TOPOLOGICAL STRUCTURES FOR GEOMETRIC MODELING

by

Kevin J. Weiler

A Thesis Submitted to the Graduate
Faculty of Rensselaer Polytechnic Institute
in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major Subject: Computer and Systems Engineering

Approved by the Examining Committee:

Prof. Michael J. Wozny, Thesis Advisor

Prof. W. Randolph Franklin, Member

Prof. Harry W. McLaughlin, Member

Prof. Mark S. Shephard, Member

Rensselaer Polytechnic Institute
Troy, New York

August 1986

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

There are many people that I'd like to thank for their help, support, and friendship during the lengthy production of this thesis. I am deeply grateful to each of them.

Dr. Michael Wozny, my advisor and doctoral committee chairman, has been helpful in both academic guidance and in the development of the research presented here. He has also put together and directs an impressive education and research laboratory, the RPI Center for Interactive Computer Graphics. And I certainly wouldn't forget Mary Johnson and her staff, who have been helpful on numerous occasions and do a superb job of keeping the administrative functions at the CICG running smoothly.

My doctoral committee, W. Randolph Franklin, Harry McLaughlin, Mark Shephard, and Michael Wozny have provided feedback, encouragement, and stimulating conversations.

Several people contributed through their careful reading and commenting on a draft of the thesis, including William Charlesworth, Philip Kennicott, Peter Noel, and Peter Wilson.

Several people three years ago commmented on the paper "Adjacency Relationships in Boundary Graph Based Solid Models" which eventually found its way into this thesis: Peter Atherton, Brian Barsky, Patrick Hanrahan, and Uri Shani.

The GE CR&D CAD Branch has provided a friendly and helpful environment which helped to foster much of this research.

Rida Farouki gave permission to use a figure from his paper [Farouki 86], which appears here as Figure 20 – 1, and has also provided comments on some portions of the thesis.

The GE Educational program provided financial support.

My current and former immediate GE CR&D management provided support for this line of research, and, when necessary, helped cut red tape: Leonidas Jones, Virgil Lucke, and Peter Wilson.

The GE CR&D graphics operation produced most of the artwork found here, with Diane Stephens coordinating the effort.

Dominick Darkangelo and John Spaeth kept my computer backed up and the laser printer running.

Most of all, I'd like to thank my family: my wife, Arlyn, for her support, patience, encouragement, and love; my son Jay, a newcomer, for his love; and my parents and relatives for their support and encouragement.

ABSTRACT

Geometric modeling technology for representing three-dimensional objects has progressed from early wireframe representations, through surface representations, to the most recent representation, solid modeling. Each of these forms has many possible representations.

The boundary representation technique, where the surfaces, edges, and vertices of objects are represented explicitly, has found particularly wide application. Many of the more sophisticated versions of boundary representations explicitly store topological information about the positional relationships among surfaces, edges, and vertices.

This thesis places emphasis on the use of topological information about the shape being modeled to provide a framework for geometric modeling boundary representations and their implementations, while placing little constraint on the actual geometric surface representations used.

The major thrusts of the thesis fall into two areas of geometric modeling.

First, a theoretical basis for two-manifold solid modeling boundary topology representations is developed. The minimum theoretical and minimum practical topological adjacency information required for the unambiguous topological representation of manifold solid objects is determined. This provides a basis for checking the correctness of existing and proposed representations. The correctness of the winged edge structure is also explored, and several new representations which have advantages over existing techniques are described and their sufficiency verified.

Second, a non-two-manifold boundary geometric modeling topology representation is developed which allows the unified and simultaneous representation of wireframe, surface, and solid modeling forms, while featuring a representable range beyond what

is achievable in any of the previous modeling forms. In addition to exterior surface features, interior features can be modeled, and non-manifold features can be represented directly. A new data structure, the Radial Edge structure, which provides access to all topological adjacencies in a non-manifold boundary representation, is described and its completeness is verified. A general set of non-manifold topology manipulation operators is also described which is independent of a specific data structure and is useful for insulating higher levels of geometric modeling functionality from the specifics and complexities of underlying data structures.

The coordination of geometric and topological information in a geometric modeling system is also discussed.

Chapter 1

INTRODUCTION

Geometric modeling technology for representing three-dimensional objects has progressed from early wireframe representations, through surface representations, to the most recent representation, solid modeling. Each involves increasing amounts of information about the shape being modeled, and provides correspondingly more sophisticated functionality. Yet each modeling form still retains unique characteristics which make it most appropriate under certain application requirements.

Each of these forms has many possible representations. One kind of representation technique that has found wide application is the boundary representation technique, where the surfaces, edges, and corner vertices of objects are represented explicitly. Many of the more sophisticated versions of boundary representations explicitly store topological information about the positional relationships among surfaces, edges, and vertices.

This thesis explores boundary based, object based, evaluated representational forms which explicitly store topological information, because these have shown wide application in industrial modeling as well as in other environments.

The thesis places emphasis on the use of topological information about the shape being modeled to provide a framework for the modeling representation and implementation, while placing little constraint on the geometric surface representations used. While there are many advantages to this approach, perhaps the most important is that it can provide a stable basis for an implementation to evolve using several geometric surface representation forms, as appropriate to the application requirements. The thesis therefore concentrates on the representation of the topological

framework itself.

The major thrusts of the thesis fall into two areas of geometric modeling.

The first is in the area of manifold boundary representations. Manifold boundary based, object based solid modeling topology representations are the basis of some of the most popular forms of manifold solid modeling representations being used today. In spite of this, most of the topological theoretical exploration of solid modeling has been limited to point set topology, which is of more value in volume based modeling representations. There has previously been little theoretical exploration of algebraic topology to provide a firm theoretical basis for boundary based solid modeling systems. The value of providing a theoretical basis for solid modeling representations is that it provides a basis for checking the correctness of existing and proposed representations and their implementations, and can provide insight which may lead to new representations previously not considered. It also provides a basis for determining the minimal amount of information needed to unambiguously represent a model. The thesis addresses this need, developing a theoretical basis for manifold solid modeling boundary topology representations. The minimum theoretical and minimum practical topological adjacency information required for the unambiguous topological representation of manifold solid objects is determined. The correctness of an existing manifold solid modeling representation is explored, and several new representations which have advantages over existing techniques are identified and proof of their sufficiency is given.

The second is in the area of non-two-manifold topology representations. Little work has been been done in the area of non-manifold boundary based object based geometric modeling representations, and non-manifold representations which explicitly store topological adjacency information are in an entirely new area of research. Yet there are several reasons why such a representation form is useful. A unified representation for combined wireframe, surface, and solid modeling by necessity requires a non-manifold representation, and is desirable since it makes it easy to use the most appropriate modeling form (or combination of forms) in a given application

without requiring representation conversion as more information is added to the model. A unified representation can also have many implementation advantages in terms of lower initial resource investments as well as lower maintenance requirements compared to multiple representation systems. The user interface in a unified representation system also tends to offer a more integrated approach to the end user since the same framework is being manipulated in all cases. Arbitrary geometric information, such as center lines, can be stored in the model with shape descriptions. Composite objects can be modeled directly. With a non-manifold representation, applications such as finite element analysis can for the first time be directly supported in the modeling representation environment, allowing communication between the modeler and analysis application in both directions using the model representation as the communication medium. Closed form Boolean operations are possible in a non-manifold representation. In addition to a non-manifold geometric modeling representation, operators to manipulate the representation greatly simplify implementations. The thesis describes a new data structure, the Radial Edge structure, which provides access to all topological adjacencies in a non-manifold boundary representation, and verifies its completeness. A general set of non-manifold topology manipulation operators is also described which is independent of a specific data structure and is useful for insulating higher levels of geometric modeling functionality from the specifics and complexities of underlying data structures.

The thesis thus contributes to the state of the art in two areas of geometric modeling; first, by establishing and utilizing a theoretical basis for manifold boundary based solid modeling topology representation systems, and second, by investigating a powerful but largely unexplored geometric modeling form through the development of the first non-manifold boundary based geometric modeling representation which explicitly represents topological adjacencies.

1.1. Organization of the Thesis

The thesis is organized into five major sections.

The first major section, "Geometric Modeling", presents an organized view of the geometric modeling field and identifies the position of this new work in the wider geometric modeling context. It provides a philosophical and technical foundation for the work described in the thesis, and also develops the terminology used throughout the thesis. It is primarily intended to provide a minimal mathematical background for the non-mathematician, and a minimal geometric modeling background for those new to modeling.

The second major section, "Manifold Solid Representations", develops a theoretical foundation for boundary based manifold solid modeling topologies, describes and proves the sufficiency of several new data structures, and, for completeness, reviews existing operators to manipulate manifold boundary graph topology representations.

The third major section, "Non-Manifold Representations", describes a new non-manifold boundary graph topology representation which provides a unified representation of wireframe, surface, solid, and non-manifold modeling forms. Completeness of the new data structure is proven. It also presents new general operators to manipulate non-manifold boundary graph topology representations.

The fourth major section, "Topology and Geometry Interface", describes some of the problems in coordinating the topological and geometric representations in geometric modeling systems, and identifies some of the potential techniques to approach these problems. It also points out the correspondence between direct representation of uses of topological elements and representation of parametric geometry surface intersections.

The fifth major section, "Conclusion", concludes the thesis, reviews the major results, and identifies areas for further research.

Five appendices follow. The first, "Topological Sufficiency Under Constraints",

examines topological sufficiency for manifold solid modeling topologies under more restrictions than the domain identified in Chapter 9. The second, "Storage and Accessing Efficiency Comparisons", provides detailed comparisons of the four manifold solid modeling data structures described in Chapter 12, in terms of storage requirements, accessing efficiency, and accessing algorithm complexity. The third, "Traversals of the Radial Edge Structure", describes detailed traversal algorithms for the non-manifold Radial Edge structure. The fourth, "Sufficiency of the Radial Edge Structure", describes detailed algorithms for the derivation of all of the non-manifold adjacency relationships from the Radial Edge structure. The fifth, "Selective Query and Traversal", details a technique for associating multiple independent attributes with topological elements which can later be used for accessing model topological adjacency information selected by combinations of attributes.

1.2. Audience

Much of the appeal of the geometric modeling field is that humans are naturally endowed with an understanding of and interest in the three-dimensional physical world and the spatial relationships of objects in it. This intuition about geometry and topology is already contained in each of us. It takes only a little more effort to study and appreciate these same relationships in the more abstract context of geometric modeling.

The major audience targeted by this thesis is the geometric modeling community. One of my goals in writing it is to demonstrate that a proper theoretical foundation is extremely beneficial in the design and implementation of geometric modeling systems, and that such a foundation is understandable and usable by modeling system implementors. In the case of this study alone, theoretical investigations led to more powerful and general representation systems than the original study was concerned with. To the end of reaching this audience, most of the theoretical parts of the thesis are stated in terms probably most familiar to the geometric modeling and computing communities, perhaps at the expense of disenchanting some who may have preferred

a more traditional notation. With the major exception of the adjacency relationship terminology central to the topic of adjacency topology, use of notation is avoided, and where possible, an intuitive overview of what is going on is attempted.

1.3. Miscellaneous

All of the material in the thesis, unless explicitly stated otherwise, describes original work, except for Chapters 5 and 13, which summarize existing terminology and review existing techniques.

Some of the material contained in this thesis has been previously published. Of note are "Topology as a Framework for Solid Modeling" [Weiler 84], which is incorporated in Chapter 4, and "Edge-based Data Structures for Solid Modeling in Curved Surface Environments" [Weiler 85a], which is incorporated in parts of Chapter 12 and Appendix B.

The thesis also incorporates work from several currently unpublished papers. These include "Adjacency Relationships in Boundary Graph Based Solid Models" [Weiler 83], which is incorporated in Chapter 11 and Appendix A, "The Radial Edge Structure: a Topological Representation for Non-Manifold Geometric Modeling" [Weiler 85b], which is incorporated in Chapters 15, 16, and 17, and "Boundary Graph Operators for Non-Manifold Geometric Modeling Representations" [Weiler 85c], which is incorporated in Chapters 15, 16, and 19.

All of the original work described here was done while pursuing a doctorate at Rensselaer Polytechnic Institute.

SECTION I

GEOMETRIC MODELING

Chapter 2

INTRODUCTION

Geometric modeling currently involves the use of computers to aid in the creation, manipulation, maintenance, and analysis of representations of the geometric shape of two- and three-dimensional objects. It is used in a wide variety of applications including industrial mechanical part design and analysis, engineering and scientific visualization, commercial video and motion picture production, artistic pursuits, and many other areas.

This thesis emphasizes the role of topology in geometric modeling.

This major section has two objectives. First, it provides some background for those unfamiliar with some of the details of geometric modeling and topology. Second, it takes a fresh look at several general geometric modeling concepts in order to provide background for later sections on two important geometric modeling representational forms, manifold and non-manifold representations.

2.1. Organization of This Section

The first of the following chapters, Chapter 3, provides a brief description of the different forms of geometric modeling representations currently available.

Next, Chapter 4 provides an intuitive introduction to the use of topology as a framework in geometric modeling implementations, and provides the philosophical basis for the approach taken in the remainder of the thesis.

Chapter 5 briefly identifies relevant terminology from graph theory, topology, and

geometric modeling.

Chapter 6 discusses topological elements and topological adjacency relationships, and describes a comprehensive terminology to describe important characteristics of adjacency relationships relevant to geometric modeling.

Finally, Chapter 7 discusses the importance of specifying the domain of geometric modeling systems and of proving sufficiency of the representation over that domain.

Chapter 3

FORMS OF GEOMETRIC MODELING

This chapter briefly describes many of the different approaches to geometric modeling representations that have evolved over the last twenty-five years.

3.1. Geometric Modeling Forms

Different forms of geometric modeling can be distinguished based on exactly what is being represented, the amount and type of information directly available without derivation, and what other information can and cannot be derived.

Historically, several different geometric modeling forms have evolved.

Wireframe modeling, one of the earliest geometric modeling techniques, represents objects by edge curves and points on the surface of the object (see Figure 3 – 1a).

Surface modeling techniques, first developed in the early 1960's, go one step further than wireframe representations by also providing mathematical descriptions of the shape of the surfaces of objects (see Figure 3 – 1b). Surface modeling techniques allow graphic display and numerical control machining of carefully constructed models, but usually offer few integrity checking features.

Solid modeling, a technique developed in the early 1970's, explicitly or implicitly contains information about the closure and connectivity of the volumes of solid shapes. It is becoming an increasingly important part of the process

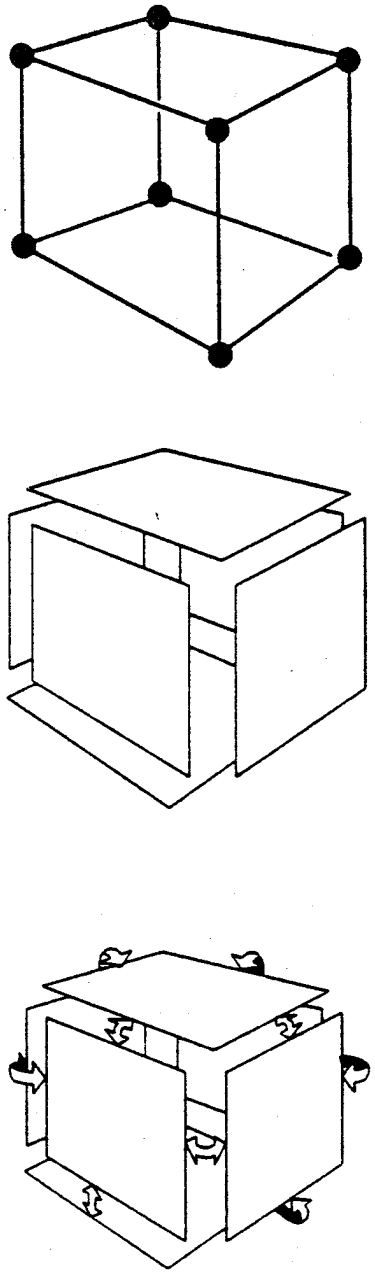


Figure 3 – 1. Wireframe, surface, and solid modeling forms

of computer aided modeling of solid physical objects for design, analysis, manufacturing, simulation, and other applications. Solid modeling offers a number of advantages over previous surface modeling techniques, because it provides a guarantee that any models which are created will form closed and bounded objects more closely related to physically realizable shapes than can be guaranteed for surface models. Figure 3 – 1c illustrates that for boundary based solid models every surface boundary is always directly adjacent to one other surface boundary, guaranteeing a closed volume. Solid models, unlike surface models, enable a modeler system to distinguish the outside of a volume from the inside, allowing mass property analysis for the determination of volume, center of gravity, and the like. Typical solid modeling systems also offer tools for the creation and manipulation of complete solid shapes, while maintaining the integrity of the representations.

Non-manifold geometric modeling, as defined here, is a new modeling form which removes constraints traditionally associated with manifold solid modeling forms by embodying all of the capabilities of the previous three modeling forms in a unified representation and extending the representational domain beyond that of the previous modeling forms (see Figure 3 – 2).

Non-manifold representations are the most recent development. Some volume based solid modeling systems have allowed some non-manifold conditions, but did not allow the full range of non-manifold conditions involving boundary objects such as surfaces and wireframe edges. Focus on full non-manifold systems allowing all such conditions is new. By definition such systems must have some boundary representation capability. The work in this thesis emphasizes non-manifold representations which explicitly store topological adjacencies.

The differences between manifold solid representations and non-manifold representations merit further discussion.

In a manifold (two-manifold) solid representation, every point on a surface is two-

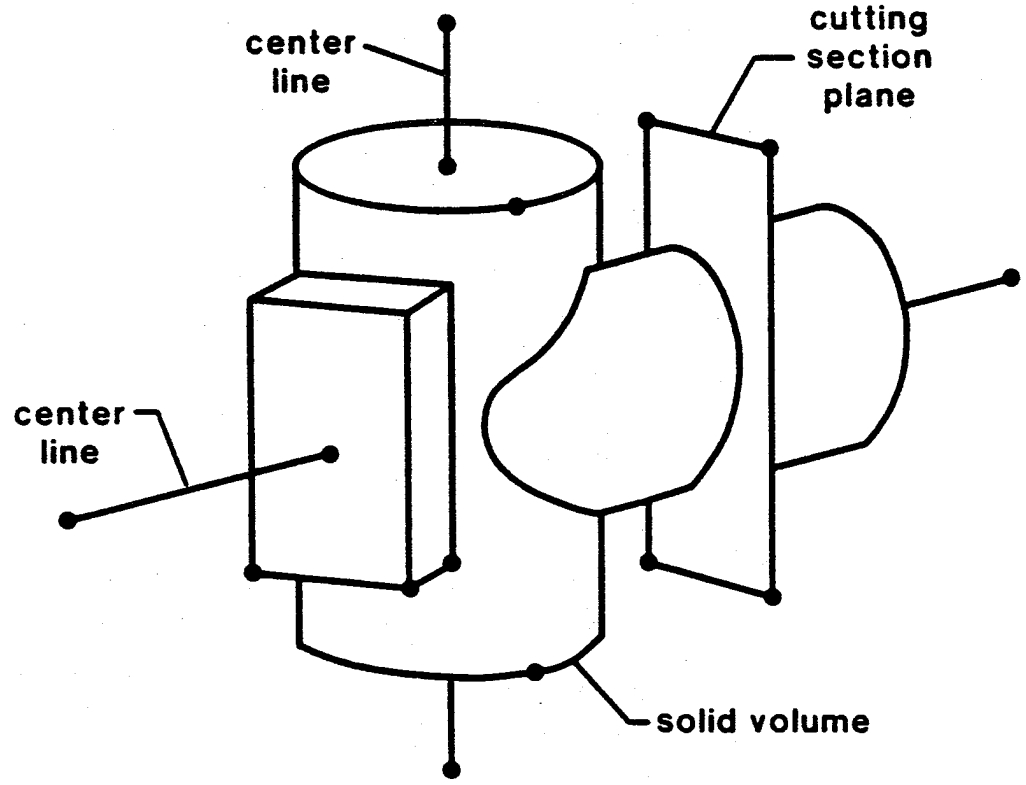


Figure 3 – 2. Example of a non-manifold geometric modeling form

dimensional; that is, every point has a neighborhood which is homeomorphic to a two-dimensional disk. In other words, even though the surface exists in three-dimensional space it is topologically "flat" when the surface is examined closely in a small enough area around any given point. Historically, boundary based solid modeling systems which store topological adjacencies have used manifold representations.

Non-manifold is a geometric modeling term referring to topological situations which are *not* two-manifold. In an environment which allows non-manifold situations the surface area around a given point on a surface might not be "flat" in the sense that the neighborhood of the point need not be a simple two-dimensional disk. This allows topological conditions such as a cone touching upon another surface at a single point, more than two faces meeting along a common edge, and wire edges emanating from a point on a surface (see Figure 3 – 3). A non-manifold representation therefore allows a general wire mesh with surfaces and enclosed volumes embedded in space.

A set of common solid modeling operations, the Boolean set operations, are not closed under manifold representations. A modification of the Boolean operations, called the *regularized set operators* [Requicha & Voelcker 77], is designed to permit only volume filling results from the Boolean operations. The regularized set operations therefore avoid a subset of the non-manifold results which can result from applying the Boolean operations on manifold inputs. However, with some manifold inputs the results of Boolean operations, regularized or not, are non-manifold and therefore not representable under manifold representations. For example, an appendage reaching out from the main volume of an object and then touching back on the surface of the same object at a single point is not directly representable with manifolds, and creation of such an object even with regularized set operations cannot yield a valid manifold result (see Figure 3 – 4). Non-manifold representations avoid these singularities by representing non-manifold situations directly instead of restricting the domain of the output.

Overall, non-manifold representations have superior flexibility, can represent a larger

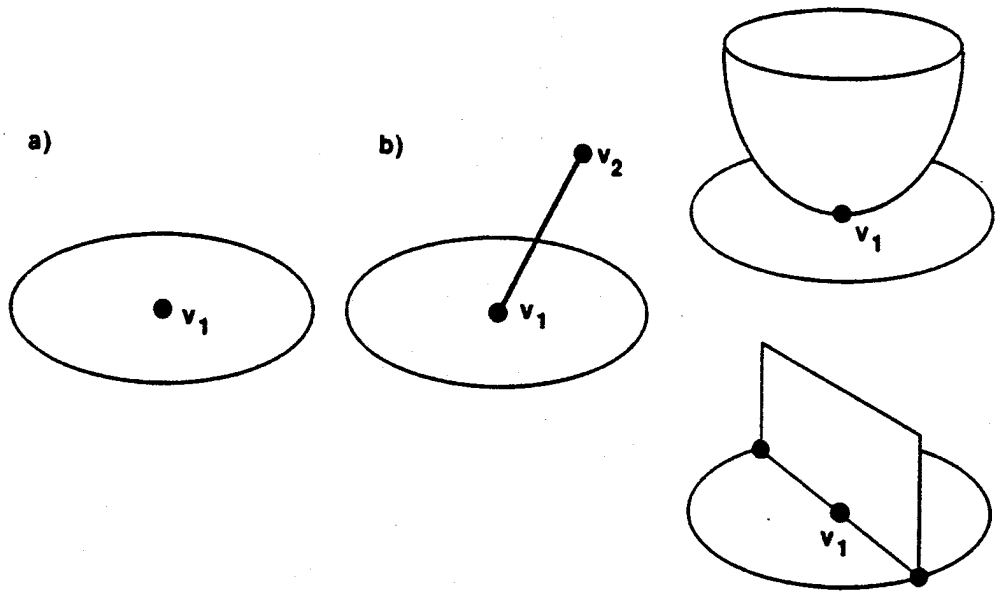


Figure 3 – 3. The 2-dimensional disk around points on a surface

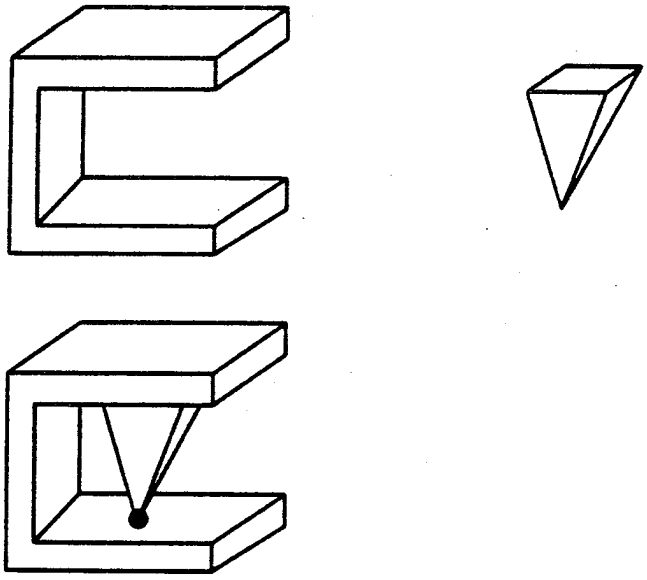


Figure 3 – 4. The Boolean union of two manifold objects yielding a non-manifold result

variety of objects, and can support a wider variety of applications than manifold representations, but at a cost of a larger size data structure. Boolean operation implementations operating on either manifold or non-manifold representations must detect and deal with non-manifold results in some fashion; however, in a non-manifold representation such results are uniformly and cleanly represented and manipulated. Thus non-manifold representations are required if accurate closed form Boolean operations with faithful representation of non-manifold results are desired. Non-manifold representations are also required if one is interested in the interior volume structures in an object and the relationships between them, such as in composite objects and finite element meshes.

Perhaps most importantly, generalized non-manifold representations can represent

wireframe, surface, and solid modeling representations simultaneously in a single uniform format. This uniformity offers significant advantages in the staging, delivery, and maintenance of geometric modeling systems.

Manifold topology representations may still be preferable, however, in situations where storage space is at a premium, and the additional advantages of non-manifold capability are not required.

3.2. A Taxonomy of Geometric Modeling Representations

More detailed analysis of the many different representations that have been developed for geometric modeling reveals a more complex picture than that shown by the basic representational form classification presented in the previous subsection.

A more detailed taxonomy of these representations is now presented.

A wide variety of representations have been developed for geometric models, each with its own strengths and weaknesses in the context of different applications. These techniques can be differentiated on the basis of at least three independent criteria concerning whether the representation is:

- boundary based or volume based
- object based or spatially based
- evaluated or unevaluated in form

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 geometric shape itself; it is spatially based when the representation is organized around the characteristics of the spatial coordinate system it uses.

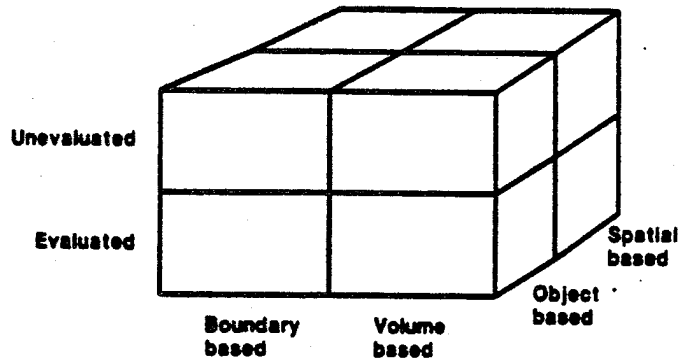
The evaluated/unevaluated characterization is roughly a measure of the amount of

work necessary to obtain information about the objects being represented with respect to a stated goal. In this thesis, for simplicity, it is assumed that the goal is obtaining enough information for wireframe or surface display of an object.

Thus many representational techniques are potentially available by choosing different combinations of values of the above criteria. The most appropriate modeling technique to use depends not only on the intended application but also on the particular phase of the application one is concerned with. Many modeling systems support multiple representational techniques to ensure their efficacy over a broad range of applications and phases of the same application.

If each of the three way criteria presented is considered to allow binary choices, then eight categories result. Several examples of the application of this classification to a variety of current geometric modeling representational schemes are presented in Figure 3 – 5. The representation names in the boxes are not the only examples that can be found for each of the classifications.

Unevaluated representation systems require some form of procedural interpretation to be used with respect to the specified application. Examples of the *unevaluated, spatial, boundary* classification include the halfspace solid representation technique where the spatial region of interest is defined by successively dividing space in halves with usually infinite surface descriptions which coincide with portions of the desired region boundary and selecting the half space on a specified side of the surface, eventually enclosing the solid region. The halfspace technique is classified here as spatial based because the surface descriptions are positioned in spatial coordinate space rather than being relative to the object. An *unevaluated, spatial, volume based* approach is the octree solid representation technique which represents solid regions of interest by hierarchically decomposing a usually cubic volume of space into successively smaller cubes. Hierarchical division and cube orientation usually follows the spatial coordinate system. An *unevaluated, object, boundary based* representation example is the procedural description of an object as a sequence of Euler operations, an edge based construction technique described later in this thesis. A popular *unevaluated, object,*



UNEVALUATED LAYER

	Boundary based	Volume based
Spatial based	Half space	Oct-tree
Object based	Euler ops	CSG

EVALUATED LAYER

	Boundary based	Volume based
Spatial based	Boundary cell enumeration	Cell enumeration
Object based	Boundary rep	Non-parametric primitives

Figure 3 – 5. Geometric modeling representation classification and examples

volume based example is CSG (Constructive Solid Geometry), where desired regions are described as a series of Boolean set operations combining primitive volumes. Sweeps, where geometric objects are swept through space, usually to produce a higher dimensional element (such as sweeping an area to obtain a volume) are another example falling into this category. Parametric primitives, standard shapes that come with size, orientation, position, and other parameters, also fall into this category.

Evaluated representation systems usually require substantially less interpretation to use with respect to the specified application. An example of a *evaluated, spatial, volume based* representation is cell enumeration, which may be as simple as a three-dimensional Boolean array, with each cell representing a cubic volume of space, with a cell having a true value if the region of interest intersects with that cell. A boundary based version of the same technique, an example of a *evaluated, spatial, boundary based* representation, is boundary cell enumeration where only the cells which intersect region boundaries have true values. An popular example of a *evaluated, object, boundary based* representation is the boundary representation, where objects are represented in terms of their boundary elements; for example, a polygon may be represented by its bounding edges, and a solid volume by its finite bounding surfaces. An *evaluated, object, volume based* representation is that of non-parametric primitives, such as a simple fixed position object; this is not a particularly flexible representation.

The application domain of particular interest to this thesis is the design, analysis, and manufacture of solid mechanical parts. Early in the design phase of such objects a high level of abstraction, a symbolic form, offers the most powerful means of performing complex design tasks — as long as the abstraction is appropriate to the design task at hand and to the designer performing it. However, during modification, analysis, and use of the constructed model, easy availability of complete information on the model is a prime consideration. For this phase of this application it has been popular to use an object based, evaluated, boundary form of geometric model which explicitly stores topological adjacency information, that is, the information specifying which topological elements such as faces, edges, and vertices touch upon one another.

The focus in this thesis will be on *evaluated, object based, boundary representations* which explicitly store topological adjacency information and can be used as a framework for the implementation of geometric modeling systems. Both manifold solid modeling and non-manifold geometric modeling representations will be addressed.

Chapter 4

TOPOLOGY AS A FRAMEWORK

This chapter provides an intuitive introduction to the use of topology as a framework in geometric modeling implementations, and provides the philosophical basis for the approach taken in the remainder of the thesis. It is intended to provide motivation for following material rather than provide a completely rigorous mathematical description of the topological aspects of geometric modeling.

4.1. Topology and Geometry

Complete geometry can be considered to represent essentially all information about the geometric shape of an object including where it lies in space and the precise geometric location of all aspects of its various elements.

Topology, by definition, is an abstraction, a coherent subset, of the information available from the geometry of a shape. More formally, it is a set of properties invariant under a specified set of geometric transformations. Invariance of these properties under transformation implies by definition that the properties represented by the topology do not include the set of information which is actually changed by such transformations. Therefore all information is not present in topology; topology is incomplete shape information which can theoretically be derived from the complete geometric specification. A carefully selected, coherent subset of information, one that supports a meaningful view of the whole, is the essence of an abstraction.

Given this idea, one can consider topological information as a fuzzy definition of an object located somewhere on the continuum between no information on the object and a complete geometric definition of the object (see Figure 4 – 1). As such, topology constrains, but does not uniquely define, the final geometry of an object. On the other hand, a complete geometric description completely defines the topology of an object, though such geometric information may not be in a form convenient for the derivation of topological information.

4.2. Different Kinds of Topology

In the context of geometric modeling, when we think of topology we most often

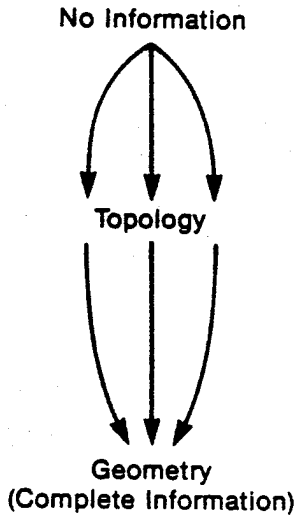


Figure 4 – 1. Topological information in the continuum of information about the geometric shape of an object

think of the adjacencies between topological elements such as vertices, edges, and faces (see Figure 4 – 2). An individual *adjacency relationship* is the adjacency, in terms of physical proximity and order, of a group of topological elements of one type (such as vertices, edges, or faces) around some other specific single topological element. An example of one topological adjacency relationship is the group of edges found in a cyclic order around each vertex on an manifold object's surface.

But an adjacency topology is only one subset of many possible subsets of geometric information — only one among many forms of topology. Knot theory topology — knots involving interlocking loops in objects which cannot be undone by geometric transformation short of intersecting the objects — is one example of a different form of topology (see Figure 4 – 3a). The amount of twist in an object of genus greater than zero is another form of topology which is totally unrelated to adjacency or knot topology (see Figure 4 – 3b). In this case all three forms of topology are orthogonal; that is, each has information which is completely independent of the other two.

We will restrict our consideration here to the adjacency form of topology since that form has so far been found the most useful in our selected application areas. Accepting this restriction, 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. This topology information can serve as a framework into which the geometric information is placed, and the topology can therefore serve as the "glue" which holds all the individual component geometry and topology information together.

4.3. Using Topology

What benefit is there to considering the topology of a geometric model apart from the complete geometric description ?

When it is a unified, coherent, high level abstraction of available information, topol-

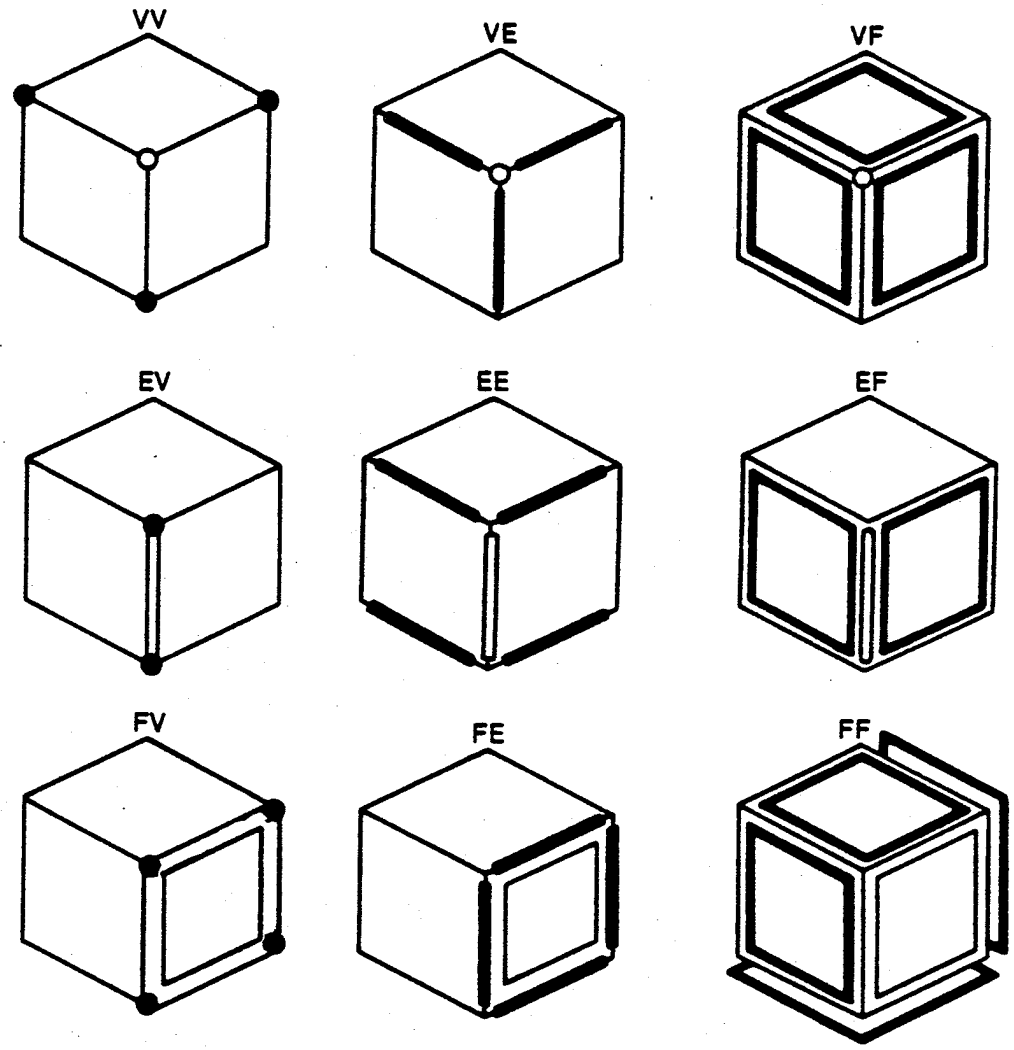


Figure 4 – 2. The nine element adjacency relationships in a manifold adjacency topology consisting of faces, edges, and vertices

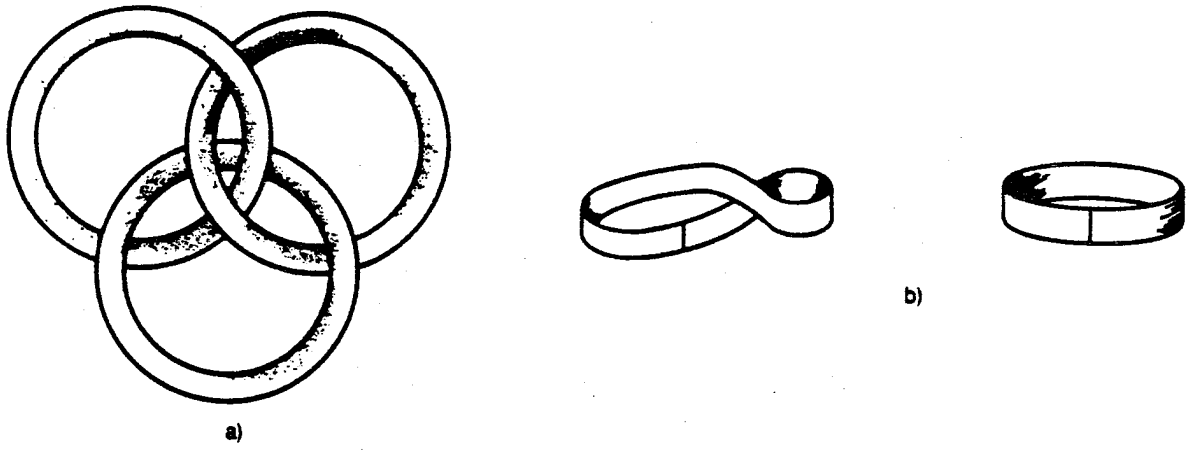


Figure 4 – 3. Different forms of topology

ogy is useful in several situations. First, it is useful whenever a concise global abstraction or summary of information can save time over being forced to view in full detail all data associated with a geometric model. Often, a top-down down hierarchical description is used for this purpose, with higher levels serving as abstractions of the lower levels. Second, during local manipulation of a small portion of an object, it is useful to be able to find directly adjacent portions of the object without having to review all data associated with the object.

Use of these two properties can simplify geometric modeling manipulation algorithms and greatly improve their efficiency. However, topology can be even more useful when it serves as a framework around which the geometric modeling representation can be built.

4.4. Topology as a Framework

By using topology as a *framework* for a geometric modeling representation we mean first that topological information is explicitly available and second that it serves as the organizing factor in the schema of the data structures used in the representation (and therefore in the algorithms which operate on the structures). Third, to provide a unified total structure, all topological information must be associated together. To date, the most commonly useful approach has been to organize the topological information in a top-down hierarchical fashion from higher to lower levels of dimensionality (see Figure 4 – 4).

The usefulness of topological information as described in the previous section is not

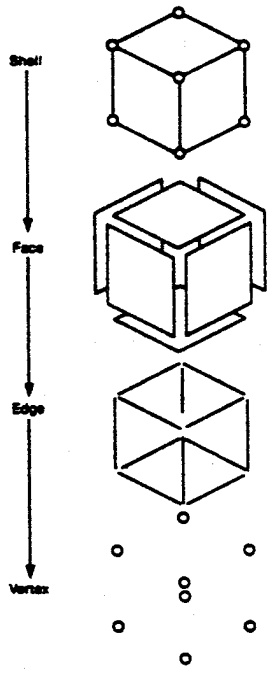


Figure 4 – 4. Top-down hierarchical representation of topological elements

the only reason topology should be considered as a framework around which a solid modeling representation can be built. There are more compelling reasons.

First, once the topological and geometric domain which the representation is intended to cover has been defined, and the corresponding topological representation has been selected, the topological portion of the *implementation* remains relatively stable. Geometric surface representation and implementation techniques are still a subject of research; the modeling field has not yet converged on any single "ultimate" or canonical geometric surface representation technique, and is still plagued by fundamental numerical accuracy problems. As a result, many different forms of geometric surface representation and implementation techniques currently exist, and more are under development. If a topological framework is used in a modeling implementation, old geometric representations can be pulled out and new ones plugged in or multiple geometric representations can be handled simultaneously without major changes to the structure of the implementation. With a stable topological framework the impact of such geometric representation changes can be minimized to small portions of the implementation and the ability to add new or replace existing geometric representations is enhanced. Thus a system implementation based on a topological framework provides for a smoother evolution of the geometric modeling system over its lifetime.

Second, because of the approximate nature of geometric representations of generalized curved surfaces as currently formulated and implemented on computers, it is possible that numerical accuracy problems can develop, such as small gaps appearing between surface patches that were intended to be adjacent. Relying on geometric information alone to determine topological relationships such as patch to patch adjacency can be an error prone proposition, particularly since arithmetic operations on the underlying scalar number representations being used are not closed form. If adjacency information is known at the time of model creation, combining a topological framework with one or more geometric surface representation techniques provides a way to represent the intended properties of an object, in spite of some types of geometric inaccuracies (though certainly not all of them).

Third, separation of topological and geometric information in a geometric modeling representation provides a more systematic approach to implementation, providing for simpler creation, verification, and analysis of the model.

4.5. Sufficient Topology

In an adjacency topology consisting of three primitive elements such as faces, edges, and vertices, there are nine possible adjacency relationships (as seen in Figure 4 – 2). If a topological representation contains enough information to recreate all nine of these adjacency relationships without error or ambiguity, it can be considered a sufficient adjacency topology representation.

A complete characterization of a sufficient representation cannot be made without first identifying the *domain*, or representational range over which the representation is intended to be valid.

Since it is not necessary in general to store all possible adjacency relationships in order to have a sufficient topological representation, identifying a sufficient minimal subset of that information becomes an issue. While only a small subset of the possible adjacency relationships can be considered sufficient and are theoretically necessary, practical topological representations useful in geometric modeling normally utilize a sufficient subset of adjacency relationships in combination with one or more other adjacency relationships. This is necessary in order to associate together all of the unique topological elements found in a particular model. For example, in a three element adjacency topology, since each adjacency relationship involves only two element types, a second adjacency relationship is necessary to associate all three element types together while maintaining the unique identity of each element (which is necessary to be able to assign unique non-topological attributes to each element, a requirement in most modeling applications). If the combination of adjacency relationships selected for a representation is sufficient, it is then not necessary to rely on geometric information to obtain any of the remaining topological adjacency relationships. Because of

possible inaccuracies in geometric data, a sufficient topological representation is therefore highly desirable.

4.6. Sufficient Topology as a Framework

When topological information is used as a framework for geometric modeling representations and their implementations, its advantages are best realized if it is independent of geometric representations. Otherwise changes cannot be made to the geometric representation portion of the system implementation without putting the entire framework at risk. In other words, a topological representation chosen as a framework for a geometric modeling system should be a sufficient topological representation. The use of a sufficient topological representation for the framework also allows a more complete consistency check against geometry, often avoiding or identifying some types of inconsistencies due to geometric inaccuracy. Furthermore, it can help avoid the inadvertent assumption of sufficient information by algorithms which manipulate the representation.

In an object based evaluated boundary form of geometric modeler it is highly desirable to utilize an adjacency topology data structure as a framework in the structure of the implementation. The abstraction implicit in this topology based organization of the data can increase the efficiency and simplicity of the modeling system. For this scheme to gain full advantage, however, the topological information used as the framework must be mathematically sufficient information, independent of the geometric information in the model. In this case the use of topology as a stable framework for the implementation structure can minimize the impact of changes in the geometric representation portions of the system implementation, can help surmount some types of geometric accuracy problems, and can simplify creation, verification, and analysis of the geometric model.

Chapter 5

TOPOLOGY AND GEOMETRY

This chapter provides a brief introduction to terminology from existing graph theory, algebraic topology, and geometric modeling that will prove useful in later sections of the thesis. It is not intended to be completely rigorous, but rather to be accessible to the average geometric modeling practitioner.

5.1. Graph Theoretic Concepts

Since the topology representations described in later chapters use graphs to represent the edges and vertices of both planar and curved surface polyhedral solids, a brief review of some ideas from graph theory will be helpful [Harary 72].

A *vertex* is a unique point. In modeling discussions we will assume it is associated with a unique three-dimensional geometric 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. A *graph* is a set of vertices and a set of distinct edges which utilize the vertices. In modeling representations, edges are often associated with closed boundaries of surface areas, and may be curved or straight non-self-intersecting finite segments of space curves.

Edges whose set includes a particular vertex are *incident* with that vertex. The *degree* of a vertex is the number of edges incident with it. Vertices which share an edge between them are considered to be *adjacent* to each other.

The *trivial graph* is a graph consisting of a single vertex.

A graph is *connected* if every two vertices are joined by some *path*, that is, there is an alternating sequence of vertices and edges which begin and end at the two vertices and where each edge in the path is incident to each vertex before and after it in the sequence.

The *connectivity* of a graph, also called the *point connectivity*, is the minimum number of vertices which, when removed along with their incident edges, results in a disconnected graph. A graph of connectivity n is said to be n-connected. This should not be confused with *line-connectivity*, which is the minimum number of edges whose removal results in a disconnected graph.

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 a multigraph edge is therefore not necessarily 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 in this thesis we are always referring to pseudographs unless noted otherwise.

A *labeled* graph is a graph where each vertex and edge is uniquely identified by some means independent of the graph.

Commonly used graph theory terminology has overlaps with terminology from the field of topology; some of this relevant terminology is therefore described in the next subsection.

5.2. Topological Concepts

Some ideas from topological theory will also be necessary to characterize the domain of the shapes that are of interest in the context of geometric modeling. An intuitive

introduction to topology may be found in [Arnold 62]. A more formal approach may be found in [Agoston 76]. The following definitions, while not completely rigorous, will be helpful in later discussions.

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

An *open disk* is that portion of a two-dimensional space which 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 open disk and is a set of points inside a sphere centered at a point and with a radius greater than zero, and excludes the sphere itself.

A subset of a topological space is *arcwise-connected* if for any two points in the subset of space there is a continuous path between them which is entirely contained within that subset of space.

A *surface*, for our purposes, is an arcwise-connected space that is topologically two-dimensional in nature. Note that although a surface is locally two-dimensional, it may geometrically exist 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 *two-manifold* surface is a topologically two-dimensional connected surface where each point on the surface has a neighborhood which is topologically equivalent to an open disk. A manifold may or may not be a closed surface. We will always be referring to two-manifolds when the word manifold is used in this thesis.

The study of three-manifolds is concerned with the shape of three-dimensional space; this area of study will not be of direct concern in this thesis and a Euclidean space will be assumed.

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. Note that while the manifold surface of a solid volume may consist of several pieces, these pieces must be joined together to form a single closed surface.

A graph can be *embedded in* (or *mapped* into) a surface if it is drawn on the surface so that no two edges intersect, except at their incident vertices. A *planar graph* is a graph which can be embedded in a planar surface. Graphs may be embedded in non-planar surfaces unrestricted in the genus of the surface. A graph may also be embedded in three-dimensional space, with or without accompanying bounded surfaces, as long as non-intersection properties are observed, that is, that no two elements intersect except at common lower dimensional boundary elements.

Faces are the connected subsets of the surface defined by a graph embedded in 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 in 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 done 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.

A *simply connected* face is a face which 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 these two holes together. A doughnut shape or torus is topologically equivalent to a sphere with one handle. The *genus* of a graph is the minimum number of handles which must be added to a sphere so that the graph can be embedded in the resulting surface without edges crossing at places other than their common vertices.

A relationship known as the *Euler-Poincaré* formula describes the relationships of numbers of elements in a planar graph:

$$V - E + F = 2$$

where V, E, and F refer to the numbers of vertices, edges, and faces in the graph, respectively.

In its more general form, where the graph may be embedded in a non-planar surface,

$$V - E + F = 2(1 - G)$$

where G refers to the genus of the graph. These formulae will be expanded further in a later chapter dealing with manifold disconnected embedded graph representations.

5.3. Geometric Modeling Concepts

The geometric modeling community has also developed its own terminology. Many of the terms describe concepts of particular interest in geometric modeling which are not addressed or addressed in less detail in other fields with different concerns.

