half][e]
```
F<E>—three field accesses/one record access
```
if (v = vert[1][e])
  then half ← 1; otherhalf ← 2
  else half ← 2; otherhalf ← 1
v ← vert[otherhalf][e]
e ← cwe[half][e]
```
F<F>—four field accesses/one record access
```
if (v = vert[1][e])
  then half ← 1; otherhalf ← 2
  else half ← 2; otherhalf ← 1
f ← face[half][e]
v ← vert[otherhalf][e]
e ← cwe[half][e]
```

Modified W-E structure adjacency relationship accessing procedures (for all connected-graph environments)

V<V>—three field accesses/one record access
```
adjacentv ← [flip(h)][e]
  {where flip(n) gives opposite of n}
olde ← e
e ← ccwe[h][e]
h ← ccwehalf [h][olde]
```
V<E>—two field accesses/one record access
```
olde ← e
e ← ccwe[h][e]
h ← ccwehalf [h][olde]
```
V<F>—three field accesses/one record access
```
f ← face[h][e]
olde ← e
e ← ccwe[h][e]
h ← ccwehalf [h][olde]
```
E(V)—two field accesses/one record access
```
v1 ← vertex[h][e]
v2 ← vertex[flip(h)][e]
```
E((E)(E))—four field accesses/one record access
```
e1 ← ccwe[flip(h)][e]
e2 ← cwe[h][e]
e3 ← ccwe[h][e]
e4 ← cwe[flip(h)][e]
```
E(F)—two field accesses/one record access
```
f1 ← face[h][e]
f2 ← face[flip(h)][e]
```
F<V>—three field accesses/one record access
```
v ← vert[h][e]
olde ← e
e ← cwe[h][e]
h ← cwehalf [h][olde]
```
F<E>—two field accesses/one record access
```
olde ← e
e ← cwe[h][e]
h ← cwehalf [h][olde]
```
F<F>—three field accesses/one record access
```
f ← face[h][e]
olde ← e
e ← cwe[h][e]
h ← cwehalf [h][olde]
```

V-E structure adjacency relationship accessing procedures (all connected-graph environments)

V<V>—two field accesses/one record access
```
e ← cwe[e]
v ← vertex[e]
```
V<E>—one field access/one record access
```
e ← cwe[e]
```
V<F>—two field accesses/one record access

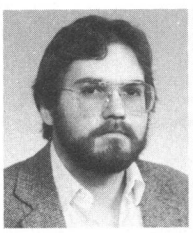
e ← *cwe*[*e*]
f ← *face*[*e*]
E(V)—three field accesses/two record accesses
*v*1 ← *vertex*[*e*]
*v*2 ← *vertex*[*othereh*[*e*]]
E((E)(E))—five field accesses/two record accesses
*e*1 ← *cwe*[*e*]
*e*2 ← *ccwe*[*e*]
etemp ← *othereh*[*e*]
*e*3 ← *cwe*[*etemp*]
*e*4 ← *ccwe*[*etemp*]
E(F)—three field accesses/two record accesses
*f*1 ← *face*[*e*]
*f*2 ← *face*[*othereh*[*e*]]
F<V>—three field accesses/two record accesses
e ← *othereh*[*ccwe*[*e*]]
v ← *vertex*[*e*]
F<E>—two field accesses/two record accesses
e ← *othereh*[*ccwe*[*e*]]
F<F>—three field accesses/two record accesses
e ← *othereh*[*ccwe*[*e*]]
f ← *face*[*e*]

F-E structure adjacency relationship accessing procedures (all connected-graph environments)

V<V>—three field accesses/two record accesses
e ← *ccwe*[*othereh*[*e*]]
v ← *vertex*[*e*]
V<E>—two field accesses/two record accesses
e ← *ccwe*[*othereh*[*e*]]
V<F>—three field accesses/two record accesses
e ← *ccwe*[*othereh*[*e*]]
f ← *face*[*e*]
E(V)—three field accesses/two record accesses
*v*1 ← *vertex*[*e*]
*v*2 ← *vertex*[*othereh*[*e*]]
E((E)(E))—five field accesses/two record accesses
etemp ← *othereh*[*e*]
*e*1 ← *ccwe*[*etemp*]
*e*2 ← *cwe*[*e*]
*e*3 ← *ccwe*[*e*]
*e*4 ← *cwe*[*etemp*]
E(F)—three field accesses/two record accesses
*f*1 ← *face*[*e*]
*f*2 ← *face*[othereh[*e*]]
F<V>—two field accesses/one record access
e ← *cwe*[*e*]
v ← *vertex*[*e*]
F<E>—one field access/one record access
e ← *cwe*[*e*]
F<F>—two field accesses/one record access
e ← *cwe*[*e*]
f ← *face*[*e*]

References

1. K. Weiler, "Topology as a Framework for Solid Modeling," *Proc. Graphics Interface 84,* Ottawa, Ont., Canada, May 1984 (extended abstract).