Non-manifold, as mentioned earlier, is a geometric modeling term referring to topological situations which are not restricted to be manifold [Braid 83] [Requicha and Voelcker 83]. In a non-manifold environment the surface area around a given point on a surface might not be topologically "flat" in the sense that the neighborhood of

the point need not be a simple two-dimensional disk. This allows topological conditions such as a cone touching upon another surface at a single point, more than two faces meeting along a common edge, and wire edges emanating from a point on a surface (see Figure 3 – 3). *Non-manifold representations* are defined here as being representations that allow non-manifold topological conditions, including those involving volume, area, curve, and point elements.

The following terms have sometimes been used inconsistently in the literature; so the definitions utilized in this thesis are given here.

Edges can be classified according to their use as boundaries by adjacent faces. A *wire or wireframe edge* is an edge embedded in space which is not a boundary of any face. A *lamina edge* is an edge which is used only once on the boundary of a single face. A *manifold edge* is an edge which is used exactly twice on the boundary of one face or exactly once each on the boundaries of two faces. A *non-manifold edge* is an edge which is used three or more times on the boundaries of one or more faces.

A *strut edge* is a manifold edge which bounds one face and has one vertex which has no other incident edges. An *isthmus edge* is a manifold edge which bounds only one face but both vertices of the edge have additional incident edges.

Adjacency relationships are the information specifying which (and in what order) topological elements such as faces, edges, and vertices touch upon one another. They are are defined in detail later in the text.

The *regularized set operations* are Boolean set operations which restrict their output so that only volume filling results may occur. Thus so-called "dangling" faces and edges which could be a result of the standard Boolean set operations are not present in the output of a regularized set operator, but non-manifold output may be present [Requicha 77].

5.4. Drawing Boundary Graphs

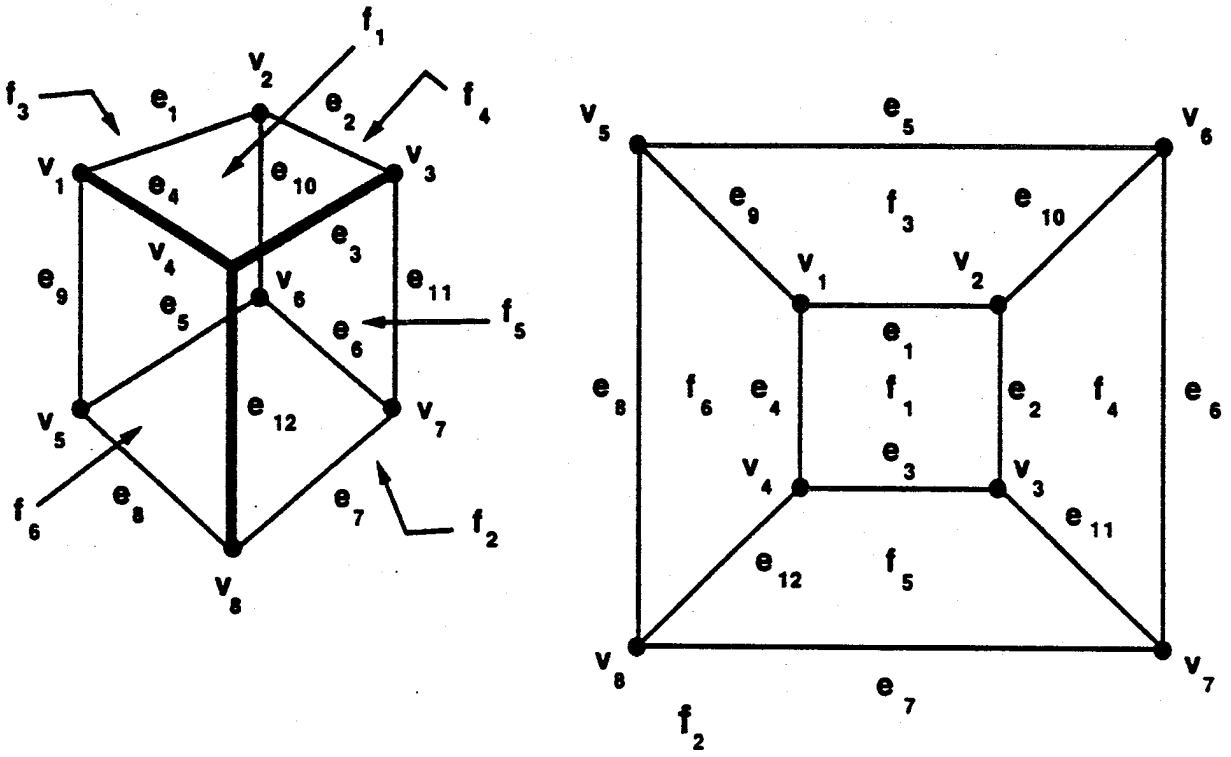
One practical issue that comes up in discussing adjacency topologies for manifold solid modeling representations is in determining how to draw the boundary graphs of solid objects on the flat sheets of paper (or CRT's) to which we've become so accustomed. It is often convenient to do so for purposes of discussion or exposition.

The graphs used in object based, boundary based, evaluated manifold solid modeling representations are graphs which have been embedded in a surface. A common form of diagramming these embedded graphs when they are planar is called a *Schlegel diagram*. A Schlegel diagram is a projection (or its combinatorial equivalent) of the vertices, edges, and faces of the embedded boundary graph of an object as seen from a point very close to the surface of the object from just outside the object. In a Schlegel diagram, as in the embedding of the graph onto its surface, edges may not cross except at their incident vertices, and vertices may not coincide. An example of a Schlegel diagram of a cube is shown in Figure 5 – 1. From here on in this subsection, when we say *graphs* we are referring to graphs which have been embedded in manifold surfaces unless explicitly stated otherwise.

Since the objects being represented in the case of manifold solids are actually closed objects, diagrams of their boundary graphs drawn on paper use the device of an "infinite" face surrounding the graph drawn on the paper. Intuitively, the diagram can be considered to be drawn on a small, nearly "flat" portion of a sphere, and this infinite face can be thought of as the "back side" of the sphere which closes the object up.

Objects of genus greater than zero, that is, objects whose boundary graphs are non-planar, need some additional mechanisms for representation on a flat piece of paper. We extend the Schlegel diagram technique here by allowing *voids* to be drawn.

Voids are labeled areas which always appear in pairs (for 2-manifold objects), with each pair having a unique label. Voids are used to connect portions of the graph together by conceptually establishing a "bridge" over the plane of the paper. Thus



a) cube *b) Schlegel diagram of cube*

Figure 5 – 1. Schlegel diagram of a cube

the edges and vertices surrounding a void appear twice in the diagram (adjacent to the two voids with the same label) and are used in opposite directions by the faces adjacent to the voids. The two faces which are adjacent to the same edge on the boundary of a void pair are actually adjacent to each other. Thus the matching pair of voids can be considered to be conceptually "glued" together.

Voids have no actual counterpart on the surface of the object being represented. They are a diagramming convenience and have no function other than to convey the

adjacency information present in the boundary graphs of objects of genus greater than zero within the confines of a planar diagram. The labeling of the voids serves to associate together each pair of voids to complete the adjacency association. The number of void pairs necessary in a diagram is equal to the genus of the boundary graph.

Figure 5 – 2 illustrates the void technique with an object of genus one. An advantage of the void technique of representing boundary graphs on a plane is that it easily allows boundary graphs with a genus greater than one in a uniform fashion.

Another technique used to draw non-planar maps on the plane when lines cross in the plane drawing but not on the actual surface, is to indicate in some notation that they do not actually meet, such as by making one of the edges dotted near the

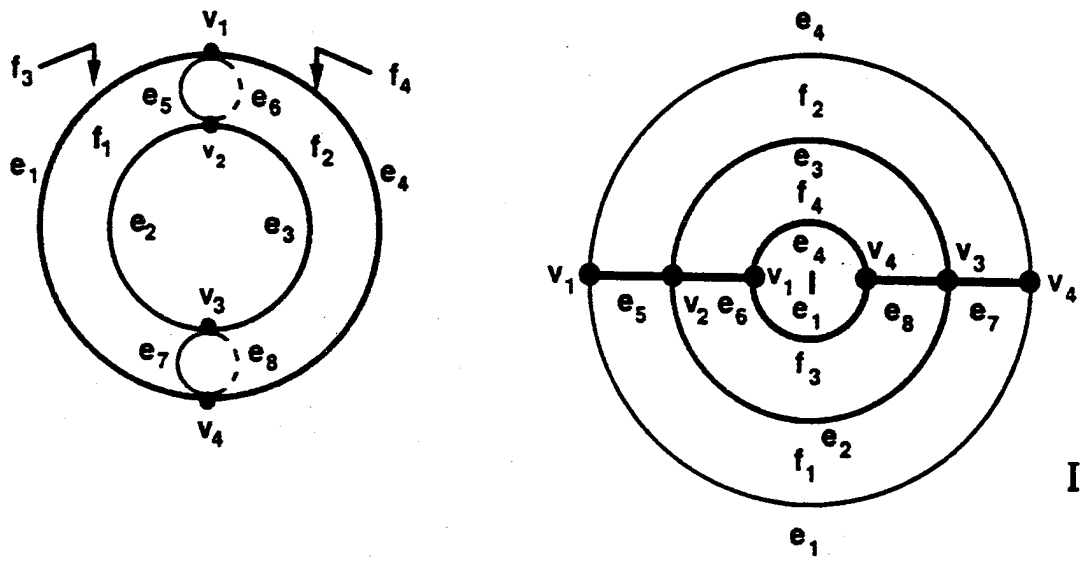


Figure 5 – 2. Boundary graph diagram of an object of genus one using voids

crossing area (see Figure 5 – 3). We'll call this technique the dotted-line technique.

Both techniques have pros and cons. The dotted-line technique has the disadvantage that it is difficult to trace face boundaries and determine the number of faces in the mapping, which is relatively easy with the void technique. On the other hand, determining the order that edges meet at a vertex is easy with the dotted-line technique but somewhat less obvious with the void technique when the vertex is located on the boundary of a void.

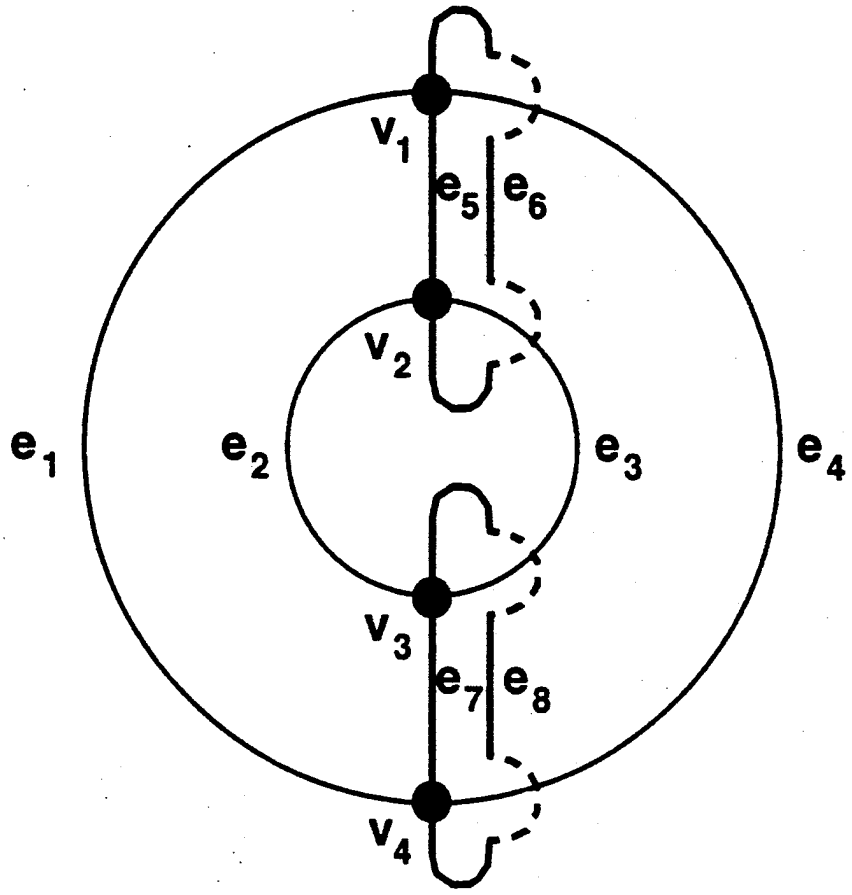


Figure 5 – 3. Crossing edge diagram of torus in Figure 5-2

Boundary graphs of non-manifold models can be drawn on the plane if suitable domain specifications are made and additional techniques are utilized. In the case of the non-manifold domain specified in this thesis in Chapter 15, individual faces (which do not include bounding edges and vertices) are restricted to be mappable to a plane. Thus their boundary graphs can be drawn on the plane in at least a piecewise fashion using additional drawing techniques.

Drawing boundary graphs of non-manifold models requires techniques similar to those used for manifold boundaries but the situations represented can be considerably more complex. For example, consider a manifold spherical surface. Push together two points on opposite sides of the sphere until they touch at the center to form a single boundary vertex. The surface is now a non-manifold surface. A drawing of its boundary graph on the plane involves the use of the infinite face technique to represent the closed surface, but the single boundary vertex shows up two places in the drawing. Similarly, non-manifold edges may show up several places in a planar drawing of a non-manifold boundary graph.

Wire edges, lamina faces, and individual regions are often drawn separately for non-manifold boundary graphs, utilizing element labels and occasional region labels to indicate actual adjacencies. This style of diagramming provides adjacency information but still does not specify the complete spatial ordering of faces around edges required for the complete description of a full non-manifold environment.

Chapter 6

TOPOLOGICAL ADJACENCY RELATIONSHIPS

Adjacency topology concerns the physical adjacencies of the topological elements embedded in space or on the surfaces of an object.

This chapter discusses topological elements and their topological adjacency relationships, and introduces a comprehensive terminology to describe characteristics of adjacency relationships relevant to geometric modeling. Topological adjacency relationships form the basis of the topological information in all of the topological representations described in this thesis.

6.1. Terminologies for Adjacency Relationships

A terminology for identifying the nine element pair adjacency relationships for connected graph manifold topologies was developed by Baer et al [Baer et al 79] for the purpose of comparing which adjacency relationships were stored in various geometric modeling systems. That terminology symbolized each adjacency relationship as a pair of symbols separated by a colon. Each of the symbols refers to one of the three element types. The first symbol is a letter which stands for the element type used as the viewpoint from which the adjacency relationship is expressed. The second symbol represents the element type which is adjacent in some way to the first element type (see Figure 6 – 1). This terminology is sufficient for the identification of the nine element adjacency relationship classes and in some cases includes additional information, but does not consistently include enough information for detailed discussion of the nature of the element adjacency relationships themselves or their interrelationships.

$v:\{v\}$	$e:\{v\ ,v\}$	$f:\{v\}$
$v:\{e\}$	$e:\{e_1, e_2, e'_1, e'_2\}$	$f:\{e\}$
$v:\{f\}$	$e:\{f\ ,f\}$	$f:\{f\}$

Figure 6 – 1. Baer et al. terminology for element adjacency relationships

An expanded terminology is needed which separately specifies the two types of information which comprise each of the nine adjacency relationships:

- identification of which of the element adjacency relationships is involved (a specification of adjacency as in the previous terminology)
- a specification of the *order* and *direction of order* of the adjacency given all the similar components in that relationship.

This last type of information is critical to boundary representation schemes but has not been previously emphasized. As will be shown later, both kinds of information are necessary to represent a complete adjacency topology using the adjacency relationships.

A more comprehensive and general terminology has been developed to explicitly include this ordering and orientation information as well as other information that will facilitate discussion of some of the properties of the adjacency relationships both as a class and in individual cases. Additions for these purposes include the ability to distinguish between a specific individual element and a group of elements (including the entire element class itself). The terminology is generalized enough to handle both manifold and non-manifold topological adjacencies.

This new terminology is used throughout the remainder of the thesis in discussions involving the topological element adjacency relationships.

6.2. Topological Element Adjacency Relationship Terminology

This terminology expresses six concepts related to the elements of graphs embedded in space or in a surface and their topological adjacency relationships. Each is described in the following sections.

6.2.1. Element Type Symbol

Three alphabetic letter *symbols* are used to specify which of the three basic topological *element types* is being referred to:

V	the vertex element type
E	the edge element type
F	the face element type

In more complex topologies, such as those allowing disconnected graphs and non-manifold conditions, additional topological element type symbols are required. They will be defined in the relevant chapters as needed.

6.2.2. Symbol Plurality

The *plurality* of an element symbol determines whether the symbol refers to one or more or all elements of the given type in a specific topology. Plurality is indicated in the following manner:

v	(lower case) - singular plurality; refers to one specific element
V	(upper case) - multiple plurality; refers to a collection of zero or more elements
$\overline{V}$	(upper case with bar) - multiple plurality; refers to a collection of *all* elements of the specified type

6.2.3. Group Ordering

A *group* is a collection of elements. Groups are allowed to have one of four *group ordering* specifications. Groups of elements are symbolized either by the proper plurality of a single symbol or by a list of element symbols. While the plurality of a symbol indicates whether the symbol refers to a single element or a group of elements, it does not identify the ordering of the group. A sequential list also does not necessarily imply an ordering. The following terminology is used to specify which ordering an element group actually has:

group	indicates that the group ordering is unspecified; it could be any of the following three orders
[*group*]	indicates an ordered linear list of elements
< *group* >	indicates an ordered cyclic list of elements
{ *group* }	indicates an unordered set of elements

As an example, $< E >$ refers to a group of edges in a cyclic ordering.

The group specification within the brackets can take either the general form indicated by the multiple plurality of a single symbol, or a specific form indicated by a series of symbols of singular plurality. Therefore if the plurality of a symbol in a group is singular, then all members of the group must be specified. As used here, all elements in a single group are usually of the same type and ordering.

Ordering refers to both sequence and direction information.

It is also useful in some cases to nest groups inside of other groups; that is, a group may consist of a list of other groups. An example of a nested group is $\{[E][E]\}$ which refers to an unordered set of two items, both of which are ordered linear lists of edges.

Parenthesis are not used as a bracketing symbol in this terminology; they therefore retain their usual mathematical meaning of associating parts of an expression whenever they are used in conjunction with the adjacency relationship terminology.

If the group specification within the group ordering brackets is a single symbol of multiple plurality, it binds to the group ordering brackets. For example, because of this binding action [*V*] means a linear ordered group of individual vertices rather a linear ordered group of nested groups of vertices. In cases where it is important to specify nesting of groups without making an ordering specification, parenthesis can be used to make the overall specification unambiguous. In the example above, a linear ordered group of nested groups of vertices (of an unspecified ordering) could be written as [(*V*)].

Two additional notational devices are used in the adjacency relationship terminology relating to groups. First, the *cardinality* of a group is specified as a superscript following the group ordering form brackets (as in $< >^{3}$), indicating that the group has the specific number of members specified by the superscript. Second, a subscript following the group ordering form brackets (as in $[]_{n}$) indicates a *reference* to the *n* th element in the group. The following section on referencing and enumeration discusses such references for unordered groups and cyclic ordered groups. Superscripts may be zero or any positive number, subscripts may be any positive number less than or equal to the number of elements in the group.

The use of the word "group" here should not be confused with other uses of the word in mathematics.

6.2.4. Adjacency Relationship

The *element adjacency relationships* indicate the topological adjacency of a group of elements with regard to a single element or element 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:

reference adjacentgroup indicates a specific adjacency relationship

Adjacency relationships deal with distinct ordered pairs of element types, so the number of adjacency relationships in an adjacency topology of n element types is n^2. In the examples initially presented three topological element types are present, creating nine distinct combinations of element types. Each distinct combination is called an *adjacency relationship class* or *type*. For example, $\bar{V}E$ refers to the adjacency relationship class involving the groups of edges which surround all of the vertices of a graph. $\bar{V}<E>$ is more specific and refers to the circular ordered lists of edges which surround vertices.

As a form of shorthand, EE can be used to signify $\bar{E}E$. This includes situations where group ordering specifications are made, so $E<V>$ can be used to signify $\bar{E}<V>$. This means that the reference element part of an adjacency relationship always refers to all elements when a multiple plurality symbol is used. Whether the general adjacency relationship concept itself or a specific adjacency in an embedded graph is being referred to is determined by the plurality of the symbols used.

An adjacency relationship carries two kinds of information: the class of the adjacency relationship and the ordering information of the adjacent group. Adjacency relationships which have unordered adjacent groups are called *unordered* adjacency relationships; relationships with linearly or circularly ordered adjacent groups are called *ordered* adjacency relationships. For example, $V<E>$ is an ordered adjacency relationship while $V\{E\}$ is an unordered adjacency relationship. The distinction is a vital one in terms of the informational sufficiency of the adjacency relationships, as will be discussed in Chapter 11.

Adjacency relationship classes can be organized into an *adjacency relationship matrix*, a standard way of presenting the descriptions of the classes. The matrix is organized into n columns and n rows, where n is the number of topological elements. The matrix starts at the upper left corner. The rows are labeled top to bottom from the lowest dimensional element (the vertex) to the highest dimensional element (in the examples given here, the face). The columns are similarly labeled left to right. For a given position in the matrix, the row labeling specifies the reference element type and

VV	VE	VF
EV	EE	EF
FV	FE	FF

Figure 6 – 2. The element adjacency relationship class matrix

the column labeling specifies the adjacent group type of the adjacency relationship class.

One can also name groups of classes based on their position within the matrix. The *main diagonal* consists of the n adjacency classes falling on the diagonal of classes from upper left to lower right. All of the classes lying above the main diagonal are the *upward hierarchical relationships*, and those below the main diagonal are the *downward hierarchical relationships*. The subset of the $(n-1)$ upward hierarchical relationship classes immediately adjacent to the main diagonal is the *upward hierarchical diagonal*, and the subset of the $(n-1)$ downward hierarchical relationships immediately adjacent to the main diagonal is the *downward hierarchical diagonal*. Figure 6 – 2 shows the element adjacency relationship matrix for the three basic topological element types.

An adjacency relationship matrix may be referred to as $\overline{A}$, with the specific matrix indicated by context. A specific adjacency relationship may then be specified in a positional notation, $\overline{A}_{row,column}$, where the adjacency relationship is located in the specified position in the adjacency relationship matrix. For example, in Figure 6 – 2, $\overline{A}_{2,3}$ refers to the *EF* adjacency relationship.

The element adjacency relationships are discussed in detail in Chapters 10 and 16 for manifold and non-manifold domains, respectively.

6.2.5. Correspondence

Correspondence is the ability to make adjacency associations between adjacency relationships which utilize the same element type in either their reference element or adjacent group. An example is the ability to make correspondences between elements in the ordered adjacent groups of two or more adjacency relationships.

The strongest form of correspondence is when two adjacency relationships have the same reference element type. Other forms of correspondence have the common element type in the adjacent group type or mixed between the adjacent group type and reference element type. Correspondences with the same reference element type are referred to as *strong correspondences* because, unlike other correspondences, their adjacent lists can be interleaved and combined in a fashion which contains more information than either of the adjacency relationships individually.

For example, using correspondence one may associate the *VE* adjacency relationship with the *VF* adjacency relationship. Then one has available not only the *edge-around-a-vertex* information and *face-around-a-vertex* information, but also all *edge-then-face-then-edge* information around a vertex. That is, the correspondence information logically *links together* the adjacency information about the various elements so that their ordering information can be coordinated.

Correspondence is symbolized as two or more adjacency relationships connected by a dash. For example, the $V<V>$ and $V<E>$ manifold adjacency relationships in correspondence are symbolized as $V<V>-V<E>$. In this case correspondence means that information about *vertex-and-edge-then-vertex-and-edge ... around-a-vertex* information is available in addition to the expected *edge-then-edge-around-a-vertex* ($V<E>$) and *vertex-then-vertex-around-a-vertex* ($V<V>$) information.

The order of appearance of the adjacency relationships in the correspondence is not significant. For example, $V<V>-V<E>$ is the same as $V<E>-V<V>$.

Strong correspondence appears to embody information not found in the individual

corresponding adjacency relationships; generating correspondence information requires information from additional adjacency relationships. For example, in the *VV-VE* correspondence above, *EV* is required to generate the correspondence.

While correspondence will be used in several places in this thesis, the topic is not treated in detail, and represents an area of possible further research.

6.2.6. Referencing and Enumeration

When dealing with a specific labeled graph, which has been mapped into a surface for manifold topologies, or embedded in space for non-manifold topologies, it is assumed there is available:

$$\bar{V} = \{\bar{V}\}$$
$$\bar{E} = \{\bar{E}\}$$
$$\bar{F} = \{\bar{F}\}$$

which are the unordered sets of all vertices, all edges, and all faces of the embedded graph. Similarly, the sets of all of any additional elements would also be available. In order to refer to specific elements of these unordered sets, an ordering shall be assumed, $[\bar{V}]$, $[\bar{E}]$, $[\bar{F}]$. Specific elements may then be referred to by the group subscripting mechanism, so that $[\bar{V}]_i$ specifies the ith element of that ordering. A shorthand form for referring to specific elements is the form v_i , e_i , and f_i , which signifies $[\bar{V}]_i$, $[\bar{E}]_i$, $[\bar{F}]_i$ respectively, and again refers to specific members of these sets, where the subscript specifies the ith element of that ordering. The ordering chosen is arbitrary, but once chosen remains constant for a given consideration of the embedded graph. Thus the embedded graph is a labeled graph.

Similarly, cyclic groups, $< N >$, are assumed to have an ordering $[N]$, so that its members may be referred to by the standard group subscripting, $< N >_i$, to indicate the ith element of the group. To arrive at such an ordering, a specific (but arbitrarily chosen) element in the cyclic list is chosen as a first element of the ordered list. Sub-

scripted list elements then refer to the ith element in this ordered list *modulo the size of the cyclic list.*

The number of elements in a set or ordered group is the cardinality designated by bracketing, as in $|\overline{V}|$, meaning the number of vertices in the entire graph, or as in $|v_i< E> |$, meaning the number of edge elements in the cyclic adjacency group associated with v_i .

An iteration over the elements of a set or list can then be specified by, for example,

$$v_j< E> _i, i \leftarrow 1..|v_j< E> |$$

which iterates over each member in the adjacent group of $v_j< E>$, or

$$v_i< e> _j, j \leftarrow 1..n$$

for short. The iteration is usually stated by itself on a line and the scope of the iteration is specified by indentation of relevant statements towards the right. The iteration may be nested, in which case the rightmost iteration varies fastest.

This terminology allows discussion of algorithms which refer to adjacency relationships of elements in a specific embedded graph.

6.2.7. Examples

The following examples further illustrate use of this terminology.

$\overline{V}$	represents the collection of all vertices in a graph.
$\overline{V}E$	represents the general adjacency relationship class of adjacent groups of edges surrounding vertex reference elements. It can also be stated as VE for short.
$V< E>$	is a more detailed description of a VE adjacency relationship class, specifying that the adjacent groups are cyclicly ordered.
v_i	represents a specific vertex, namely $[\overline{V}]_i$, the ith element in the group of all vertices in a graph.

$v_i< E>$	represents the specific *VE* adjacency relationship consisting of the circularly ordered adjacent group of edges surrounding the reference element vertex $[\bar{V}]_i$.
$v_n< e> _3$	indicates a reference to the third edge element in the cyclic list of the adjacent group edge elements surrounding the vertex reference element $[\bar{V}]_n$.
$E\{V\}^2$	represents the adjacency relationship class *EV* with unordered adjacent groups of vertices surrounding a edge reference elements, and further specifies that there are always exactly two vertices in each adjacent group.
$E\{< E> \}^2$	represents the adjacency relationship class *EE* with unordered adjacent groups, each member of which is itself a cyclicly ordered group of edges. The description further specifies that there are always exactly two $< E>$ groups in the adjacent group of each edge. This could also be written out as $E\{< E> < E> \}$.
$v_k < e_r\ e_s\ e_t >$	represents a fully detailed description of the circularly ordered list of edges around a specific vertex. Note that here, all members of the group must be enumerated, since the adjacent group consists of specific elements rather than a single multiple plurality group symbol.
VE - VF	indicates that the *VE* and *VF* adjacency relationships are maintained in correspondence with each other.
VV - VE - VF	indicates that all three of these adjacency relationships are maintained in correspondence.
$e_i\{[E]\}^2$	represents the relationships of edges adjacent to each end of a particular edge e_i. See Chapter 10 for a more detailed explanation.
$(e_i< L> _j)< V>$	is the $L< V>$ adjacency for a specific loop. The loop reference element is found by taking a reference to the jth element of the $e_i< L>$ adjacency relationship's adjacent group. The parentheses are not strictly required here but do provide clarity.
$\bar{E}_i, i \leftarrow 1..n$	is an iteration over each edge in an entire graph.
$\bar{V}_j< E> _k, j \leftarrow 1..n, k \leftarrow 1..n$	is a nested iteration over each edge in the adjacent group of the $V< E>$ adjacency relationship for every vertex in a graph. The edge is referenced within the scope of the iteration by its complete expression $\bar{V}_j< E> _k$. The index k varies fastest in this iteration.

$\overline{A}_{1,3}$

represents a specific adjacency relationship in an adjacency relationship relationship matrix using the *row, column* positional notation. The adjacency relationship matrix being referred should be clear from context. In this example, referring to the adjacency relationship matrix in Figure 6 – 2, the adjacency relationship specified is the *VF* adjacency relationship.

Chapter 7

TOPOLOGICAL DOMAIN AND SUFFICIENCY

A computer *representation* of an application consists not only of static data and data structures but also of the operators and procedures applied against them. The two are inextricably intertwined.

The *domain* of a representation is the complete set of possibilities for which the representation is valid. The domain addressed by any representation should be carefully specified; it is the only measure of success of the representation and is the starting point for any formal proof of correctness.

The *correctness* of a representation depends on:

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

Early influential work by Requicha [Requicha 77] emphasized consideration of the topological aspects of domain, but much of this work used a point set topological approach, which is less directly applicable to boundary representations than to other representation forms. The approach taken here utilizes algebraic topology, which is directly related to the adjacency topologies addressed in this thesis.

This section will address the importance of providing a specification for the intended domain of geometric modeling representations, as well as the importance of

determining their topological sufficiency over that domain.

7.1. Domain

Traditionally, the careful specification of the domain for geometric modeling representations, especially boundary based representations, was rarely done — often leaving open the question of their validity for various applications. Considering the amount of effort required to construct a significant robust geometric modeling system, implementors can ill afford to base an implementation around a representation structure which 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 the domain which a modeling system is intended to address is the first step in such an examination of the sufficiency of a representation.

The domain, in this case the topological domain, must be specified as completely as possible. The domain specification is usually made by stating an initial environment followed by a series of further restrictions on that environment. Two types of restrictions can be made.

First, *representational restrictions* places further limits on the gross topological conditions affecting the geometric shapes that are allowed to exist in the representation, directly affecting what is representable in the representation. For example, placing restrictions on the allowable genus of an object, such as stating that the genus must always be zero, reduces the number of possible shapes that are representable, in this case making doughnut (torus) shapes unrepresentable.

Second, *procedural restrictions* place additional conditions on the representation, but do not directly change what is representable in the representation, only the exact manner in which it is represented. For example, restricting individual faces from having handles does not mean that surfaces with handles are not representable, only that a face boundary must be present on a handle. Thus, the allowable partitioning of the surface is further restricted, but anything representable without the restriction is

transformable into something which is representable without changing the intended shape.

7.2. Topological Sufficiency

Topological sufficiency of a representation is regarded here as the ability to completely and unambiguously represent adjacency topologies. Completeness implies the ability to generate all of the topology information from the representation. Unambiguity implies that for any unique set of data in the representation, there is only one possible set of topology information that can result from interpretation of the representation, that is, there is a one-to-one mapping between a representation and the full topology information.

Sufficiency can be regarded at two levels, theoretical and practical sufficiency. Theoretical sufficiency is the absolute minimum information required to unambiguously reproduce a complete adjacency topology, while practical sufficiency is the minimum required in a practical geometric modeling representation.

7.2.1. Theoretical Sufficiency

Sufficiency of a representation is the ability to recreate all of the topological element adjacency relationships without error or ambiguity. In this context, it is the ability of a specified subset of the element adjacency relationships taken from a specific mapped graph to provide enough information to uniquely reproduce the original embedding of the graph except for labels of the element type(s) which are not in the original subset of adjacency relationship(s) chosen. The embedding constructed from the adjacency relationship subset must be identical to the original in all of the element adjacency relationships, reflecting the ability of the adjacency relationship subset to represent the topology of a mapped graph exactly and completely. Note that this definition does not allow the use of geometry associated with elements (if any) for derivation of any additional topological information.

In general, it is not necessary to store information on all the adjacency relationship classes in the topology to achieve sufficiency. In fact, at least in the manifold domain, there can be single adjacency relationships and combinations of single insufficient adjacency relationships which can be used to achieve topological sufficiency over a specified domain.

7.2.2. Practical Sufficiency

All elements in an embedded graph geometric modeling structure must be bound together in some fashion in order to produce a single cohesive representation of an object. Thus all elements must be related to each other by label, since in a practical modeling system additional information is potentially uniquely associated with each individual element by label.

This means that any representation which includes n topological element types for which reproducible labels are desired, must allow the derivation of at least $n-1$ adjacency relationships involving all n element types. This is the key to understanding the difference between theoretical minimal sufficient topological information and the minimal sufficient topological information practical in a geometric modeling system.

For example, in a labeled graph environment consisting of three topological element types, at least two or more adjacency relationships are necessary to bind all of the different element types together, since each individual relationship can only refer to at most two element types. Thus in a practical modeling representation for this environment sufficient combinations of two individually insufficient adjacency relationships are just as interesting for geometric modeling representations as individually sufficient relationships (as long as they involve all three element types), since two adjacency relationships are required anyway.

SECTION II

MANIFOLD SOLID REPRESENTATIONS

Chapter 8

INTRODUCTION

This major section discusses object based evaluated boundary based manifold solid modeling representations which explicitly represent information about the adjacencies of topological elements. To date, all of these representations, with only partial exceptions, have been manifold representations.

Manifold representations are currently in use in many commercial boundary based solid modeling systems, as well as in prototype industry standards, and reflect a heavy investment in manifold technology by industry. When storage space is at a premium and the flexibility and unified representational advantages of non-manifold representations will never be required in a representation, manifold representations will continue to be used in preference over non-manifold representations. Thus manifold topology systems will likely be around for some time, and are worthy of detailed theoretical analysis.

8.1. Organization of This Section

This major section is organized into the following five chapters concerning manifold topology representations.

First, the domain of interest is described in Chapter 9.

Chapter 10 describes the manifold adjacency relationships.

Next, Chapter 11 details the theoretical sufficiency of various combinations of the manifold adjacency relationship information.

Chapter 12 describes several data structures for manifold topology representations and provides proof of their sufficiency.

Lastly, Chapter 13 describes operators for manipulating manifold topologies.

Chapter 9

DOMAIN

In this section we are interested in restricting our range of topological representational capability from the domain of all topological possibilities to only that portion which corresponds to physically realizable solids with manifold surfaces. Making such restrictions will simplify our stated goal of unambiguously representing topologies of manifold solid polyhedra using boundary graph based techniques.

The domain conditions identified in this chapter will provide the context which will be assumed in the rest of this major section on manifold solid modeling representations, unless explicitly noted otherwise.

9.1. Topological Considerations

Our primary assumption in this section is that of a manifold domain in a three-dimensional Euclidean space. We are going to restrict the range of solid objects of interest to those with compact (closed) orientable 2-manifold surfaces. This eliminates the possibility of vertices, edges, and one sided faces which "hang off" the mapped boundary graph of the object. Thus non-manifold objects such as Figure 3 – 2 are excluded from consideration here (but are treated in the next major section of the thesis).

In many current modeling applications the final result does not require non-manifold objects; such objects are not physically realizable as single objects since they can be connected through infinitely thin vertices or edges. Thus a manifold representation is

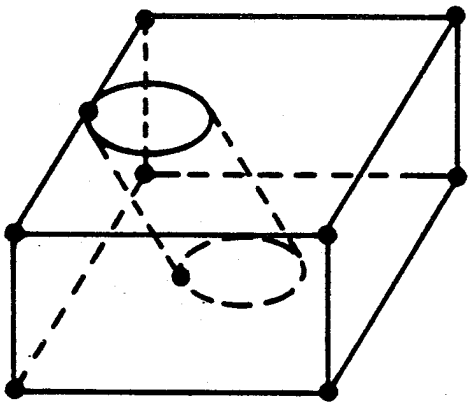
often adequate for representing the final result, if only solid shape information is desired. This does have the effect of restricting the modeling sequence of operations, however, since non-manifold objects would not be allowed even as intermediate results; these restrictions can be removed by the use of more advanced non-manifold representations. This major section of the thesis will accept this limitation and require a manifold representation at all stages of modeling.

[Requicha 77] and [Mantyla 81] discusses objects with "pseudo-manifold" surfaces. Basically such objects have non-manifold regions at certain points and curves, but, unlike the non-manifold object in Figure 3 – 2, their volumes are completely connected by regions consisting only of interior points (Figure 3 – 3). Because of this, such objects, while not manufacturable in the practical sense because infinitely thin portions of the solid cannot be machined or manufactured, do represent possible design goals in that they are still single, connected objects. One approach for representing such objects could be by simply adding edges and vertices to the manifold representation until all surfaces were manifolds and then identifying and associating together elements involved in the originally non-manifold regions explicitly. Geometry is not modified in this scheme so that the originally non-manifold regions are still geometrically coincident though no longer directly adjacent topologically without use of the additional association information. We will not include such objects within our representational range of interest here because of the additional complexity an adequate representational scheme for pseudo-manifolds would imply, while still not providing the generality or uniformity of a true non-manifold approach because implementations of these techniques require special case procedural detection and handling of non-manifold conditions.

The ability to represent boundary graphs as pseudographs which allow self loops and multigraphs is very desirable because such situations occur naturally during typical modeling operations, particularly those involving the Boolean operations (see Figure 9 – 1). While such situations can be simulated by dividing each multiple and self loop edge into several edges, this approach requires additional intelligence on the part of the modeler to detect and deal with such situations. Much of the power of

boundary graph based solid modeling systems derives from their ability to preserve and quickly deliver surface coherence information; unnecessarily increasing the number of elements necessary to represent an object decreases this performance.

a) self loops created by subtraction of a cylinder from a rectangular solid



b) multigraph created by subtraction of a sphere from a rectangular solid

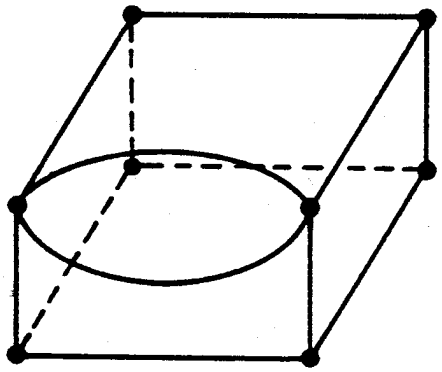


Figure 9 – 1. Self loops and multigraphs resulting from common modeling operations

9.2. Geometric Considerations

In a sense the topology of an object is a "fuzzy" geometry specification which prescribes certain limits which a geometric instantiation must maintain. Thus geometric instantiations of topological representations by definition are subject to certain geometric restrictions in order to preserve their topological integrity. It is worthwhile discussing some of the geometric implications of the topological restrictions we will be making.

Perhaps the most important geometric restriction on the geometric instantiation of a manifold polyhedron topology in this regard is that the manifold surfaces of the topology may not intersect except at the specified adjacent face boundaries. This is necessary to keep the surface homeomorphic to an open disk as required in the definition of a manifold. If the geometric instantiation of the object surface intersects itself at a point, curve, or area then the combination of the object topology and geometry representations is invalid under the requirements we have identified so far. At such intersections the surface becomes non-manifold and non-orientable with repsect to a single volume when one considers the entire surface of the object at one time.

Non-manifold objects such as those described above can be the result of common modeling operations such as the Boolean operations. Requicha discusses constraints on the Boolean set operations (the *regularized* set operations [Requicha 77]) which guarantee that all resulting surfaces are used as boundaries of space filling volumes. It is the responsibility of manifold modeling system implementations that depend on manifold characteristics for their topological integrity to ensure that all possible modeling operations result in manifold objects or at least declare non-manifold results invalid since they are unrepresentable in the manifold domain. Non-manifold modeling systems can avoid this problem entirely.

Another geometric restriction involves the valid range of the geometric definition of an individual face of a manifold solid object model which uses boundary graph based representation techniques. Every embedding of a graph into a surface must be a

two-cell embedding. That is, each face is homeomorphic to an open disk. Every face, whether singly or multiply connected, must be mappable to a plane. This means that each face is topologically "flat" and cannot contain handles. Otherwise one could arbitrarily add any number of handles to each face and information about such global features as genus would have to be contained in geometric surface definitions rather than the boundary graph structure. Allowing this would remove many of the advantages of boundary graph based representations of solid models since detailed geometric information would have to be consulted to determine some of the global characteristics such as genus of the entire surface. At any rate, flexible geometric representations of such multi-handled surface types independent of topological information appear intractable with today's geometric surface representation techniques, particularly when one considers the intersections of such surfaces. Since this is an undesirable situation, we will therefore restrict all face geometric surface definitions to form surfaces which are topologically "flat" and mappable to a plane.

We can include the surface of a sphere under this constraint if we omit at least one point. A truncated cylindrical surface is mappable to a plane and implies a disconnected graph. Both connected and disconnected graph conditions will be discussed in this section.

Following [Requicha 80a], we also restrict geometric surface descriptions to have "finiteness" properties, that is, they are well behaved in the sense of having finite surface area and not having infinitely varying surface properties.

9.3. Domain Characterization

We will now describe the domain over which we are examining these solid modeling representations in more detail. We are specifically interested in representations of the class of manifold solid objects with the following (not necessarily distinct) characteristics:

Compact Orientable 2-Manifolds - The surfaces of the objects are compact

orientable 2-manifolds in a three-dimensional Euclidean space. This implies that no faces are allowed which self-intersect, or intersect with each other, forcing the adjacency topology to explicitly carry all surface intersection information through adjacency information. Thus in a traversal of edges bordering faces, every edge is traversed exactly twice, and no non-manifold conditions are allowed. The orientability guarantees that the interior of a solid volume is distinguishable from its exterior. Note also that we are talking about single volumes completely connected by interior points.

Embedded Graph Adjacency Topology - Their topologies are represented by 2-cell 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 ability is very desirable because such situations occur naturally during typical modeling operations involving curved surfaces, particularly those involving the Boolean set operations. More restricted graphs are briefly considered in Appendix A.

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 contain no handles - This ensures 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-Poincaré formula). It is important to note that a face does not include its boundary; otherwise faces of objects like the one in Figure 12 – 4 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. 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 - In initial discussions on sufficiency and data structures, we will assume the graphs are connected graphs, and their individual faces are simply connected. There are no other connectivity restrictions other than being 1-connected. This restriction will be lifted in a later parts of the relevant discussion.

The compact orientable manifold, embedded graph, and connected graph conditions ensure the validity of the basic Euler-Poincaré equation.

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 are equally valid for representing both planar and nonplanar faced solid objects with curved or straight line edges. However, there is a much wider variety of embedded graph configurations that are 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 actually several reasons for using labeled graphs. First, it is desirable to have the ability to associate non-topological and possibly non-unique attributes with topological elements for application purposes (including associating geometric coordinate values with a vertex). Second, adjacency relationship information, even if sufficient, does not in general uniquely identify an element. Third, all element types will in general be required in a solid modeling representation since we want the relationships of all topological element types to be derivable and associated with each other by label.

Holes in faces and internal cavities in solids can be represented with disconnected graphs. Both of these situations are not directly allowed by the connected graph condition, but this restriction will be removed, and an expanded version of the Euler-Poincaré equation will be presented to support removal of this restriction in a subsection on disconnected graphs in the following chapter.

Chapter 10

TOPOLOGICAL ADJACENCY RELATIONSHIPS

The basic concepts behind the topological adjacency relationships have been described in Chapter 6; this chapter describes the specific topological adjacency relationships found in the manifold domain specified in the previous chapter.

10.1. The Manifold Topological Elements

Since topological element adjacency relationships concern the relationships between individual topological elements, we must now define the elements more carefully before describing the adjacency relationships themselves.

At least seven distinct element types, including six basic topological element types are involved in a manifold evaluated object based boundary topology representation. They can be seen as being related in a hierarchical fashion, where lower dimensional elements are used as boundaries of higher dimensional elements.

A *model* is a single three-dimensional topological modeling space, consisting of one or more distinct regions of space. A model is not strictly a topological element as such, but acts as a repository for all topological elements contained in a geometric model, allowing the manipulation of multiple geometric models by a modeling system.

A *region* is a volume of space. There is always at least one in a model. Only one region in a model may have infinite extent; all others have a finite extent, and when more than one region exists in a model, all regions have a boundary. For example, a single solid would require two regions in the model, one for the inside of the object, and one for the outside (which has an infinite extent). For manifold solid modeling

it is usually assumed that there is only one volume of interest (where there would be only two regions in a model) so in this situation it is not necessary to directly represent regions in an adjacency relationship topology. The only times that more than two regions show up in a manifold solid topology is when a solid model has several interior voids or when voids have additional shells within them. Even in these cases regions are usually not represented directly, since there is a one-to-one correspondence between shells and regions in a manifold model. Regions will therefore not be considered further in this section.

A *shell* is an oriented boundary surface of a region. Shells are applicable to disconnected graph topologies. A single region may have more than one shell, as in the case of a solid object with a void contained within it. A region may have no shell only where all space exists as a single region, as in the initial state where no modeling has been done, or after all components of a model have been deleted. A shell must consist of a connected set of faces which form a closed volume.

A *face* is a bounded portion of a shell. It is oriented. Note that an orientable element implies only that it is possible to assign an orientation, while an oriented element actually specifies a particular orientation. Strictly speaking, a face consists of the piece of surface it covers, but does not include its boundaries.

A *loop* is a connected boundary of a single face. Loops are applicable to disconnected graph topologies. A face may have one or more loops; for example a simple polygonal face would require one loop, and a face with a hole in it would require two loops. Loops normally consist of an alternating sequence of edges and vertices in a complete circuit, but may consist of only a single vertex. Loops are also oriented.

An *edge* is a portion of a loop boundary between two vertices. Topologically, an edge is a bounding curve segment which may serve as part of a loop boundary for the one or two faces which meet at that edge. Every edge is bounded by a vertex at each end (possibly the same one). An edge is orientable, though not oriented; it is the *use* of an edge which is oriented.

A *vertex* is simply a unique point in space, that is, no two vertices may exist at the same geometric location (although the topology alone does not specify an exact geometric location beyond these topological constraints). Single vertices may also serve as boundaries of faces.

Thus, discounting models and regions, there are three topological elements of interest for connected graph manifold adjacency topologies and five topological elements of interest for disconnected graph manifold adjacency topologies.

Although not directly represented in adjacency relationships as described here, at least two additional structure types of topological element adjacency *uses* associated with the edge, and vertex elements may also be defined. Their purpose is to represent the use of a specific basic topological element in the adjacent group of an adjacency relationship; in some representations they are represented directly.

An *edge-use* is an oriented bounding curve segment on a loop of a face and represents the use of an edge by that loop. There are always two uses of a single edge in a manifold model.

A *vertex-use* is a structure representing the adjacency use of a vertex by an edge, or a loop.

10.2. The Manifold Connected Graph Topological Adjacency Relationships

The nine manifold element adjacency relationships of topological elements in manifold embedded graphs, as expressed in the new adjacency relationship terminology, are shown in Figure 10 – 1. A diagram of the ordered element adjacency relationships, along with the unordered relationships (which lack ordering information) is shown in Figure 10 – 2. The unordered relationships have been included in these figures for later discussions of orderedness of topological models under certain constraints. Figure 10 – 2 includes ordered adjacency relationships with the edge as a reference element, although the ordering is not intrinsic to the relationships and can

only be induced by correspondence (see Section 10.2.1). An expanded example of the actual values of adjacency relationships in a particular embedded graph is shown in Figure 10 – 3.

Variations on how each relationship is represented and defined are possible. These variations involve the semantics of the adjacency relationships and not necessarily storage representation formats. The adjacency relationship definitions shown in the figures include a few cases which reflect choice as to the exact meaning of the relationship.

The *EE* adjacency relationship can be defined at least two different ways. In both cases the adjacent group of the reference edge is an unordered list of length two. The length of two is due to an edge having two ends and the list is unordered since there is no means of identifying one end of an edge from the other solely in terms of its edge adjacencies. The difference in the two definitions given is in how the members of the adjacent group themselves are defined.

class	*ordered*	*unordered*	
VV	$V<V>$	$V\{V\}$	
VE	$V<E>$	$V\{E\}$	
VF	$V<F>$	$V\{F\}$	
EV		$E\{V\}^2$	*see text*
EE	$E\{[E]\}^2$	$E\{\{E\}\}^2$	*EE def. A, see text*
EF		$E\{F\}^2$	*see text*
FV	$F<V>$	$F\{V\}$	
FE	$F<E>$	$F\{E\}$	
FF	$F<F>$	$F\{F\}$	*FF def. A, see text*

Figure 10 – 1. The ordered and unordered adjacency relationships for manifold topologies

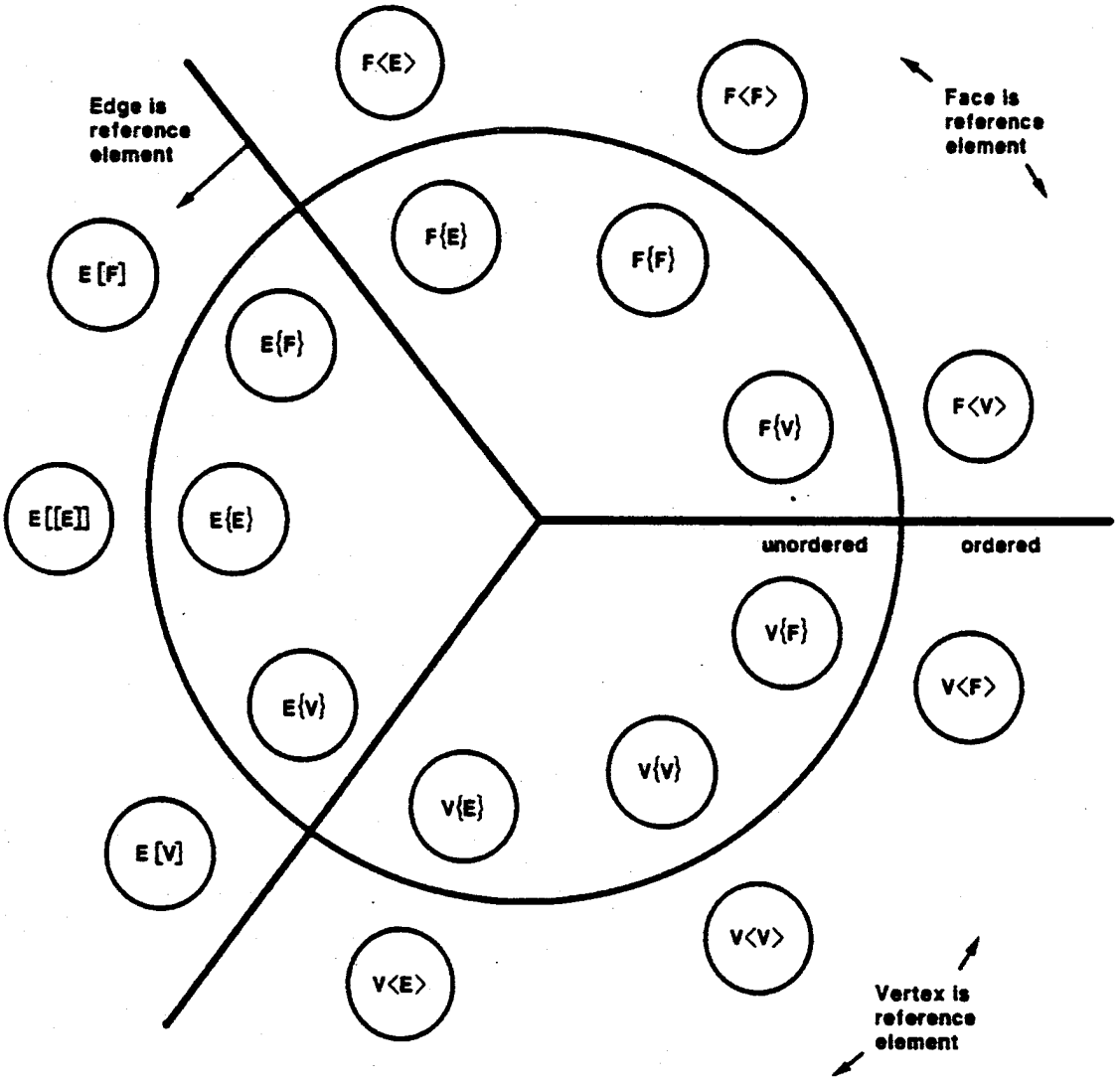
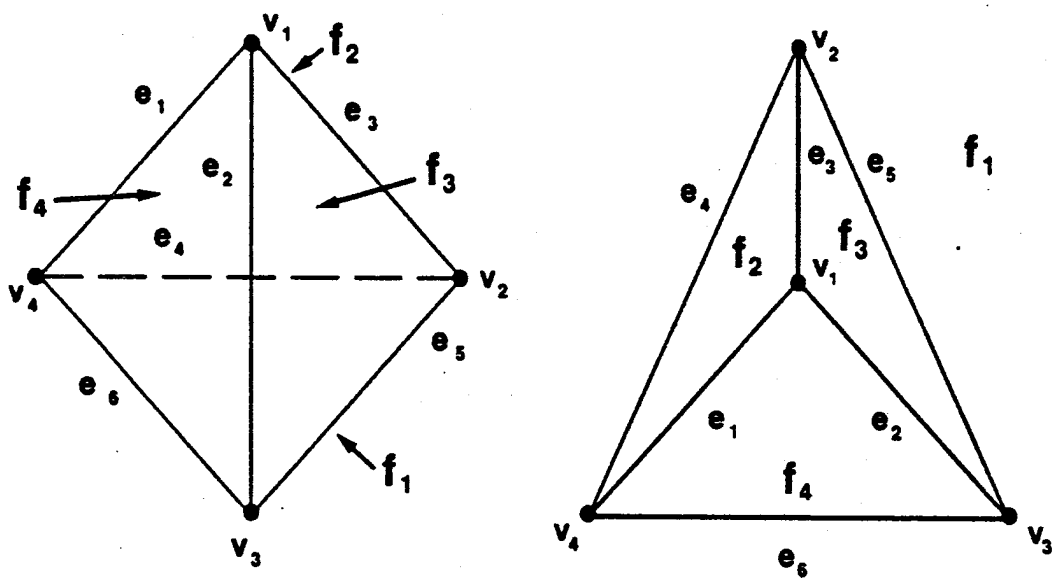


Figure 10 – 2. Diagram of the ordered and unordered element adjacency relationships

a) a solid tetrahedron and its labeled embedded graph structure



b) adjacency relationships

V< V>
v_1< V> = v_1< v_2 v_3 v_4>
v_2< V> = v_2< v_3 v_1 v_4>
v_3< V> = < v_1 v_2 v_4>
v_4< V> = < v_2 v_1 v_3>

V< E>
v_1< E> = v_1< e_1 e_3 e_2>
v_2< E> = v_2< e_5 e_3 e_4>
v_3< E> = v_3< e_6 e_2 e_5>
v_4< E> = v_4< e_4 e_1 e_6>

V< F>
v_1< F> = v_1< f_2 f_3 f_4>
v_2< F> = v_2< f_1 f_3 f_2>
v_3< F> = v_3< f_1 f_4 f_3>
v_4< F> = v_4< f_1 f_2 f_4>

E{V}
e_1{V}= {v_1 v_4}
e_2{V}= {v_1 v_3}
e_3{V}= {v_1 v_2}
e_4{V}= {v_2 v_4}
e_5{V}= {v_2 v_3}
e_6{V}= {v_3 v_4}

E{[E]}^{2}
$e_1\{[E]\}^2$ = e_1 {[e_3 e_2][e_6 e_4]}
$e_2\{[E]\}^2$ = e_2 {[e_1 e_3][e_5 e_6]}
$e_3\{[E]\}^2$ = e_3 {[e_2 e_1][e_4 e_5]}
$e_4\{[E]\}^2$ = e_4 {[e_5 e_3][e_1 e_6]}
$e_5\{[E]\}^2$ = e_5 {[e_3 e_4][e_6 e_2]}
$e_6\{[E]\}^2$ = e_6 {[e_2 e_5][e_4 e_1]}

E{F}
e_1{F}= e_1 {f_2 f_4}
e_2{F}= e_2 {f_3 f_4}
e_3{F}= e_3 {f_2 f_3}
e_4{F}= e_4 {f_1 f_2}
e_5{F}= e_5 {f_1 f_3}
e_6{F}= e_6 {f_1 f_4}

F< V>
f_1< V> = f_1< v_2 v_4 v_3>
f_2< V> = f_2< v_4 v_2 v_1>
f_3< V> = f_3< v_1 v_2 v_3>
f_4< V> = f_4< v_1 v_3 v_4>

F< E>
f_1< E> = f_1< e_4 e_6 e_5>
f_2< E> = f_2< e_4 e_3 e_1>
f_3< E> = f_3< e_3 e_5 e_2>
f_4< E> = f_4< e_2 e_6 e_1>

F< F>
f_1< F> = f_1< f_3 f_2 f_4>
f_2< F> = f_2< f_1 f_3 f_4>
f_3< F> = f_3< f_2 f_1 f_4>
f_4< F> = f_4< f_2 f_3 f_1>

Figure 10 – 3. Actual adjacency relationships for a tetrahedron

In the first *EE* definition, $E\{[E]\}^2$, (or $E[[E]]^2$ in correspondence) referred to here as *EE* definition A, each of the two members of the adjacent group is itself a group of two linearly ordered edges, symbolized by $[E]$. The two edges, in order, refer to the left and right nearest neighbor of the reference edge clockwise and counterclockwise from the reference edge respectively about that end of the reference edge. Such rotational directions are as seen from outside the solid volume looking directly towards the surface. This definition of *EE* has the advantage of requiring a short constant length implementation data structure.

In the second *EE* definition, $E\{< E >\}^2$, referred to here as *EE* definition B, each member of the adjacent group is a cyclicly ordered group of edges, symbolized as $< E >$. Each member $< E >$ of the adjacent group refers to the cyclicly ordered list of all of the edges surrounding one end of the edge. To effectively use the relationships, the reference edge would usually need to be found in the $< E >$ group in order to determine the relationship of the reference edge to other edges, and further, an indication of which occurrence of the edge in the adjacent group was relevant to a particular end of the edge would need to be maintained for self loop edges.

An example illustrating the differences in the two definitions are shown in Figure 10 – 4.

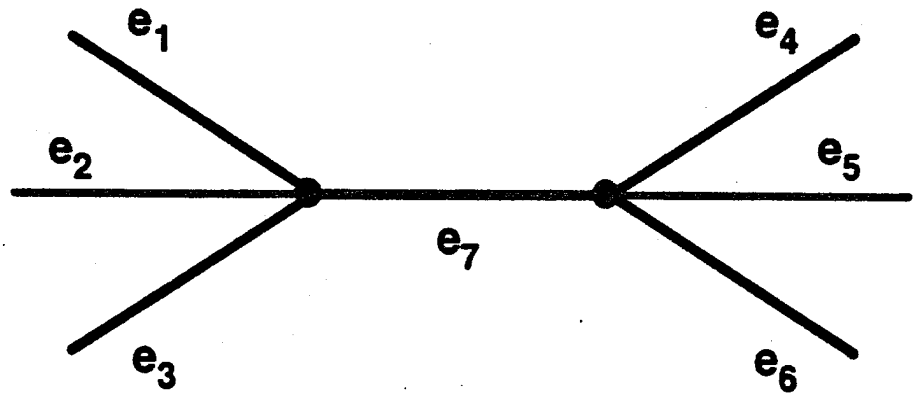
FF will be defined in terms of the adjacency of faces to vertex and/or edge elements in the boundary of the reference face. Even then, the *FF* relationship can still be defined at least two semantically different as well as syntactically different ways. The difference is in how the adjacent goup is defined (see also Figure 10 – 5):

Definition A - only faces adjacent to edges surrounding the reference face are in the adjacent group

Definition B - faces adjacent to edges *and* vertices of the reference face are in the adjacent group; no differentiation of the two is made in the adjacent group

Unless specified otherwise, we will be referring to definition *A* when *FF* is mentioned. The preferred definition is largely a matter of taste; definition *A* is chosen here

a) topology



b) $E\{[E]\}^2$ *form (definition A) generates:*

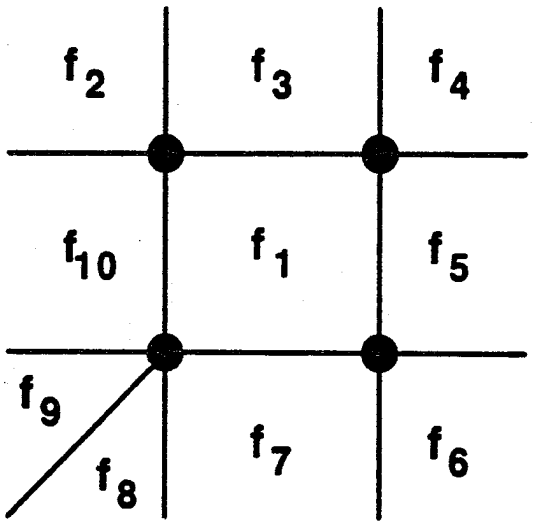
$$e_7\{[e_3\ e_1]\ [e_4\ e_6]\}$$

c) $E\{<E>\}^2$ *form (definition B) generates:*

$$e_7\{<e_3\ e_2\ e_1\ e_7>\ <e_7\ e_4\ e_5\ e_6>\}$$

Figure 10 – 4. EE adjacency relationship formats

a) topology



b) definition A

$$f_1\langle F\rangle = f_1 \langle f_{10} f_3 f_5 f_7 \rangle$$

c) definition B

$$f_1\langle F\rangle = f_1 \langle f_{10} f_2 f_3 f_4 f_5 f_6 f_7 f_8 f_9 \rangle$$

Figure 10 – 5. Two definitions of the F< F> adjacency relationship

because of its simplicity and because of the convenience of having the same number of members in its adjacent group as $F\langle V\rangle$ and $F\langle E\rangle$.

In the $F\langle V\rangle$ relationship, strut or isthmus edges and self loops in faces are represented as shown in Figure 10 – 6.

The $V\langle F\rangle$ relationship is defined such that the adjacent group enumerates all faces encountered between all the edges surrounding a vertex. The number of elements in the adjacent group of $F\langle V\rangle$ is therefore the same as in the $V\langle E\rangle$ relationship.

For example, in the case of v_1 in Figure 10 – 6, the adjacent group is $< f_2 f_1 f_1 >$. This is the maximum amount of information we can ascribe to $V < F >$.

10.2.1. Edge Adjacency Relationships

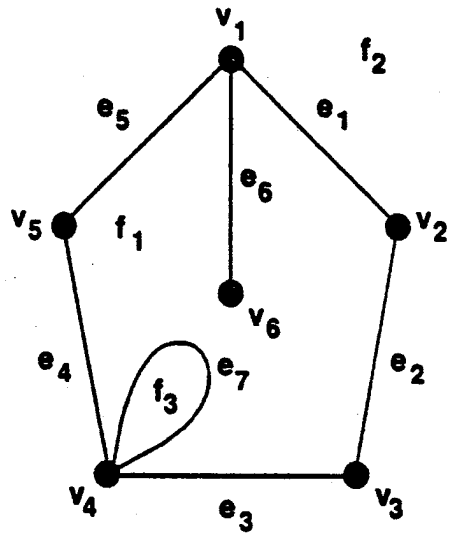
The element adjacency relationships where the edge is the reference element have several characteristics which are different from the other adjacency relationships and are worth mentioning at this point.

The relationships where the edge is the reference element are the only relationships in which the adjacent groups can be of fixed length, are essentially unordered, and can not be truly cyclic.

EV and *EF* are defined as $E\{V\}$ and $E\{F\}$ and are exceptional in that they are the only adjacency relationships which always have exactly two members in their adjacent group ($|e\{V\}| = 2$ and $|e\{F\}| = 2$). Without combining information together, there is no basis for differentiating one end or one side of the edge from the other in any individual adjacency relationship involving the edge as the reference element. Thus *EV* and *EF* are by themselves unorderable without referring to other elements for positioning. One might argue that *EV* and *EF* have cyclic ordered adjacent groups of length two, but this is semantically equivalent to an unordered list in terms of ordering information, and it is unclear if a claim can be made for any cyclic nature of the two ends of an edge. Therefore, since *EV* and *EF* can't reflect any true ordering they are represented as unordered element adjacency relationships, and there are no ordered versions of *EV* and *EF*.

Similarly, the adjacent group of the *EE* relationship is also unordered. In this case, however, it is listed as an ordered adjacency relationship because some relative ordering information is retained in each of the individual members of the adjacent group in both definitions discussed.

Any two or all three of the adjacency relationships with the edge as the reference ele-



a) F< V > relationship

$$f_1\langle V\rangle = f_1 \langle v_5 v_1 v_6 v_1 v_2 v_3 v_4 v_4\rangle$$

b) V < F > relationship

$$v_1\langle F\rangle = v_1 \langle f_2 f_1 f_1\rangle$$

c) V < E > relationship

$$v_1\langle E\rangle = v_1\langle e_1 e_6 e_5\rangle$$

d) E {[E]}² relationship (EE definition A)

$$e_6\{[E]\}^2 = e_6\{[e_6\, e_6][e_5 e_1]\}$$
$$e_7\{[E]\}^2 = e_7\{[e_7 e_4][e_3 e_7]\}$$

e) E {< E > }² relationship (EE definition B)

$$e_6\{\langle E\rangle\}^2 = e_6\{\langle e_1 e_6 e_5\rangle \langle e_6\rangle\}$$
$$e_7\{\langle E\rangle\}^2 = e_7\{\langle e_4 e_7 e_7 e_3\rangle \langle e_4 e_7 e_7 e_3\rangle\}$$

Figure 10 – 6. Adjacency relationship example involving *strut* edges and self loops

ment may be put into correspondence. In this case the correspondence information may be represented by having the adjacent group assume an ordering for coordination only, as in $E[V]$ and $E[F]$ for $E\{V\}$ and $E\{F\}$ in correspondence, and using the ordering to coordinate between the two relationships. Although the ordering is arbitrary for the first relationship chosen, it provides a basis for ordering the remaining elements, allowing the correspondence to be made. Thus while $E\{V\}^2$, $E\{[E]\}^2$, and $E\{F\}$ all consist of unordered adjacent groups, the imposition of an ordering can be used to represent the correspondence between all of them, creating $E[V]^2$-$E[[E]]^2$-$E[F]^2$ This ordering would be used for correspondence and does not represent information inherently present in the specific adjacency relationships in correspondence.

An example of the representation of strut and self loop edges in terms of both definition A and definition B of the *EE* relationship are shown in Figure 10 – 6d and 10 – 6e. Ends of strut edges, since they are not adjacent to any other edges, are represented in definition A, $E\{[E]\}^2$, as a set including the reference edge twice for the corresponding member of the adjacent group. In definition B, however, a strut edge does have a single member in its adjacent group member $< E >$, which is the reference edge itself.

An advantage of the *EE* definition A is that with the correspondence $E[V]^2$-$E[[E]]^2$-$E[F]^2$, efficient clockwise and counterclockwise traversals around the edges and vertices surrounding a face can be made. This allows traversal of the entire graph without resorting to local searches through cyclic lists of elements of arbitrary length, as would be necessary with *EE* definition B unless the cyclic adjacent groups were somehow marked to indicate the location of the reference element edge in the cyclic list.

10.2.2. Correspondence

The twenty-seven pairs and the six triplets of possible correspondences of the nine possible adjacency relationships are listed in Figure 10 – 7. Nine of the possible

thirty-six pairs of adjacency relationships (thirty-six since the number of unique unordered pairs in a group of n objects is $\frac{n(n-1)}{2}$) do not allow correspondence because they involve the adjacency relationships where one of the pair of adjacency relationships has the same reference and adjacent group element type and the other of the pair of adjacency relationships consists of the two element types not found in the first relationship.

---

same reference element type

EV - EF	VE - VF	FV - FE
EV - EE	VE - VV	FV - FF
EE - EF	VV - VF	FE - FF
EV - EE - EF	VV - VE - VF	FV - FE - FF

same adjacent group element type

VV - EV	VE - EE	VF - EF
VV - FV	VE - FE	VF - FF
EV - FV	EE - FE	EF - FF
VV - EV - FV	VE - EE - FE	VF - EF - FF

mixed same reference and adjacent group element type

VE - EV	VF - FV	EF - FE
VE - EF	VF - FE	EF - FV
VE - FV	VF - EV	EV - FE

Figure 10 – 7. Correspondences between the nine adjacency relationships

---

10.3. Adjacency Relationships for Disconnected Graphs

Although early boundary modelers (such as [Baumgart 72]) had simpler data structures which represented topology information using the element adjacency relations much as they have already been described, several later boundary based solid modelers (including [Eastman & Weiler 79] and [Braid et al 78]) have expanded the number of basic elements to remove both the surface and volume connectivity restrictions encountered with the original representations. The basic idea is that the new element types "bridge" the gap between common boundaries of the same face or volume. Since the same conditions can be represented in a connected graph representation, the changes are more a practical matter of convenience and a clean representation rather than an extension of theory.

The *loop* structure modification was originally created to eliminate the otherwise unnecessary artifact edges used to associate "inner" hole contours with the "outer" face boundaries (see Figure 10 – 8). The addition of the loop structure generalizes the representations to allow disconnected graphs within single surfaces of a solid volume.

The *shell* structure extension was made to allow multiple shelled objects (solid volumes with internal cavities) without resorting to artifact faces created solely to provide a connected graph representation of the desired separate surfaces (see Figure 10 – 9). Similarly, this addition generalizes the representation schemes to allow single volumes to contain multiple surfaces.

Both of these additions together modify the Euler-Poincaré equation:

$$V - E + F = 2 - 2G$$

to the following form:

$$V - E + F - (L - F) = 2 (S - G)$$

where L is the number of loops and S is the number of shells or surfaces in the

object being represented.

10.3.1. Loops

Note that the quantity (L - F) in the Euler-Poincaré equation above is the number of contours of multiply connected faces "in addition" to the first contour in faces of the object being represented. The effect of subtracting the "additional" contours on the left side of the equation is identical to the effect of including an additional artifact edge, since the edges appear with a negative sign on the left side of the equation. Thus the overall effects of the two different multiple contour face representation techniques are identical in terms of their effect on the Euler characteristic of the topology.

While the artifact edge technique is convenient from a theoretical point of view for its simplicity, it has several problems from a practical standpoint. In a geometric modeling situation, where models are constantly modified during the design of an object, the artifact edges may be split several times, increasing the computational costs of manipulating the model. The system must also be able to decide which vertices of

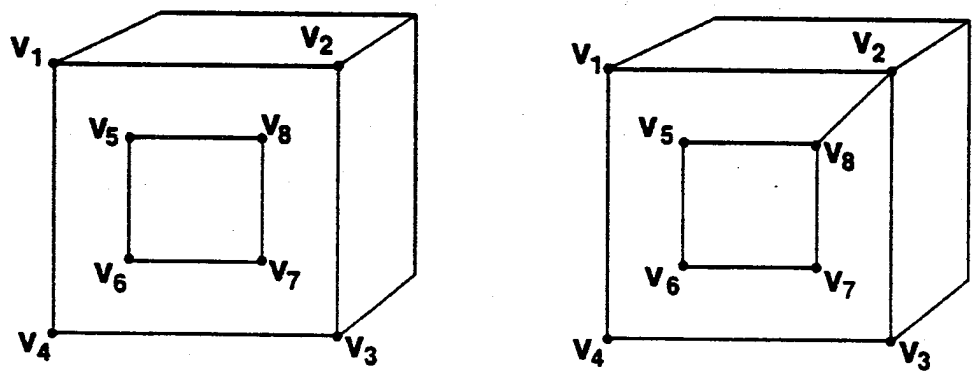


Figure 10 – 8. Artifact edges to associate separate boundaries of a face

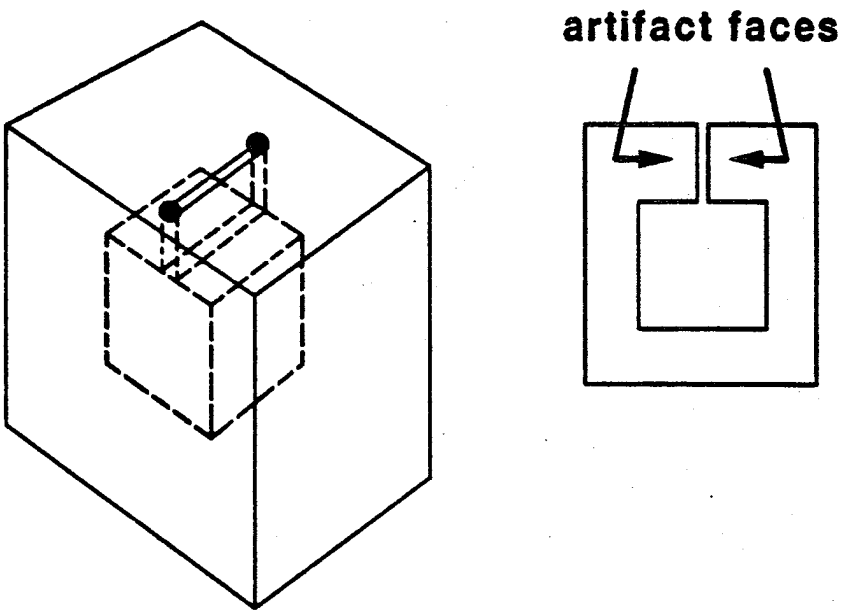


Figure 10 – 9. Artifact faces to associate separate boundaries of a volume

the contours to use when locating the artifact edge in the structure during its creation. In systems which actually display artifact edges, the appearance of these edges to produce a hole in a face is conceptually disturbing to users whose modeling requests (such as remove the volume of a cylindrical shape from a block) did not imply "extra" lines on faces with holes.

10.3.2. Shells

Object representations with multiple shells could be represented by a list of several separate surface topologies. Unlike artifact edges used to represent holes in faces, independent shells cannot be represented by "artifact faces" without significant computational effort and additional tags to differentiate "real" faces from artifact faces.

This is necessary because two matching coincident "faces" are required to tie together an outer shell to an inner shell. Normally, geometrically coincident faces which are topologically separate would not otherwise occur in such representations.

Some additional topological information not derivable from the other adjacency relationships can also be stored at the shell level. This information is the characterization of which shell is the outer shell of the finite object and which are the inner shells entirely contained by the outer shells. The usefulness of representing this information in the topological model instead of deriving it from geometric information is again dependent upon its frequency of use in a given application. Such information can reduce computational cost dramatically in situations such as the determination of whether a point is interior or exterior to a solid since it allows a hierarchical spatial search to be performed.

10.3.3. Disconnected Graph Adjacency Relationships

If the loop and shell elements are considered as additional topological element types, then several new adjacency relationships emerge, as well as changes in the semantics of the old adjacency relationships. There can be many variations on the way these relationships are specified; one way is shown in the adjacency relationship matrix in Figure 10 – 10.

Since we are only allowing manifold surfaces on objects, the adjacency relationships $V\{S\}$, $E\{S\}$, $L\{S\}$, $F\{S\}$, may only have one member in their adjacent group. $L\{F\}$ is part of the definition of a face and therefore has only one member in its adjacent group. Since edges have only two sides on a manifold, similar to the initial set of adjacency relationships, the adjacent groups of $E\{V\}$, $E\{L\}$, $E\{F\}$, and each of the member groups of the adjacent groups of $E\{[E]\}^2$ have exactly two members.

$V<L>$ is defined as the cyclically ordered list of loops which use a vertex (see Figure 10 – 11). LL could be defined as the list of loops adjacent to the reference loop by

$V<V>$	$V<E>$	$V<L>$	$V<F>$	$V\{S\}^1$
$E\{V\}^2$	$E\{[E]\}^2$	$E\{L\}^2$	$E\{F\}^2$	$E\{S\}^1$
$L<V>$	$L<E>$	$L<L>$	$L\{F\}^1$	$L\{S\}^1$
$F\{<V>\}$	$F\{<E>\}$	$F\{L\}$	$F\{<F>\}$	$F\{S\}^1$
$S\{V\}$	$S\{E\}$	$S\{L\}$	$S\{F\}$	$S\{S\}^0$

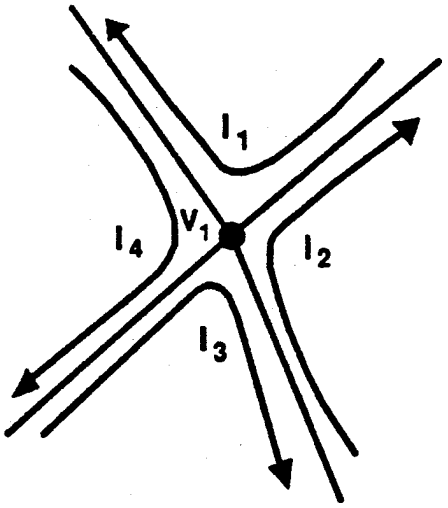
Figure 10 – 10. The manifold disconnected graph adjacency relationship matrix

sharing an edge (*LL* definition A), or as the list of other loops used in the face to which the loop belongs (*LL* definition B), as shown in Figure 10 – 12. We will use *LL* definition A here.

$F\{L\}$ is the list of loops belonging to a face.

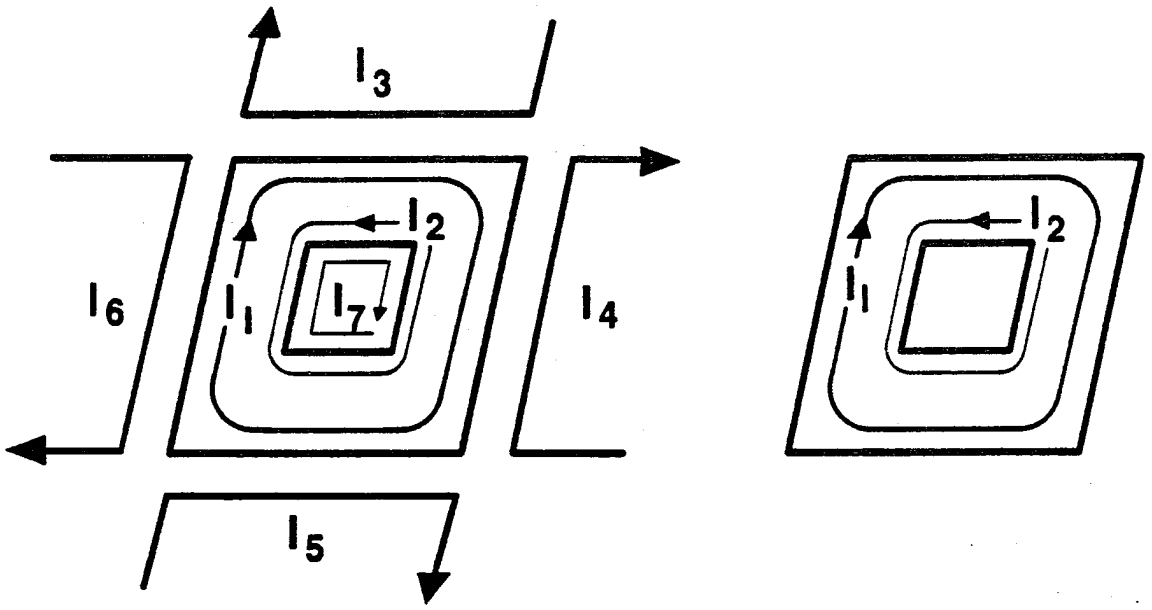
FV, *FE*, and *FF* adjacent groups may have multiple members, each member of which is a group, one for each loop in the face. For a given face f_i, there will therefore be exactly $|f_i\{L\}|$ members in the unordered adjacent groups of $F\{<V>\}^{|f_i\{L\}|}$, $F\{<E>\}^{|f_i\{L\}|}$, and $F\{<F>\}^{|f_i\{L\}|}$, with each adjacent group consisting of $|f_i\{L\}|$ members consisting of $<V>$, $<E>$, and $<F>$ groups, respectively. These $<V>$ and $<E>$ adjacent groups are equivalent to $L<V>$ and $L<E>$ for each loop of the face.

$S\{S\}$ has no members in its adjacent group since shells may not touch in manifold environments. $S\{F\}$ simply provides a lists of faces in a shell; the remaining adjacency relationships using a shell as the reference element can be derived from $S\{F\}$ and similar adjacency relationships using the face as the reference element.



$$v_1\langle L\rangle = \langle l_1\, l_2\, l_3\, l_4\rangle$$

Figure 10 – 11. V< L> adjacency relationship example



a) LL Definition A:

$l_1\{L\} = \{l_3 l_4 l_5 l_6\}$

$l_2\{L\} = \{l_7\}$

b) LL Definition B:

$l_1 < L > = < l_2 >$

$l_2 < L > = < l_1 >$

Figure 10 – 12. LL adjacency relationship example

A model normally keeps a simple list of shells. More complex structures may be desirable in some situations to differentiate the outermost shell or completely capture the shells of multiple objects or an ability to differentiate containment relationships between shells (such as with hierarchical tree structured lists).

The new *FL* and *LF* adjacency relationships involving the loop element type embody the connective information allowing faces to have multiple disconnected contours. Such connective information was only available through adjacency relationships involving edges in the previous system of adjacency relationships, which is why the artifact edge technique was developed to simulate disconnected contours.

Since the addition of the shell element type occurs at a hierarchically higher level above the existing elements, its effects on the other element adjacency relationships are minimal.

Chapter 11

TOPOLOGICAL SUFFICIENCY

Proving the topological sufficiency of a geometric modeling representation is an important part of the process of verifying the correctness of a representation over a specified domain. The most concise way to prove topological sufficiency of a representation is to start from information about the theoretical minimum information necessary to attain sufficiency.

This chapter develops the theoretical minimum topological adjacency information necessary for manifold boundary geometric modeling representations. This is done by examining the topological element adjacency relationships for topological sufficiency. Sufficiency of specific data structures is discussed in Chapter 12, which describes the data structures.

The topological element adjacency relationships are first considered for sufficiency individually, and are then considered for sufficiency in combination. The findings are then summarized.

Some readers may wish to skip directly to the summary subsection at the end of this chapter on a first reading.

11.1. Sufficiency of the Manifold Element Adjacency Relationships

To examine the topological sufficiency of a representation or of its specific implementation data structures we first need to find what *information* is sufficient, in other words which set of adjacency relationships are sufficient.

The overall objective of this chapter is to characterize the theoretical sufficiency of

various subsets of the manifold ordered element adjacency relationships, and in particular each individual adjacency relationship, to represent manifold curved surface domain polyhedral topologies.

It can be proven with simple counterexamples that none of the individual unordered element adjacency relationships are sufficient to specify a complete manifold polyhedron topology under the conditions identified in Chapter 9. Although insufficiency of two of the unordered adjacency relationships will be proven here, this section will concentrate on examining the sufficiency of the ordered element adjacency relationships and their ability to unambiguously produce a complete polyhedron topology representation under the conditions identified in Chapter 9. This includes the topic of whether some combinations of individually insufficient adjacency relationships are together sufficient.

First, sufficiency will be defined, then sufficiency of three of the nine individual element adjacency relationships will be proven, and then insufficiency of the remaining six will be proven. The sufficiency of some pairs of individually insufficient element adjacency relationships will also be considered. Finally, a summary will characterize the findings.

11.1.1. The Individually Sufficient Adjacency Relationships

Three element adjacency relationships, $V<E>$, the specific EE adjacency relationship $E\{<E>\}^2$, and $F<E>$, are individually sufficient to represent polyhedral topologies.

All three sufficient element adjacency relationships have the edge element type as the type of their adjacent group.

11.1.1.1. V < E> Sufficiency

A theorem due to Edmonds [Edmonds 60] determined that the directed cyclic orders of the edges around the vertices in an embedded graph are sufficient information to

completely and uniquely describe polyhedron topologies (see [White 73] and [Graver & Watkins 77]). The cyclicly ordered edge-around-a-vertex information is equivalent to the definition of the $V<E>$ element adjacency relationship given here. Therefore the $V<E>$ adjacency relationship by itself is sufficient for representing polyhedral topologies unambiguously.

A major result of the theorem is an embedding enumeration algorithm, called the Edmonds embedding technique, which can produce all of the 2-cell embeddings in an orientable surface of a given graph (the connectivity information $V\{V\}$ or $E\{V\}$). The algorithm operates by turning the lists of edges incident to each vertex into a cyclic list, which creates a specific instantiation of $V<E>$ information. By permuting the orders of the edges in the cyclic lists, all possible $V<E>$ adjacency relationships can be created. The theorem states that each possible ordering corresponds to a specific embedding of the graph in an oriented surface. Thus, by permuting the $V<E>$ adjacent group information created in this way, each of the possible embeddings can be produced.

Generating the actual embedding from a specific instantiation of the $V<E>$ information (see [White 73]) involves constructing the boundaries of the faces of an embedding from the $V<E>$ information, and then "sewing" the face boundaries together by matching up their edges much like assembling a picture puzzle. Every edge of an embedded graph is used exactly twice in the $V<E>$ adjacent groups of edges, and each such use of an edge is associated with one of the two directed edges between the two vertices of the edge. Since this map construction or embedding technique results in a mapped graph where every edge is used twice and in opposite directions, the map is closed and oriented.

To illustrate how the boundaries of the faces of an embedded graph can be determined, the following algorithm which is part of the Edmonds embedding technique (see [White 73] and [Young 63]) is presented in adjacency relationship terminology.

To traverse the boundary of a face in a clockwise direction given the clockwise cyclicly ordered $V<E>$ information:

1. Select a vertex v_i. This will be called the *original vertex.*
2. Select an edge which is a member of $v_i<E>$, say $v_i<E>_a$. This will be called the *original edge.*
3. Find some $v_k<E>_b$ such that $v_k<E>_b = v_i<E>_a$ and if $i = k$ then $a \neq b$.
4. Find $v_k<E>_c$, the successor edge to $v_k<E>_b$ in the traversal of the face boundary, from the $v_k<E>$ information using b. $v_k<E>_c$ is simply the edge preceding $v_k<E>_b$ in the cyclic sequence $v_k<E>$, that is $v_k<E>_{b-1}$.
5. Until v_k = the *original vertex* and $v_k<E>_c$ = the *original edge,* go to step 3, using v_k as the new v_i and $v_k<E>_c$ as the new $v_i<E>_a$.

The traversal of the boundary of a face in the embedded graph from the $V<E>$ information alone is now complete. To construct all face boundaries from the $V<E>$ information the above process is repeated until all edges have been used twice during the traversals of the face boundaries.

Note that the $F<V>$-$F<E>$ adjacency relationships in correspondence can also be created during the face boundary traversal. These relationships are used to "sew" together the face boundaries into a complete embedding by an identification process which matches up each use of an edge so that each of its two adjacent faces uses the edge in opposite directions in their boundary. This is done by making sure the vertices of the two uses of the edge "match up" when the two faces are made adjacent along their common boundary. In the case of an isthmus or strut edge the two sides or uses of the edge are sewn together on the same face boundary.

Using this embedding technique, $V<E>$ information taken from a specific embedding can be used to generate an embedding which will be identical to the original up to the label of the faces. If desired, after the embedding process is complete, the faces of the embedded graph may be uniquely labeled, and other adjacency relationships derived.

11.1.1.2. E {< E> } Sufficiency

It is apparent that $E\{<E>\}^2$, *EE* definition B contains nearly identical information to $V<E>$ since each member of the adjacent group contains information identical to the entire adjacent group of the $V<E>$ relationship of one of the vertices to which the edge is incident. The difference is that the reference vertex of each $<E>$ adjacent group is unknown, and there are multiple copies of each $<E>$ group. In fact, for every vertex of degree n there are n copies of an adjacent group equivalent to the vertex's $v_i<E>$ adjacent group. $E\{<E>\}^2$ cannot be used directly for traversal of the edges bounding a face because there is no foolproof way of determining which edge of the $<E>$ groups to use when multiple self loops occur at a vertex of the reference edge. In order to determine face boundaries and embeddings from the $E\{<E>\}^2$ information, $V<E>$ information must first be created.

$V<E>$ information can be created from $E\{<E>\}^2$ information by a simple algorithm which eliminates the duplicate copies of the $<E>$ groups and then labels the vertices. Establishing the equivalence between $E\{<E>\}^2$ and $V<E>$ in this way will prove the sufficiency of $E\{<E>\}^2$.

The algorithm is:

1. If there are no $E\{<E>\}^2$ adjacency relationships, then our embedded graph is the trivial graph and we are finished.
2. Otherwise, for each adjacent group member $<E>$ of every $e_i\{<E>\}^2$ create an equivalent $<E>$ and place it in set A, a set of all $<E>$'s found in all $E\{<E>\}^2$'s.
3. Until set A is empty:
    a) Remove some $<E>$, a member of set A, from set A.
    b) Find and eliminate the other $(n-1)$ members of set A which exactly match in membership and cyclic order the $<E>$ originally removed from set A in step 3a, where $n = |<E>|$.
    c) Place the $<E>$ originally removed from set A in step 3a into set B.

4. Until set B is empty:

   a) Remove $< E >$, any member of set B.

   b) Create a unique label of a vertex, i .

   c) Join the label with the $< E >$ to create a $v_i <$ E> adjacency relationship.

Given the ability to generate $V < E >$ information uniquely from $E\{< E >\}^2$ information, we can claim sufficiency for $E\{< E >\}^2$ (EE definition B).

Theorem 11-1: The $E\{< E >\}^2$ adjacency relationship is sufficient to unambiguously represent adjacency topologies of polyhedra.

proof: Given the above algorithm, one can convert $E\{< E >\}^2$ into $V < E >$. The algorithm is correct because by the *EE* definition B of $E\{< E >\}^2$ there must be n copies of the $< E >$ cyclic ordered groups of edges surrounding each vertex of degree n, one for every edge incident to a vertex. Given the one-to-one correspondence between $V < E >$ and $E\{< E >\}^2$ using this algorithm, $E\{< E >\}^2$ is then sufficient by the Edmonds theorem.

11.1.1.3. F < E> Sufficiency

The ordered cyclic list of edges surrounding a face preserves the orientation and embedding of the face. Because each edge can only be used twice, and because the orientation information is preserved, an embedding technique can be constructed to create the complete embedding from the $F < E >$ information.

The embedding process is similar to the Edmonds embedding technique and is basically an identification process which matches up each of the two directed uses of a given edge as well as each of the uses of a vertex. The identification process will form a closed and oriented surface. Unlike the Edmonds technique, information on vertex identity is not directly available and vertex identification must be made solely through the use of edge information.

The embedding procedure involves the examination of all adjacent groups of all $f_i< E>$:

1. *edge identification procedure*

   If any $f_a< E>_b = f_c< E>_d$, where a may or may not equal c (but if $a= c$, as in the case of a strut edge, then $b\neq d$), then the boundaries of f_a and f_c are adjacent along this edge. The two uses of the edge are the only uses of the edge and are of opposite orientation in the boundary cycles of f_a and f_c.

2. *vertex identification procedure*

   Every edge has two ends or vertices. A traversal of the boundary cycle of a face first encounters one vertex of the edge, called the starting vertex of the edge, then encounters the edge itself, and then encounters the second vertex of the edge, called the ending vertex of the edge with respect to the face boundary cycle. There are two rules for vertex identification:

   A) The starting vertex of an edge $f_n< E>_i$ is the same vertex as the ending vertex of the edge $f_n< E>_{i-1}$ directly previous to $f_n< E>_i$ in the face boundary cycle. The ending vertex of an edge $f_n< E>_i$ is the same vertex as the starting vertex of the edge $f_n< E>_{i+1}$ directly following $f_n< E>_i$ in the face boundary cycle.

   B) For any matching uses of an edge $f_a< E>_b = f_c< E>_d$, the starting vertex of edge $f_a< E>_b$ is the same vertex as the ending vertex of edge $f_c< E>_d$ and the ending vertex of edge $f_a< E>_b$ is the same vertex as the starting vertex of edge $f_c< E>_d$.

As a direct result of rule *A* in the vertex identification procedure, if $| f_a< E> | = 1$ or $| f_c< E> | = 1$ then the starting and ending vertices of the edge are in fact the same vertex. Intuitively this makes any previously discovered common uses of the starting and ending vertices in the partially embedded graph coalesce so that they converge upon the same vertex. If $| f_a< E> | \neq 1$ and $| f_c< E> | \neq 1$ then the two vertices

of any given edge in the face boundary cycles may or may not be distinct. In other words, a self loop must be encountered before it will be recognized that two potential vertices are in fact the same vertex, since the vertices have not been labeled.

As a result of vertex identification rules *A* and *B* combined, if any two-edge sequence $f_m< E>_i$, $f_m< E>_{i+1}$ in the face boundary cycle of face f_m matches in opposite order a sequence $f_n< E>_j \ldots f_n< E>_{j+1+n}$, a sequence in the face boundary cycle of face f_n in which there may be $n = 0$ or more edges between $f_n< E>_j$ and $f_n< E>_{j+1+n}$, then the ending vertex of $f_m< E>_i$ is the starting vertex of $f_m< E>_{i+1}$ is the ending vertex of $f_n< E>_j$ is the starting vertex of $f_n< E>_{j+1}$ is the ending vertex of $f_n< E>_{j+n}$ is the starting vertex of $f_n< E>_{j+1+n}$ (see Figure 11 – 1).

$F< E>$ information is similar to the information which had to be constructed in the

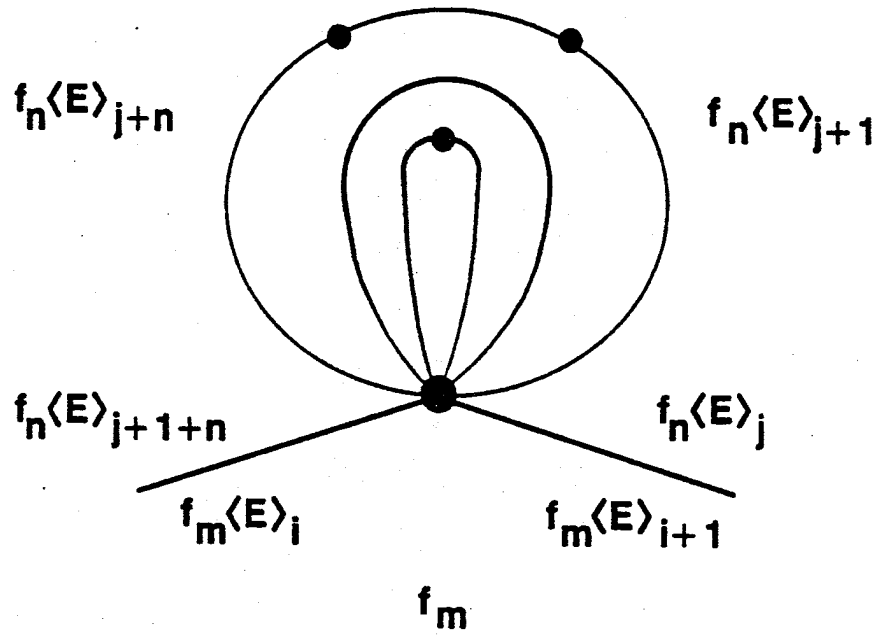


Figure 11 – 1. Result of application of vertex identification rules A and B

first part of the Edmonds embedding technique, except that the additional $F< V>$ information is not directly available and must be constructed. Direct information on the vertices of the edges of the face boundaries is not necessary to match up edges during embedding because uses of the edges already reflect orientation. Therefore each of the two uses of a given edge in the $F< E>$ information necessarily uses the edge in a direction opposite to the other use. Vertex identification is more involved than for $V< E>$ during embedding since vertices are not directly represented in $F< E>$ and the information must be derived from edge adjacencies by the rules given above.

If desired, after the embedding process (the identification process) is complete, the vertices of the embedded graph may be uniquely labeled, and other adjacency relationships derived.

Theorem 11-2: The $F< E>$ adjacency relationship is sufficient to unambiguously represent the adjacency topologies of curved surface polyhedra.

proof: Using the identification process above, since every instance of an edge on a face boundary is matched with another instance of the same edge on a face boundary, and every vertex use is connected, the resulting embedding is closed. Since the two instances of each edge are of opposite orientation for the two adjoining faces, the embedding is oriented. The order of the rules applied in the identification or "sewing" process does not affect the outcome since all affects are local. At every step in the sewing process for any given edge instance remaining to be sewn there is only one possible other edge instance in a boundary to which it can be matched. By the vertex identification rules there are a finite number of steps to determine common vertex identity. The process is therefore deterministic, and the embedding produced unique.

A point of minor interest is the representation of the trivial graph in the three sufficient adjacency relationships. In $V< E>$ it is represented as a single adjacency relationship with an empty adjacent group. In $E\{< E> \}^2$ and $F< E>$ there is no direct

way of representing vertices unattached to edges, so they must represent the trivial graph as simply the absence of any adjacency relationships.

11.1.2. The Insufficient Individual Adjacency Relationships

Six of the nine element adjacency relationships are individually insufficient for unambiguously representing the topologies of polyhedra. These six relationships are $E\{V\}$, $E\{F\}$, $V<V>$, $F<V>$, $V<F>$, and $F<F>$. Additionally, *EE* definition A, $E\{[E]\}^2$, is also insufficient.

Intuitively, the proofs utilize counterexamples to the unambiguous reconstruction of a mapping of the graph from the adjacency relationship information under consideration. In these counterexamples, it is shown that for a given adjacency relationship of the type under consideration there exists more than one mapping. This is not acceptable for the unambiguous representation of the topology of polyhedra and proves the insufficiency of the particular adjacency relationship under consideration for representing topologies of polyhedra.

Proofs of the insufficiencies are most easily given in the form of counterexamples. All of the insufficiency proofs in this paper have the same basic format, so the format is described once and referred to from the insufficiency theorems in the following sections. The general format of the proofs is:

General Format for Insufficiency Theorems 11-3 through 11-5:

Theorem: The *X* adjacency relationship information is not sufficient to unambiguously represent the manifold adjacency topologies of curved surface polyhedra.

proof (by contradiction): If the *X* adjacency relationship is sufficient to unambiguously represent the manifold adjacency topologies of curved surface polyhedra, then one could reconstruct the unique mapping of the embedded graph of the object shown in the Figure *X part a* (along with all of the adjacency relationships up to the labels of the other element type(s) not involved in the *X* adjacency relationship) from its *X* adjacency relationship information

alone (Figure *X part b*). Note, however, that another mapping consistent with the *X* adjacency relationship information can be found which is not consistent with other adjacency relationships in the original, meaning the two labeled mappings are not homeomorphic (Figure *Xpart c*). The *X* adjacency relationship is ambiguous and does not contain enough information to uniquely represent the topology (mapped graph) shown in the figure. Therefore the *X* adjacency relationship is insufficient to unambiguously represent manifold curved surface polyhedra topologies.