2. F. Harary, *Graph Theory,* Addison-Wesley, Reading, Mass., 1972.
3. B. Arnold, *Intuitive Concepts in Elementary Topology,* Prentice-Hall, Englewood Cliffs, N.J., 1962.
4. M. Agoston, *Algebraic Topology,* Marcel Dekker, New York, 1976.
5. A. Baer, C. Eastman, and M. Henrion, "Geometric Modelling: a Survey," *Computer-Aided Design,* Vol. 11, No. 5, Sept. 1979, pp. 253-272.
6. K. Weiler, "Adjacency Relationships in Boundary-Graph-Based Solid Models," June 1983 (to be published).
7. B. Baumgart, "Winged-edge Polyhedron Representation," Stanford Artificial Intelligence Report No. CS-320, Oct. 1972.
8. B. Baumgart, "A Polyhedron Representation for Computer Vision," *AFIPS Proc.*, Vol. 44, 1975, pp. 589-596.
9. C. Eastman and K. Weiler, "Geometric Modeling Using the Euler Operators," *Conf. Computer Graphics in CAD/CAM Systems,* May 1979, pp. 248-259.
10. I. Braid, R. Hillyard, and I. Stroud, "Stepwise Construction of Polyhedra in Geometric Modelling," CAD Group Document No. 100, Univ. of Cambridge Computer Laboratory, Oct. 1978.
11. M. Mantyla and R. Sulonen, "GWB: A Solid Modeler with Euler Operators," *IEEE Computer Graphics and Applications,* Vol. 2, No. 7, Sept. 1982, pp. 17-31.
12. M. Mantyla, "A Note on the Modeling Space of Euler Operators," *Computer Vision, Graphics and Image Processing,* Vol. 26, 1984, pp. 45-60.
13. D. Stoker, "CRIPL-Edge Data Structure," unpublished, Carnegie-Mellon Univ., May 1974.
14. C. Eastman and H. Henrion, "GLIDE: A Language for Design Information Systems," *Computer Graphics,* (Proc. Siggraph 77), Vol. 11, No. 2, Summer 1977, pp. 24-33.
15. J. Edmonds, "A Combinatorial Representation for Polyhedral Surfaces," *American Mathematical Society Notices,* Vol. 7, Oct. 1960, pp. 646.
16. J. Graver and M. Watkins, *Combinatorics with Emphasis on the Theory of Graphs,* Springer-Verlag, New York, 1977.
17. A. White, *Graphs, Groups, and Surfaces,* Mathematical Studies 8, North-Holland, Amsterdam, 1973.
18. P. Hanrahan, "Creating Volume Models from Edge-Vertex Graphs," *Computer Graphics* (Proc. Siggraph 82), Vol. 16, No. 3, July 1982, pp. 77-84.



Kevin Weiler is a graphics scientist with General Electric Corporate Research and Development. From 1977 to 1981 he was at Carnegie-Mellon University, where he was senior scientist and managed the CAD-Graphics Laboratory. His research interests include geometric modeling, interactive graphics, and design databases.

Weiler received an MS in computer graphics from Cornell University in 1978 and a BA from the University of Maryland in 1975. He is currently a PhD candidate in computer and systems engineering at Rensselaer Polytechnic Institute. He is a member of IEEE and ACM.

Weiler's address is General Electric Corporate R&D, Bldg. 37-569, PO Box 43, Schenectady, NY 12301.