Proofs of insufficiency for the remaining six element adjacency relationships now follow. Where appropriate, comments are made regarding causes of the insufficiency and restrictions which would allow the particular element adjacency relationship to be sufficient.

For completeness, proofs of the insufficiency of $E\{V\}$ and E{F} are given even though they are not ordered adjacency relationships.

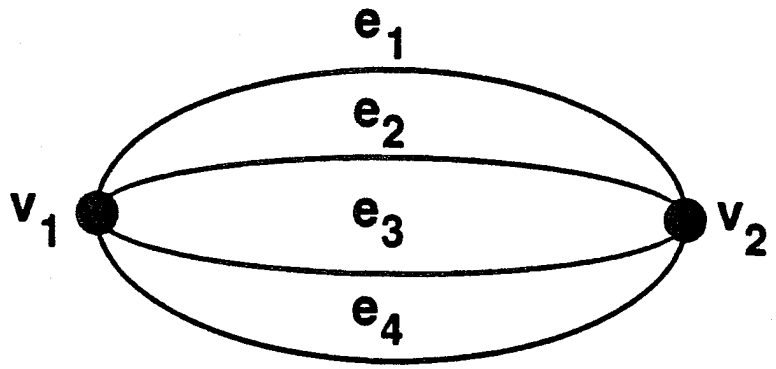
Theorem 11-3: Each of the $E\{V\}$, $E\{F\}$, $V<V>$, $F<V>$, $V<F>$, and $F<F>$ adjacency relationships are individually not sufficient to unambiguously represent the adjacency topologies of curved surface polyhedra.

proof (by contradiction): Using the insufficiency proof form, and the Figures 11 – 2, 11 – 3, 11 – 4, 11 – 5, 11 – 6, and 11 – 7, respectively, we can see that each is insufficient by counterexample.

Given that the $E\{V\}$ and $E\{F\}$ adjacency relationships are not truly ordered element adjacency relationships, it is not surprising they are not sufficient.

While each of the two element adjacency relationships $V<V>$ and $F<V>$ are insufficient for the general case, they are each sufficient if the range of representation is restricted to disallow multigraphs and self loops. Intuitively, it is possible to see this is true because it is only with multigraphs and self loops that edges are not uniquely identified by the set of their two endpoints. If the restriction is made and pseudographs are not allowed, then it is fairly straightforward to develop a function

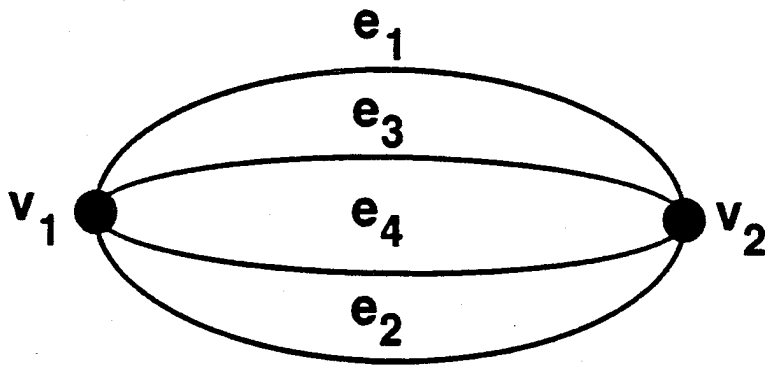
a) Mapping 1



b) E{V} information

$$e_1\{V\} = e_1\{v_1\ v_2\} \quad e_3\{V\} = e_3\{v_1\ v_2\}$$
$$e_2\{V\} = e_2\{v_1\ v_2\} \quad e_4\{V\} = e_4\{v_1\ v_2\}$$

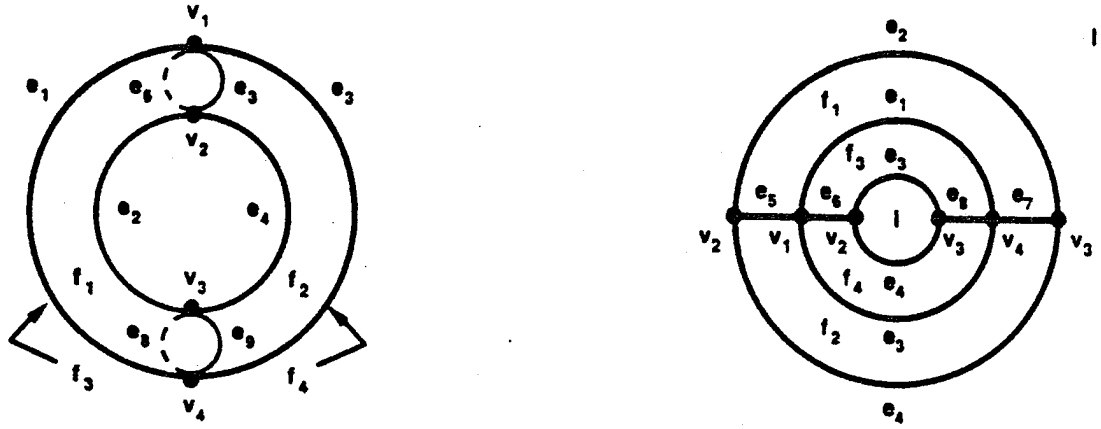
c) Mapping 2



Note: while this is a convenient proof, use of pseudographs are not necessary to prove E{V} insufficient; an example is the hypercube.

Figure 11 – 2. Insufficiency of the E{V} adjacency relationship

a) Mapping 1 (and object)



b) E{F} information

$$
\begin{array}{llll}
e_1\{F\} = e_1\{f_1\ f_3\} & & e_5\{F\} = e_5\{f_1\ f_2\} \\
e_2\{F\} = e_2\{f_1\ f_3\} & & e_6\{F\} = e_6\{f_3\ f_4\} \\
e_3\{F\} = e_3\{f_2\ f_4\} & & e_7\{F\} = e_7\{f_1\ f_2\} \\
e_4\{F\} = e_4\{f_2\ f_4\} & & e_8\{F\} = e_8\{f_3\ f_4\}
\end{array}
$$

c) Mapping 2

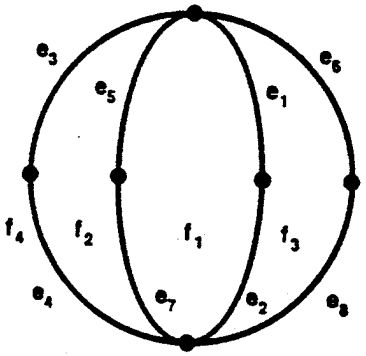
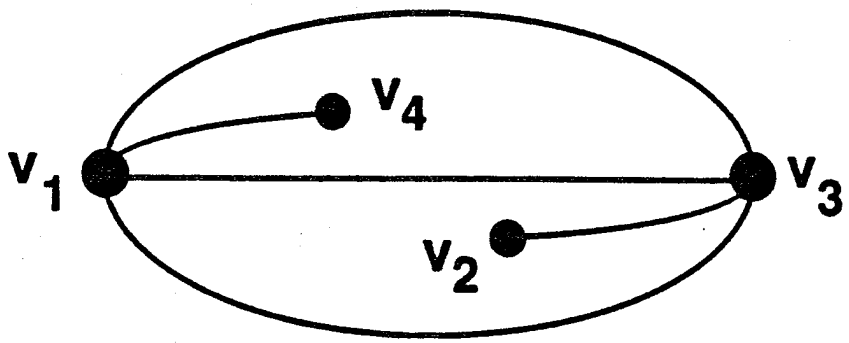


Figure 11 – 3. Insufficiency of the E{F} adjacency relationship

a) Mapping 1



b) V< V> information

$v_1< V> = v_1< v_3\ v_3\ v_3\ v_4>$ $\quad v_3< V> = v_3< v_1\ v_2 v_1\ v_1>$

$v_2< V> = v_2< v_3>$ $\quad v_4< V> = v_4< v_1>$

c) Mapping 2

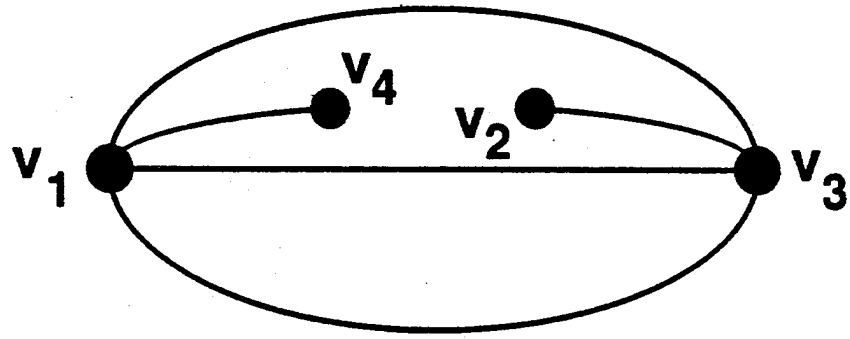
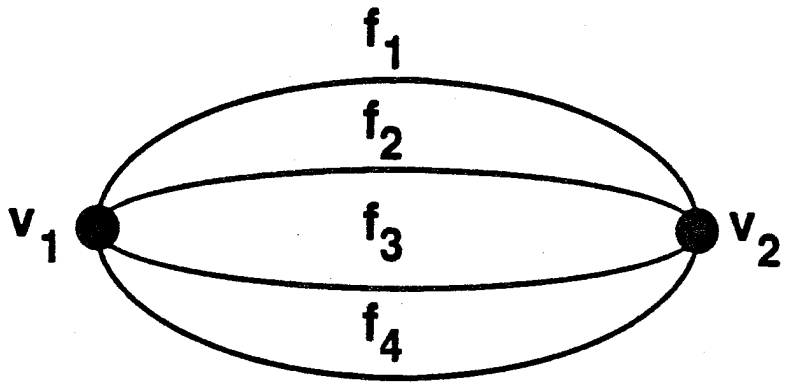


Figure 11 – 4. Insufficiency of the V< V> adjacency relationship

a) Mapping 1



b) F< V> information

$$f_1\langle V\rangle = f_1\langle v_1\ v_2\rangle \qquad f_3\langle V\rangle = f_3\langle v_1\ v_2\rangle$$
$$f_2\langle V\rangle = f_2\langle v_1\ v_2\rangle \qquad f_4\langle V\rangle = f_4\langle v_1\ v_2\rangle$$

c) Mapping 2

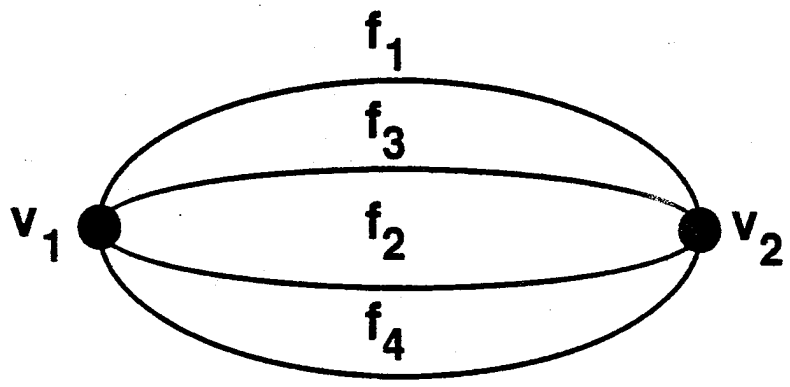
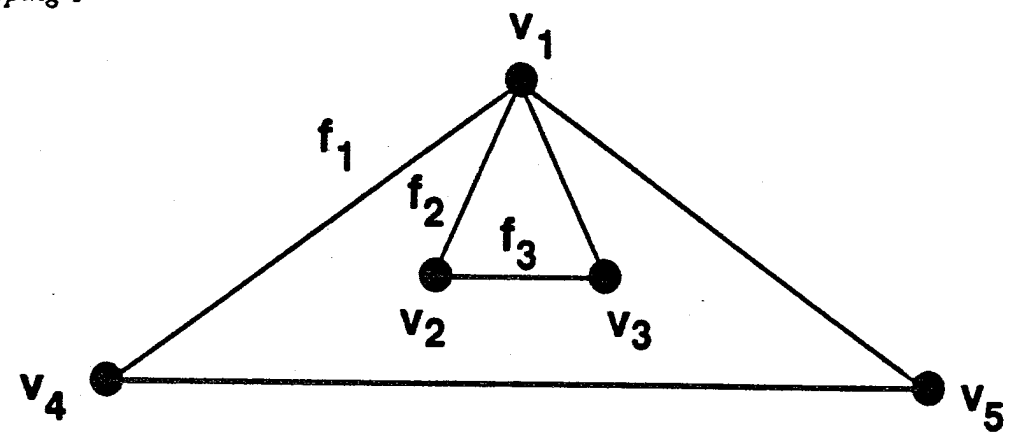


Figure 11 – 5. Insufficiency of the F< V> adjacency relationship

a) Mapping 1



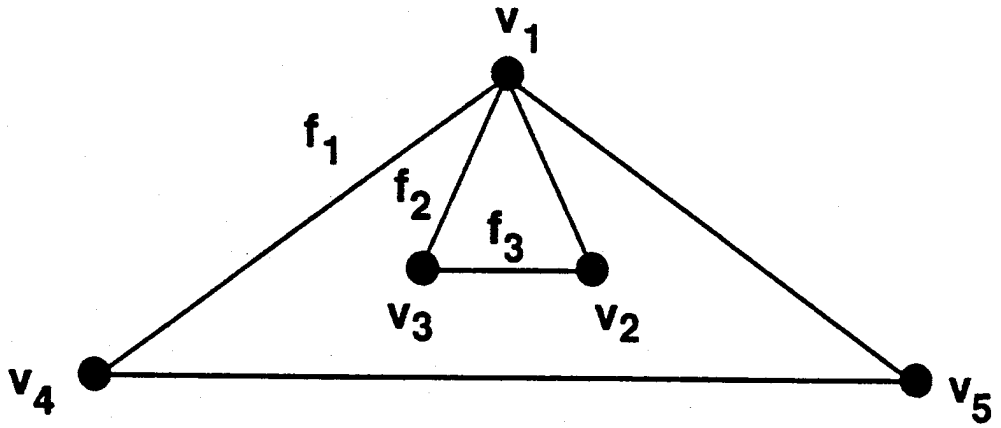
b) V< F> information

$v_1< F> = v_1< f_1\ f_2\ f_3\ f_2>$ $\quad v_4< F> = v_4< f_1\ f_2>$

$v_2< F> = v_2< f_3\ f_2>$ $\quad v_5< F> = v_5< f_1\ f_2>$

$v_3< F> = v_3< f_3\ f_2>$

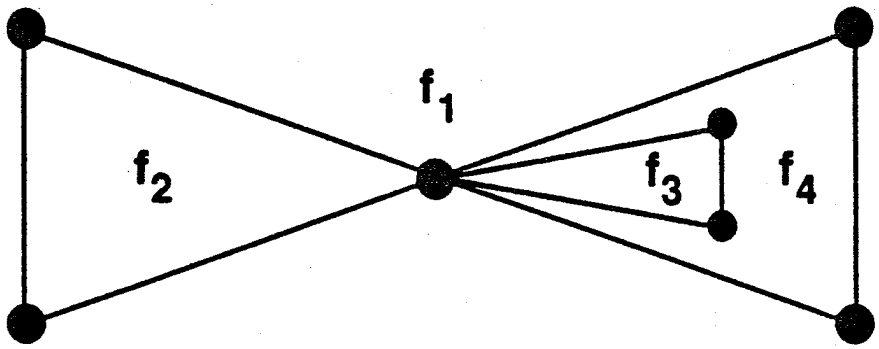
c) Mapping 2



(note that orientation of f_3 has changed)

Figure 11 – 6. Insufficiency of the V< F> adjacency relationship

a) Mapping 1



b) V< F> information

f_1< F> = f_1< f_2 f_2 f_2 f_4 f_4 f_4> f_3< F> = f_3< f_4 f_4 f_4>
f_2< F> = f_2< f_1 f_1 f_1> f_4< F> = f_4< f_1 f_1 f_1 f_3 f_3 f_3>

c) Mapping 2

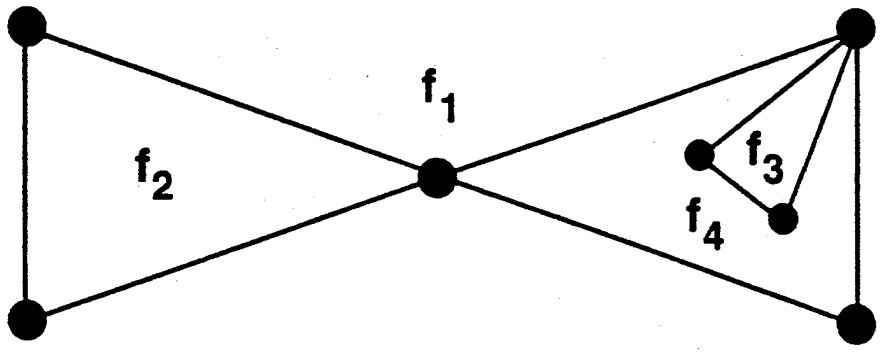


Figure 11 – 7. Insufficiency of the F< F> adjacency relationship

that transforms $V<V>$ into $V<E>$ and $F<V>$ into $F<E>$. Since both $V<E>$ and $F<E>$ are sufficient without restriction, $V<V>$ and $F<V>$ would be sufficient under these restrictions. Sufficiency under this restriction is addressed in detail in Appendix A.

In a fashion similar to V< V> and $F<V>$ under constraint, if we constrain the adjacent groups of $E\{F\}$ to be unique so that the reference edge element can be uniquely identified, $V<F>$ and $F<F>$ can be transformed to $V<E>$ and $F<E>$ respectively and can be considered sufficient under constraint (see Appendix A).

EE definition A, $E\{[E]\}^2$, is also insufficient:

Theorem 11-4: The *EE* definition A adjacency relationship, $E\{[E]\}^2$, is insufficient to unambiguously represent the adjacency topology of curved surface polyhedra.

proof (by contradiction): Using the insufficiency proof form, and the Figure 11 – 8, we can see that $E\{[E]\}^2$ is insufficient by counterexample.

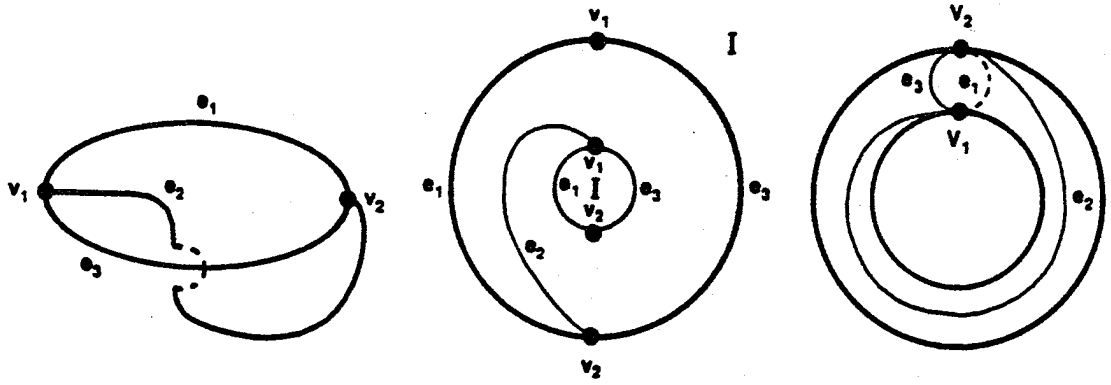
11.1.3. Sufficiency of Combinations of Adjacency Relationships

Since some of the individual element adjacency relationships are insufficient, it is interesting to consider whether combinations of individually insufficient element adjacency relationships are together sufficient.

Out of thirty-six possible unique unordered pairs of the nine adjacency relationships, twenty-one already involve sufficient relationships. Of those remaining, three have no basis for correspondence and do not appear in the list of twenty-seven correspondences of Figure 10 – 7. We are therefore left with twelve pairs of possible interest.

As previously mentioned, practical modeling systems need to label all three element types so that additional application related information may be associated with the elements. This means that at least two adjacency relationships will be needed in these

a) Mapping 1



b) E{V} information

$$e_1\{[E]^2\}^2 = e_1\{[\,e_2\ e_3][\,e_2\ e_3]\}$$
$$e_2\{[E]^2\}^2 = e_2\{[\,e_3\ e_1][\,e_3\ e_1]\}$$
$$e_3\{[E]^2\}^2 = e_3\{[\,e_1\ e_2][\,e_1\ e_2]\}$$

c) Mapping 2

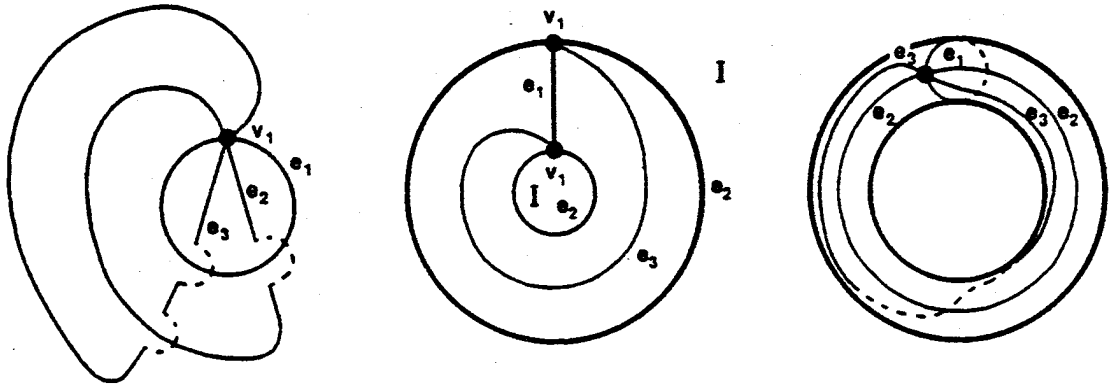


Figure 11 – 8. Insufficiency of the E{[E]} adjacency relationship

systems so that all three element types are labeled and properly associated to be able to combine the adjacency information. This makes it interesting to ask whether any two of the individually insufficient element adjacency relationships which involve all three element types are together sufficient to represent the topology of polyhedra.

Of the remaining twelve possible pairs of adjacency relationships, only five pairs of element adjacency relationships involve all three element types. One pair, consisting of $E\{V\}$-$E\{F\}$, has the same element type as reference element and therefore has the strongest correspondence. Two more pairs, $E\{V\}$-$F<V>$ and $E\{F\}$-$V<F>$, have the same element type in their adjacent groups, and the last two pairs, $E\{V\}$-$V<F>$ and $E\{F\}$-$F<V>$ are mixed with the common element type in both the reference element and adjacent group.

As will be now shown, none of these five pairs of element adjacency relationships are sufficient to unambiguously represent the topologies of polyhedra. The form of the proofs is identical to that used in the proofs of insufficiency for the individual relationships and so will not be repeated here. The only difference is that pairs of relationships instead of single relationships will be considered.

In the $E\{V\}$-$E\{F\}$ pair it will be assumed that we have access to both of the adjacency relationships in strong correspondence since this will allow the maximal amount of information to be available. If $E[V]$-$E[F]$ in correspondence is not sufficient (as we will prove next) then the pair $E\{V\}$ and $E\{F\}$ together without correspondence is also not sufficient since even less information is available.

Theorem 11-5: Each of the $E[V]^2$-$E[F]^2$, $E\{V\}^2$-$F<V>$, $E\{F\}^2$-$V<F>$, $E\{V\}^2$-$V<F>$, and $E\{F\}^2$-$F<V>$ adjacency relationship pairs with correspondence is insufficient to unambiguously represent the adjacency topologies of curved surface polyhedra.

proof (by contradiction): Using the insufficiency proof form, and the Figures 11 – 9, 11 – 10, 11 – 11, 11 – 11, and 11 – 12, respectively, we can see that $E[V]^2$-$E[F]^2$, $E\{V\}^2$-$F<V>$, $E\{F\}^2$-$V<F>$, $E\{V\}^2$-$V<F>$, and $E\{F\}^2$-$F<V>$ are each insufficient.

Another, more complex combination of particular interest is the so called "winged-edge" structure polyhedral topology representation which is discussed in more detail in Chapter 12. This representation is essentially $E[V]^2$ - $E[[E]]^2$ - $E[F]^2$ in correspondence, utilizing each of the adjacency relationships with the edge as the reference element including the *EE* definition A. All three have been proven individually insufficient previously in this thesis. However, see Section 12.3.2 for proof of the conditions required for sufficiency of the adjacency relationship pair $E[V]^2$-$E[[E]]^2$.

The seven other pairs of the original twelve pairs of interest are not examined here since they do not involve all three element types. Additionally, if one also examines others pairs involving *EE* definition A, several more pairs of possible interest can be generated.

Combinations of three or more individually insufficient element adjacencies are also not examined here.

11.2. Sufficiency of the Disconnected Graph Adjacency Relationships

Disconnected graph topology representations can always be reduced to connected graph domain by the addition of artifact edges and faces to eliminated loops and shells.

The introduction of the new element types does produce some differences in the sufficiency of the various adjacency relationships for representing polyhedral topologies. The new adjacency relationships together contain the same information available with the old adjacency relationships, but the information in some cases has been distributed over a greater number of adjacency relationships. This directly affects the sufficiency of the new element adjacency relationships.

Intuitively, the addition of the loop element effectively "spreads out" the information for sufficient representation of a polyhedron from the from the information previously available in the *FE* relationship over several new element adjacency relation-

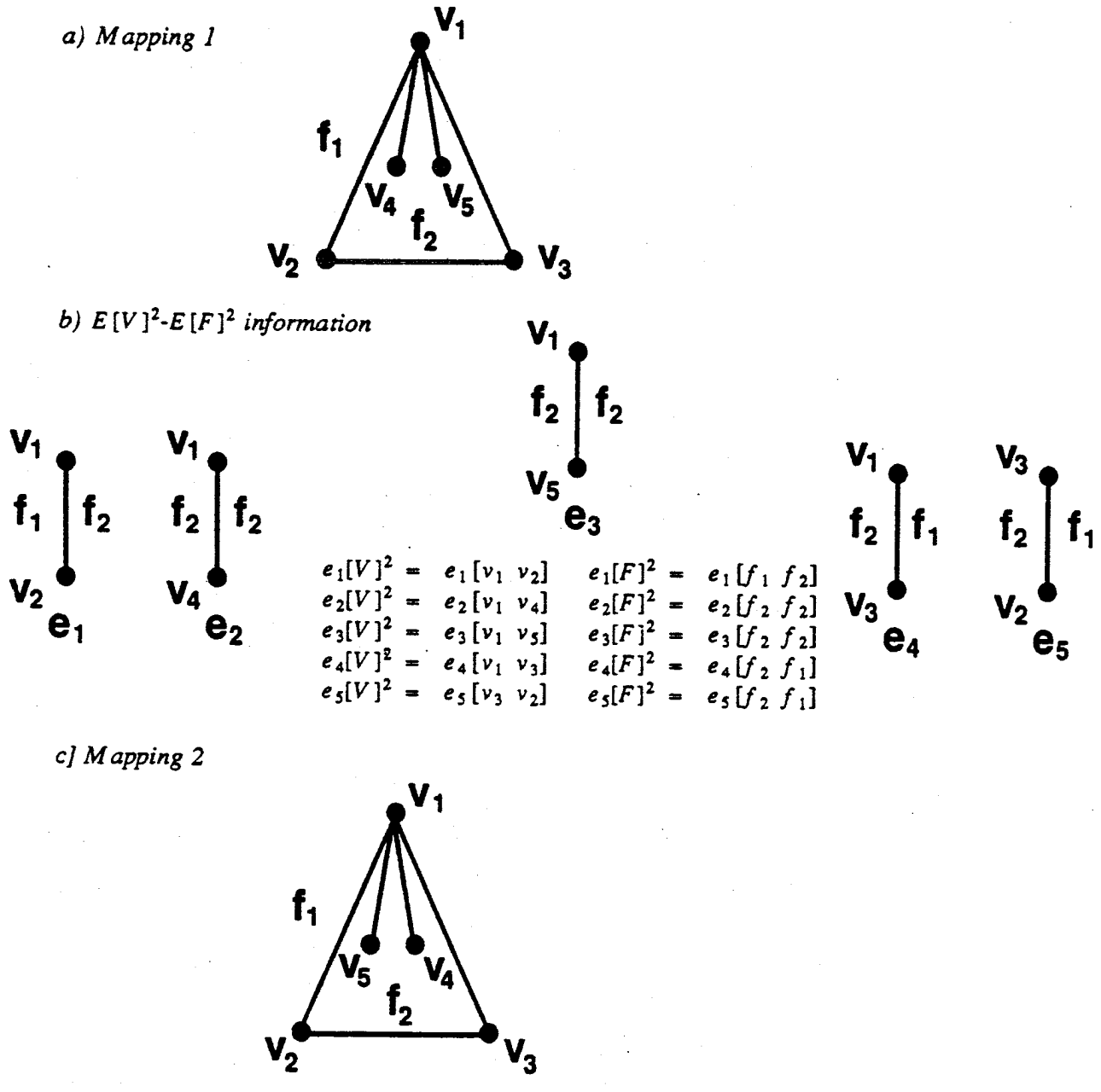
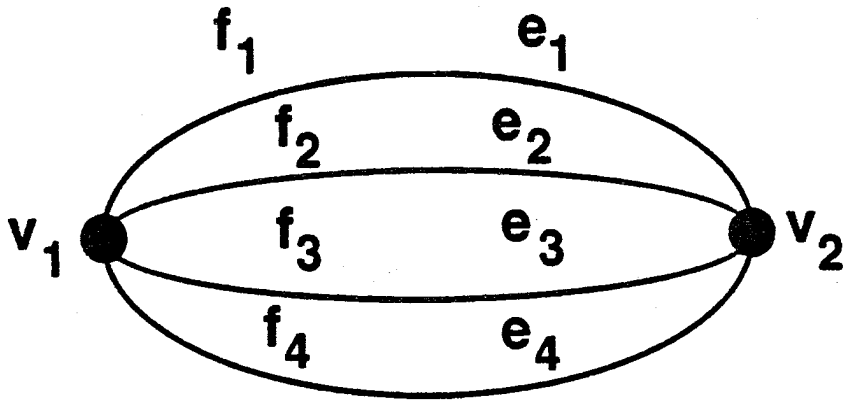


Figure 11 – 9. Insufficiency of the E[V]-E[F] adjacency relationships in strong correspondence

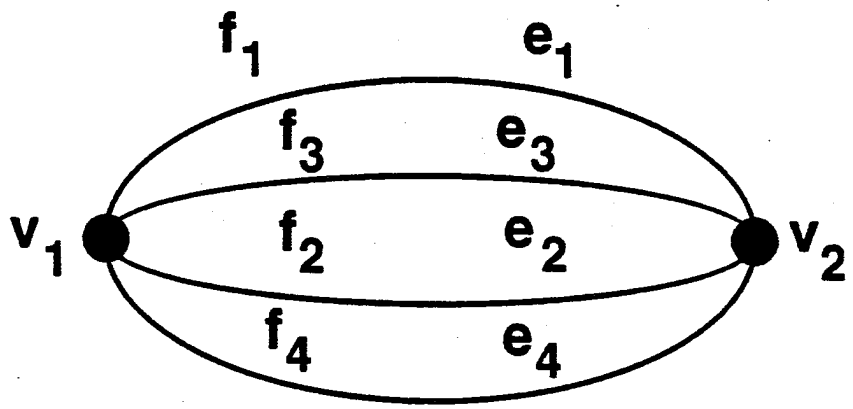
a) Mapping 1



b) $E\{V\}^2$-$F<V>$ information

$$
\begin{array}{llll}
e_1\{V\}^2 = e_1\{v_1\ v_2\} & f_1<V> = f_1<v_1\ v_2> \\
e_2\{V\}^2 = e_2\{v_1\ v_2\} & f_2<V> = f_2<v_1\ v_2> \\
e_3\{V\}^2 = e_3\{v_1\ v_2\} & f_3<V> = f_3<v_1\ v_2> \\
e_4\{V\}^2 = e_4\{v_1\ v_2\} & f_4<V> = f_4<v_1\ v_2>
\end{array}
$$

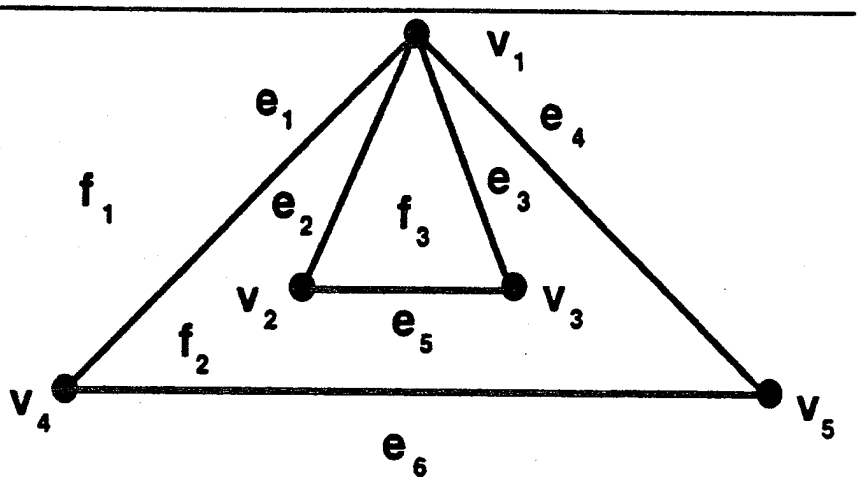
c) Mapping 2



Note that both mappings have identical $F<V>$, $E\{V\}$ information yet differ in $E\{F\}$.

Figure 11 – 10. Insufficiency of the E{V}-F< V> adjacency relationships in correspondence

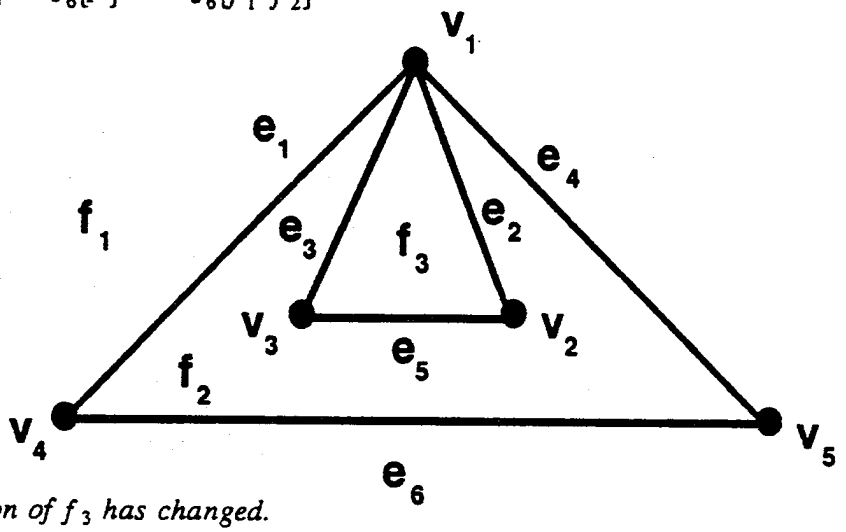
a) Mapping 1



b) $E\{F\}^2$, $E\{V\}^2$, and $V<F>$ information ($E\{F\}$ and $E\{V\}$ not in correspondence)

$e_1\{V\}^2$	=	$e_1\{v_1\ v_4\}$	$e_1\{F\}^2$	=	$e_1\{f_1\ f_2\}$	$v_1<F>$	=	$v_1<f_1\ f_2\ f_3\ f_2>$
$e_2\{V\}^2$	=	$e_2\{v_1\ v_2\}$	$e_2\{F\}^2$	=	$e_2\{f_2\ f_3\}$	$v_2<F>$	=	$v_2<f_3\ f_2>$
$e_3\{V\}^2$	=	$e_3\{v_1\ v_3\}$	$e_3\{F\}^2$	=	$e_3\{f_3\ f_2\}$	$v_3<F>$	=	$v_3<f_3\ f_2>$
$e_4\{V\}^2$	=	$e_4\{v_1\ v_5\}$	$e_4\{F\}^2$	=	$e_4\{f_2\ f_1\}$	$v_4<F>$	=	$v_4<f_1\ f_2>$
$e_5\{V\}^2$	=	$e_5\{v_2\ v_3\}$	$e_5\{F\}^2$	=	$e_5\{f_2\ f_3\}$	$v_5<F>$	=	$v_5<f_1\ f_2>$
$e_6\{V\}^2$	=	$e_6\{v_4\ v_5\}$	$e_6\{F\}^2$	=	$e_6\{f_1\ f_2\}$			

c) Mapping 2

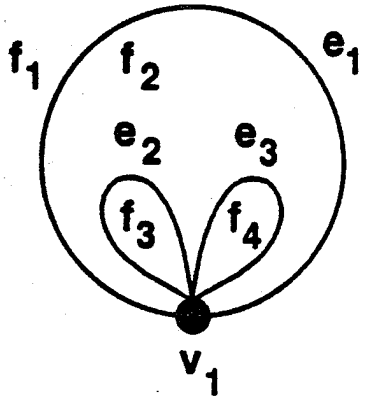


Note orientation of f_3 has changed.

Note that both mappings have identical $E\{F\}$-$V<F>$ and $E\{V\}$-$V<F>$ information yet differ in $F<V>$, $F<E>$, $V<E>$, and $E\{(E)(E)\}$.

Figure 11 – 11. Insufficiency of the E{F}-V< F> adjacency relationships in correspondence and the E{V}-V< F> adjacency relationships in correspondence

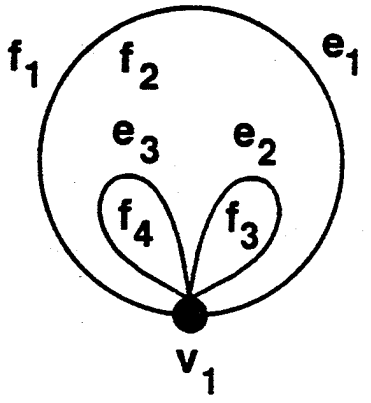
a) Mapping 1



b) $E\{F\}^2$-$F<V>$ information

$$
\begin{array}{llll}
e_1\{F\}^2 = e_1\{f_1\ f_2\} & f_1<V> = f_1<v_1> \\
e_2\{F\}^2 = e_2\{f_2\ f_3\} & f_2<V> = f_2<v_1\ v_1\ v_1> \\
e_3\{F\}^2 = e_3\{f_2\ f_4\} & f_3<V> = f_3<v_1> \\
 & f_4<V> = f_4<v_1>
\end{array}
$$

c) Mapping 2



Note that both mappings have identical $E\{F\}^2$, $E\{V\}^2$, and $F<V>$ information yet differ in $F<E>$, $E\{[E]\}^2$, and $V<E>$

Figure 11 – 12. Insufficiency of the E{F}-F< V> adjacency relationships in correspondence

ships ($F\{L\}$ and $L<E>$) in order to explicitly represent the separate multiple contours of the faces. Thus in this new system $L<E>$ is not sufficient by itself to unambiguously represent polyhedral topologies, but requires $F\{L\}$ or $L[F]^1$. The new system is primarily a change of form for convenience and efficiency; no additional information (that is, no information which is not derivable from existing information) was brought to the model compared to the artifact edge technique.

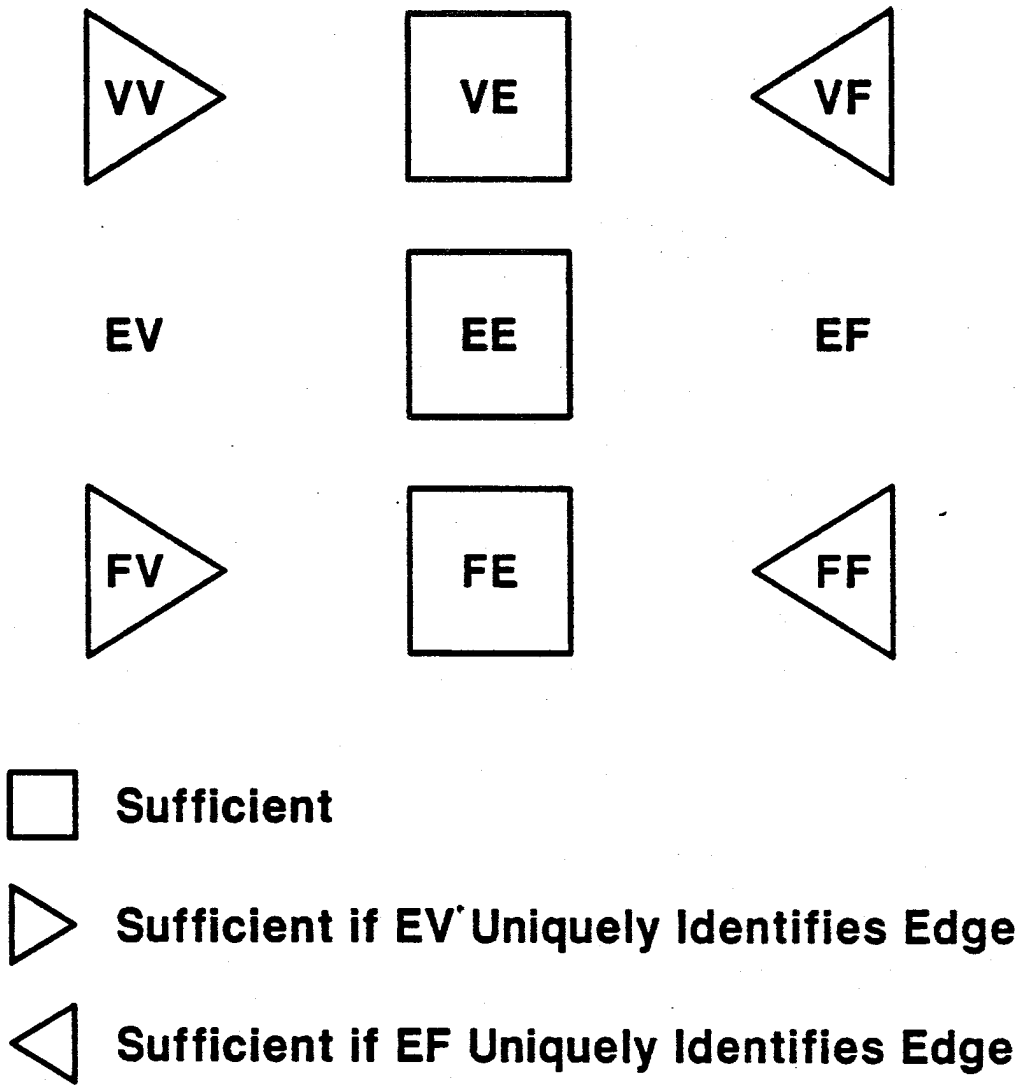
Similarly, $V<E>$ is no longer sufficient because it does not contain the information giving connectivity across the surface of a face with multiple boundaries; but $V<E>$ along with $V<L>$ and either $L[F]^1$ or $F\{L\}$ would be sufficient. There are many other possibilities, but these are not treated in detail here.

11.3. Summary of Findings

Of the nine manifold connected graph adjacency relationships, three of the ordered adjacency relationships are individually sufficient over the domain specified in Chapter 9. The three are the $V<E>$, $F<E>$, and some forms of the EE relationship. These three sufficient relationships are those which use the edge element type in their adjacent groups.

It has been proven here that six of the nine element adjacency relationships are individually insufficient to unambiguously represent the topologies of polyhedra under the domain specified.

Much of this work is based on a theorem due to Edmonds ([Edmonds 60],[Graver & Watkins 77],[White 73]) 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. 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 given here, however, involves a topological identification procedure.



note that only one rarely used form of EE is actually sufficient (see Section 11.2.1.2)

Figure 11 – 13. Adjacency relationship matrix showing sufficiency of the individual ordered adjacency relationships

Only twelve of the possible thirty-six pairs of element adjacency relationships do not involve an already sufficient adjacency relationship and do allow correspondence to be made. It turns out that there are five possible pairs out of the twelve of individually insufficient element adjacency relationships in correspondence which reference all three topological element types. Unfortunately, none are sufficient, however.

Thus, although a minimum of two adjacency relationships are 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) in order for the representation to be informationally sufficient.

A detailed proof of the sufficiency of the pair $E[[E]^2]^2$-$E[V]$ in correspondence may be found in Section 12.3.2 under the discussion of the winged edge structure.

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. This subject is covered in detail in Appendix A.

Figure 11 – 13 summarizes these results and notes sufficiencies under restrictions. It is interesting to note that the findings appear to support a kind of symmetry in the diagram when the element adjacency relationships are organized in a regular fashion as in the adjacency relationship matrix.

Chapter 12

TOPOLOGICAL DATA STRUCTURES

There are many possible topologically sufficient representation schemes for evaluated manifold boundary object solid models. The focus in this chapter is on several representational schemes for edge based representations. 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.

There are many reasons why this particular form of representation is interesting, but one reason is the historical popularity of the existing winged edge representation and an accompanying, though more general, set of operators called the Euler operators.

Detailed analysis of the existing winged edge structure in light of the information on topological sufficiency of the previous chapter led to three new data structures. All four data structures are described here with proof of their sufficiency.

12.1. Edge Based Graph Data Structures

Four different edge based data structures for representing manifold embedded graph topologies useful in solid modeling are presented: The *winged edge structure*, the *modified winged edge structure*, the *vertex-edge structure*, and the *face-edge structure*. For brevity, these will also be referred to as the *W-E*, *modified W-E*, *V-E*, and *F-E* structures, respectively.

The winged edge structure, will be familiar to many. The modified winged edge structure and the face-edge and vertex-edge structures originally introduced in [Weiler 85a] are new. All four are based on use of the edge element as the reference

element from which adjacencies with other elements are determined. Two of the structures, the winged edge and the modified winged edge structures, keep the edge information as a single unit, while the face-edge and vertex-edge structures 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 are identical in the form of their storage format, but differ greatly in the semantic interpretation of their storage format.

The data structures are discussed in the context of a computer implementation, and are described in detail after a preliminary section on supporting data structures. For each data structure a description is given followed by a proof of sufficiency.

A detailed comparison of the four data structures in terms of storage requirements, accessing efficiency, and algorithmic complexity is given in Appendix B.

12.2. Support Data Structures

Most of the topological information for the structures described here is embodied in the edge structures. Before the actual edge based data structures are described, however, we will consider the structure of the other data structure elements in the embedded graph representation. We will show a representation for these other elements which is essentially the same regardless of which of the four edge data structures is used.

Data structures for two of the three element types, faces and vertices, and a structure to pull together the entire ensemble of elements found on a single surface, called the *shell*, are now described.

Descriptions of these support structures are shown in Figure 12 – 1. 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 relation-

ships embodied.

Data objects refer to each other by the use of pointers. The naming convention for the pointers in the data structures described is:

from-element-type to-element-type ptr

where the topological element types are symbolized by the letters *s, f, l, e,* and *v* for

---

a) Pascal declarations

```
type    face_ptr = ^face;
        edge_ptr = ^edge;               { used for W-E and modified W-E edges }
        edgeuse_ptr = ^edgeuse;         { edge halves - used for F-E and V-E edges }
        vertex_ptr = ^vertex;

        shell_attrib_ptr = ^shell_attrib;
        face_attrib_ptr = ^face_attrib;
        edge_attrib_ptr = ^edge_attrib;
        edgeuse_attrib_ptr = ^edgeuse_attrib;
        vertex_attrib_ptr = ^vertex_attrib;

        ptr_type = SHELLptr, FACEptr, EDGEptr, EDGEUSEptr, VERTEXptr;

        shell = record
                sa_ptr: shell_attrib_ptr;       { attributes }
                sf_ptr: face_ptr                { heads circular doubly linked list }
                end;

        face = record
                sf_next, sf_last: face_ptr;     { doubly linked list of faces }
                fa_ptr: face_attrib_ptr;        { geometry and other attributes }
                case downptr: ptr_type of       { EDGEptr if any edges on boundary }
                        VERTEXptr:  (fv_ptr: vertex_ptr);
                        EDGEptr:  (fe_ptr: edge_ptr {or feu_ptr for edge halves} )
                end;

        vertex = record
                va_ptr: vertex_attrib_ptr       { geometry and other attributes }
                end;
```

Figure 12 – 1a. Pascal description of the support data structures

---

b) Storage allocation description

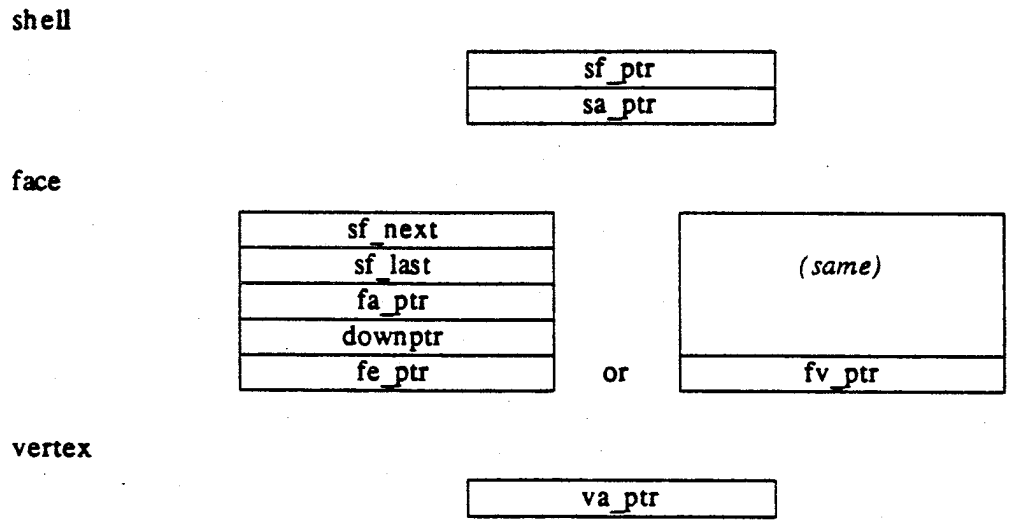


Figure 12 – 1b. Pascal description of the support data structures

shell, face, loop, edge, and vertex, respectively. There is sometimes an additional name before the "ptr" suffix when there is more than one pointer of the given type combination. Circular linked lists of lower dimension elements maintained by higher dimension elements often use pointers embedded in the lower dimension elements. The pointers are usually named in the form:

higher-dimension-type lower-dimension-type next

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 hierarchically organized top-down allows increased time efficiency in many modeling applications because objects can often be processed at higher levels of abstraction which 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 more abstract (and more concise) levels of the structure can be consulted 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 additional non-topological and non-geometric data.

Strictly speaking the boundary of a face refers to the ordered alternating sequence of edges and vertices which 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 given here 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 will point 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. This solution is more general than others which treat the situation as a special case of the shell, as will be seen when the structures are extended to handle disconnected graphs in a later section of this chapter.

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.

12.3. The Winged Edge Structure

12.3.1. Description

The winged edge or W-E 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.

The winged edge structure was originally developed by Baumgart at Stanford in the early seventies [Baumgart 72]. It served to model environments of planar polyhedral solids in computer vision research applied towards robotics. The winged edge structure has often been applied in the solid modeling field to represent the boundary graph of the topological adjacencies of faces, edges, and vertices embedded in the surface of planar faced polyhedral solid models.

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 ([Eastman & Weiler 79],[Braid et al 80]), an example of which is discussed here in a later section. These enhanced winged edge representations were incorporated into these groups' respective solid modeling systems, GLIDE [Eastman & Henrion 77], [Eastman & Thornton 79] and Build2 [Braid

79].

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 "winged edge" name results from the graphical appearance of the adjacent edges when drawn in relation to the reference edge (see Figure 12 – 2). Note that this implies a labeled graph environment, where all three basic elements are uniquely labeled or named.

As seen in Figure 12 – 2, 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 latter set of adjacencies is divided into two sections, each section 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 which directly follow or precede the reference edge in the edge cycles surrounding the two faces adjacent to the reference edge. The *ee_cw_ptr* and *ee_ccw_ptr* field names used here therefore refer to their use in determining the cycle of edges surrounding a face, as viewed from just outside the solid volume looking towards the surface. This is different from the original Baumgart field names which used *cw* (clockwise) and *ccw* (counter-clockwise) 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 $E[V]$-$E[[E]]^2$-$E[F]$ adjacency relationships in correspondence, where the adjacent edge information is represented as two ordered lists of length two, one for each endpoint of the edge. The adjacent groups of the three adjacency relationships are ordered here to allow the correspondence. The other edge based data structures embody similar information, though with subtle but important differences.

a) Pascal description

```
side = 1..2;

edge = record
       ev_ptr: array [side] of vertex_ptr;
       ee_cw_ptr,ee_ccw_ptr: array [side] of edge_ptr;
       ef_ptr: array [side] of face_ptr;
       ea_ptr: edge_attrib_ptr          { geometry and other attributes }
       end;
```

b) Storage allocation description

ev_ptr[1]	ev_ptr[2]
ee_cw_ptr[1]	ee_cw_ptr[2]
ee_ccw_ptr[1]	ee_ccw_ptr[2]
ef_ptr[1]	ef_ptr[2]
ea_ptr	

c) Diagram

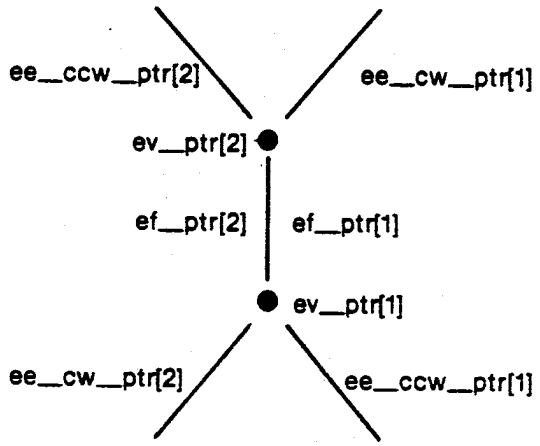


Figure 12 – 2. The winged edge data structure

12.3.2. Sufficiency

Recently there has been much interest in using the winged edge 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 pseudo-graph. The validity of these structures must therefore be examined in this wider domain.

It is worth noting that the discussion of sufficiency 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, by distributing and partitioning the information in different ways across different data structure elements. A data structure is sufficient itself as long as its information content allows for the derivation of some sufficient adjacency relationship.

As stated before, the W-E representation is essentially the $E[V]^2$-$E[[E]^2]^2$-$E[F]^2$ adjacencies in correspondence, utilizing all of the adjacency relationships which use the edge as the reference element. All three are individually insufficient (the $E[[E]^2]^2$ form of the *EE* adjacency relationship is shown to be individually insufficient in [Weiler 83] and here in Section 11.2.2). As mentioned in [Hanrahan 82], 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 $E[V]^2$-$E[[E]^2]^2$ in correspondence is sufficient to generate unique topological embeddings for the curved surface case, but only if some additional mechanisms (but not additional information) are available.

Theorem 12-1: *The pair of $E[V]^2$-$E[[E]^2]^2$ 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 uniquely construct $V< E>$ information from the $E[V]^2$-$E[[E]^2]^2$ adjacency relationship pair in correspondence. If at any point during the derivation of the next edge clockwise (looking towards the surface from just above it and outside of the solid volume) 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 $E[V]^2$-$E[[E]^2]^2$ adjacency relationship pair is insufficient.

Given access to $E[V]^2$-$E[[E]^2]^2$, 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. The only possible source of confusion would be if the next edge e_2 were a self loop. In that case, it is not known 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 all edges were examined to detect errors and backtracking were done. But one can look at each end of e_2 and only use the end where the *counterclockwise* edge from e_2 is e_1 (note that this makes the *ee_ccw_ptr* fields mandatory in the W-E structure). The only possible source of confusion then would be 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 12 – 3a). 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 12 – 3b).

Given this configuration any additional structure in the graph can only exist in sequential positions A and/or B (see Figure 12 – 3b). No matter how these additional structures are configured, the $V< E>$ relationship for v_1 will be of the form

$$v_1\langle E\rangle = v_1\langle e_1\ e_2\ edges\text{-}of\text{-}A\ \ e_1\ e_2\ edges\text{-}of\text{-}B\rangle\ .$$

The key feature of the $v_1\langle E\rangle$ 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 sequences in the $V\langle E\rangle$ adjacent group forms a circular list.

Another way of looking at it is that the $E[[E]^2]^2$ adjacencies are the same for both positions A and B and there is no basis in adjacency topology for distinguishing between them given $E[V]^2$-$E[[E]^2]^2$.

Note, however, that in order 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 and infinite sequences could be generated for $v_1\langle E\rangle$. 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 $E[V]^2$-$E[[E]^2]^2$ 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 state of procedures which 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, all three possible configurations of additional structures are shown: *1)* no additional structure (Figure 12 – 3b) *2)* one additional structure at A *or* B (Figure 12 – 3c) *3)* two additional structures at A *and* B. These structures may be disconnected (Figure 12 – 3d) or connected (Figures 12 – 3e and 12 – 3f). In any of the cases it is the repeating sequence in the circular group which is controlled by the original adjacencies of e_1 and e_2 about v_1 that guarantees a choice of *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, which can be guaranteed by use of a marker field.

It should be noted that objects with multiple self loops are not necessarily totally

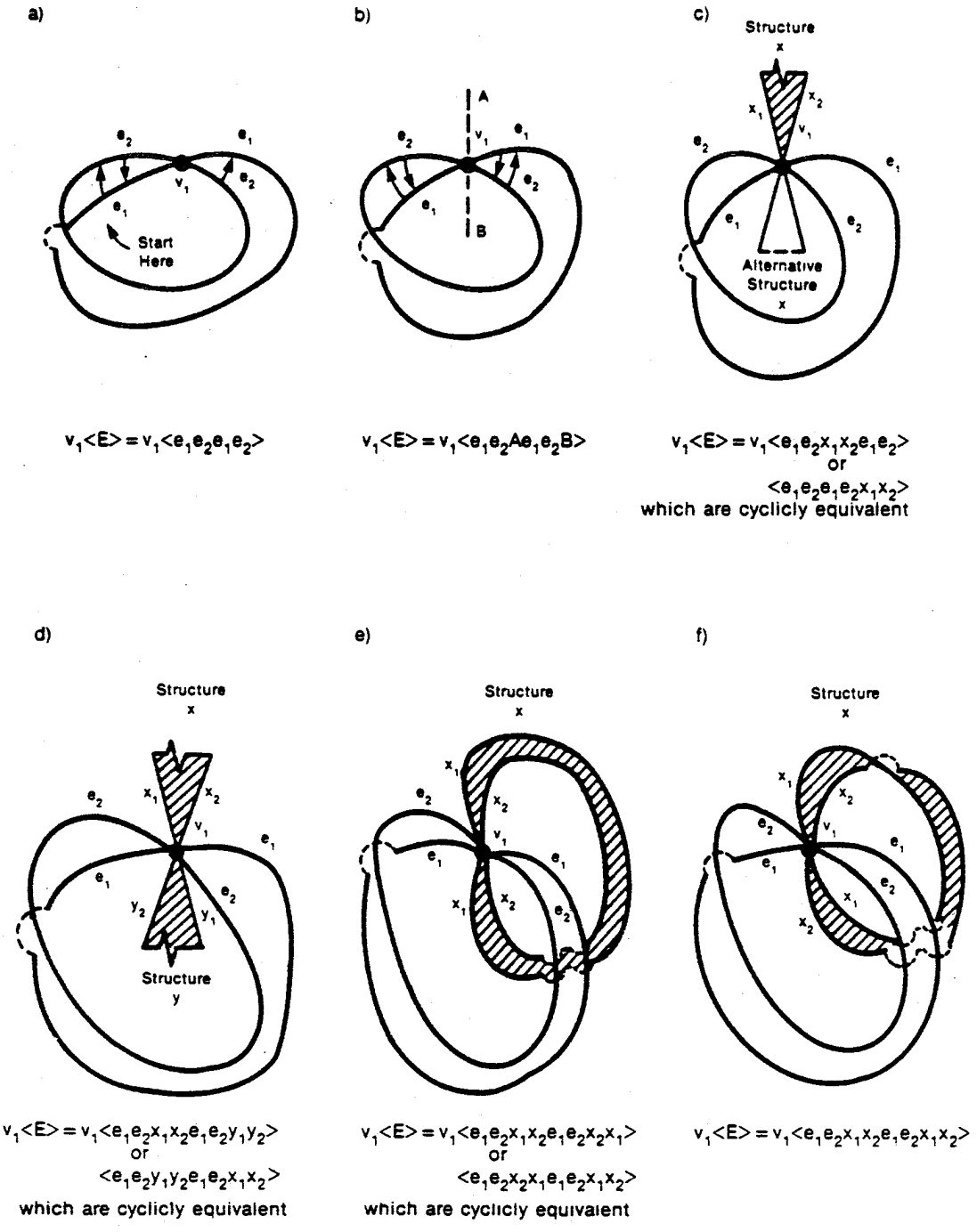


Figure 12 – 3. Sufficiency of E[V]-[[E]]

esoteric or of little concern; Figure 12 – 4 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 shown will also create repeating or identical sequences in the adjacent groups of the $F< E>$ information. We would also 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. Note that the purpose of using these additional mechanisms of extended pointers and mark bits is to easily distinguish 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, $E[V]^2$-$E[[E]^2]^2$ is sufficient to represent polyhedral topologies by way of the Edmonds theorem.

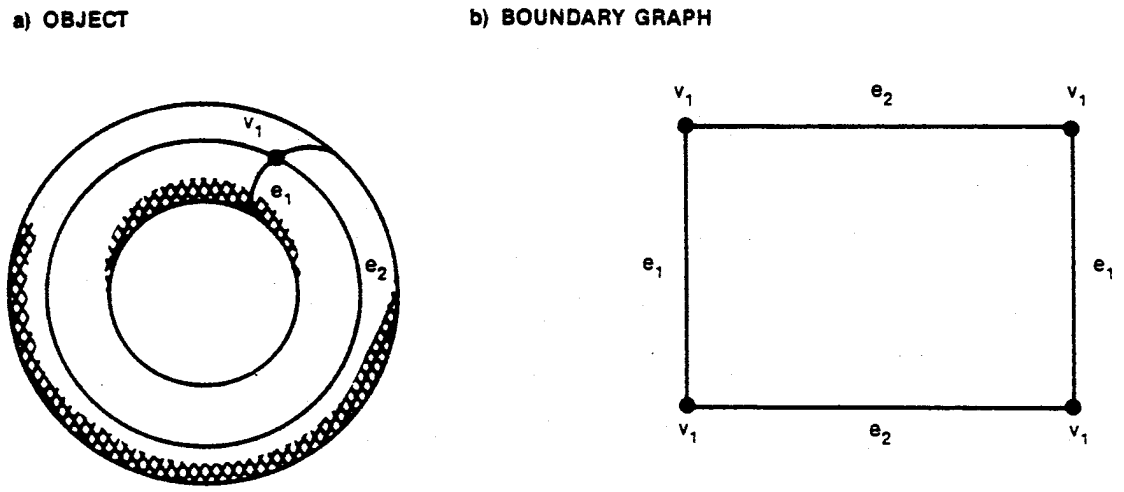


Figure 12 – 4. An object with multiple self loops using the same vertex

Perhaps more importantly, this verifies in detail that the winged edge structure $E[V]^2$-$E[[E]^2]^2$-$E[F]^2$, 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 $E[V]^2$-$E[[E]^2]^2$. As has been demonstrated, it is also sufficient for curved surface polyhedra, however, some additional and complex mechanisms, but not additional information, are necessary in order to effectively use the winged edge structure.

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.

12.4. The Modified Winged Edge Structure

12.4.1. Description

The modified winged edge or modified W-E 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 *ee_cw_ptr* and *ee_ccw_ptr* pointers are accompanied by edge *ee_cw_half* and *ee_ccw_half* 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 12 – 5; its diagram description is similar to the W-E

structure.

12.4.2. Sufficiency

Being a superset of the information contained in the W-E structure, the modified W-E structure is also sufficient. In fact, another proof of the sufficiency of the modified W-E structure can be constructed which is similar to the proof of sufficiency of the F-E structure, described later, 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.

---

a) Pascal description

```
side = 1..2;

edge = record
       ev_ptr: array [side] of vptr;
       ee_cw_ptr,ee_ccw_ptr: array [side] of eptr;
       ee_cw_half,ee_ccw_half: array [side] of side;
       ef_ptr: array [side] of fptr;
       ea_ptr: edge_attrib_ptr          { geometry and other attributes }
       end;
```

b) Storage allocation description

ev_ptr[1]		ev_ptr[2]	
ee_cw_ptr[1]	ee_cw_half[1]	ee_cw_ptr[2]	ee_cw_half[2]
ee_ccw_ptr[1]	ee_ccw_half[1]	ee_ccw_ptr[2]	ee_ccw_half[2]
ef_ptr[1]		ef_ptr[2]	
ea_ptr			

Figure 12 – 5. The modified winged edge data structure

The modified W-E structure avoids the computational complexity of algorithms for the W-E structure by explicitly including information concerning the which of two possible *uses* of an edge element is intended, via the *ee_cw_half* and *ee_ccw_half* fields. This simplifies accessing algorithms compared to the W-E structure, especially in curved surface environments. The additional fields do cause more complexity in accessing than for the V-E and F-E structures, however, as can be seen in the accessing procedures described in the Appendix B.

12.5. The Vertex-Edge Structure

12.5.1. Description

The vertex-edge or V-E 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 *ends* which is found adjacent to other edge ends around a vertex.

The structure is shown in Figure 12 – 6. The adjacency of edges around a vertex represents a circular ordered list and is represented using the *ee_cw_ptr* fields. The opposite vertex (shown as the solid dark circle), one of the adjacent faces, and the other end of the edge are also available through pointers. The *ee_ccw_ptr* field is optional, but is usually included for access time efficiency. The reference element vertex is shown in the diagram as the outlined circle.

The opposite vertex information was chosen to be included in the vertex-edge structure for efficiency in recovering the $V<V>$ adjacency relationship (see Appendix B). The choice of which face should be included is an arbitrary one.

12.5.2. Sufficiency

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

a) Pascal description

```
edgeuse = record
        ev_ptr: vertex_ptr;
        ee_cw_ptr, ee_ccwptr: edgeuse_ptr;
        ef_ptr: face_ptr;
        ee_mate_ptr: edgeuse_ptr;
        ea_ptr: edgeuse_attrib_ptr        { geometry and other attributes }
        end;
```

b) Storage allocation description

ev_ptr
ee_cw_ptr
ee_ccw_ptr
ef_ptr
ee_mate_ptr
ea_ptr

c) Diagram

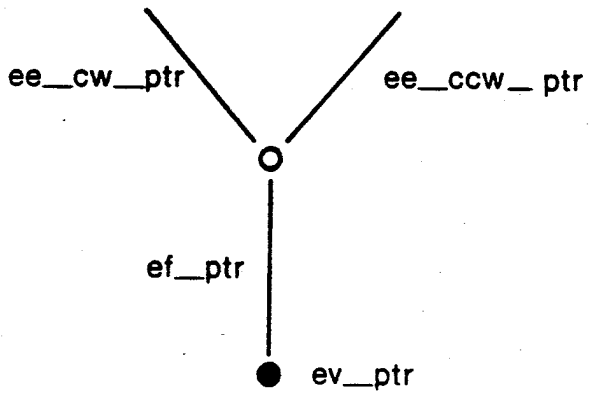


Figure 12 – 6. The vertex-edge data structure

without some additional connectivity restrictions (see Figure A – 1).

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 *ee_cw_ptr* pointers of edge end structures. First one can find one edge adjacent to each vertex by using the *ee_mate_ptr* field of an edge whose *ev_ptr* field matches the vertex in question. Then, for each vertex, follow the *ee_cw_ptr* 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.

12.6. The Face-Edge Structure

12.6.1. Description

The face-edge or F-E 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 12 – 7. The adjacency of edges around a face represents a circular ordered list and is represented using the *ee_cw_ptr* fields. Access to one vertex, the opposite adjacent face, and the other side of the edge is also available through pointers. The *ee_ccw_ptr* field is optional, but is usually included for access time efficiency.

The opposite face information was chosen to be included in the face-edge structure for efficiency in recovering the $F< F>$ adjacency relationship (see Appendix B). The choice of which vertex should be included is an arbitrary one; the one chosen here is shown as the solid dark circle in the diagram.

Each side of an edge is used only once as a boundary of a face, and the side implies an orientation towards that face. This orientation is specified here as the area to the

a) Pascal description

```
edgeuse = record
        ev_ptr: vertex_ptr;
        ee_cw_ptr, ee_ccwptr: edgeuse_ptr;
        ef_ptr: face_ptr;
        ee_mate_ptr: edgeuse_ptr;
        ea_ptr: edgeuse_attrib_ptr        { geometry and other attributes }
        end;
```

b) Storage allocation description

ev_ptr
ee_cw_ptr
ee_ccw_ptr
ef_ptr
ee_mate_ptr
ea_ptr

c) Diagram

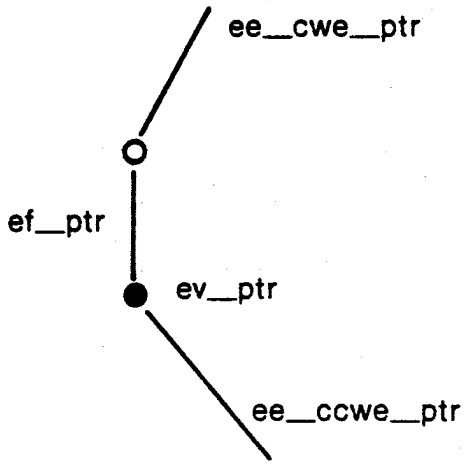


Figure 12 – 7. The face-edge data structure

right of the edge side when traveling along the edge side from the vertex specified in the *ev_ptr* field to the other vertex of the edge. Each edge is therefore used twice and in opposite directions by face boundaries when the entire embedded graph is considered.

The CRIPL-edge representation structure was an early edge based representation with an edge structure similar to the face-edge structure but intended for the planar faced domain [Stoker 74]. It utilized *ev_ptr*, *ee_cw_ptr*, and *ef_ptr* fields, but not an *ee_mate_ptr* field. As will be seen, the missing *ee_mate_ptr* 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 [Eastman & Henrion 77] until replaced by an enhanced winged edge representation [Eastman & Weiler 79].

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

12.6.2. Sufficiency

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 *ev_ptr* and *ee_cw_ptr* fields. This can be done by traversing all of the edge half structures which represent the edge sides surrounding each face, following the *ee_cw_ptr* fields of the edge half structures until one arrives back at the starting edge half structure. The vertex field is necessary for determining the adjacency of the faces, as explained below. The *ev_ptr* and *ee_cw_ptr* fields alone are therefore sufficient information for the planar faced domain.

Finding the other side of the edge is possible without the *ee_mate_ptr* field since 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 together by using the vertex information to find the identity of the full edges to which the halves belong. Identifying the edge sides together brings the whole surface together, much like a picture 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 only be used in one direction, and this direction is unambiguous, due to the convention that an edge side is a boundary of the face area found to its right when traveling from its specified vertex to its sec< ond 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 $E\{V\}$ does not in general unambiguously identify a specific edge.

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

12.7. Topological Elements and Their Uses in Adjacency Relationships

It is important to distinguish between the occurrence of edge element identity information in adjacency relationships and a representation of the edges themselves.

As can be seen from the analysis of the sufficiency of the last three data structures versus that for the winged edge structure, explicitly representing the use of the ele-

ments in the adjacency relationships is unambiguous and produces more straightforward access algorithms.

This is what was meant 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 refers to the end of the edge immediately adjacent to the vertex. In the case of the F-E structure, we are referring to a *side* of the edge used to bound a face and the edge half there refers to the side rather than end of the edge. Since by the definition of the V-E and F-E structures the end and side information is 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 from one another (in one case which of the two ends should be used, and in the other case which of the two sides should be used) without additional processing.

This is why 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. The W-E structure uses an edge as a single structure rather than representing each of its two uses in adjacencies separately. This results in a situation where one must use pointers to full edges for a given use of each edge in the adjacent group of an adjacency relationship. The problem with this is that it leaves the burden of determining which half of the edge was intended to be used in the adjacent group up to the algorithms which manipulate the structure (as well as leaving it to the proofs to show there is no ambiguity). Representing uses of edges (sides or ends) in adjacent groups eliminates such complexity.

This particular weakness of the W-E structure is addressed by the modified W-E

structure. While the modified winged edge structure essentially provides access to edge uses, the information is still distributed between its *half* fields and the edge pointers, which causes greater algorithmic complexity for accessing than is the case for the V-E and F-E structures which provide a direct representation of the edge uses. (see Appendix B).

To carry out the concept of direct use representation in a more uniform manner, vertex uses can be specified for the V-E and F-E structures. This would have allowed upward hierarchical access from the vertex to all edges using the vertex. This wasn't done in the data structures presented here because the "extra" information was not necessary for sufficiency, and the applications considered during design primarily used top-down traversals. In many applications, however, traversal in both directions is more important and vertex uses should be specified. When using parametric space representations with the F-E structure, vertex uses would also be important for specifying coordinate locations in parametric space (see Chapter 20).

12.8. 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 *ee_ccw_ptr* pointers could be removed even with curved surfaces, the *ef_ptr* field of the V-E and F-E structures could point to the other face, and the *ev_ptr* 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 cases, other element types can be used as reference elements. Such alternatives are not discussed here.

Adding vertex uses can also be an advantage in many applications, as described

above, at a cost of increased storage requirements.

12.9. Extensions for Disconnected Graphs

Until this section the assumption was made that the embedded boundary graphs being used were connected graphs. This restriction is now dispensed with in favor of extensions of the previously given support data structures and topological elements in order to allow direct representation of faces with multiple boundaries and objects with multiple shells.

12.9.1. 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 total graph is disconnected.

Multiply connected faces can be simulated by the artifact edge technique where an additional edge joins each boundary contour to some other contour on the same face (see Figure 10 – 8). The artifact edge therefore has the same face adjacent to both of its sides. This technique is not as desirable as an explicit 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, which can be made worse when a model is subjected to many modeling operations which further subdivide the artifact edges (such as Boolean operations or section cuts).

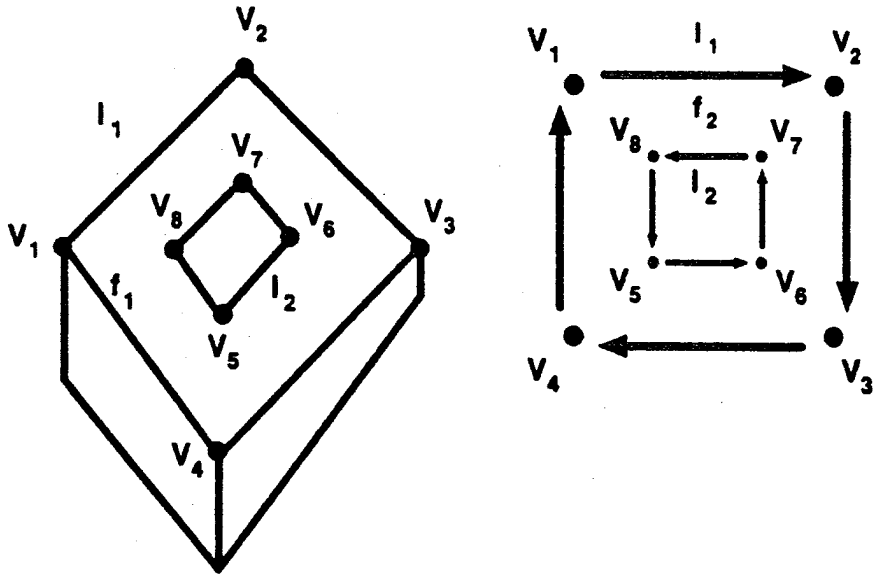
Changes to the graph data structures to directly support multiply connected faces without use of the artifact edge technique do not affect the data structures at the edge level, but rather at the face level.

There are several ways to modify the face structures already described to explicitly handle multiply connected faces. One explicit technique is an additional fixed length structure called a *loop* which is required for each boundary contour associated with a face. In this scheme the face points to a list of loop records, which contains one loop record for each contour associated with the face. Each loop record points to an edge (for the winged edge structure) or some form of edge-half (for the other structures), or, if it is a single vertex contour, to the vertex. The pointers in the edge records themselves store enough information to obtain the complete boundary contour definition. An alternative explicit implementation is to simply have a variable length face record which keeps a pointer for an edge (or the single vertex) for 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 or efficient at providing data objects with dynamically variable length.

As shown in Figures 12 – 8, and 12 – 9, the loop structure simply provides the face structure with a mechanism to maintain a linked list of pointers to boundary contours using fixed size record structures. Thus each contour adjacent with the face has a loop structure in the linked list which 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 where the particular contour consists of a single vertex rather than a series of edges. This 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, in a situation where 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.



$f_1\{L\} = f_1\{l_1\ l_2\}$
$l_1\langle V\rangle = l_1 < v_1\ v_2\ v_3\ v_4 >$
$l_2\langle V\rangle = l_5 < v_6\ v_7\ v_8\ v_9 >$

Figure 12 – 8. Loop structure adjacency relationships

Pascal declarations

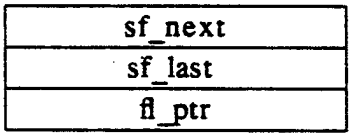
```
type   loop_ptr = ^loop;

       face = record
              sf_next,sf_last: face_ptr;
              fl_ptr: loop_ptr
              end;

       loop = record
              fl_next: loop_ptr;
              case downptr: ptr_type of
                     VERTEXptr:  (lv_ptr: vertex_ptr);
                     EDGEptr:    (le_ptr: edge_ptr {or edgeuse_ptr})
              end;
```

Storage allocation description

face



loop

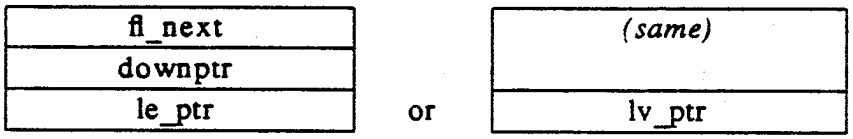


Figure 12 – 9. Modified and additional data structures for loop

Naturally, if backpointers are used in the edge records, as is the case with the four edge structures presented, what were previously edge-to-face pointers would 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

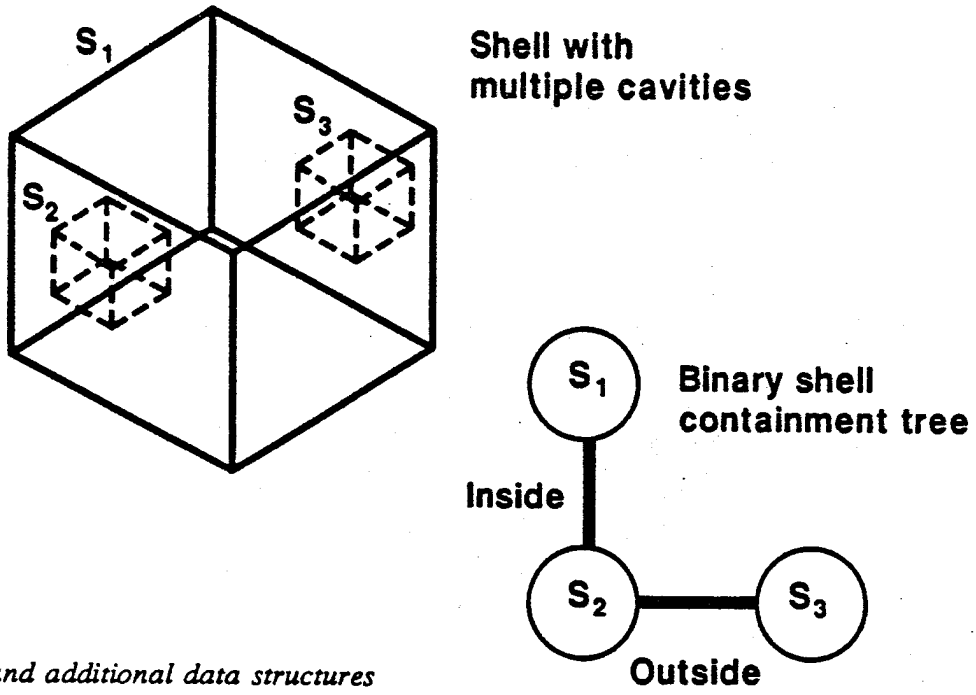
biased towards the top-down hierarchical approach of maintaining relationships between higher to lower level dimensionality elements. The use of this structure seems more natural with the W-E and F-E structures for this reason. A representation using the V-E edge structure would probably be vertex centered 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.

12.9.2. Multiple Shell Objects

Another situation encountered in solid modeling is where a solid object contains one or more hollow cavities, but still consists 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. This requires changes in the data structure above the face level. This can be handled most simply as 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. This can be done utilizing a binary tree structure where 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 12 – 10). Maintaining this additional topological information can increase efficiency in many geometric modeling applications, such as interference detection, for example. This approach can be used to represent not only single solid volumes with multiple voids, but can also be used to represent 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 [Eastman & Weiler 79], although they are equally applicable to



modified and additional data structures

Pascal declarations

```
type    shell_ptr = ^shell;

        shell = record
                ss_inside_ptr,ss_outside_ptr: shell_ptr;
                sf_ptr: face_ptr
                end;
```

Storage allocation description

shell

ss_inside_ptr
ss_outside_ptr
sf_ptr

Figure 12 – 10. Shell structure

the other three structures presented here.

Chapter 13

EULER OPERATORS

The Euler operators are a set of operators which can manipulate manifold boundary graph based topology representations in a low level, incremental and systematic fashion, constructing a topology primarily edge by edge. Euler operators can be used with any of the four previously described edge based data structures.

This chapter describes the basic functions of the Euler operators and describes in detail the external interface of a specific implementation.

13.1. The Euler Operators

The Euler operators were originated by Baumgart [Baumgart 72] as a means of manipulating the winged edge data structures. The operators 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 create and manipulate the model of the embedded graph on an edge by edge basis in a systematic way independent of the actual data structure.

The advantage of this approach is that it provides a flexible base for higher level operators while insulating them from the details and complexities of the data structures utilized. Indeed while an *implementation* of the Euler operators is specific to the data structure actually used (for example, any of the four data structures described in the previous chapter), the *external interface* to the operators can remain the same, and the implementation of all higher level operations can be identical regardless of the data structure chosen.

There are many variations on how the Euler operators can be implemented. The version of the operators described here were designed and implemented by the author at Carnegie-Mellon University and were originally part of the GLIDE system [Eastman & Weiler 1979].

The description is included here because it offers an example of how the Euler Operators have been provided for a complete implementation of a manifold solid modeling system, and because they have strongly influenced the design of the new non-manifold operators described in a later section. Alternative versions of the Euler operators have also been defined for GEOMED [Baumgart 1974], Build2 [Braid et al 1978], and more recently GWB [Mantyla 1982]. A discussion of the theoretical sufficiency of the Euler operators to cover the representation space is given in [Mantyla 84].

13.2. The Basic Operators

Five of the basic Euler operators presented below, *MSFLV*, *MEV*, *ME*, *GLUE*, and *KE* are sufficient to create any topology, but others are included to add convenience and flexibility to the surface construction process.

The names of the Euler operators traditionally follow those originally defined by Baumgart. They describe, with a few exceptions, the effect the operators have on the existence of topological elements as well as the genus of the graph. The *M* stands for "Make" or create, and *K* stands for "Kill" or delete. Each of these is followed by letters signifying the types of topological elements created or deleted; *S*, *F*, *L*, *E*, and *V* stand for shell, face, loop, edge, and vertex, respectively, and *G* stands for "genus". Thus *MEV* stands for "make edge, vertex", and *KEMSFL* stands for "kill edge, make shell, face, loop". Other operators, such as *GLUE* and *ESPLIT*, have names describing their more generic functionality.

The eight basic operators and their subcases, and a few additional operators are shown in Table 13 – 1. The destructive operators which are complements to the con-

structive operators are shown in the same row alongside each constructive operator. Also shown are compound operators which could be implemented as sequences of the basic operators, as well as an additional miscellaneous operator. More detailed functional descriptions will follow.

The operator names are shown in upper case; below each in lower case are the names of the subcases which each operator can distinguish and handle automatically.

The Euler Operators create small incremental changes in the numbers of the components in a topology. These effect of these changes, as well as their effect on the genus of the graph, are described in Table 13 – 2. The subscripted numbers are variable quantities whose values depend on the number of elements of the specific type which involved in the operation. The type in this case is indicated by the subscripted letter. In some cases, such as for *kflevs*, the number of elements involved may be the same for more than one element type, in which case the subscript used is the same for all of those which are related and must have the same quantity. Note that these incremental changes, when substituted into the Euler-Poincaré equation, will always balance the equation. Thus if Euler operations are treated as atomic (non-interruptible) operators, the data structures are always constrained to represent a valid manifold topology at every stage.

13.3. Direction-Edge-Vertex Positioning Specification

One of the problems in designing an interface to the Euler operators is in how to unambiguously specify the exact placement of new edges. For example, in Figure 13 – 1, if we know we want to attach a new edge to an existing vertex v_1, should the new edge be attached to the vertex above or below the edge between vertices v_1 and v_4 ? Various implementations have solved this problem in different ways.

One approach is to restrict the constructive operators so that it is not possible to create a situation like that shown in Figure 13 – 1, where the result could be ambigu-

Table 13 - 1. The Euler Operators

constructive	destructive	compound	miscellaneous
MSFLV	*KSFLEV*	*MME*	*LMOVE*
MEV	*ESQUEEZE(KEV)*	*ESPLIT*	
ME	*KE*	*KVE*	
mefl	*kefl*		
mekl	*keml*		
meksfl	*kemsfl*		
GLUE	*UNGLUE*		
kflevmg	*mflevkg*		
kflevs	*mflevs*		

Table 13 - 2. Operator Effect on the Numbers of Topological Elements

operator	*changes in number of topological elements*					
	Shells	Faces	Loops	Edges	Vertices	Genus
MSFLV	+1	+1	+1		+1	
MEV				+1	+1	
ME						
mefl		+1	+1	+1		
mekl			−1	+1		
meksfl	−1	−1	−1	+1		
GLUE						
kflevmg		−2	−2	$-n_e$	$-n_e$	+1
kflevs	−1	−2	−2	$-n_e$	$-n_e$	
KSFLEV	−1	$-n_f$	$-n_l$	$-n_e$	$-n_v$	$-n_g$
ESQUEEZE				−1	−1	
KE						
kefl		−1	−1	−1		
keml			+1	−1		
kemsfl	+1	+1	+1	−1		
UNGLUE						
mflevkg		+2	+2	$+n_e$	$+n_v$	−1
mflevs	+1	+2	+2	$+n_e$	$+n_v$	

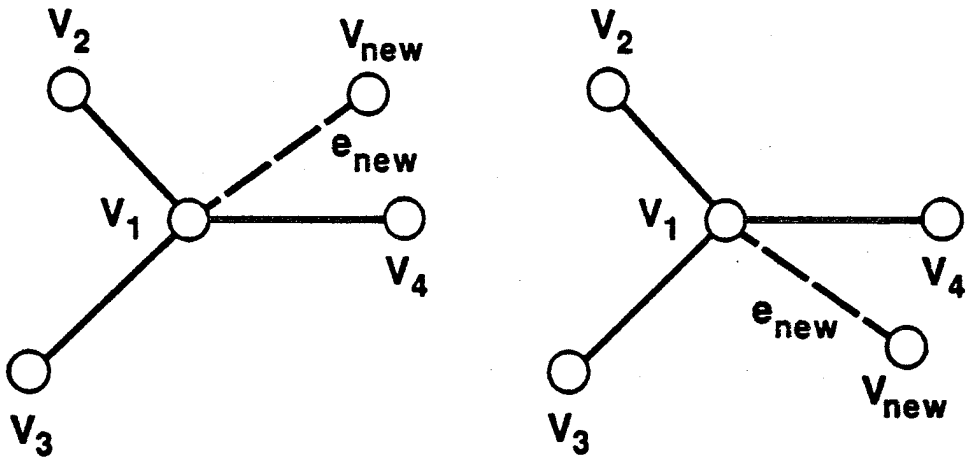


Figure 13 – 1. Specification of placement for an edge

ous unless additional information is specified. This has the effect of restricting the order in which the operators can be applied. The advantage of the approach is that no explicit positioning specification is necessary. The disadvantage is that knowledge of the restrictions on the order in which operators can be applied to achieve a given result must be embedded in all the algorithms which use the Euler operators. The destructive operators must similarly be restricted so as not to create graph configurations which could lead to ambiguous situations for the constructive operators.

Braid, Hillyard, and Stroud [Braid et al 78] used a mixed mode approach where the additional information needed to unambiguously specify the desired action of the operation could either be the identity of related topological components (such as specifying which loop an *MEV* operation should place its new edge into) or the relation of the new component to an existing one (such as specifying that an *MEV* operation should place its new edge clockwise of some specified existing edge).

The direction-edge-vertex edge placement specification technique [Eastman & Weiler 79] uniformly requires the explicit inclusion of the required positioning information in all situations where additional information is necessary for disambiguation of the semantics of the Euler operators. In this technique the exact position of a new edge is specified unambiguously with a vertex, edge, and rotation direction (clockwise or counter-clockwise). The new edge will use the vertex specified as one of its end-points and will lie in the specified rotation direction from the specified edge about the specified vertex (see Figure 13 – 2). The new edge can be said as being "(counter)clockwise from edge e_1 about vertex v_1". The direction is specified as being clockwise or counter-clockwise from the point of view of an observer looking towards the vertex from just outside the solid volume above the surface in which the vertex is embedded. The advantage of this technique is that it provides an unambiguous specification of placement without any restrictions on ordering the sequence of

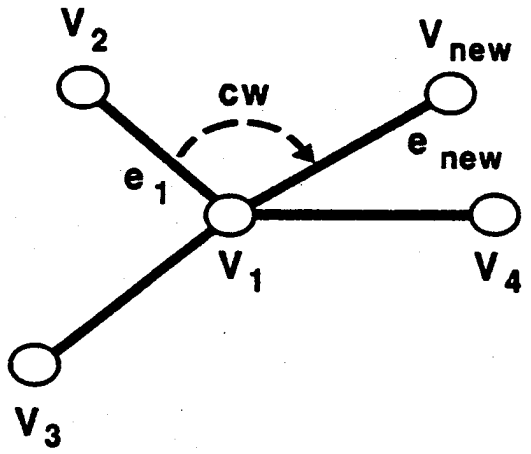


Figure 13 – 2. Direction-edge-vertex edge placement specification

operators. The disadvantage is that the positioning information must be specified explicitly, creating an explicitly more complex interface.

13.4. A Specification of the Euler Operators

Specific functional descriptions of the individual operators follow.

The version of the operators described here were designed and implemented by the author at Carnegie-Mellon University and were originally part of the GLIDE system [Eastman & Weiler 1979]. The naming convention has been changed, however; the older name *body* used in the original version has been replaced with the name *shell* for consistency with the rest of the material here.

The operators described use the direction-edge-vertex specification described earlier. Thus direction parameters must be specified as *clockwise* or *counter-clockwise*.

The interface to each operator is described in a Pascal style, listing its input parameters, followed by its set of output parameters specified as *var* (call-by-reference) parameters. Optional input parameters are italicized; if not specified they should be *nil* valued pointers or *unspecified* rotational directions. This calling sequence description is then followed by a detailed description of the operator functionality and the various subcases handled by the operator. References to topological element types in the calling sequence descriptions refer to pointers to the elements rather than the elements themselves.

The operator specifications are independent of any specific underlying data structure. The operator specifications are followed by diagrams illustrating their function in Figures 13 – 3, 13 – 4, and 13 – 5. The diagrams follow the same order as the specifications.

13.4.1. Basic Operators

MSFLV (var face_ptr: newf; var loop_ptr: newl; var vertex_ptr: newv)

"Make Shell, Face, Loop, Vertex" creates a new manifold surface in the topology, and is therefore the first operator used in any topology construction. The new shell resulting from the operation is always treated implicitly rather than explicitly, since all further operations deal with lower level topological elements, and never explicitly require the identity of the shell. *MSFLV* creates a new shell, the face *newf*, the loop *newl*, and the vertex *newv*. The single vertex created, *newv*, can be used as a starting point for subsequent construction of additional topological features on the manifold surface.

MEV (vertex_ptr: v; *edge_ptr: e; dir_type: dir;*

var edge_ptr: newe; var vertex_ptr: newv)

"Make Edge, Vertex" creates a new edge and vertex. The new edge *newe* starts at the existing vertex *v* and ends at the new vertex *newv*. If the optional placement arguments *e* and *dir* are specified, *newe* will be positioned in direction *dir* (clockwise or counter-clockwise) from edge *e* about vertex *v*, as seen when looking towards the manifold surface from just outside the volume above the vertex *v*.

ME (*vertex_ptr: v1; edge_ptr: e1; dir_type dir1*;

vertex_ptr: v2; edge_ptr: e2; dir_type dir2;

var edge_ptr: newe; var face_ptr: newf; var loop_ptr: newl)

"Make Edge" creates an edge between the existing vertices *v1* and *v2*. If optional placement is specified, the new edge, *newe*, will be direction *dir1* (clockwise or counter-clockwise) about vertex *v1* from edge *e1*, and direction *dir2* (clockwise or counter-clockwise) about *v2* from *e2*.

mefl: *"make edge, face, loop"* occurs when the new edge will close off one portion of the face it is on from the rest of the face. In this case, the new face, *newf*, and loop, *newl* will lie to the *dir1* side of *newe* about *v1*.

mekl: *"make edge, kill loop"* occurs when the new edge will not

close off one portion of the face it is on from the rest of the face. In this case, the vertices *v1* and *v2* were on different loops of the same face, but afterwards will be located on the same loop. The surviving loop is the loop associated with *v1*.

meksfl: *"make edge, kill shell, face, loop"* occurs when the two specified vertices are on different shells. The new edge links together the two shells into a single shell. The shell of *v1* is the surviving shell.

GLUE (face_ptr: f1; edge_ptr: e1; face_ptr: f2; edge_ptr: e2)

"Glue Faces" merges two single loop faces together, deleting both faces and loops and one set of edges and vertices, with the effect of joining together the volumes which the two faces are bounding. Both loops must have the same number of edges and vertices, and must have no edges in common. The merge is performed so that *e1* of *f1* and *e2* of *f2* are merged into the same edge. The surviving set of edges and vertices are those of *f1*.

kflevmg: *"kill face, loop, edge, vertex, make genus"* occurs when both faces exist on the same shell. The glue operation increases the genus of the shell by one, which is topologically equivalent to adding a handle to the surface.

kflevs: *"kill face, loop, edge, vertex, shell"* occurs when the two faces exist on different shells. The glue operation merges the two shells together into a single shell, with the shell of *f1* being the survivor.

13.4.2. Complement Operators

KSFLEV (vertex_ptr: v)

"Kill Shell, Face, Loop, Edge, Vertex" determines the shell of the specified vertex *v* and deletes the shell and all its constituent topological elements (including the specified vertex).

ESQUEEZE (edge_ptr: e; *vertex_ptr: v*; var vertex_ptr: vsurvivor)

"Edge Squeeze" (also known as *"Kill Edge, Vertex"*) "squeezes" the ends of the specified edge *e* together, deleting the edge and a vertex while preserving adjacencies. The optional parameter *v*, if specified, designates which vertex of the edge *e* will survive; in any case, the

surviving vertex is indicated by the *vsurvivor* return parameter.

KE (edge_ptr: e1; *vertex_ptr: v1;* var loop_ptr: newl)

"Kill Edge" deletes the specified edge *e*.

kefl: *"kill edge, face loop"* occurs when the edge to be deleted separates two different faces. In this case, the edges of the two loops using the deleted edge are merged and one face and loop are deleted. The surviving face and loop are those found to the right of the edge to be deleted, when traversing the edge from the optionally specified vertex *v1* to the other vertex. Any other loops of the deleted face are moved to the surviving face.

keml: *"kill edge, make loop"* occurs when the edge to be deleted occurs twice on a loop of a single face. In this case, a new loop, *newl*, will be generated on the same face.

KEMSFL (edge_ptr: e1; *vertex_ptr: v1;*

var face_ptr: newf; var loop_ptr: newl)

"Kill Edge, Make Shell, Face, and Loop" deletes the specified edge, *e*, which is required to have the same face on both sides. The two disconnected graph components that result are each treated as separate shells. *KEMSFL* is shown as a subcase of *KE* in the tables and diagrams because of its functional similarity to other subcases of *KE*. Differentiating *kemsfl* from *keml* cannot be done without explicit indication of intent, however, which is why a separate operator, *KEMSFL* is provided. The face and loop to be left on the original shell are those found to the right of the edge to be deleted, when traversing the edge from the optionally specified vertex *v1* to the other vertex. Any other loops of the original face are also left on the specified face.

UNGLUE (edge_ptr: e1; var face_ptr: newf1,newf2; var loop_ptr: newl1,newl2)

"Unglue Faces" takes a single circuit of edges starting with edge *e1* which have marked using an edge marking facility, separates the model along the circuit, replicating edges and vertices as necessary. The process creates two new faces *newf1* and *newf2*, and their respective loops *newl1* and *newl2* which utilize the edges on each side of the separated circuit. This keeps the volume closed in order to maintain a closed manifold representation. The circuit marked for the *UNGLUE*

must be complete, have no struts or self loops, and must not cross itself.

mflevkg: *"make face, loop, edge, vertex, kill genus"* occurs when the separation induced by the operation leaves the graph still connected. In this case the specified circuit lies on a handle of the shell which has a genus of one or more. The handle is removed, and the single shell with genus reduced by one is the result.

mflevs: *"make face, loop, edge, vertex, shell"* occurs when the separation induced by the operation creates a disconnected graph. Each component of the result is treated as a separate shell; thus two separate volumes is the result.

13.4.3. Composite Operators

MME (integer: number; *vertex_ptr: v; edge_ptr: e; dir_type: dir;*
var edge_ptr: ebeg,eend; var vertex_ptr: vend)

"Make Multiple Edges" creates a connected chain of *number* edges starting at the specified vertex *v*. If the optional placement arguments *e* and *dir* are specified, *ebeg*, the first edge created, will be positioned in direction *dir* (clockwise or counter-clockwise) from edge *e* about vertex *v*, as seen when looking towards the manifold surface from just outside the volume above the vertex *v*. The action is equivalent to a series of *MEV*'s, and if vertex *v* is not specified, a *MSFLV* followed by a series of *MEV*'s.

ESPLIT (edge_ptr: e; *vertex_ptr: v;* var edge_ptr: newe; var vertex_ptr: newv)

"Edge Split" splits the specified edge *e* into two connected edges, *e* and *newe*. A new vertex, *newv*, is created between these two edges. The optional parameter *v*, if specified, designates which vertex of the edge *e* will be found on the new edge. The effect of this operator could be simulated by application of the *KE* operator followed by *MEV* and *ME* operators, but unlike *ESPLIT*, edge *e* would be entirely replaced rather than modified in place and, by side effect, a face could be deleted and replaced with a new one, perhaps shifting ownership of interior loops.

E(vertex_ptr: v)

"Kill Vertex, Edge" deletes the vertex specified by *v* and any edges using this vertex. If necessary, faces and their loops are also deleted. Ownership of additional loops of deleted face falls the remaining surrounding face. Three cases may occur. First, when the vertex is the only boundary of a shell, it is equivalent to a *KSFLEV*. Second, when the vertex is a single vertex loop of a face, it is equivalent to an application of *ME(mekl)* followed by an *ESQUEEZE*. Third, when there are *n* edges using a vertex, the result is equivalent to *n - 1 KE*'s followed by *ESQUEEZE*.

Miscellaneous Operators

OVE(loop_ptr: l; face_ptr: f)

"Loop Move" moves the loop *l* from its current face to the face *f*. This can be useful for moving loops from an original face over to the new face created by application of the *ME(mefl)* operator. It is not strictly an Euler operator since it doesn't involve any changes to the terms of the Euler equation.

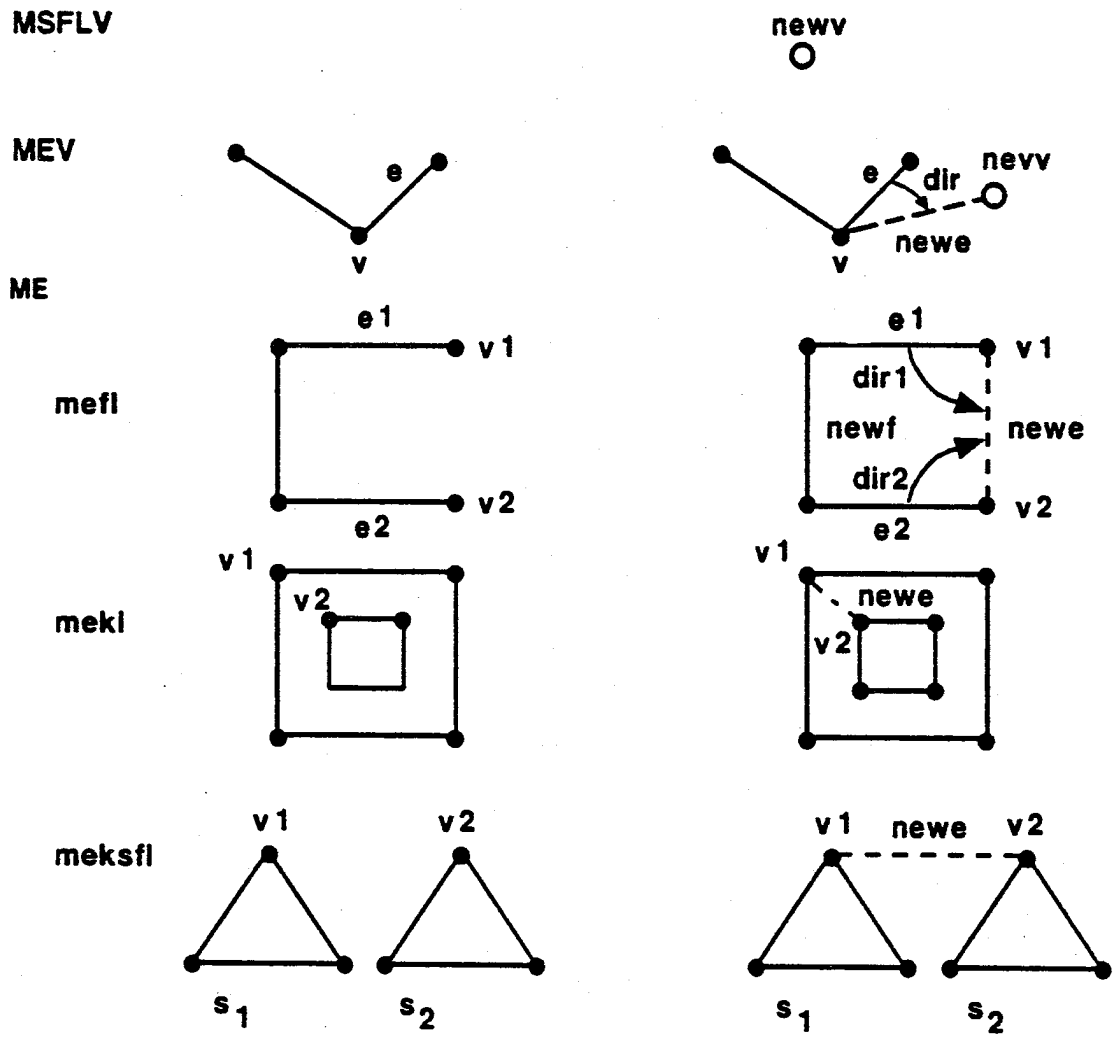


Figure 13 – 3. Action of the Euler operators

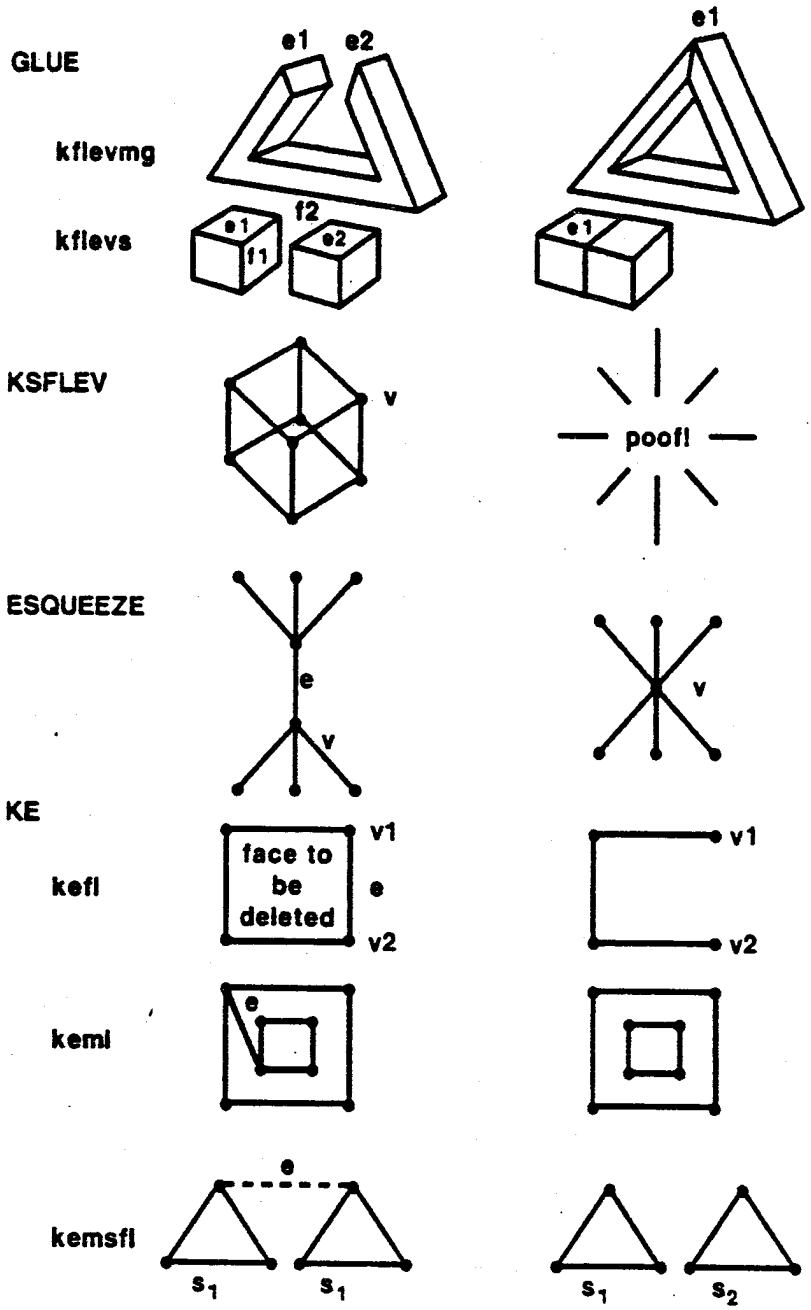


Figure 13 – 4. Action of the Euler operators

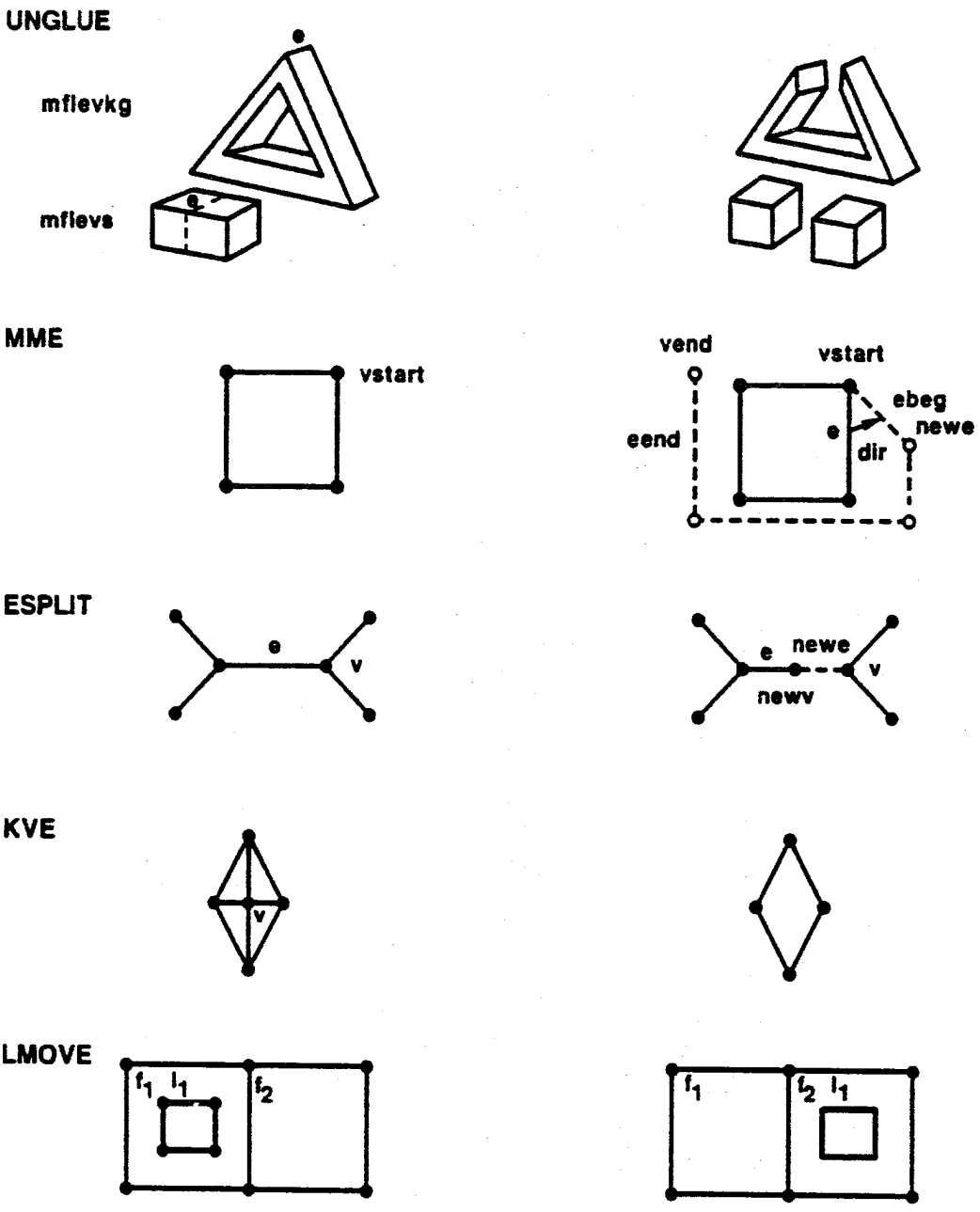


Figure 13 – 5. Action of the Euler operators

13.5. Building Higher Level Functions on the Euler Operators

As previously stated, a great deal of the attraction of the Euler operators is that they provide a flexible base for higher level operators while insulating those new operators from the details and complexities of the actual data structures utilized.

They are flexible, because they are fairly low level operators which systematically manipulate the model of the embedded graph on an edge by edge basis, providing automatic topological integrity checking. Almost any other kind of commonly found modeling operator or procedure can be built on top of the Euler operators, including parametric primitives, sweeps, and chamfers.

Boolean operators may also be implemented using the Euler operators. Many implementations of the Boolean set operations in Euler operation based systems, however, opt not to use the Euler operators in some circumstances in favor of direct data structure manipulation. This is done mostly to delay integrity checking until the end of the set operation and removes algorithmic restrictions caused by topological integrity requirements enforced by the Euler operators. It is sometimes claimed that delays in integrity maintenance may also improve efficiency.

For detailed examples of how the Euler operators can be used to build some of these higher level operators, see [Eastman & Weiler 79] and [Braid 79].

SECTION III

NON-MANIFOLD REPRESENTATIONS

Chapter 14

INTRODUCTION

Non-manifold is a geometric modeling term referring to topological situations which are not restricted to be two-manifold. *Non-manifold representations* are defined here as geometric modeling representations which allow volume, both manifold and non-manifold surface, curve, and point elements in a single uniform environment. This allows topological surfaces which are not constrained to be homeomorphic to a two-dimensional topological disk at every point (such as when a cone touches upon another surface at a single point, when more than two faces meet along a common edge, or when a wire edge begins at a point on a surface (see Figure 3 – 3). A non-manifold representation therefore allows a general wire mesh with surfaces and volumes embedded in space and can be a functional superset of wireframe, surface, and traditional manifold solid modeling forms (see Figure 3 – 2).

Non-manifold conditions naturally arise as the result of closed form Boolean set operations, even when input is restricted to be manifold. Representation of interior features of models also require a non-manifold domain. Of special interest, non-manifold representations can allow a uniform representation of any combination of wireframe, surface, and solid modeling forms.

Little work has been been done in the area of non-manifold geometric boundary modeling, and non-manifold boundary representations which explicitly store topological adjacency information is an entirely new area of research. While the occurrence of non-manifold results from Boolean operations with manifold inputs has been noted, and the existence of non-manifold equivalents to the Euler operators conjectured [Requicha & Voelcker 83], the topic has not previously been addressed by geometric

modeling technology.

This major section describes a non-manifold domain useful for geometric modeling, a data structure, called the Radial Edge structure, for an object based evaluated non-manifold boundary topology representation along with proof of its completeness, and general low-level operators, call the non-manifold topology operators, to manipulate non-manifold topologies.

14.1. Application Areas for Expanded Modeling Capabilities

Several geometric modeling application areas can be supported by non-manifold representations in ways different from existing manifold solid representations.

1. *Modeling* - The new uniform non-manifold representation allows wireframe, surface, solid, and non-manifold modeling techniques to be utilized simultaneously in the same modeling system using the same representation. This allows a smooth transition in modeling applications from wireframe to surface to solid including the automatic detection of solid enclosures without any need for restructuring or translation. Non-manifold boundary representations also allow storage of arbitrary geometric information, such as center line axes and cutting planes, along with the shape description directly in a single model. Composite objects consisting of several distinct materials, such as that used in aircraft and other applications, can be modeled with adjacencies explicitly available in the model without extensive derivation. This flexibility can reduce overall implementation and maintenance costs, and allows development of a uniform user interface to serve all common aspects of the modeling system. It also provides more flexibility in the implementation and marketing of a geometric modeling system. Closed form implementations of the Boolean operators are possible.

2. *Analysis* - FEM (Finite Element Method) meshing can be performed on the

same representation as the original modeling representation and, using the modeling representation as the communication medium, results can be passed directly back to the modeler for modification, bypassing the traditional manual process of updating models based on FEM analysis results. This may lead to integrated tools which automatically perform certain kinds of modification of the original model based on analysis results directly available from the model representation, and eventually, it could lead to tools which model and analyze simultaneously, optimizing the design as modeling proceeds. Non-manifold results of Boolean operations are allowable for the representation and analysis of points, curves, and areas of overlap as well as volumes of overlap.

3. *Composite Objects* - The extended domain of the new representation will support the representation of interior structures directly. Areas of common boundary and volume are represented explicitly, allowing specification and analysis of such relationships during the design phase, removing the need to re-derive these relationships during analysis.

4. *VLSI (Very Large Scale Integration)* - Non-manifold representations can support advanced integrated circuit fabrication through easy calculation of material area and volume adjacencies, allowing for analysis of electrical properties. It can support current two and one-half dimensional and future three-dimensional chip building capabilities.

14.2. Organization of This Section

This section is organized into the following five chapters concerning non-manifold topology representations.

First, the domain of interest is described in Chapter 15.

Chapter 16 describes the non-manifold adjacency relationships.

Chapter 17 describes the Radial Edge data structure for non-manifold topology representation.

Next, Chapter 18 briefly outlines information related to the theoretical sufficiency of the non-manifold adjacency relationships, and discusses completeness of the Radial Edge structure.

Last, Chapter 19 describes general operators for manipulating non-manifold topologies.

Chapter 15

DOMAIN

This chapter describes the non-manifold domain addressed in this major section. The domain conditions will provide the context which will be assumed in the rest of this major section on non-manifold geometric modeling representations, unless explicitly noted otherwise.

15.1. Specification of Domain

The non-manifold representations addressed here are assumed to be boundary based object based evaluated forms of geometric modeling representations, where topological adjacency information is used as a framework for the entire representation. A series of further specifications on the geometric and topological domain for a non-manifold representation follows.

1. *Non-manifold Surfaces* - The representation is a non-manifold topological representation which allows the uniform representation of wireframe, surface, and solid modeling representations, allows Boolean operations in a closed form, and provides an extended domain which includes representation of the interior features of objects.

   The representation contains topological information in a graph structure embedded in three-dimensional Euclidean space. This embedded topological boundary graph structure provides a framework for the remaining geometric model information. The entire non-manifold structure is finite

in extent. Any surfaces in it are orientable in the sense that the identity of the volume on each side of the surface is known.

The representation must provide the ability to represent arbitrary three-dimensional meshes embedded in space. A cycle in the mesh may or may not have a surface piece (a *face)* associated with it. A group of adjacent faces may entirely enclose a volume of space; in this case the closed volume is given a unique identity and the elements adjacent to it are known. Thus any combination of wireframe, surface, and solid modeling techniques is permissible within the constraint that all element intersection information (intersection of surfaces, edges) is explicitly represented in the embedded graph structure.

2. *Manifold Faces* - A *face* is defined as a connected and bounded portion of a surface, but does not include its boundary. While an entire surface may be non-manifold, the individual faces of an object are required to be manifold. This means that no face is allowed which self-intersects (except at its boundary). This forces the topology to carry all surface (as well as edge) intersection information. Thus the non-manifold characteristics of the representation occur only at the boundaries of individual faces which are otherwise manifold. A single non-manifold surface face may therefore be represented by ensuring a boundary occurs along all non-manifold points and curves.

3. *Faces Mappable to a Plane* - Every individual face is required to be mappable to a plane without cutting or creating new boundaries in the face. This forces the topological framework to carry all genus information. Note this is a further restriction not implied by the previous specification. This restriction is the same as saying that faces may not contain handles, noting that faces do not include their boundaries.

4. *Non-intersection Properties* - Regions may not intersect with each other except along their boundaries. Faces may not intersect each other except along their

boundaries. Edges in the embedded graph structure may not intersect except at their endpoints. Vertices must be distinct in three space. Further, as a corollary to the face non-intersection property, edges may not intersect faces except along or at their boundaries. Thus topological elements of two given types may only intersect each other at a level of hierarchy (top down the levels are regions, shells, faces, edges, vertices) at least one level lower than the lowest of the two levels. This restriction is necessary to prevent topological elements from penetrating faces and volumes without knowledge of the embedded graph representation structure.

5. *Finiteness* - Vertices are at finite positions in space, edges are finite in length, faces are finite in surface area, and enclosed regions are finite in volume. This includes the semianalytic requirement discussed by Requicha [Requicha 80a], where surfaces must not have infinitely varying oscillations. The shapes allowable must be representable with a finite number of topological elements.

6. *Pseudographs* - Generalized graphs, pseudographs, are allowed. This means that self loops and multigraphs are allowed. This allows curved edges without constraint on the geometry (other than the embedded graph constraint that edges must not intersect except at endpoints).

7. *Disconnected Graphs* - Disconnected graphs are allowed. This allows multiply connected faces without the necessity for "artifact edge" bridges between multiple contour boundaries belonging to the same face. It also allows direct representation of multiple shelled objects, such as an object with one or more voids in it.

8. *Labeled Graphs* - All graph elements are labeled. Since all labeled elements are unique as a result, this allows non-geometric information to be associated with them. This has implications on the minimum number of element adjacency relationships required in the topological representation, since the

label of every element must be mentioned in the combination of all adjacency relationships in the topological representation.

Chapter 16

ADJACENCY RELATIONSHIPS

This chapter describes the specific topological adjacency relationships found in the non-manifold domain specified in the previous chapter. The basic concepts behind the topological adjacency relationships have been described in Chapter 6.

16.1. The Non-Manifold Topological Elements

Since topological element adjacency relationships concern the relationships between individual topological elements, we must now define the elements more carefully before describing the adjacency relationships themselves.

At least seven distinct element types, including six basic topological element types are involved in a non-manifold evaluated object based boundary topology representation. They can be seen as being related in a hierarchical fashion, where lower dimensional elements are used as boundaries of higher dimensional elements.

The portions of the descriptions which differ from their manifold counterparts are italicized.

A *model* is a single three-dimensional topological modeling space, consisting of one or more distinct (though perhaps adjacent) regions of space. A model is not strictly a topological element as such, but acts as a repository for all topological elements contained in a geometric model, allowing the naming and manipulation of multiple models by a geometric modeling system.

A *region* is a volume of space. There is always at least one in a model. Only one

region in a model may have infinite extent; all others have a finite extent, and when more than one region exists in a model, all regions have a boundary. For example, a single solid would require two regions in the model, one for the inside of the object, and one for the outside (which has an infinite extent).

A *shell* is an oriented boundary surface of a region. A single region may have more than one shell, as in the case of a solid object with a void contained within it. A region may have no shell only where all space exists as a single region, as in the initial state where no modeling has been done, or after all components of a model have been deleted. A shell may consist of a connected set of faces which form a closed volume *or may be an open set of adjacent faces, a wireframe, or a combination of these, or even a single point.*

A *face* is a bounded portion of a shell. *It is orientable, though not oriented, as two region boundaries (shells) may use different sides of the same face. Thus only the use of a face by a shell is oriented.* Strictly speaking, a face consists of the piece of surface it covers, but does not include its boundaries.

A *loop* is a connected boundary of a single face. A face may have one or more loops, for example a polygon would require one loop and a face with a hole in it would require two loops. Loops normally consist of an alternating sequence of edges and vertices in a complete circuit, but may consist of only a single vertex. *Loops are also orientable but not oriented, as they bound a face which may be used by up to two different shells. Thus, it is the use of a loop that is oriented.*

An *edge* is a portion of a loop boundary between two vertices. Topologically, an edge is a bounding curve segment which may serve as part of a loop boundary for *one or more faces* which meet at that edge. Every edge is bounded by a vertex at each end (possibly the same one). An edge is orientable, though not oriented; it is the *use* of an edge which is oriented.

A *vertex* is simply a topologically unique point in space, that is, no two vertices may exist at the same geometric location (although the topology alone does not specify any

exact geometric location beyond these constraints). Single vertices may also serve as boundaries of faces *and as complete shell boundaries.*

Although not directly represented in the adjacency relationships as described here, at least *four* additional types of topological element adjacency *uses* associated with the *face, loop,* edge, and vertex elements may also be defined. In some representations they may be directly represented.

A face-use is one of the two uses (sides) of a face. Face-uses, the use of a face by a shell, are oriented with respect to the face geometry.

A loop-use is one of the uses of a loop associated with one of the two uses of a face. It is oriented with respect to the associated face-use.

An *edge-use* is an oriented bounding curve segment *on a loop-use of a face-use and represents the use of an edge by that loop-use, or if a wireframe edge, by endpoint vertices.* Orientation is specified with respect to edge geometry. There may be *many* uses of a single edge in a model, *but there will always be an even number of edge-uses (since each use by a face produces two edge-uses, one for each face side). A wireframe edge produces two edge-uses, one for each end of the edge.*

A *vertex-use* is a structure representing the adjacency use of a vertex by an edge as an endpoint, by a loop in the case of a single vertex loop, *or by a shell in the case of a single vertex shell.*

16.2. Adjacency Relationships in a Non-Manifold Model

The topological information stored in a non-manifold boundary representation consists of the existence and adjacencies of the six basic topological elements. Queries and traversals of the topological representation are related to accessing this adjacency information.

An *adjacency relationship* is the adjacency (in terms of physical proximity and order) of

a group of topological elements of one type (vertices, edges, loops, faces, shells, or regions) around some other specific single topological element.

Thirty-six topological element adjacency relationships are possible in a non-manifold boundary representation, as outlined in Figure 16 – 1.

Information related to specific adjacency relationships might be stored directly in a representation, but need not be; as long as information involving a sufficient set of adjacency relationships is available, information about all other adjacency relationships is derivable.

An expanded example showing actual values of the non-manifold adjacency relationships for an object which is a non-manifold superset of the example in Figure 10 – 3 is given in Figure 16 – 2. The figure shows a tetrahedron with a wire emanating from one vertex, a lamina face sharing one edge, and an additional single vertex

$V\{V\}$	$V\{E\}$	$V\{L\}$	$V\{F\}$	$V\{S\}$	$V\{R\}$
$E\{V\}^2$	$E\{<[E]^2>\}$	$E\{<L>\}$	$E\{<F>\}$	$E\{<S>\}$	$E\{<R>\}$
$L\{<V>\}^2$	$L\{<E>\}^2$	$L\{<<L>>\}^2$	$L\{F\}^1$	$L\{S\}^2$	$L\{R\}^2$
$F\{\{<V>\}\}^2$	$F\{\{<E>\}\}^2$	$F\{L\}$	$F\{\{<<F>>\}\}^2$	$F\{S\}^2$	$F\{R\}^2$
$S\{V\}$	$S\{E\}$	$S\{L\}$	$S\{F\}$	$S\{S\}$	$S\{R\}^1$
$R\{V\}$	$R\{E\}$	$R\{L\}$	$R\{F\}$	$R\{S\}$	$R\{R\}$

Figure 16 – 1. Adjacency Relationship Matrix of the Non-manifold Topological Elements

shell. Region r_2 is inside of the tetrahedron volume, which has shell s_2 ; everything outside the interior of the tetrahedron is adjacent to region r_1. The diagram in the figure consists of a pictorial view of the non-manifold object, followed by a planar graph representation of the object in three parts, part *a* showing the tetrahedron, part *b* showing the lamina face with its single vertex loop, and part *c* separately showing the wire edge and single vertex shell. These are followed by the adjacency relationships. For brevity, those adjacency relationships which have two orientations inducing identical adjacent groups in opposite order are only shown in one orientation. This includes $E\{<[E]>\}^2$, $E\{<L>\}^2$, $E\{<F>\}^2$, $E\{<S>\}^2$, and $E\{<R>\}^2$, where each end of the edge induces the opposite orientation implied by the outer unordered group brackets, and $F\{\{<V>\}\}^2$, $F\{\{<E>\}\}^2$, $F\{\{<<F>>\}\}^2$, $L\{<V>\}^2$, $L\{<E>\}^2$, and $L\{<<L>>\}^2$, where the two sides of a face induce the opposite orientation implied by the outer unordered group brackets.

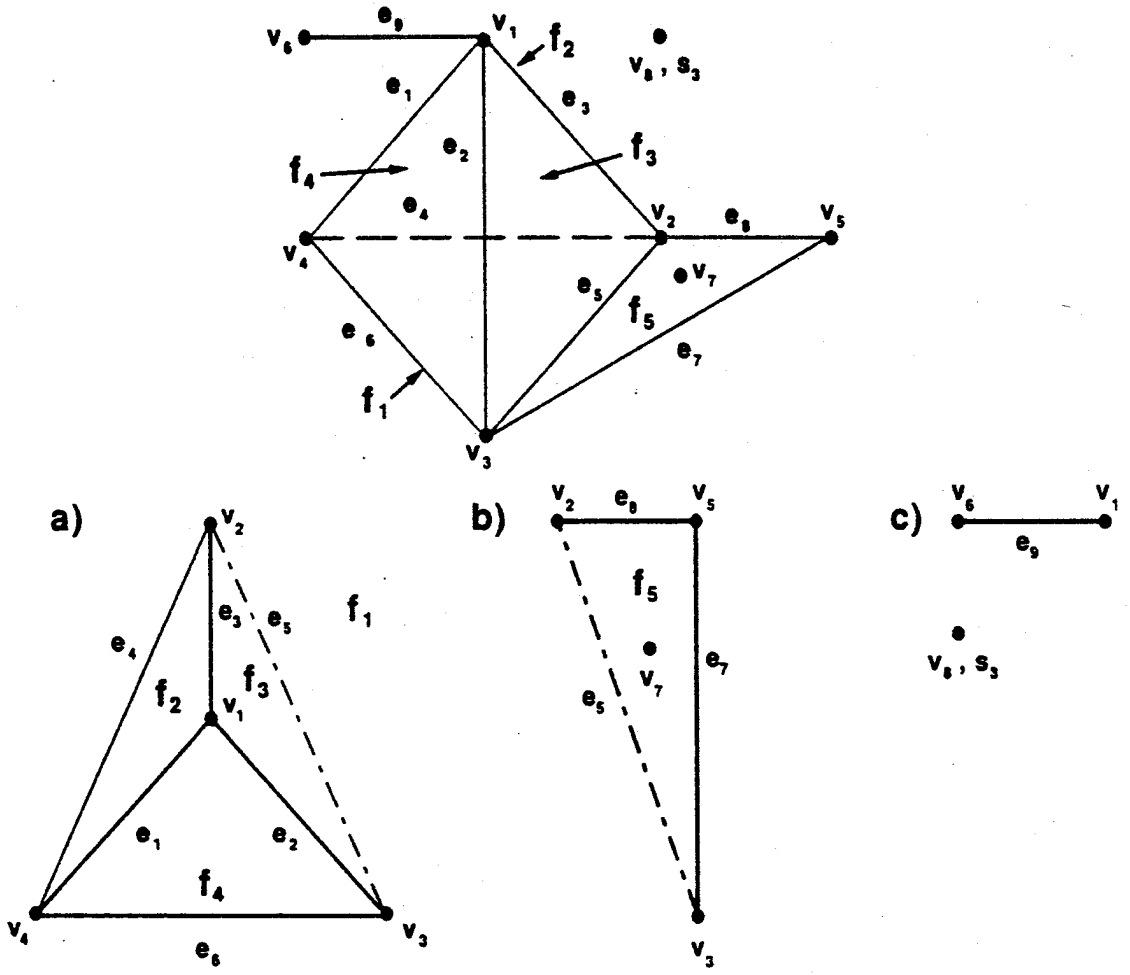


Figure 16 – 2a. Actual adjacency relationships for a non-manifold object

$V\{V\}$	$V\{E\}$	$V\{L\}$
$v_1\{V\} = v_1\{v_2\ v_3\ v_4\ v_6\}$	$v_1\{E\} = v_1\{e_1\ e_3\ e_2\ e_9\}$	$v_1\{L\} = v_1\{l_2\ l_3\ l_4\}$
$v_2\{V\} = v_2\{v_3\ v_1\ v_4\ v_5\}$	$v_2\{E\} = v_2\{e_5\ e_3\ e_4\ e_8\}$	$v_2\{L\} = v_2\{l_1\ l_3\ l_2\ l_5\}$
$v_3\{V\} = v_3\{v_1\ v_2\ v_4\ v_5\}$	$v_3\{E\} = v_3\{e_6\ e_2\ e_5\ e_7\}$	$v_3\{L\} = v_3\{l_1\ l_4\ l_3\ l_5\}$
$v_4\{V\} = v_4\{v_2\ v_1\ v_3\}$	$v_4\{E\} = v_4\{e_4\ e_1\ e_6\}$	$v_4\{L\} = v_4\{l_1\ l_2\ l_4\}$
$v_5\{V\} = v_5\{v_2\ v_3\}$	$v_5\{E\} = v_5\{e_8\ e_7\}$	$v_5\{L\} = v_5\{l_5\}$
$v_6\{V\} = v_6\{v_1\}$	$v_6\{E\} = v_6\{e_9\}$	$v_6\{L\} = v_6\{\}$
$v_7\{V\} = v_7\{\}$	$v_7\{E\} = v_7\{\}$	$v_7\{L\} = v_7\{l_6\}$
$v_8\{V\} = v_8\{\}$	$v_8\{E\} = v_8\{\}$	$v_8\{L\} = v_8\{\}$

$V\{F\}$	$V\{S\}$	$V\{R\}$
$v_1\{F\} = v_1\{f_2\ f_3\ f_4\}$	$v_1\{S\} = v_1\{s_1\ s_2\}$	$v_1\{R\} = v_1\{r_1\ r_2\}$
$v_2\{F\} = v_2\{f_1\ f_3\ f_2\ f_5\}$	$v_2\{S\} = v_2\{s_1\ s_2\}$	$v_2\{R\} = v_2\{r_1\ r_2\}$
$v_3\{F\} = v_3\{f_1\ f_4\ f_3\ f_5\}$	$v_3\{S\} = v_3\{s_1\ s_2\}$	$v_3\{R\} = v_3\{r_1\ r_2\}$
$v_4\{F\} = v_4\{f_1\ f_2\ f_4\}$	$v_4\{S\} = v_4\{s_1\ s_2\}$	$v_4\{R\} = v_4\{r_1\ r_2\}$
$v_5\{F\} = v_5\{f_5\}$	$v_5\{S\} = v_5\{s_1\}$	$v_5\{R\} = v_5\{r_1\}$
$v_6\{F\} = v_6\{\}$	$v_6\{S\} = v_6\{s_1\}$	$v_6\{R\} = v_6\{r_1\}$
$v_7\{F\} = v_7\{f_5\}$	$v_7\{S\} = v_7\{s_1\}$	$v_7\{R\} = v_7\{r_1\}$
$v_8\{F\} = v_8\{\}$	$v_8\{S\} = v_8\{s_3\}$	$v_8\{R\} = v_8\{r_1\}$

$E\{V\}^2$	$E<[E]^2>$
$e_1\{V\} = e_1\{v_1\ v_4\}$	$e_1<[E]> = e_1<[e_4\ e_3][e_6\ e_2]>$
$e_2\{V\} = e_2\{v_1\ v_3\}$	$e_2<[E]> = e_2<[e_5\ e_3][e_6\ e_1]>$
$e_3\{V\} = e_3\{v_1\ v_2\}$	$e_3<[E]> = e_3<[e_4\ e_1][e_5\ e_2]>$
$e_4\{V\} = e_4\{v_2\ v_4\}$	$e_4<[E]> = e_4<[e_5\ e_6][e_3\ e_1]>$
$e_5\{V\} = e_5\{v_2\ v_3\}$	$e_5<[E]> = e_5<[e_6\ e_4][e_7\ e_8][e_2\ e_3]>$
$e_6\{V\} = e_6\{v_3\ v_4\}$	$e_6<[E]> = e_6<[e_4\ e_5][e_1\ e_2]>$
$e_7\{V\} = e_7\{v_3\ v_5\}$	$e_7<[E]> = e_7<[e_5\ e_8]>$
$e_8\{V\} = e_8\{v_2\ v_5\}$	$e_8<[E]> = e_8<[e_7\ e_5]>$
$e_9\{V\} = e_9\{v_1\ v_6\}$	$e_9<[E]> = e_9<>$

Figure 16-2b. Actual adjacency relationships for a non-manifold object

$E<L>$	$E<F>$
$e_1<L> = e_1<l_2\ l_4>$	$e_1<F> = e_1<f_2\ f_4>$
$e_2<L> = e_2<l_3\ l_4>$	$e_2<F> = e_2<f_3\ f_4>$
$e_3<L> = e_3<l_2\ l_3>$	$e_3<F> = e_3<f_2\ f_3>$
$e_4<L> = e_4<l_1\ l_2>$	$e_4<F> = e_4<f_1\ f_2>$
$e_5<L> = e_5<l_1\ l_5\ l_3>$	$e_5<F> = e_5<f_1\ f_5\ f_3>$
$e_6<L> = e_6<l_1\ l_4>$	$e_6<F> = e_6<f_1\ f_4>$
$e_7<L> = e_7<l_5>$	$e_7<F> = e_7<f_5>$
$e_8<L> = e_8<l_5>$	$e_8<F> = e_8<f_5>$
$e_9<L> = e_9<>$	$e_9<F> = e_9<>$

$E<S>$	$E<R>$
$e_1<S> = e_1<s_1\ s_2>$	$e_1<R> = e_1<r_1\ r_2>$
$e_2<S> = e_2<s_1\ s_2>$	$e_2<R> = e_2<r_1\ r_2>$
$e_3<S> = e_3<s_1\ s_2>$	$e_3<R> = e_3<r_1\ r_2>$
$e_4<S> = e_4<s_1\ s_2>$	$e_4<R> = e_4<r_1\ r_2>$
$e_5<S> = e_5<s_1\ s_1\ s_2>$	$e_5<R> = e_5<r_1\ r_1\ r_2>$
$e_6<S> = e_6<s_1\ s_2>$	$e_6<R> = e_6<r_1\ r_2>$
$e_7<S> = e_7<s_1>$	$e_7<R> = e_7<r_1>$
$e_8<S> = e_8<s_1>$	$e_8<R> = e_8<r_1>$
$e_9<S> = e_9<s_1>$	$e_9<R> = e_9<r_1>$

$L<V>$	$L<E>$
$l_1<V> = l_1<v_2\ v_4\ v_3>$	$l_1<E> = l_1<e_4\ e_6\ e_5>$
$l_2<V> = l_2<v_4\ v_2\ v_1>$	$l_2<E> = l_2<e_4\ e_3\ e_1>$
$l_3<V> = l_3<v_1\ v_2\ v_3>$	$l_3<E> = l_2<e_3\ e_5\ e_2>$
$l_4<V> = l_4<v_1\ v_3\ v_4>$	$l_4<E> = l_2<e_2\ e_6\ e_1>$
$l_5<V> = l_5<v_2\ v_5\ v_3>$	$l_5<E> = l_2<e_8\ e_7\ e_5>$
$l_6<V> = l_6<v_7>$	$l_6<E> = l_6<>$

$L<<L>>$	$L\{F\}^1$
$l_1<<L>> = l_1<<l_1\ l_5\ l_3> <l_1\ l_2> <l_1\ l_4>>$	$l_1\{F\} = l_1\{f_1\}$
$l_2<<L>> = l_2<<l_2\ l_1> <l_2\ l_3> <l_2\ l_4>>$	$l_2\{F\} = l_2\{f_2\}$
$l_3<<L>> = l_3<<l_3\ l_2> <l_3\ l_5\ l_1> <l_3\ l_4>>$	$l_3\{F\} = l_3\{f_3\}$
$l_4<<L>> = l_4<<l_4\ l_2> <l_4\ l_3> <l_4\ l_1>>$	$l_4\{F\} = l_4\{f_4\}$
$l_5<<L>> = l_5<<l_5\ l_1\ l_3> <l_5> <l_5>>$	$l_5\{F\} = l_5\{f_5\}$
$l_6<<L>> = l_6<<>>$	$l_6\{F\} = l_6\{f_5\}$

Figure 16-2c. Actual adjacency relationships for a non-manifold object

$L\{S\}^2$	$L\{R\}^2$
$l_1\{S\} = l_1\{s_1 s_2\}$	$l_1\{R\} = l_1\{r_1 r_2\}$
$l_2\{S\} = l_2\{s_1 s_2\}$	$l_2\{R\} = l_2\{r_1 r_2\}$
$l_3\{S\} = l_3\{s_1 s_2\}$	$l_3\{R\} = l_3\{r_1 r_2\}$
$l_4\{S\} = l_4\{s_1 s_2\}$	$l_4\{R\} = l_4\{r_1 r_2\}$
$l_5\{S\} = l_5\{s_1 s_1\}$	$l_5\{R\} = l_5\{r_1 r_1\}$
$l_6\{S\} = l_6\{s_1 s_1\}$	$l_6\{R\} = l_6\{r_1 r_1\}$

$F\{< V >\}$	$F\{< E >\}$
$f_1\{< V >\} = f_1\{< v_2 v_4 v_3 >\}$	$f_1\{< E >\} = f_1\{< e_4 e_6 e_5 >\}$
$f_2\{< V >\} = f_2\{< v_4 v_2 v_1 >\}$	$f_2\{< E >\} = f_2\{< e_4 e_3 e_1 >\}$
$f_3\{< V >\} = f_3\{< v_1 v_2 v_3 >\}$	$f_3\{< E >\} = f_3\{< e_3 e_5 e_2 >\}$
$f_4\{< V >\} = f_4\{< v_1 v_3 v_4 >\}$	$f_4\{< E >\} = f_4\{< e_2 e_6 e_1 >\}$
$f_5\{< V >\} = f_5\{< v_2 v_5 v_3 > < v_7 >\}$	$f_5\{< E >\} = f_5\{< e_8 e_7 e_5 > < >\}$

$F\{L\}$	$F<<F>>$
$f_1\{L\} = f_1\{l_1\}$	$f_1<<F>> = f_1<<f_1 f_5 f_3> <f_1 f_2> <f_1 f_4>>$
$f_2\{L\} = f_2\{l_2\}$	$f_2<<F>> = f_2<<f_2 f_1> <f_2 f_3> <f_2 f_4>>$
$f_3\{L\} = f_3\{l_3\}$	$f_3<<F>> = f_3<<f_3 f_2> <f_3 f_5 f_1> <f_3 f_4>>$
$f_4\{L\} = f_4\{l_4\}$	$f_4<<F>> = f_4<<f_4 f_2> <f_4 f_3> <f_4 f_1>>$
$f_5\{L\} = f_5\{l_5 l_6\}$	$f_5<<F>> = f_5<<f_5 f_1 f_3> <f_5> <f_5>>$

$F\{S\}^2$	$F\{R\}^2$
$f_1\{S\} = f_1\{s_1 s_2\}$	$f_1\{R\} = f_1\{r_1 r_2\}$
$f_2\{S\} = f_2\{s_1 s_2\}$	$f_2\{R\} = f_2\{r_1 r_2\}$
$f_3\{S\} = f_3\{s_1 s_2\}$	$f_3\{R\} = f_3\{r_1 r_2\}$
$f_4\{S\} = f_4\{s_1 s_2\}$	$f_4\{R\} = f_4\{r_1 r_2\}$
$f_5\{S\} = f_5\{s_1 s_1\}$	$f_5\{R\} = f_5\{r_1 r_1\}$

Figure 16-2d. Actual adjacency relationships for a non-manifold object

$S\{V\}$	$S\{E\}$
$s_1\{V\} = s_1\{v_1\ v_2\ v_3\ v_4\ v_5\ v_6\ v_7\}$	$s_1\{E\} = s_1\{e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9\}$
$s_2\{V\} = s_2\{v_1\ v_2\ v_3\ v_4\}$	$s_2\{E\} = s_2\{e_1 e_2 e_3\ e_4 e_5 e_6\}$
$s_3\{V\} = s_3\{v_8\}$	$s_3\{E\} = s_3\{\}$
$S\{L\}$	$S\{F\}$
$s_1\{L\} = s_1\{l_1 l_2 l_3 l_4 l_5\}$	$s_1\{F\} = s_1\{f_1 f_2 f_3 f_4 f_5\}$
$s_2\{L\} = s_2\{l_1 l_2 l_3 l_4\}$	$s_2\{F\} = s_2\{f_1 f_2 f_3 f_4\}$
$s_3\{L\} = s_3\{\}$	$s_3\{F\} = s_3\{\}$
$S\{S\}$	$S\{R\}^1$
$s_1\{S\} = s_1\{s_2\}$	$s_1\{R\} = s_1\{r_1\}$
$s_2\{S\} = s_2\{s_1\}$	$s_2\{R\} = s_2\{r_2\}$
$s_3\{S\} = s_3\{\}$	$s_3\{R\} = s_3\{r_1\}$
$R\{V\}$	$R\{E\}$
$r_1\{V\} = r_1\{v_1\ v_2\ v_3\ v_4\ v_5\ v_6\ v_7\ v_8\}$	$r_1\{E\} = r_1\{e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9\}$
$r_2\{V\} = r_2\{v_1\ v_2\ v_3\ v_4\}$	$r_2\{E\} = r_2\{e_1 e_2 e_3\ e_4 e_5 e_6\}$
$R\{L\}$	$R\{F\}$
$r_1\{L\} = r_1\{l_1 l_2 l_3 l_4 l_5\}$	$r_1\{F\} = r_1\{f_1 f_2 f_3 f_4 f_5\}$
$r_2\{L\} = r_2\{l_1 l_2 l_3 l_4\}$	$r_2\{F\} = r_2\{f_1 f_2 f_3 f_4\}$
$R\{S\}$	$R\{R\}$
$r_1\{S\} = r_1\{s_1 s_3\}$	$r_1\{R\} = r_1\{r_2\}$
$r_2\{S\} = r_2\{s_2\}$	$r_2\{R\} = r_2\{r_1\}$

Figure 16-2e. Actual adjacency relationships for a non-manifold object

16.2.1. Adjacency Relationship Semantics

The non-manifold domain is a more complex one than the manifold domain, and the semantics of the adjacency relationships reflect some of the complexity.

As in the manifold domain, there are multiple interpretations for the semantics of some of the adjacency relationships. The definitions utilized in this thesis are described here.

The adjacency relationships where the type of the reference element and the adjacent group are the same are particularly prone to multiple interpretations. The $V\{V\}$ adjacency relationship is defined here as the set of all vertices which are adjacent to the reference vertex by being at the other end of the edges specified by the $V\{E\}$ adjacency relationship. The $E\{< [E]^2 >\}^2$ adjacency relationship is defined as the set of clockwise and counterclockwise edges to the reference edge for each loop found radially around the edge. The $L\{< < L > >\}^2$ adjacency relationship is defined as the cyclic ordered list of radially adjacent loops sharing an edge with the reference loop for each edge in the cyclic list of edges in the reference loop; the outermost brackets are for the two orientations the information can take based on which side of the face one views the relationships from. The $F\{\{< < F > >\}\}^2$ adjacency relationship is defined as the cyclic ordered list of radially adjacent faces sharing an edge with the reference face for each edge in the cyclic list of edges in each loop of the reference face; the outermost brackets are for the two orientations the information can take based on which side of the face one views the relationships from. The $S\{S\}$ adjacency relationship is defined as the set of all shells which share a face with the reference shell. The $R\{R\}$ adjacency relationship is defined as the set of all regions which share a face with the reference region.

The adjacency relationships where the edge is the reference element also are open to multiple interpretations. The $E\{< L >\}^2$ adjacency relationship is defined as the set of loops using the edge, with each use listed radially around the reference edge, with the same radial ordered group occurring twice, once in opposite order, once from the per-

spective of each end of the edge. The $E\{<F>\}^2$ adjacency relationship is defined as the set of faces using the edge on a loop boundary, with each use listed radially around the reference edge, with the same radial ordered group occurring twice, once in opposite order. The $E\{<S>\}^2$ adjacency relationship is defined as the set of shells using the edge on a face boundary, with each use listed radially around the reference edge, with the same radial ordered group occurring twice, once in opposite order. The $E\{<R>\}^2$ adjacency relationship is defined as the set of regions using the edge on a shell boundary, with each use listed radially around the reference edge, with the same radial ordered group occurring twice, once in opposite order. A wireframe edge would therefore have two empty $E\{<F>\}^2$ radial ordered adjacent groups but would have one member in each of its $E\{<S>\}^2$ and $E\{<R>\}^2$ radial ordered adjacent groups.

The $L\{<V>\}$ and $L\{<E>\}$ adjacency relationships would have the expected meaning of representing the ordered lists of vertices and edges around a loop; the outermost brackets are for the two orientations the information can take based on which side of the face is used to view the relationships.

The $F\{\{<V>\}\}$ and $F\{\{<E>\}\}$ adjacency relationships are similar to the $L\{<V>\}$ and $L\{<E>\}$ adjacency relationships above, except that an additional unordered list bracket pair encloses the innermost group to provide for the multiple loops that may be found in a face.

The remaining adjacency relationships follow their expected definition in terms of their function as downward or upward hierarchical adjacency relationships. A more complete interpretation of the semantics of the adjacency relationships, specifically applied to the Radial Edge structure, can be found in the completeness proof in Chapter 18 and Appendix D.

Correspondence is not discussed here in detail, but there are several characteristics of the non-manifold environment which allow correspondence between adjacency relationships, including the two ends of an edge, the radial ordering of loops around an edge, the two sides of a face, and the loops in a face. These characteristics are

utilized in the correspondence information kept in the Radial Edge structure discussed in Chapters 17 and 18.

Chapter 17

TOPOLOGICAL DATA STRUCTURES

This chapter discusses a specific data structure for the representation of an object based evaluated non-manifold boundary topology representation which explicitly stores adjacency relationship information.

First, design issues for the non-manifold environment outlined in Chapter 15 are discussed. Next, the Radial Edge data structure is described. The detection of volume enclosure, a condition maintained by operators manipulating the structure, is then discussed in relationship to the data structure.

17.1. Design Issues in Non-Manifold Representations

Many issues arise in the design of a representation to support non-manifold environments, some similar to those found with manifold representations, and some unique to non-manifold representations. Specific resolutions to these issues are described in the following subsection discussing the actual data structures.

One perspective on issues in non-manifold modeling can be seen from a comparison of non-manifold with manifold modeling environments.

There are many differences between the manifold and non-manifold environments. The non-manifold environment, being able to model objects unrepresentable in manifold environments, is correspondingly more complex. The non-manifold domain in an intuitive sense has a higher level of representational dimension than that of a manifold domain, simply because a manifold domain restricts itself to surface junctures which are topologically two dimensional, while non-manifold domains support

more complex junctures. There are also some similarities, however. Many of the issues in manifold representation design exist in non-manifold representation design not only at the same dimensional level, but also appear in a similar fashion at a higher level of dimension.

Several representation design issues of concern in a non-manifold environment are now discussed.

17.1.1. Direct Representation of Adjacency Uses

A key simplification principle found with manifold representations that equally applies in a non-manifold environment is that directly representing the *use* of topological representation structures (*uses* of the topological face, loop, edge, and vertex elements) in the adjacency relationship information, rather than the topological elements themselves, simplifies accessing by eliminating the need for procedural decision making during traversals of the topology structures [Weiler 85a]. This is usually done at a cost of at least some increase in storage requirements.

Even simple operations such as traversal of the edges around a loop of a face can not use the edge identity as an indicator of where it is in the loop traversal since the same edge may be used twice in the loop (as in a strut edge). In fact, an edge identity with a vertex identity is also not sufficient since self loops are allowed, and when non-manifold edges occur, even edge identity with orientation information is not sufficient since an edge may be used many times in both directions. When the uses of elements are represented directly, however, these problems disappear since positioning in a list of adjacencies is uniquely defined.

17.1.2. Non-Manifold Conditions Along an Edge

A situation where more than two faces meet along a common edge is a major headache for manifold representation application developers since it can appear as a

result of the standard and the regularized Boolean operations, yet is not directly representable in manifold representations (see Figure 17 – 1).

How this situation is handled is a key issue in the design of a non-manifold representation. Representing non-manifold situations directly tends to simplify manipulation and modeling algorithms and remove special case considerations.

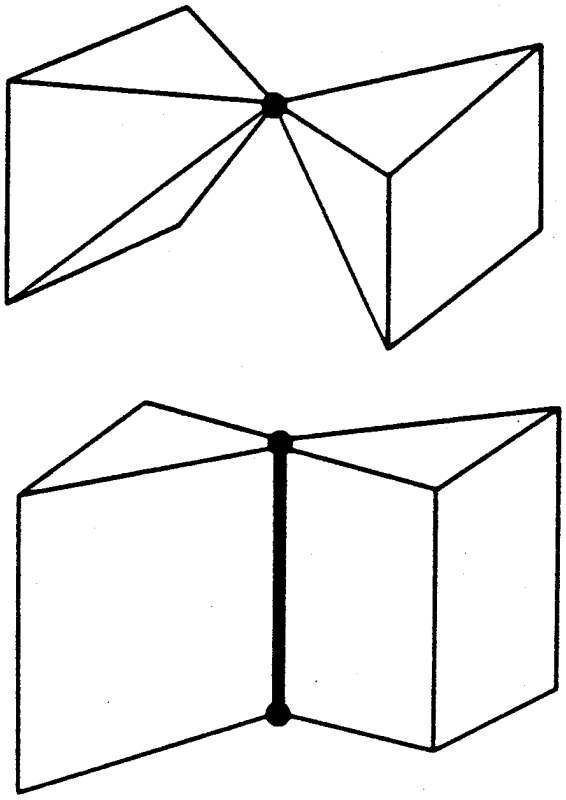


Figure 17 – 1. Non-manifold conditions at a point and along an open curve

17.1.3. Non-Manifold Condition at a Vertex

A similar problem for manifold representations is the situation where several distinct volumes or faces are connected only by a single vertex (see Figure 17 – 1). In some cases this can be represented while still using a manifold representation by decomposing the object by a process of duplicating vertices. This manifold solution has the undesirable characteristic of losing adjacency information (unless additional information is added, essentially creating a partial non-manifold representation as a special case). Again, these non-manifold situations are possible as a result of Boolean operations, even when the input is manifold; correct implementations of these operators for manifold representations therefore must either decompose the output or give up.

A non-manifold representation must preserve such adjacency information so that the information is available locally. Since the vertex is the only common structure between such adjacent structures, the vertex structure is the logical place to store such adjacency information. This also is a logical place to store connectivity information for wireframe edges.

Note that in manifold representations an upward pointer from a vertex to higher dimensional element structure levels was optional; in a non-manifold representation it is logically mandatory. It is therefore necessary to consider whether it is also as necessary to isolate *uses* of a vertex as it was necessary to isolate uses of faces and edges.

Another concept related to non-manifold vertices is *separation surfaces*, detailed later.

17.1.4. Non-Manifold Wireframe Representation

A *wire edge* is defined as a single edge, possibly a self loop or multiply connected edge, which has no adjacent face. Each end of the edge may or may not be attached to other edges. These adjacent edges may or may not also be wire edges. A *wireframe* is a collection of connected wire edges.

With only a little extra care, a non-manifold representation can be designed to be flexible enough to equally accept wireframe, surface, or solid models, or any combination of them at the same time. Non-manifold solid models, fully developed with interior partitions, have many complex adjacency relationships.

Several desirable properties for a wireframe representation should be preserved. The representation structure should implicitly or explicitly keep track of what shell (what boundaries of what volumes) any given wire edge or shell vertex is part of. The adjacencies between edges should be available, sorted by which end of the edge was adjacent. The two vertices at each end of an edge should be available.

17.1.5. Separation Surfaces

Another situation to be considered is a complete surface formed by the juncture of faces around a vertex that effectively separates the space immediately around the vertex into two half-spaces, distinguishable from each other because the surfaces are orientable (see Figure 17 – 2). These surfaces are called *separation surfaces*, and may be composed of one or more faces as long as they together form a continuous manifold or lamina which creates the half-space division at the vertex.

This means an edge attached to a vertex at the center of a separation surface could be intended to fall on one side or the other of the separation surface (see Figure 17 – 2).

At a single vertex there may exist many separation surfaces, which effectively form a tree of separation adjacency relationships between the surfaces. For example, in Figure 17 – 3, the separation surface tree for the illustrated vertex has three branches at the top level, and one of the branches itself has two sub-branches. The resulting separation surface tree is shown in the figure symbolically as well as pictorially. This kind of information must be available in a non-manifold representation; there are many ways it may be represented. A purely topological approach might represent the separation surface adjacency tree directly; a hybrid approach might involve the use of

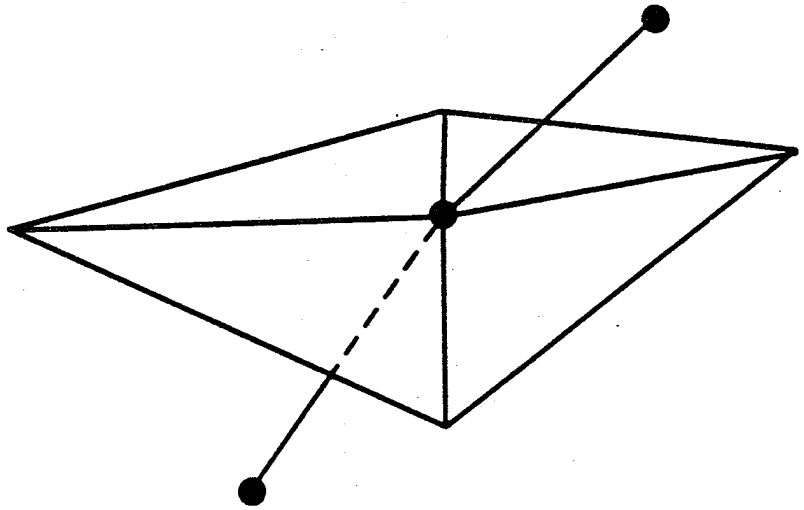


Figure 17 – 2. A separation surface completely surrounds a vertex and divides the space around the vertex into two half spaces.

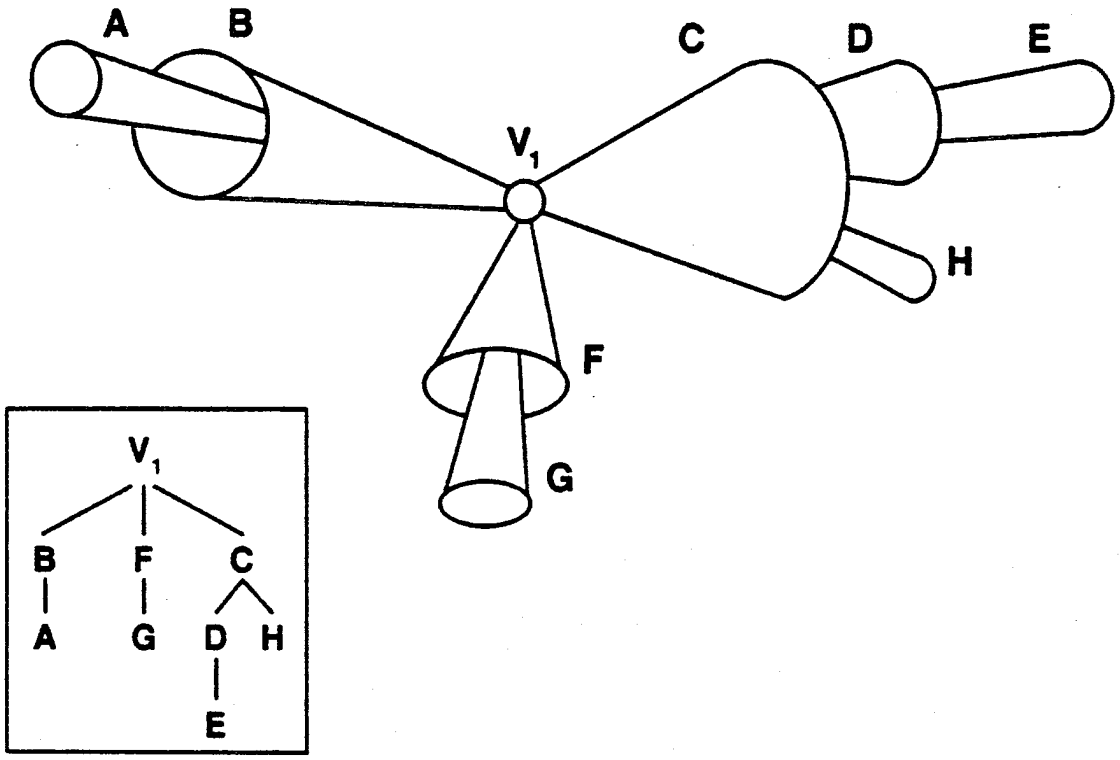


Figure 17 – 3. A nested tree of separation surfaces

both topology and geometry to determine such information.

If separation surface information is supported directly by the representation, topological elements may be inserted into the model with respect to their adjacencies to separation surfaces. This would have the effect of reducing operation ordering dependencies during model creation and manipulation. Keeping such information easily available in a representation is problematic, however, since the most convenient

places to store such information change as the model is manipulated. Furthermore, the separation surface tree information must either be derived using geometry and topology together or must be produced by a somewhat complex interface protocol for any manipulation operators provided.

Notice that a similar problem exists for manifold representations concerning where struts attached to a vertex should lie when a new edge is attached to a vertex. Some systems, simply put limitations on how and when such struts could be made. Other systems handle it by explicit designation of the adjacencies on the manifold (see Chapter 13). No such direct specification exists in three space, however, unless separation surfaces are an explicit part of the representation and the operators provided use such specifications.

17.2. A Description of the Radial Edge Data Structure

17.2.1. Design Decisions

Several decisions were made during design of the Radial Edge non-manifold data structure with respect to the design issues raised in the previous subsection and some of the practical constraints identified in the previous chapters.

1. The top-down hierarchical relationships of the topological elements, from higher dimensional elements to lower dimensional elements, and the bottom-up hierarchical relationships of the topological elements, from lower dimensional elements to higher dimensional elements are directly represented in the data structures, as shown in Figure 17 – 4. The terms "up pointer" and "down pointer" used in later data structure descriptions refer to these relative positions in this hierarchy.

2. The $V\{E\}$ adjacency relationship consisting of the unordered list of edges incident to a vertex is represented in order to capture the adjacencies of separate volumes touching at a single non-manifold point, as well as to capture the edge adjacencies in a wireframe. Since the vertex is the only common structure between such adjacent structures in these situations, the vertex and vertex-use structures are the logical place to store such adjacency information.

3. The $E<L>$ adjacency relationship consisting of the ordered list of loops surrounding an edge is represented. This is required because the same volume may be adjacent to an edge from several directions at once; the radial face ordering around an edge is necessary to allow the

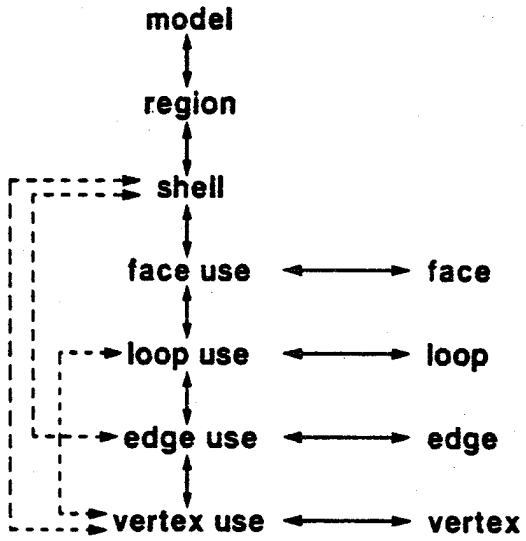


Figure 17 – 4. Radial Edge structure relationships

adjacencies of the volumes at the edge to be correctly represented. This a key feature of the Radial Edge data structure, giving rise to the name "Radial Edge".

4. No separation surface information is represented. It was felt intuitively that such information would be expensive to maintain under the effects of typical modeling operations on such a representation, especially in view of the expected frequency of use of such information in normal modeling situations.

5. The adjacency *uses* of the face, loop, edge, and vertex elements are directly represented. In particular, representing each face with two face-use (face side) structures and each edge with an edge-use structure for each use by each face-use are some of the other key ideas of the Radial Edge structure described here.

6. Wireframe edges are represented by two edge-uses, one for each end of the edge. Connectivity to other edges is maintained through the vertex-use structures.

In fact, it is not necessary to have any direct representation of the basic face, loop,

edge, and vertex elements themselves; representations of their uses are sufficient to indicate their position in the model. It is convenient, however, from a system architecture point of view if programmers using the operators to manipulate the data structures deal with the more intuitive concept of topological elements rather than topological uses of elements as much as possible. Additionally, dealing only with basic elements at the interface level helps insulate higher levels of a geometric modeling system from data structure dependencies. This is one of the few justifications for the representation of the face, loop, edge, and vertex elements directly in this representation. It may also be desirable on the basis of high speed traversal of all faces, edges, and vertices in a model, although the overhead in the Radial Edge structure for these operations is not overwhelming except perhaps in the largest models. Geometry may also be stored in the face, edge, and vertex elements directly, but programming modularity, desire for multiple geometric representations, ease of manipulation, and the desire to represent variable and symbolic geometric dependencies and constraints make implementing a separate geometry representation more desirable.

For parametric surface geometry representations, the edge-use approach of the Radial Edge structure provides a one-to-one correspondence of topological elements to oriented parametric space curve segment geometry elements. This is a particularly useful bookkeeping feature when curve geometries exist only as parametric space curves; otherwise procedural testing is necessary. This is discussed in Chapter 20.

There are several secondary design issues which, while not as fundamental as those discussed previously in this chapter, are nevertheless important. These design issues mostly concern tradeoffs of data structure space for speed and/or simpler manipulation algorithms. Examples include decisions regarding search vs. upward pointers, variable size structures vs. lists of fixed size structures, and doubly vs. singly linked lists. The basic strategy chosen here is to utilize explicit upward pointers to avoid necessity for search, to utilize fixed size structures, and to use doubly linked circular lists. These choices trade space for speed and simplicity of algorithms. Other choices are possible, but optimal choices would involve careful statistical analysis of actual usage patterns. This latter approach might yield overall better space and time

performance, but is not foolproof, since usage patterns can change drastically based on even minor changes in heavily used application code.

The main purpose of the shell and loop structures is to function as variable length list mechanisms which allow disconnected graph conditions within surfaces (the loop), and between surfaces as well as between other boundary structures (the shell). In the case of a shell, the highest dimension of the boundary elements may range from a surface (2D), an edge (1D), or a vertex(0D). In a loop it may range from an edge to a vertex. Note that while the element-use structures also effectively provide a variable length list mechanism, their primary advantage is still in providing unique identification for each element usage to simplify later adjacency queries and traversals.

17.2.2. Data Structures

The data structures of the Radial Edge structure are described in the form of Pascal data structures in Figures 17 – 9 to 17 – 16.

Figure 17 – 5 illustrates some of the adjacency relationships represented in the edge-use structure. Two adjacent faces are shown; the edge they share in this case gives rise to four edge-use structures, one by each of the two sides of each of the two faces. Any given edge-use structure keeps track not only of the *eueu_mate_ptr* edge-use structure found on the opposite side of the face, but also of the *eueu_radial_ptr* edge-use structure on the face-use radially adjacent to the face-use of the given edge-use. In this way, the full radial ordering of faces about the edge can be maintained. Figure 17 – 6 depicts the *eueu_radial_ptr* and *eueu_mate_ptr* edge-use relationships in a cross-sectional view.

Figure 17 – 7 depicts how edge-uses in a loop-use are connected for the representation of the cyclic ordered list of edges around a loop. Figure 17 – 8 shows pointers for a wireframe vertex touched upon by several edges and illustrates how connectivity is maintained through common vertices.

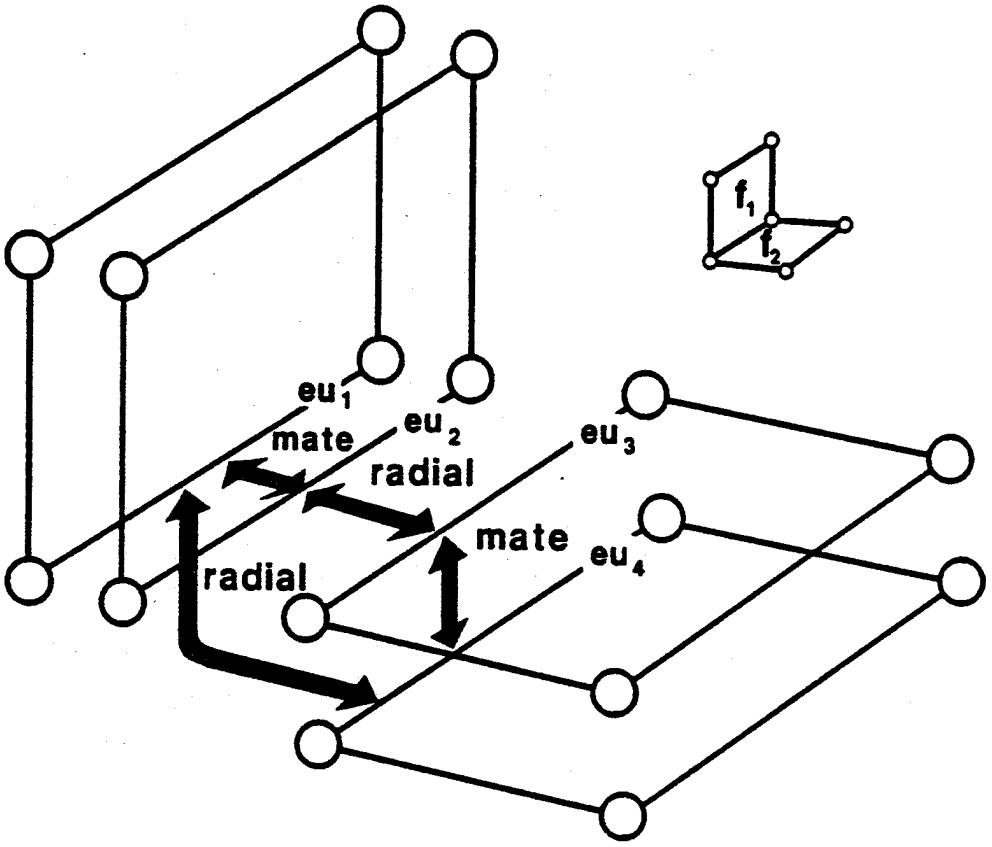


Figure 17 – 5. Radial Edge representation of two faces joining along a common edge showing how the four edge uses of the common edge (each side of each face uses the edge) are connected

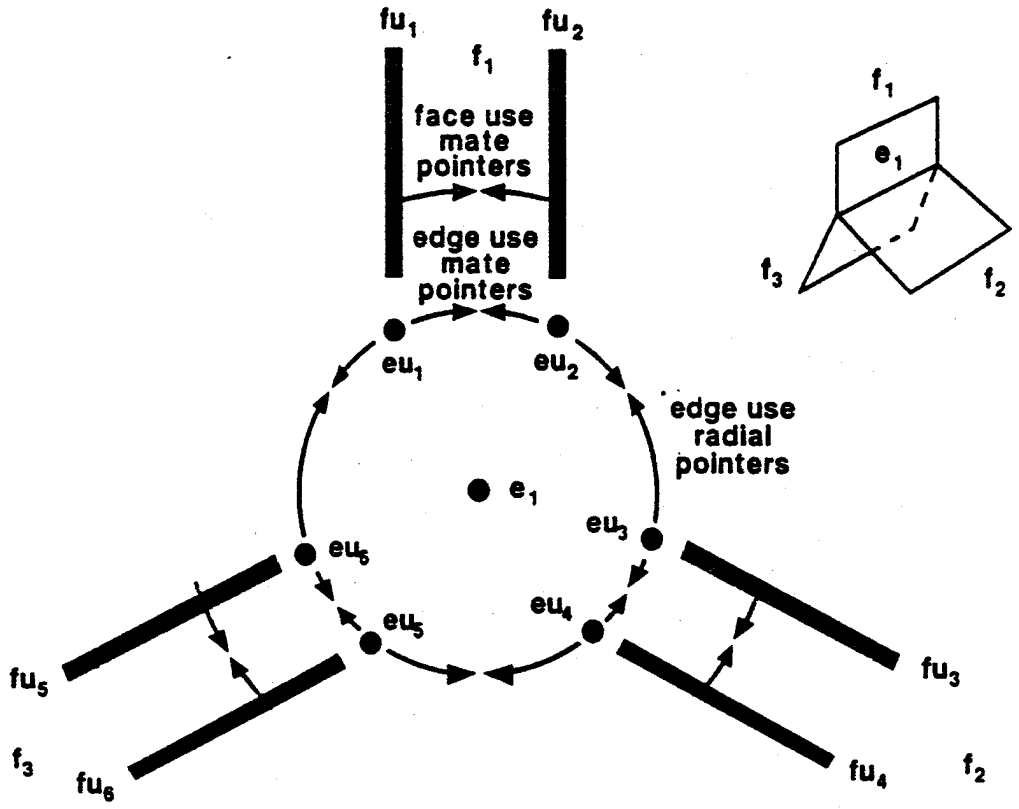


Figure 17 – 6. Cross-section of three faces sharing a common edge in the Radial Edge representation

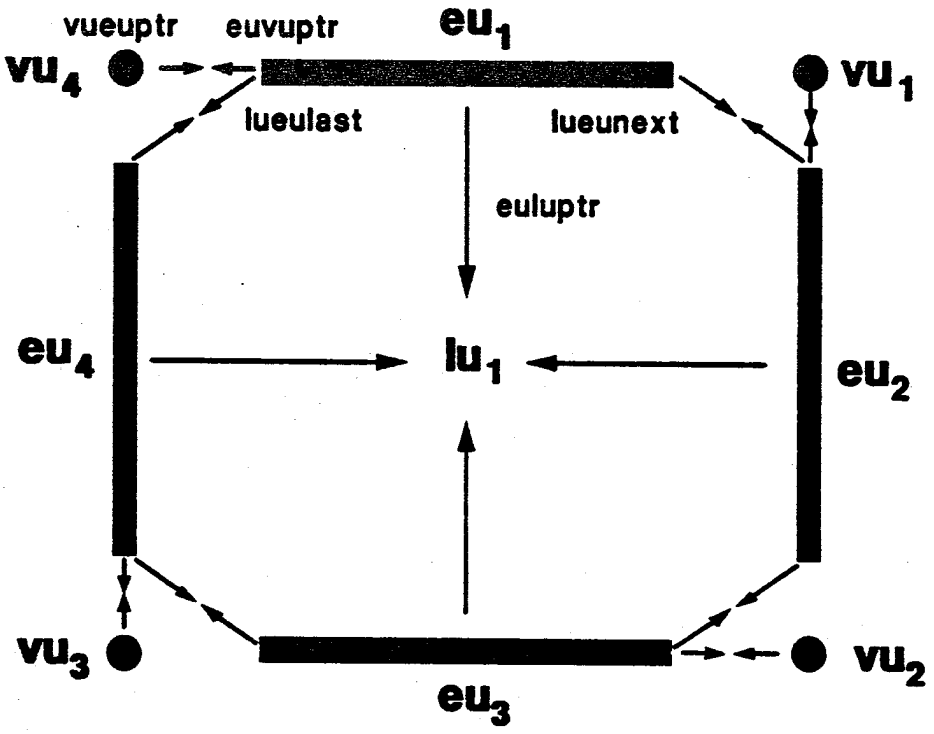


Figure 17 – 7. Plan view of a loop of edges in the Radial Edge structure

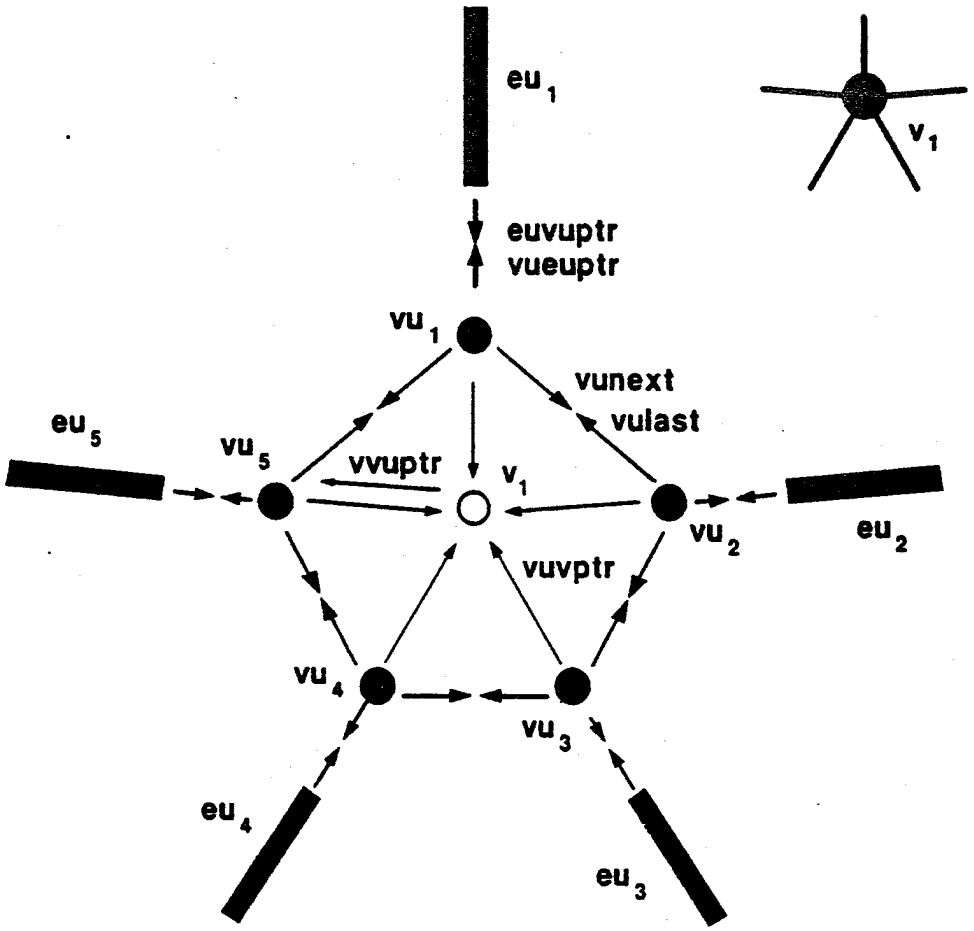


Figure 17 – 8. Radial Edge representation of a vertex and its uses by five incident edges. Vertex-use to edge-use pointers are required for representation of adjacencies of wire edges incident to a vertex.

As in the descriptions of manifold data structures, data objects refer to each other by the use of pointers. Similarly, the naming convention for the pointers in the data structures described is:

from-element-type to-element-type ptr

where the topological element types are symbolized by the letters *r, s, f, l, e, v, fu, lu, eu,* and *vu* for region, shell, face, loop, edge, vertex, face-use, loop-use, edge-use, and vertex-use, respectively. There is sometimes an additional name before the "ptr" suffix when there is more than one pointer of the given type combination. Circular linked lists of lower dimension elements maintained by higher dimension elements often use pointers embedded in the lower dimension elements. The pointers are usually named in the form:

higher-dimension-type lower-dimension-type next

It should be noted that the actual orientations of orientable elements are associated with the usage structures rather than basic element structures since it is the use of an element that forces an orientation. The orientation is meaningful primarily with regard to geometry stored in the geometry attribute of the face and edge basic topological elements. A consistent interpretation of orientation is therefore necessary for face normals and edge directions.

The model structure simply maintains a down pointer to a list regions in the model, and a region structure maintains a list of shells bounding the region. The shell structure maintains a down pointer to the highest dimensional element type which bounds the shell: a list of face-uses, a wire edge-use, or a single vertex-use.

The face, loop, edge, and vertex records are merely convenient places to put attribute information such as geometry, and also assist by providing unique identities for basic topological elements if operators are based on basic elements rather than use elements. They are not topologically necessary in the representation, however.

The face-use structure represents the use of one side of a face by a shell. It maintains a down pointer to an associated list of loop-use structures, as well as a pointer to the face-use of the other side of the face.

```
type

        { topological element structures }
model_ptr = ^model;
region_ptr = ^region;
shell_ptr = ^shell;
face_ptr = ^face;
loop_ptr = ^loop;
edge_ptr = ^edge;
vertex_ptr = ^vertex;
        { topological element adjacency usage structures }
faceuse_ptr = ^faceuse;
loopuse_ptr = ^loopuse;
edgeuse_ptr = ^edgeuse;
vertexuse_ptr = ^vertexuse;

        { pointer type indicator values }
ptr_type = (MODELptr, REGIONptr, SHELLptr, FACEptr, LOOPptr, EDGEptr,
        VERTEXptr, FACEUSEptr, LOOPUSEptr, EDGEUSEptr, VERTEXUSEptr);

        { attribute/geometry structures }
model_attrib_ptr = ^model_attrib;
region_attrib_ptr = ^region_attrib;
shell_attrib_ptr = ^shell_attrib;
face_attrib_ptr = ^face_attrib;
faceuse_attrib_ptr = ^faceuse_attrib;
loop_attrib_ptr = ^loop_attrib;
loopuse_attrib_ptr = ^loopuse_attrib;
edge_attrib_ptr = ^edge_attrib;
edgeuse_attrib_ptr = ^edgeuse_attrib;
vertex_attrib_ptr = ^vertex_attrib;
vertexuse_attrib_ptr = ^vertexuse_attrib;

        { usage vs. element orientation type }
orientation_type = (SAMEorientation,OPPOSITEorientation,UNSPECIFIEDorientation);

{ note:  general pointer variable naming convention is the concatenated string
                                from-element to-element "_ptr"
        where the element types are m,r,s,f,l,e,v,fu,lu,eu, and vu.
        All "next","last" pointers are for circular doubly linked lists.
}
```

Figure 17 – 9. General types for Radial Edge structure in Pascal notation

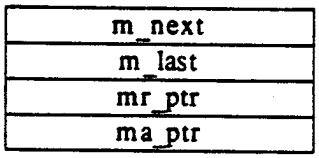
```
var
Models: model_ptr;              { root of data structure; list of all models }

type
```

a) Pascal Declaration

```
model = record
        m_next,m_last: model_ptr;  { list of all active models }
        mr_ptr: region_ptr;        { list of regions in modeling space }
        ma_ptr: model_attrib_ptr   { attribs }
        end;
```

b) Storage allocation description



a) Pascal Declaration

```
region = record
        rm_ptr: model_ptr;              { owning model }
        mr_next,mr_last: region_ptr;    { regions in model list of regions }
        rs_ptr: shell_ptr;              { list of shells in region}
        ra_ptr: region_attrib_ptr       { attribs }
        end;
```

b) Storage allocation description

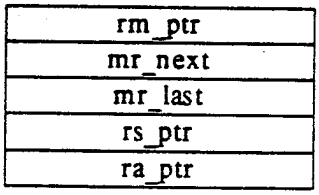
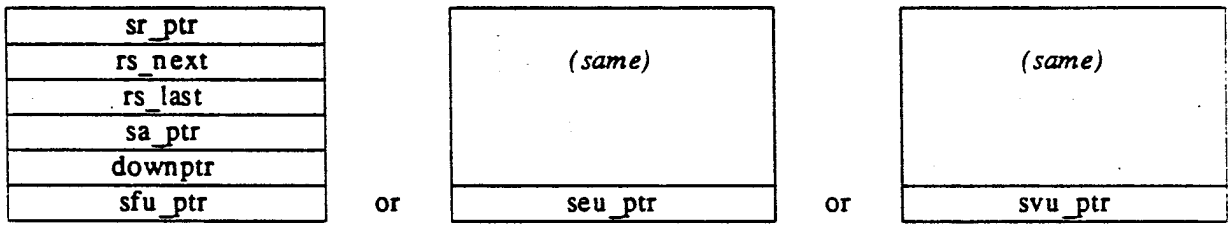


Figure 17 – 10. Types for Radial Edge basic topological elements in Pascal notation

a) Pascal Declaration

```
shell = record
        sr_ptr: region_ptr;                          { owning region }
        rs_next,rs_last: shell_ptr;                  { shells in region's list of shells }
        sa_ptr: shell_attrib_ptr;                    { attribs }
        case downptr: ptr_type of                    { mutually exclusive alternatives }
                FACEUSEptr: (sfu_ptr: faceuse_ptr);          { list of face-uses in shell }
                EDGEUSEptr: (seu_ptr: edgeuse_ptr);          { shell is wireframe }
                VERTEXUSEptr: (svu_ptr: vertexuse_ptr)       { shell is single vertex }
        end;
```

b) Storage allocation description



a) Pascal Declaration

```
face = record
       ffu_ptr: faceuse_ptr;          { list of uses of this face - use fu mate field }
       fa_ptr: face_attrib_ptr        { attribs including geometry }
       end;
```

b) Storage allocation description

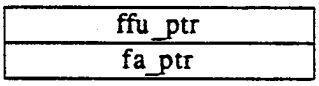


Figure 17 – 11. Types for Radial Edge basic topological elements in Pascal notation

a) Pascal Declaration

```
loop = record
       llu_ptr: loopuse_ptr;     { list of uses of this loop - use eu mate eulu fields }
       la_ptr: loop_attrib_ptr   { attribs }
       end;
```

b) Storage allocation description

llu_ptr
la_ptr

a) Pascal Declaration

```
edge = record
       eeu_ptr: edgeuse_ptr;     { list of uses of this edge - use eu radial/ mate fields }
       ea_ptr: edge_attrib_ptr   { attribs including geometry }
       end;
```

b) Storage allocation description

eeu_ptr
ea_ptr

a) Pascal Declaration

```
vertex = record
       vvu_ptr: vertexuse_ptr; { list of uses of this vertex - use vunext fields }
       va_ptr: vertex_attrib_ptr { attribs including geometry }
       end;
```

b) Storage allocation description

vvu_ptr
va_ptr

Figure 17 – 12. Types for Radial Edge basic topological elements in Pascal notation

a) Pascal Declaration

```
faceuse = record                          { will always be exactly two uses of face }
        fus_ptr: shell_ptr;               { owning shell }
        sfu_next,sfu_last: faceuse_ptr;   { fu's in shell's list of fu's }
        fufu_mate_ptr: faceuse_ptr;       { opposite side of face }
        fulu_ptr: loopuse_ptr;            { list of loops in face-use }
        orientation: orientationtype;     { compared to that of face geom definition }
        fuf_ptr: face_ptr;                { face definition and attributes }
        fua_ptr: faceuse_attrib_ptr       { attribs }
        end;
```

b) Storage allocation description

fus_ptr
sfu_next
sfu_last
fufu_mate_ptr
fulu_ptr
orientation
fuf_ptr
fua_ptr

Figure 17 – 13. Types for Radial Edge adjacency usage topological elements in Pascal notation

a) Pascal Declaration

```
loopuse = record
        lufu_ptr: faceuse_ptr;                  { owning face-use }
        fulu_next,fulu_last: loopuse_ptr;       { lu's in fu's list of lu's }
        lulu_mate_ptr: loopuse_ptr;             { loopuse on other side of face }
        lul_ptr: loop_ptr;                      { loop definition and attributes }
        lua_ptr: loopuse_attrib_ptr;            { attribs }
        case downptr: ptr_type of               { mutually exclusive alternatives }
                EDGEUSEptr: (lueu_ptr: edgeuse_ptr);        { list of eu's in lu }
                VERTEXUSEptr:(luvu_ptr: vertexuse_ptr)      { loop is one vertex only }
                end;
```

b) Storage allocation description

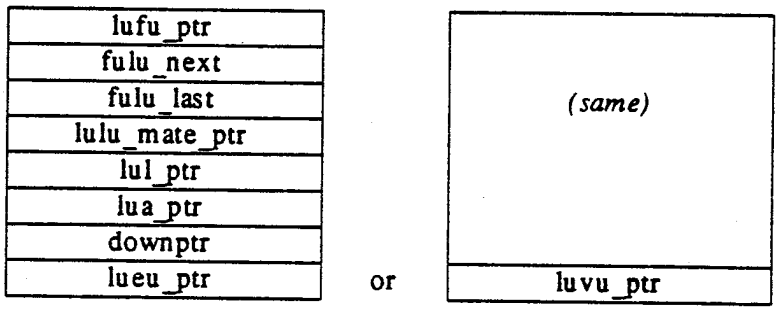


Figure 17 – 14. Types for Radial Edge basic topological elements in Pascal notation

a) Pascal Declaration

```
edgeuse = record
        euvu_ptr: vertexuse_ptr;        { starting vu of eu in this orientation }
        eueu_mate_ptr: edgeuse_ptr;     { eu on other fu of face or end of wire }
        eue_ptr: edge_ptr;              { edge definition and attributes }
        eua_ptr: edgeuse_attrib_ptr;    { parametric space geom }
        case upptr: ptr_type of         { mutually exclusive alternatives }
                SHELLptr: (
                        eus_ptr:shell_ptr               { owning shell }
                        );
                LOOPUSEptr: (
                                                { cw/ccw eu's in lu's ordered eu list }
                        lueu_cw_ptr,lueu_ccw_ptr: edgeuse_ptr;
                        eueu_radial_ptr: edgeuse_ptr;   { eu on radially adjacent fu }
                        orientation: orientationtype;   { compared to geom }
                        eulu_ptr: loopuse_ptr           { owning loop }
                        )
        end;
```

b) Storage allocation description

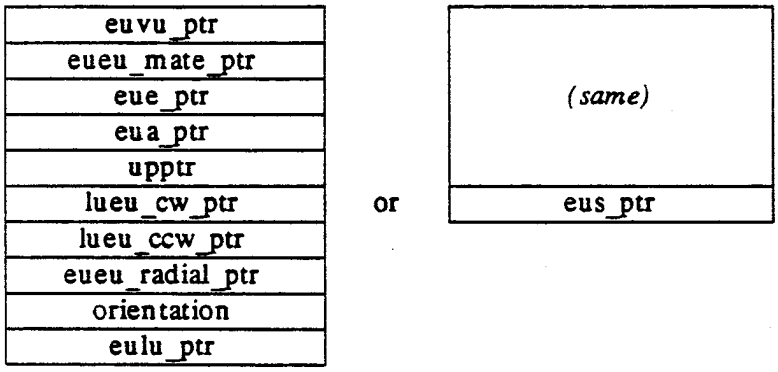


Figure 17 – 15. Types for Radial Edge basic topological elements in Pascal notation

a) Pascal Declaration

```
vertexuse = record
        vu_next,vu_last: vertexuse_ptr; { list of all vu's of vertex }
        vuv_ptr: vertex_ptr;                          { vertex definition and attributes }
        vua_ptr: vertexuse_attrib_ptr;                { parametric space geom & attribs }
        case upptr: ptr_type of                       { mutually exclusive alternatives }
                SHELLptr:(vus_ptr: shell_ptr);        { no fu's or eu's on shell }
                LOOPUSEptr: (vulu_ptr: loopuse_ptr); { loop consists of only this vu }
                EDGEUSEptr: (vueu_ptr: edgeuse_ptr){ eu causing this vu }
        end;
```

b) Storage allocation description

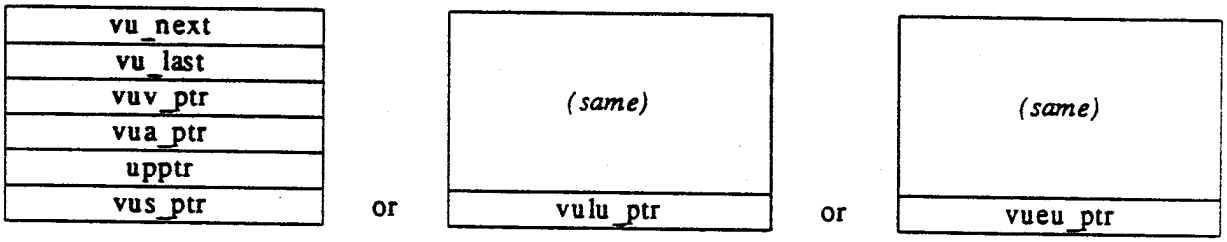


Figure 17 – 16. Types for Radial Edge adjacency usage topological elements in Pascal notation

The loop-use structure also maintains a pointer to the equivalent loop-use structure on the other side of the face. It has a down pointer either to a list of connected edge-use structures or to the single vertex-use in the case of a single vertex loop.

The edge-use structure has two configurations. A wireframe edge is represented by two edge-use structures, one for each end of the edge, and each maintains a direct pointer to the shell it bounds as well as to the other edge-use. If the edge-use bounds a face, it also maintains pointers to allow forward and reverse traversal of the loop-use it is associated with. A downpointer to a vertex-use is also maintained; the

vertex-use at the other end of the edge-use is found from the corresponding field of the mate edge-use. The radial and mate pointers which give ordered access to the edge-uses associated with faces which radially use the edge is a very important aspect of the Radial Edge structure.

The vertex-use structure maintains an up pointer to the lowest dimensional element directly using it: an edge-use in the case of a wire or loop-use edge, a loop-use in the case of a single vertex loop, or a shell in the case of a single vertex shell.

17.2.3. Geometry and Other Attributes

Geometry information is not directly described here since many forms of geometric surface, curve, and vertex coordinate representations are possible, and their definition is not necessary to understand the adjacency topology structures.

Typically, a vertex would have coordinate geometry or procedural coordinate descriptions associated with it. Edges would usually either directly store or refer to curve information, and faces would maintain or refer to a description of the geometric surface geometry. This geometry would be directly associated with the basic topological face, edge, and vertex structures. Models, shells, and sometimes faces can have spatial extent information such as bounding boxes for the efficiency of applications.

Geometry information as it relates to edge-uses and vertex-uses in a system using parametric surface geometry formulations is described in Chapter 20.

Orientation information, as found in the face-use and edge-use structures are binary values which indicate whether the orientation required by the structure agrees or disagrees with the orientation specified by geometry attributes. Orientation information is not strictly related to the adjacency topology information but is included in these structure definitions because they are necessary to specify orientations with the non-manifold topology operators described in Chapter 19. They are included for no other reason.

In most implementations the geometry implementation can be layered on top of the topology implementation so that the topology and geometry packages can be implemented and maintained separately.

Many other kinds of attribute can be associated with the various elements. In mechanical engineering applications, volumes, densities, and other mass properties may be attributes of region and other element types. Color, translucency, and surface finish are other common properties.

17.2.4. Variations in Data Structures

The many variations possible are primarily related to speed vs. storage issues. Algorithmic complexity is also an important factor affecting the implementors' ability to write and maintain applications. The usual relationship is that reducing storage requirements will increase processing time and algorithmic complexity.

Doubly linked lists are not required for virtually any of the lists maintained in the Radial Edge structure, although search would be required to replace the lost functionality for those lists which are ordered. Deletion is usually slower in singly linked lists, however. The optimization trick here is in statistically determining the lists for which search is relatively inexpensive in a given application.

Some of the upward pointers can be eliminated without great hardship; again a statistical profile of the applications using the representation would be useful in making these decisions.

Face, loop, edge, and vertex structures are not required. Even operators based on specifying basic topological elements can be implemented without them if unique naming mechanisms are provided.

In applications requiring heavy use of wireframe elements, storage may be saved by representing wire edges in a single edge-use structure which has pointers to both vertex-use structures at each end of the edge. This does mean that the manipulation

operators must perform procedural checking for wire edges as a special case, however.

Variable length structures could also be investigated to reduce overhead from loop and shell structures, and some of the use structures. As can be seen in the algorithmic complexity comparison of the four manifold data structures in Appendix B, however, there can be drastic complexity costs involved.

17.3. Detecting Volume Closure by Face Additions

A great many new situations arise in a non-manifold environment which do not directly exist in standard manifold environments, simply because the domain is much larger and is correspondingly more complex. Yet there are also correspondences between non-manifold and manifold environments, often similar to correspondences between problems in three dimensions versus problems in two dimensions.

An example is the problem of determining when a new volume has been enclosed as a result of adding a new face to an existing structure in a non-manifold environment. This is a basic task which must be performed repeatedly during modeling operations. The same problem does not directly exist in the manifold environment since all face operations take place on an existing manifold, and any new volumes must be explicitly created by creating a new manifold. However, a corresponding but simpler problem that exists in the manifold environment is a new face being created by the addition of an edge between two existing points.

In a manifold environment, connecting an edge between two points may close off a new face, but only if there is already some other connection in the graph structure between the two endpoints. The only way to discover such situations is through the equivalent of a potentially global search around the edges adjacent to the surface starting on one side of the new edge. If one eventually reaches the other side of the edge from which one started, then a new face has *not* been enclosed. Such algorithms are automatically carried out in manifold modeling systems by operators such as the Euler operators.

In a similar fashion, determination of the enclosure of a new volume in a non-manifold environment should also be made by an automated algorithm in order to maintain the integrity of the representation, since the representation must always know what volumes exist and where their boundaries are. It is not reasonable to expect applications to provide such information since the information is essentially available in the modeling structure itself, and applications should not be concerned with intimate details of the representation system.

As in the manifold environment problem of connecting an edge between two points, some form of search is necessary in order to determine when a new region has been enclosed by the addition of a face in a non-manifold environment. Similarly, the search is essentially a potentially global search of the face-uses adjacent to the original volume in which the face addition operation is being performed. To determine whether a potential new volume is enclosed by the addition of the new face, one must traverse all face-uses adjacent to the original volume to be sure that the potential new volume is not open to the original volume at some location. If a new volume actually has been enclosed, then the search restricts itself only to the face sides (uses) directly adjacent to the new volume. The basic algorithm relies on the principle that if one starts out on one of the sides (face-uses) of the new face and traverses all face sides (face-uses) adjacent to it by volume (and recursively, adjacent to each of those), and eventually the other side of the face from which one started is reached, then a new volume has *not* been formed. If the traversal is complete but did not include the other side of the new face, a new volume has been formed.

The ease with which this basic idea can be implemented is primarily determined by the ease with which information for traversal can be obtained from the data structures. The Radial Edge structure is optimized for obtaining such adjacency information, however, and the algorithm for determining volume enclosure is correspondingly simple.

The algorithm can be described in terms of the two procedures presented here. Record structure access is given in Pascal notation.

```
        { traverse all face-uses (face sides) adjacent by common volume
                to the chosen original face-use (face side) }
traverse(fu):
    1. mark the face-use fu that you are currently on.
    2. traverse all edge-uses eu of all loop-uses of the face-use fu. For each pointer to
        edge-use eu:
        a) newfu ← eu^.eueu_radial_ptr^.eulu_ptr^.lufu_ptr
        b) if newfu is not marked, then traverse(newfu)

        { determine enclosure of new volume after face addition of newf }
enclosure(newf):
    1. traverse(newf^.ffu_ptr)
    2. if ( newf^.ffu_ptr^.fufu_mate_ptr is marked )
        then return FALSE              { no enclosure }
        else return TRUE               { enclosure found }
```

Note that this specialized traversal effectively ignores all wire edges and any faces lying at the end of these wires. This is because any change in the status of a region due to face creation cannot be propagated through an infinitely thin wire (through a 0-dimensional boundary point at its endpoint); it must occur due to a change in accessibility involving faces adjacent through a 1-dimensional boundary element (a curve) to the one just created.

Chapter 18

TOPOLOGICAL SUFFICIENCY

There are three main issues regarding the topological sufficiency of non-manifold boundary topology representations which store adjacency relationships.

First, what is the theoretical minimal amount of topological information required to reconstruct a non-manifold topology for the specified domain ?

Second, what is the practical minimal amount of topological information required to reconstruct a non-manifold topology for the specified domain ? In other words, what is the minimal topological information required to be stored in an implementation of a geometric modeling system ?

Third, are the data structures to be used, in this case the Radial Edge structure, sufficient ?

This chapter outlines some aspects of these issues.

The first issue is still an open research question. This chapter does not prove minimal theoretical topological adjacency relationship information but does conjecture which adjacency relationship information is particularly critical to non-manifold topology representations.

The second issue, as discussed here, is straightforward given the solution to the first, using the same techniques established for obtaining a practical minimum sufficient set of adjacency relationships as used for manifold representations.

Finally, proof of completeness of the Radial Edge structure is then discussed.

18.1. Minimal Theoretical Sufficiency for Non-Manifold Environments

Identification of a minimal sufficient set of adjacency relationships and a formal proof of their sufficiency remains an open problem at this time. A rigorous mathematical basis for minimal sufficiency of non-manifold boundary topology representations, and operators which manipulate them, has not yet been developed.

Several characteristics of the non-manifold adjacency relationships related to sufficiency can be noted on an informal basis, however.

The adjacency relationships which seem to be most convenient in non-manifold modeling representations are the downward and upward hierarchical diagonal adjacency relationships. Examining the relationship between these two sets of adjacency relationships is instructive with regard to exploring sufficiency.

The downward hierarchical diagonal adjacency relationships from $E\{V\}$ to $R\{S\}$ would normally be expected to fall among the commonly useful relationships because they directly indicate the hierarchical relationships between topological elements of various dimensionality.

All of the upward hierarchical diagonal adjacency relationships can easily be derived from their counterpart downward hierarchical adjacency relationships (where $E\{V\}$ is the counterpart to $V\{E\}$, for example), except for the $E\{< L >\}$ adjacency relationship which seems to have substantially different information from its counterpart $L\{< E >\}$ adjacency relationship.

The $E\{< L >\}$ adjacency relationship, indicates the radial ordering of loops using an edge, and therefore (with the other hierarchical adjacency relationships) the radial ordering of all higher dimensional elements about the reference edge. This would therefore appear to be required in order to maintain the adjacency between faces unambiguously.

Certainly for a practical system requiring a labeled graph, a set of five adjacency relationships spanning all the element types (such as the downward hierarchical diagonal

adjacency relationships) would be required.

This set is not minimal if labels are not important, however, as in the theoretical case being considered here. $S\{F\}$, the faces around a shell, could be derived because of the connectivity between faces offered by the $E\{< L >\}$ adjacency relationship, assuming that $L\{F\}$ or $F\{L\}$ was available. $R\{S\}$ cannot be derived from the other downward hierarchical diagonal adjacency relationships, since, for example, a single vertex shell could be located in virtually any region. $R\{V\}$ could be useful as a more powerful alternative in this regard since it does not depend on shell labels.

Other information which must be derivable includes similar situations where direct edge-to-edge connectivity is not available, such as the regions associated with single vertex shells and wireframes, and the faces or loops associated with single vertices forming loops.

18.2. Practical Sufficiency for Non-Manifold Environments

As in the case of manifold modeling representations, a practical non-manifold implementation would not only include information related to the sufficient set of adjacency relationships, but would also include information related to additional adjacency relationships so that information associating all element types is available. This ties the representation together; all elements are uniquely labeled and non-topological data may then be unambiguously associated with the topological model. Thus boundary graph based geometric modeling systems normally require information equivalent to at least $(n-1)$ adjacency relationships (where n is the number of basic topological element types) to associate all element types together into a cohesive whole. In this case $n=6$, so at least five adjacency relationships would be required as a practical minimum sufficiency requirement.

From the previous discussion on minimal sufficiency, it might be conjectured that a set of adjacency relationships equivalent to the downward hierarchical diagonal adjacency relationships and $E\{< L >\}$ adjacency relationship would be a practical minimum

sufficient set (shown surrounded by boxes in Table 16 – 1), as long as additional information about the region associations of wireframe edges and single vertex shells, and face associations of single vertex loops are also maintained.

The next subsection discusses how the Radial Edge structure maintains this information.

18.3. Sufficiency of the Radial Edge Structure

18.3.1. Adjacency Relationships in the Radial Edge Structure

The adjacency relationship matrix containing the thirty-six non-manifold topological element adjacency relationships is given in tables 18 – 1 and 18 – 2, and provides a more detailed description of the adjacency relationships as found in the Radial Edge structure.

The major difference between this adjacency relationship matrix and the general one shown in table 16 – 1 is that the correspondences are being shown, making some of the unordered groups ordered. One other difference is that the radial ordering of the interior group of the *LL* and *FF* adjacency relationships are represented by ordered pairs consisting of those elements before and after the given element in the radial ordered list under consideration.

The upward and downward hierarchical diagonal adjacency relationships, which are all directly represented in the Radial Edge structure (as will be shown shortly), are boxed in the tables. In addition, six other adjacency relationships are also represented directly under the special conditions of a single vertex shell ($V\{S\}$ and $S\{V\}$), a single vertex loop ($V\{L\}$ and $L[<V>]$), and a wireframe portion of a shell ($E[<S>]$ and $S\{E\}$).

Note that linear ordered lists are used in a number of adjacent groups where the

Table 18 – 1. Left Half of Adjacency Relationship Matrix for the Radial Edge Structure

reference element type	*adjacent group element type*		
	vertices	edges	loops
vertex	$V[V]^{\lvert V[E]\rvert}$	$V[E]$	$V\{L\}$
edge	$E[V]^2$	$E[<[E]^{\lvert E<L>\rvert}>]^{\lvert E[V]\rvert}$	$E[<L>]^{\lvert E[V]\rvert}$
loop	$L[<V>^{\lvert L<E>\rvert}]^{\lvert F[S]\rvert}$	$L[<E>]^{\lvert F[S]\rvert}$	$L[<[L]^2>^{\lvert L<E>\rvert}]^{\lvert F[S]\rvert}$
face	$F[[<V>]]^{\lvert F[S]\rvert}$	$F[[<E>]]^{\lvert F[S]\rvert}$	$F[L]$
shell	$S\{V\}$	$S\{E\}$	$S\{L\}$
region	$R\{V\}$	$R\{E\}$	$R\{L\}$

Table 18 – 2. Right Half of Adjacency Relationship Matrix for the Radial Edge Structure

reference element type	*adjacent group element type*		
	faces	shells	regions
vertex	$V\{F\}$	$V\{S\}$	$V\{R\}$
edge	$E[<F>^{\lvert E<L>\rvert}]^{\lvert E[V]\rvert}$	$E[<S>^{\lvert E<L>\rvert}]^{\lvert E[V]\rvert}$	$E[<R>^{\lvert E<L>\rvert}]^{\lvert E[V]\rvert}$
loop	$L\{F\}^1$	$L[S]^{\lvert F[S]\rvert}$	$L[R]^{\lvert F[S]\rvert}$
face	$F[[<[F]^2>]^{\lvert F[L]\rvert}]^{\lvert F[S]\rvert}$	$F[S]^2$	$F[R]^{\lvert F[S]\rvert}$
shell	$S\{F\}$	$S\{S\}$	$S\{R\}^1$
region	$R\{F\}$	$R\{S\}$	$R\{R\}$

general description of the non-manifold adjacency relationships indicated unordered adjacent groups. This indicates situations where correspondence information is available from the Radial Edge structure; the ordering between corresponding adjacency relationships with the same reference element is in correspondence. Further, the adjacency relationships which are in correspondence with a directly represented adjacency relationship have the cardinality of their adjacent group specified as the cardinality of the adjacency relationship originating the correspondence.

18.3.2. Completeness of the Radial Edge Structure

It is necessary to prove two aspects of a topological data structure to prove that it is sufficient. First, it must be proven *complete*, meaning that all adjacency relationships are derivable from the data existing in the data structure. Second, it must be proven *unambiguous*, meaning that there is a one-to-one correspondence between the topological representation and the full topological adjacency relationship information. This means that a unique set of data in the data structure will result in a unique set of full adjacency information. Note that we are not talking about a canonical form for representation of a given shape but rather a unique representation for a given set of full topological adjacency relationship information.

Proving unambiguity for a specific data structure is likely as difficult as finding the minimal sufficient adjacency relationships for the non-manifold environment. Since this information has not yet been found, this section will concentrate on proving completeness of the Radial Edge structure rather than proving full sufficiency. This provides some (but not total) assurance of its sufficiency. Therefore, while it is conjectured that the Radial Edge structure is sufficient, it is not proven here.

Proving completeness, that all adjacency relationships are derivable from the data existing in the Radial Edge data structure in the specified non-manifold environment, involves thirty-six separate derivations, one for each adjacency relationship.

In general, given the upward hierarchical diagonal adjacency relationships, the upward hierarchical relationships above the upward hierarchical diagonal may be derived from their neighboring adjacency relationships, where

$$\overline{A}_{row,column} = f\left(\overline{A}_{row,column-1} , \overline{A}_{column-1,column} \right)$$

Similarly, given the downward hierarchical diagonal adjacency relationships the downward hierarchical relationships below the downward hierarchical diagonal may also be derived from their neighboring adjacency relationships:

$$\overline{A}_{row,column} = f\left(\overline{A}_{row,column+1} , \overline{A}_{column+1,column} \right)$$

Roughly speaking, this means that an adjacency relationship AC can be obtained from AB and BC adjacency relationships. An example of this derivation in adjacency relationship terminology is the following, where $L[<V>]^2$ is derived from $L[<E>]^2$ and $E[V]^2$ information for loops consisting of more than a single vertex:

```
                                    "iterate over reference element type"
L̄i , i←1..n
        create adjacency relationship L̄i< V >  with empty < V>  adjacent group
                                    "initialize status variable"
        v_last←(L̄i[< E> 1]1)[V]1
                                    "iterate over adjacent group"
        L̄i[< E> j]1 , j←1..n
                if v_last= (L̄i[< E> j]1)[V]1
                        then v_last←(L̄i[< E> j]1)[V]2
                        else v_last←(L̄i[< E> j]1)[V]1
                append v_last to L̄i< V >  adjacent group
        output L̄i< V >
```

Derivation of the main diagonal is a less regular process in terms of the positions of the required adjacency relationships in the matrix.

There are several choices of notation that can be used in showing how the adjacency relationship information is derived, however.

While the derivation process can be described in adjacency relationship notation as shown, in this case it does not accurately reflect the derivation process that would actually be used with the Radial Edge structure. The Radial Edge structure contains enough correspondence information that determination of which vertex is associated with the traversal of each edge in $L[<E>]^2$ is explicit and need not be derived with conditionals, as implied in the algorithm above. This is because the individual components in the adjacency relationship terminology refers to entire topological elements rather than the *uses* of the topological elements. The Radial Edge structure represents these uses directly, so the actual process to generate adjacency relationships is often simpler.

A programming notation using accessing and traversal operations similar to the ones to be described in Chapter 19 but tailored specifically to the Radial Edge structure can

also be used to describe the derivation algorithms in a way that corresponds more closely with the information actually present in the Radial Edge structure. For example, the equivalent function to the above conversion algorithm in a programming notation would be:

```
foreach_loop_in_model(l,m,status1)
        create adjacency relationship l< V>  with empty < V>  adjacent group
        foreach_edgeuse_in_loopuse(eu,l^.llu_ptr,status2)
                append eu^.euvu_ptr^.vuv_ptr to l< V>
        output l< V>
       end {foreach}
```

Complete algorithms for the derivation in both cases are actually more complex than the versions presented above since they must account for single vertex loops and the fact that two different orders of the adjacent group are possible based on orientation. A complete version of the conversion algorithm is given in Appendix D.

The proof of completeness of the Radial Edge structure is demonstrated by showing that all thirty-six non-manifold adjacency relationships can be derived.

The detailed algorithms proving completeness are contained in Appendix D. First, it is shown that all ten of the upward and downward hierarchical diagonal adjacency relationships are directly available from the Radial Edge structure. Second, the remaining twenty upward and downward hierarchical adjacency relationships are derived from the existing adjacency relationship information. Lastly, the final six adjacency relationships on the main diagonal of the adjacency relationship matrix are derived.

Since all of the adjacency relationships can be derived from information present in the Radial Edge structure, it is complete.

Traversal as well as other routines are defined in relationship to the Radial Edge structure are used in the Algorithms and are defined in Appendix C.

Chapter 19

NON-MANIFOLD OPERATORS

Operators to build, modify, and traverse non-manifold boundary graph representations, which also insulate higher levels of modeling functionality from specifics and complexities of the data structure, are a necessity for a well structured implementation of a non-manifold modeling system.

While the possibility of non-manifold boundary graph operators have been conjectured [Requicha & Voelcker 83], no work has been published in this area to date.

This chapter outlines a set of basic construction operators developed for building and modifying non-manifold geometric modeling boundary graph topology representations. This particular set of operators was designed for their primitive functionality, allowing other more complex operators to be built using them, for their convenience in the construction process, and for their conceptual compatibility with existing manifold operators, the Euler operators. The external interface to these operators is independent of the actual underlying data structures used; for interface simplicity they refer to topological elements rather than topological adjacency uses whenever possible.

19.1. The Non-Manifold Topology Operators

A key feature of the operators described is that they impose little restriction on the order in which they can be applied during the construction or manipulation of a model. Some sets of operators can provide reduced complexity by restrictions on the application order; this has the unfortunate side effect that knowledge of such restric-

tions must be embedded in higher level applications based on the operators. This is regarded as undesirable and therefore such restrictions are minimized.

The non-manifold topology operators can be classified by two characteristics.

First, some of the operators described here are specialized to handle either *manifold* or *non-manifold* situations, because in some cases substantially different kinds of specifications are required to unambiguously construct a model. The separate manifold versions of the operators incidentally provide compatibility with existing higher level geometric modeling functions originally designed for manifold situations using the Euler operators, although some additional information is required because of the non-manifold environment. When no difference in specification is required, *general* operators are provided which will handle both manifold and non-manifold situations.

Second, the operators can also be classified as to whether they are functionally *constructive* or *destructive* in terms of the number elements existing in the model after application of the operators. Several other operators which can be easily implemented as an application of a series of other constructive and/or destructive operators are classified as *compound* operators although, as will be described, low level implementations can offer some advantages in continuity of the identity of adjacent elements.

Some operations, such as deleting an edge, require different actions based on the situation in which they are applied. When these situations do not require additional specification from the user of the operators, a general operator is provided to handle all subcases. These subcases are shown in Table 19 – 1 with their associated operator, although they are not directly specified during use.

The names of the operators utilize a naming convention similar to the Euler operators, as described in Chapter 13, describing the effect of each operator on the numbers of topological elements in the boundary graph. The names consist of the letters *M* and *K* (standing for "Make" or "create" and "Kill" or "delete") each of which is followed an underscore and by one or more of the letters *M*, *R*, *S*, *F*, *L*, *E*, and *V*, symbolizing the element types model, region, shell, face, loop, edge, and ver-

tex, respectively. The underscore distinguishes the non-manifold operator names from the traditional Euler operator names. In some cases, strictly manifold version of the operators exist, in which case they are preceded by a capital *M* to indicate they are manifold operators. Major operator names are capitalized, and subcases which can be automatically detected are written in lower case. Thus *mekl* stands for "make edge, kill loop", which is an automatically distinguished subcase of *MM_E*.

Table 19 – 2 outlines the changes in the numbers of topological elements as a result of application of each of the operators. In some cases the number of elements involved is variable; these are indicated by subscripted variables in the table.

Table 19 – 3 shows the complementary relationship between specific constructive and destructive operators.

19.1.1. Non-Manifold Positioning Specification

As in the manifold Euler operators, positioning of elements must be specified completely to avoid ambiguity in the semantics of the non-manifold topology manipulation operators. The non-manifold environment is much more complex, however, and more sophisticated positioning information is required.

One possible non-manifold counterpart to the manifold *direction-edge-vertex* positioning specification is the non-manifold *f_orientation-face-edge-e_orientation* positioning specification. This specification technique is useful in some situations for specifying how the edge of a face should be glued into the radial ordering of faces around another edge: the given edge of the face can be glued to the target edge on the *f_orientation* side of face *face* about the target edge *edge*. Note, however, that an *e_orientation* is also required to specify which part of the face should be the reference for the positioning when the face uses the edge twice in a manifold manner, as in the case of a cylindrical face which meets itself along that edge.

In general, however, even this is insufficient if the face specified is a non-manifold

Table 19 – 1. Topology Representation Construction Operators

	general	non-manifold	manifold
constructive	*M_MR*	*M_EV*	*MM_EV*
	M_SV	*M_E*	*MM_E*
		me	*mefl*
		meks	*mekl*
	M_RSFL	*M_F*	
		mfl	
		mflrs	
destructive	*K_V*	*K_F*	
	kvfle	*kflrs*	
	kve	*kfl*	
	kvl	*kflms*	
	kvlms		
	kvs		
	kvrsfl		
	kvsfle		
	kvrsfle		
	K_E		
	ke		
	kems		
	keml		
	kefl		
	keflms		
	K_M		
	G_V		
	gvksv		
	gvkv		
	G_E		
	geke		
	gekfle		
	gekev		
	geksev		
	G_F		
	gfksflev		
	gfkflev		
compound	*ESPLIT*		
	ESQUEEZE		
	esqeezekev		
	esqeezeke		

Table 19 – 2. Operator Effect on Numbers of Topological Elements

operator	*changes in number of topological elements*						
	Models	Regions	Shells	Faces	Loops	Edges	Vertices
M_MR	+1	+1					
M_SV			+1				+1
M_RSFL		+1	+1	+1	+1		
K_V							
kvfle				$-n_f$	$-n_l$	$-n_e$	−1
kve						$-n_e$	−1
kvl					$-n_l$		−1
kvlms			$+n_s$		$-n_l$		−1
kvs			−1				−1
kvrsfl		−1	−1	−1	−1		−1
kvsfle			−1	−1	−1	−1	−1
kvrsfle		$-n_r$	$-n_s$	$-n_f$	$-n_l$	$-n_e$	−1
K_E							
ke						−1	
kems			+1			−1	
keml					+1	−1	
kefl				−1	−1	−1	
keflms			$+n$	−1	−1	−1	
K_M	−1	$-n_r$	$-n_s$	$-n_f$	$-n_l$	$-n_e$	$-n_v$
G_V							
gvksv			−1				−1
gvkv							−1
G_E							
geke						−1	
gekfle				−1	−1	−1	
gekev						−1	−2 or −1
geksev			−1			−1	−2
G_F							
gfksflev			−1	−1	−1	$-n_{ens}$	$-n_{vns}$
gfkflev				−1	−1	$-n_{ens}$	$-n_{vns}$
ESPLIT						+1	+1
ESQUEEZE						−1	−1
esqueezekev						−1	−1
esqueezeke						−1	
M_EV						+1	+1
M_E							
me						+1	
meks			−1			+1	
M_F							
mfl				+1	+1		
mflrs		+1	+1	+1	+1		
K_F							
kflrs		−1	−1	−1	$-n_l$		
kfl				−1	$-n_l$		
kflms			+1	−1	$-n_l$		
MM_EV						+1	+1
MM_E							
mefl				+1	+1	+1	
mekl					−1	+1	

Table 19 – 3. Complementary Relationships Between Construction Operators

constructive	destructive
M_MR	*K_M*
M_SV	*K_V (kvs)*
M_RSFL	*K_F (kflrs)*
M_EV	*ESQUEEZE (esqueezekev)*
MM_EV	*ESQUEEZE (esqueezekev)*
ESPLIT	*ESQUEEZE (esqueezekev)*
M_E	*K_E*
me	*ke*
meks	*kems*
MM_E	*K_E*
mefl	*kefl*
mekl	*keml*
M_F	*K_F*
mfl	*kfl*
mflrs	*kflrs*

face which uses the edge more than once with the same orientation. The only unambiguous specification to handle this situation is to directly utilize edge *uses* or their equivalents in the specification (such as a position in an adjacent group of an adjacency relationship). Edge uses can be used to specify precisely where along the face boundary the glue is supposed to take place.

The need for this more detailed specification arises with the *G_E*, *G_F*, and *M_F* operators when the non-manifold portions of non-manifold faces are involved.

Thus at least two versions of interface specifications for the non-manifold topology operators can be specified. One version, useful in a more limited adjacent environment including some but not all non-manifold conditions, utilizes orientations with respect to adjacent *basic topological elements*. These operators recognize and complain if asked to handle situations where an ambiguity could occur. The other directly utilizes *topological element uses* for specification applicable to all non-manifold conditions.

The specifications given here are primarily of the basic topological element variety, although the glue operations are specified in the element use form, and both forms of the *M_F* operator are given.

19.1.2. A Specification of the Non-Manifold Operators

Specific functional descriptions of the individual operators follow. The interface to each operator is first described in a Pascal style, listing its input parameters (optional parameters are italicized; if not specified they should be *nil* valued pointers or *unspecified* valued orientations), followed by its set of output parameters specified as *var* (call-by-reference) parameters. This calling sequence description is then followed by a detailed description of its functionality and the various subcases handled by the operator. References to topological element types in the calling sequence descriptions refer to pointers to the elements rather than the elements themselves.

The operator specifications given are independent of any specific underlying data structure, within the assumption that separation surface information [Weiler 86a] is not utilized.

Each operator is constrained to meet additional practical criteria. Each returns a value indicating whether the function was completed successfully, or if not, the reason for failure. An operator may not modify the data structures unless no errors will occur and it has sufficient storage to successfully complete the operation. Thus each operator may be regarded as an atomic operation, and the data structure will be consistent both before and after the operator is executed.

Definition of terms used in the operator specifications which have not already been defined are now given. An *orientation* may refer to face orientations, meaning a specific side of the face, or may refer to edge orientations, meaning a specific direction from one end to the other. An *orientation specification* may have values of *same, opposite,* or *unspecified,* and refers to agreement or disagreement with the geometric orientation specified for the face or edge. *Closing off* a region with a new face means

that the creation of the face has divided a region into two distinct new regions; that is, it is not possible to connect a point inside one new region with a point inside the other without penetrating a face, edge, or vertex. The same concept applies in two dimensions when closing off a face.

The operator specifications are followed by diagrams in the same order illustrating their function in Figures 19 – 1, through 19 – 5.

19.1.3. General Operators

M_MR(var model_ptr: newm; var region_ptr: newr)

"Make Model, Region" creates a new model *newm* containing a new region *newr*.

M_SV(region_ptr: r; var shell_ptr: news; var vertex_ptr: newv)

"Make Shell, Vertex" creates a new shell *news* in region *r*, consisting of the single vertex *newv*.

M_RSFL(vertex_ptr: v; region_ptr: r;
var region_ptr: newr; var shell_ptr: news; var face_ptr: newf;
var loop_ptr: newl)

"Make Region, Shell, Face, Loop" creates a new region *newr* inside region *r*, with a shell *news* which consists of the single face *newf*, which has the single loop *newl* consisting of the existing single vertex *v*. The operator can be thought of as creating a spherical surface containing region *newr* which touches upon vertex *v* in region *r*.

K_V(vertex_ptr: v)

"Kill Vertex" deletes the vertex *v* and any edges which touch upon it, deleting loops, faces, shells, and regions as necessary. *K_V* will not delete a vertex when deletion of the vertex (and/or edges incident to it) would cause the creation of a non-manifold face. In this case an error will result

and no action will be performed. There are more subcases than the ones listed below involving combinations of the described graph conditions (for example, one can always add a single wire edge which is not a self loop to the situations described below for which the operator has an "*e*" in its name, and the result will also create a single vertex shell), and the situations handled can actually be more complex than is described below (for example, one can always add self loop wire edges to the situations described below for which the operator has an "*e*" in its name and the result is that those edges will also be deleted).

kvfle: *"kill vertex, faces, loops, edges"* occurs when the vertex lies on a surface and has one or more incident manifold edges which separate different faces. In this case the edges are deleted, deleting a face and a loop for each edge separating two faces.

kve: *"kill vertex, edge"* occurs when the vertex has incident manifold isthmus or strut, or wireframe edges whose deletion does not result in a disconnected graph.

kvl: *"kill vertex, loop"* occurs when the vertex has no incident edges and is a single vertex loop vertex on one or more faces which have other boundaries. If there is more than one face, they all must have one or more common boundaries other than the vertex.

kvlms: *"kill vertex, loop, make shell"* occurs when the vertex has no incident edges and is a single vertex loop vertex on more than one face, all of which which have other boundaries, but at least two of which have no other common connection to each other. In this case the faces become separated and additional shells are generated. *(not shown in diagram)*

kvs: *"kill vertex, shell"* occurs when the vertex has no incident edges and is a single vertex shell. If it was the only vertex in the model, only a single region will remain in the essentially empty model.

kvrsfl: *"kill vertex, shell, face, loop"* occurs when the vertex has no incident edges and is a single vertex loop vertex on a face which has no other boundary. If it was the only vertex in the model, only a single region will remain in the essentially empty model.

kvsfle: *"kill vertex, shell, face, loop, edge"* occurs when the vertex is an endpoint of a single self loop edge on a lamina face which has no other boundaries. If it was the only vertex in the model, only a single region will remain in the essentially empty model.

kvrsfle: *"kill vertex, region, shell, face, loop, edge"* occurs when the vertex is an endpoint of one or more self loop edges on two or more faces which have no other boundaries. If it was the only vertex in the model, only a single region will remain in the essentially empty model. *(not shown in diagram)*

K_E(edge_ptr: e; *vertex_ptr: v; face_ptr: fsurvivor;*

var loop_ptr: newl; var shell_ptr: news)

"Kill Edge" deletes the edge *e*. *K_E* will not delete an edge when deletion

of the edge would cause the creation of a non-manifold face; this condition is always true of non-manifold edges where an edge is used three or more times a single face. In this case an error will result and no action will be performed.

ke: *"kill edge"* occurs when the specified edge is a wireframe edge that is not the only connectivity path between its two vertices.

kems: *"kill edge make shell"* occurs when the specified edge is a wireframe edge which is the only path of connectivity between its two vertices. If specified, the vertex *v* is the vertex with the original shell; the other vertex of the deleted edge is part of the new shell.

keml: *"kill edge make loop"* occurs when the specified manifold edge lies in a face and is an "isthmus" or "strut" edge, that is, it occurs twice in the loop of the face.

kefl: *"kill edge, face, loop"* occurs in two cases. The first is when the specified manifold edge lies between two different faces. In this case, the face specified by *fsurvivor* is not deleted. The deleted face has its loop appropriately merged with the surviving face's loop, and any other loops of the deleted face become part of the surviving face. The second case is when a lamina edge is deleted, causing the face and loop using it to be deleted.

kefims: *"kill edge, face, loop, make shell"* occurs when the edge is a lamina edge boundary of a face which has multiple loops which are not otherwise connected except through the face. The face and its loops are destroyed when the edge is deleted, and the elements connected to each former loop become part of their own separate shell.

K_M(model_ptr: m)

"Kill Model" deletes the model *m* and all of its constituent topological elements.

G_V(vertexuse_ptr: vu1, vu2)

"Glue Vertex" merges the vertices of *vu1* and *vu2* together, preserving the adjacencies of elements. The vertex of *vu1* is the surviving vertex; the vertex of *vu2* is deleted. Both *vu1* and *vu2* must be adjacent to the same region or an error will result and no action will be performed.

gvksv: *"glue vertex, kill shell, vertex"* occurs when the two specified vertices are not connected by any path (not located on the same shell). The surviving shell is the shell of vertex

gvkv: *"glue vertex, kill vertex"* occurs when the two specified vertices are already connected by some path (located on the same shell).

G_E(edgeuse_ptr: eu1,eu2)

"Glue Edge" merges the edge of *eu1* together with the edge of *eu2*, preserving the adjacencies of elements. The edge of *eu1* is the surviving

edge; the edge of *eu2* and any of its vertices which are not shared with the edge of *eu1* are deleted. The orientations with which to glue the edges together are specified by the edge-use input parameters themselves; they are glued together in the specified orientation. Both *eu1* and *eu2* must be adjacent to the same region or an error will result and no action will be performed. Note that if the edges of *eu1* and *eu2* share any vertices, the acceptable orientations for glue operations are already fixed; if they are improperly specified, an error will result and no action will be performed.

geke: *"glue edge, kill edge"* occurs when the two edges already share the same vertices.

gekfle: *"glue edge, kill faces, loops, edge"* occurs when the two edges share the same two vertices and both edges form the loop boundary of one or more faces, then the merging of the two edges eliminates any of these faces and loops.

gekev: *"glue edge, kill edge, vertex"* occurs when the two edges share one vertex.

geksev: *"glue edge, kill shell, edge, vertex"* occurs when the two edges are not connected by any path (are not on the same shell). The surviving shell is the shell of edge *e1* in the region common to the two edges.

G_F(faceuse_ptr: fu1; edgeuse_ptr: eu1; faceuse_ptr: fu2; edgeuse_ptr: eu2)

"Glue Faces" merges the single loop faces of *fu1* and *fu2* together, preserving the adjacencies of elements. The face-use input parameters specify which side of each face to glue together. *eu1* and *eu2* specify exactly how and in what direction the two loop boundaries match up; the loops are glued together with the edge-uses in opposite orientation. The face of *fu1* is the surviving face; the face of *fu2* and its loop, and edges and vertices not shared with the face of *fu1*, are deleted. The face sides specified by both *fu1* and *fu2* must be adjacent to the same region and must each have one only loop with the same number of edges with self loop, isthmus, and strut edges in an identical order in both loops or an error will result and no action will be performed. Note that if the faces of *fu1* and *fu2* share any edges or two or more vertices the acceptable orientations for glue operations may already be fixed; if they are improperly specified, an error will result and no action will be performed. The manifold glue operation often found in implementations of the Euler operators can be emulated by

performing an additional *K_F* after the *G_F* to remove the surviving face of *ful*.

gfksflev: *"glue face, kill shell, face, loop, edges, vertices"* occurs when the specified faces are not connected by any path (are not on the same shell). The surviving shell is the shell using the *orientfl* side of *fl*.

gfkflev: *"glue face, kill face, loop, edges, vertices"* occurs when the specified faces are connected by some path (are on the same shell).

gfksflv: *"glue face, kill face, loop, edges, vertices"* occurs when the specified faces have a single loop consisting of one vertex and bound single face shells which are adjacent to the same region. The surviving shell is the shell of *fl*. *(not shown in diagram)*

ESPLIT(edge_ptr: e; *vertex_ptr: v;* var edge_ptr: newe; var vertex_ptr: newv)

"Edge Split" splits the specified edge *e* into two connected edges, *e* and *newe*. A new vertex, *newv*, is created between these two edges. The optional parameter *v*, if specified, designates which vertex of the edge *e* will be found on the new edge. For manifold situations the effect of this operator could be simulated by application of the *K_E* operator followed by *MM_EV* and *MM_E* operators, but unlike *ESPLIT*, edge *e* would be entirely replaced rather than modified in place and, by side effect, a face could be deleted and replaced with a new one, perhaps shifting ownership of interior loops. In non-manifold situations where the edge is used three or more times by one or more faces, *K_E* will not allow deletion of the edge since non-manifold faces would be created, so *ESPLIT* is the only alternative.

ESQUEEZE(edge_ptr: e; *vertex_ptr: v*; var vertex_ptr: vsurvivor)

"Edge Squeeze" "squeezes" the ends of the specified edge *e* together, deleting the edge and a vertex while preserving adjacencies. The optional parameter *v*, if specified, designates which vertex of the edge *e* will survive; in any case, the surviving vertex is indicated by the *vsurvivor* return parameter. For manifold situations the effect of this operator could be simulated by application of the *K_E* operator followed by the *G_V* operator, but unlike *ESQUEEZE*, by side effect a face could be deleted and replaced with a new one, perhaps shifting ownership of interior loops.

specified by *e_orients*. Note that this is sufficient for some but not all cases where the edge is non-manifold, and is not a sufficient specification for creating faces forming non-manifold surfaces. In these cases the *U_M_F* version of the operator should be used. Individual specifications in the specification lists are optional whenever the specific edges in *edges* are wireframe and/or lamina edges. In this case *nil* and *unspecified* orientations are placed in the proper position in the face and face orientation lists. If all edges meet these criteria, the entire specification lists themselves are optional. The specifications given must meet the connectivity constraints of the existing graph. Otherwise, an error will result and no action will be performed.

mfl: *"make face, loop"* occurs when the new face will not close off one portion of the region it is in from the rest of the region.

mflrs: *"make face, loop, region ,shell"* occurs when the new face does close off one portion of the region it is in from the rest of the region. In this case the new region, *newr*, and shell, *news*, lie to the specified orientation side of the face of the first face specified in *facelist*.

U_M_F(edgeuselist: edgeuses; *e_orientlist: e_orients;*

var face_ptr:newf; var loop_ptr: newl;

var region_ptr: newr; var shell_ptr: news)

"Element Use Make Face" is the full non-manifold version of "Make Face" utilizing the element use input specification. It creates a new face *newf* with its single loop *newl* bounded by the single circuit of edges as specified in *edgeuses*. The single specification list *edgeuses* specifies not only the edges to use, but also states that the new face will lie between any face owning the specified edgeuse and the face found radially opposite to the specified edgeuse. The optional input specification list *e_orients* is used to specify which orientation of an edge to use in cases of self loop edges. Similar restrictions to and subcases of *M_F* apply to *U_M_F*, except that it handles all non-manifold situations. Note that this element use version of the *M_F* operator is only required in situations involving edges used more than once in a single orientation by a single face, and for creating faces which themselves form non-manifold surfaces.

G_F(faceuse_ptr: fu1; edgeuse_ptr: eu1; faceuse_ptr: fu2; edgeuse_ptr: eu2)
"Glue Faces" merges two single loop faces together, preserving the adjacencies of elements. The orientation of each face is specified to tell which side of each face to glue together. An edge from each face along with orientation are also specified to tell exactly how and in what direction the two loop boundaries match up. One face is the surviving face; The other face and its loop, and edges and vertices not shared with the surviving face are deleted. The specified sides of both faces must be adjacent to the same region and must have one loop with the same number of edges or an error will result and no action will be performed. The manifold glue operation often found in implementations of the Euler operators can be effected by performing an additional *KF* after the *G_F* to remove the surviving face.

ESPLIT(edge_ptr: e; *vertex_ptr: v;* var edge_ptr: newe; var vertex_pr: newv)
"Edge Split" splits an edge into two connected edges. A new vertex is created between these two edges. An optional parameter specifies which vertex of the old edge will be found on the new edge.

ESQUEEZE(edge_ptr: e; *vertex_ptr: v;* var vertex_ptr: vsurvivor)
"Edge Squeeze" "squeezes" the ends of an edge together, deleting the edge and a vertex while preserving adjacencies. An optional parameter specifies which vertex of the edge will survive.

4.2. Non-Manifold NMT Operators

M_EV(vertex_ptr: v; region_ptr: r; var edge_ptr: newe; var vertex_ptr: ne 27.br *"Make Edge, Vertex"* creates a new wire edge which connects an existing vertex with a new vertex. The new edge and vertex will exist in the specified region. The existing vertex must be adjacent to this region or an error will result and no action will be performed.

M_E(vertex_ptr: v1,v2; region_ptr: r; var edge_ptr: newe)
"Make Edge" creates a new wire edge between two specified vertices The new edge will exist in a specified region. Both specified vertices must be adjacent to this region or an error will result and no action will be performed.

M_F(edgelist: edges; *facelist: faces; f_orientlist: f_orients; e_orienlist: e_orients;*
var face_ptr:newf; var loop_ptr: newl;
var region_ptr: newr; var shell_ptr: news)
"Make Face" creates a new face with a single loop bounded by the single circuit of edges as specified in an edge list. The list of edges specified by *edgelist* must form a true circuit or an error will result and no action will be performed. Specification lists for loops and orientations are of the same length as the edge list and are used to specify the radial positioning of the new face whenever the edge specified is already a manifold edge or a non-manifold edge. The new face will be attached to the edge so that it lies to the specified orientation side of the face identified by the loop and orientation specification corresponding to that edge. Individual specifications in the specification lists are optional whenever the specific edges in the edge list are wireframe and/or lamina edges. In this case no loops or orientations are specified for the proper position in the loop and orientation lists. If all edges meet these criteria, the the entire specification lists themselves are optional. To assist in the creation of multiple loop faces and additional operator called *M_L* is used. The *M_F* specification technique cannot be used in all non-manifold

situations; in these cases *UM_F* and M_L operators which utilize edge-use information can be used.

K_F(face_ptr: f; *orientationtype: orient*)
"Kill Face" deletes a face and all loops associated with it. It does not delete any edges or vertices.

4.3. Manifold NMT Operators

MM_EV(vertex_ptr: v; *edge_ptr: e; dir_type: dir;* { *CW or CCW* }
face_ptr: f; *orientationtype: orient;*
var edge_ptr: newe; var vertex_ptr: newv)
"Manifold Make Edge, Vertex" creates a manifold edge and a vertex. A new edge starts at the specified existing vertex and ends at the new vertex. The edge and vertex are created in a specified face If optional placement is specified, the new edge will be located in a specific orientation with respect to the existing manifold elements, using an edge-vertex-direction specification [EAST79]. The vertex and edge specified must be on the boundary of a specified face or an error will result and no action will be performed.

MM_E(vertex_ptr: v1; *edge_ptr: e1; dir_type dir1;* { *CW or CCW* }
vertex_ptr: v2; *edge_ptr: e2; dir_type dir2;*
face_ptr: f; *orientationtype: orient;*
var edge_ptr: newe; var face_ptr: newf; var loop_ptr: newl)
"Manifold Make Edge" creates an edge between two existing vertices. The edge is created in a specified face. If optional placement is specified, the new edge will be located in a specific orientation with respect to the existing manifold elements, using an edge-vertex-direction specification [EAST79]. The vertices and edges specified must be on the boundary of the specified face or an error will result and no action will be performed.

4.4. Other NMT Operators

Several other operators, not described in detail here, are also useful.

Some are composite operators, equivalent to sequences of other operators. Operations such as *MM_LV*, which make a single loop vertex in a face (equivalent to *MM_EV* followed by *K_E*), and *M_MRSV*, which simply calls *M_MR* and *M_SV* in sequence, are convenient.

Others do not manipulate the graph structure in the same way as the operators described above. Examples are move operations, such as *SMOVE*, to move shells into different regions and models, and *LMOVE*, to move loops into different faces adjacent to the same region. Copy operations are also useful, such as *SCOPY*, to copy shells and place them into different regions and models, and *MCOPY*, to copy entire models.

K_F(face_ptr: f; *orientationtype: orient*)

"Kill Face" deletes the face *f* and all loops associated with it. It does not delete any edges or vertices. There are actually more subcases than described here.

kflrs: *"kill face, loop, region, shell"* occurs when the specified face has different regions on each side. In this case, deletion of the face brings together the two regions. If specified, the surviving region is the region lying to the *orient* side of the face. All loops associated with the face are also deleted.

kfl: *"kill face, loop"* occurs when the specified face has the same region on both sides. All loops associated with the face are also deleted.

kflms: *"kill face, loop, make shell"* occurs when the specified face has the same region on both sides, has one or more loops consisting entirely of lamina edges. All loops associated with the face are also deleted, but an additional shell is generated for each loop which had no connection to the boundaries of the other loops except through the face.

19.1.5. Manifold Operators

MM_EV(vertex_ptr: v; *edge_ptr: e; dir_type: dir;* { *CW or CCW* }

face_ptr: f; *orientationtype: orient;*

var edge_ptr: newe; var vertex_ptr: newv)

"Manifold Make Edge, Vertex" creates a manifold edge and a vertex. The new edge *newe* starts at existing vertex *v* and ends at the new vertex *newv*. The edge and vertex are created in the face *f*. If optional placement is specified, *newe*, will be positioned in direction *dir* from edge *e* about vertex *v*, as seen when looking towards the *orient* side of face *f*. Vertex *v*, and if specified, edge *e* must be on the boundary of face *f* or an error will result and no action will be performed.

MM_E(vertex_ptr: v1; *edge_ptr: e1; dir_type dir1;* { *CW or CCW* }

vertex_ptr: v2; *edge_ptr: e2; dir_type dir2;*

face_ptr: f; *orientationtype: orient;*

var edge_ptr: newe; var face_ptr: newf; var loop_ptr: newl)

"Manifold Make Edge" creates an edge between the existing vertices *v1* and *v2*. The edge is created in the face *f*. If optional placement is

specified, the new edge, *newe*, will be direction *dir1* about vertex *v1* from edge *e1*, and direction *dir2* about *v2* from *e2*, as seen when looking towards the *orient* side of face *f*. Vertices *v1* and *v2*, and if specified, edges *e1* and *e2* must be on the boundary of face *f* or an error will result and no action will be performed. Note that the *meksfl* case of the Euler operators is not relevant in the non-manifold environment.

mefl: *"make edge, face, loop"* occurs when the new edge will close off one portion of the face it is on from the rest of the face. In this case, the new face, *newf*, and loop, *newl* will lie to the *dir1* side of *newe* about *v1*, as seen when looking towards the *orient* side of face *f*.

mekl: *"make edge, kill loop"* occurs when the new edge will not close off one portion of the face it is on from the rest of the face. In this case, the vertices *v1* and *v2* were on different loops of the same face, but afterwards will be located on the same loop. The surviving loop is the loop associated with *v1*.

19.1.6. Other Operators

Several other operators, not described in detail here, are also useful but do not manipulate the graph structure in the same way as the other operators described above. Examples are move operations, such as *SMOVE*, to move shells into different regions and models, and *LMOVE*, to move loops into different faces adjacent to the same region. Copy operations are also useful, such as *SCOPY*, to copy shells and place them into different regions and models, and *MCOPY*, to copy entire models.

19.2. Sufficient Set of Construction Operators

While many operators can be designed to promote efficiency or convenience for given applications, one interesting issue is to determine a minimal set of operators which can define any model in the representation. There can be many such minimal sets of operators, since many different operators can be designed which have overlapping functionality. These operators may incrementally push construction of a model towards a given specification in different sized steps, with eventually the same result.

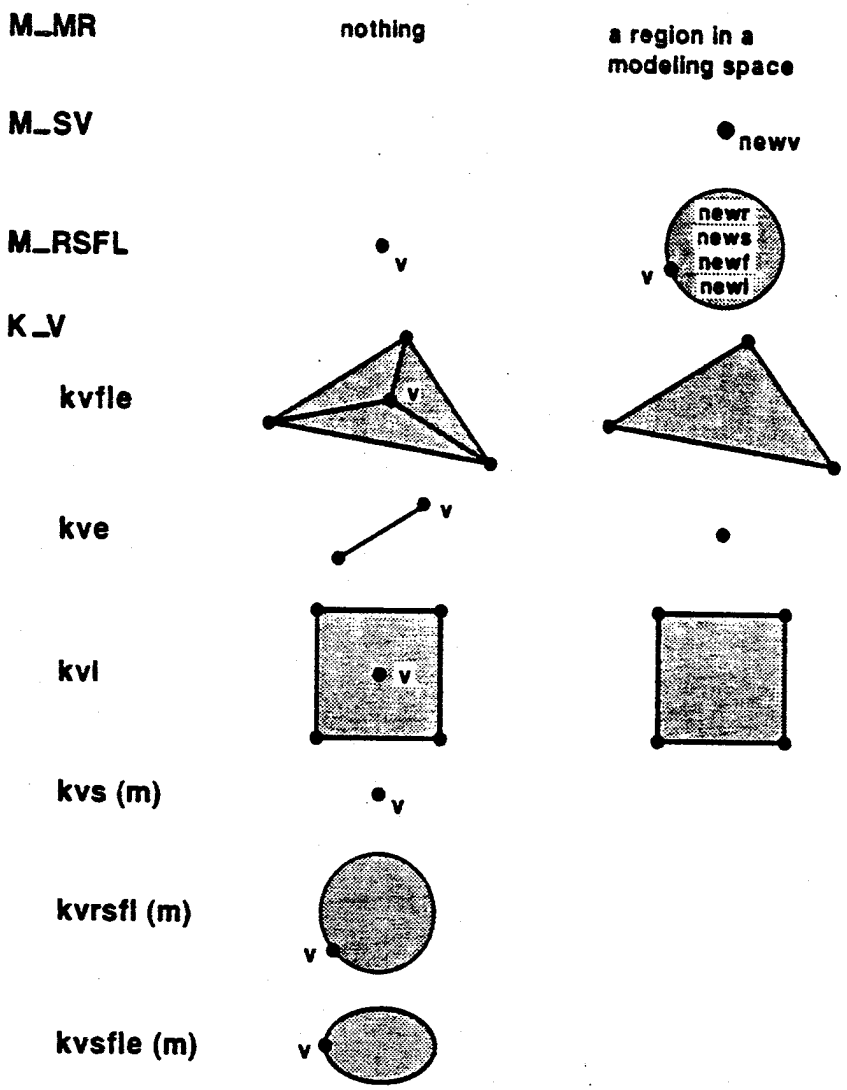


Figure 19 – 1. Action of the non-manifold topology operators

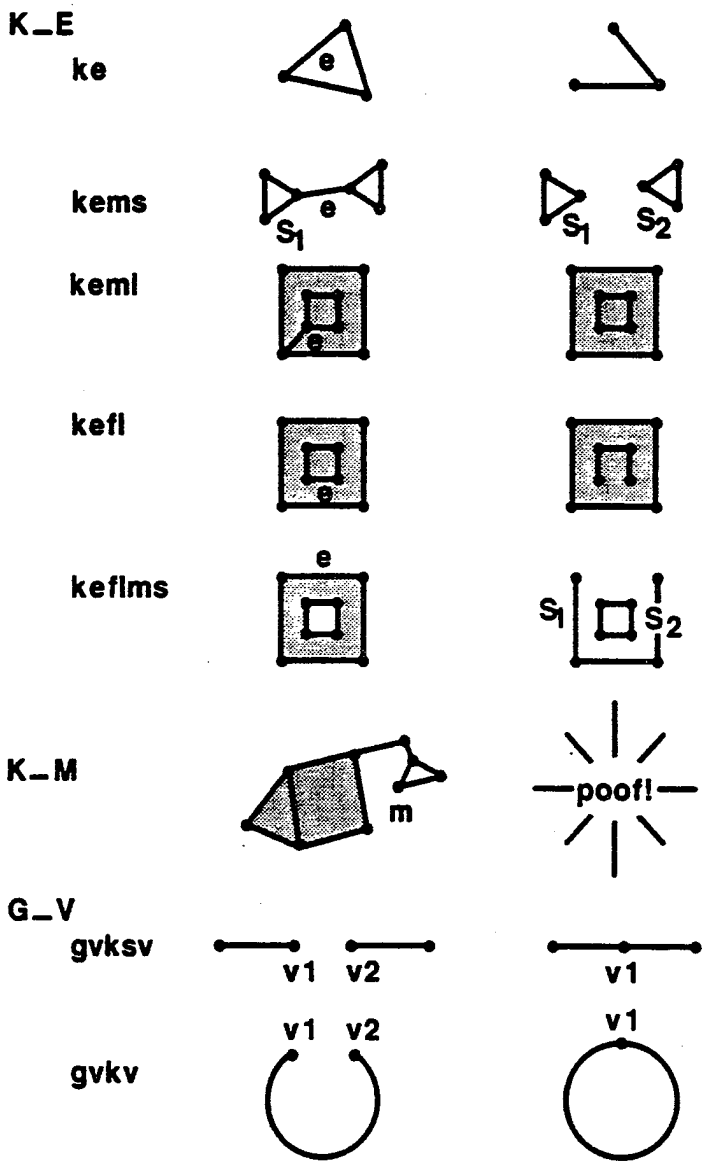


Figure 19 – 2. Action of the non-manifold topology operators

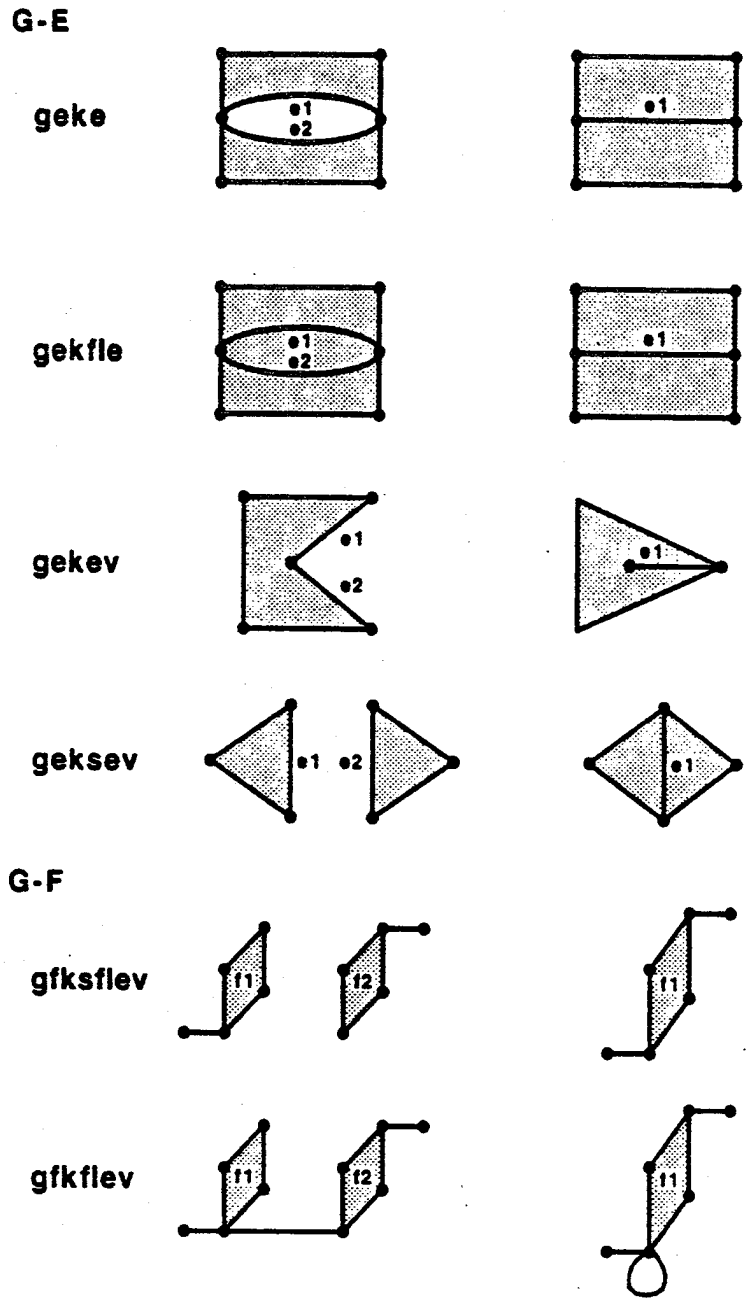


Figure 19 – 3. Action of the non-manifold topology operators

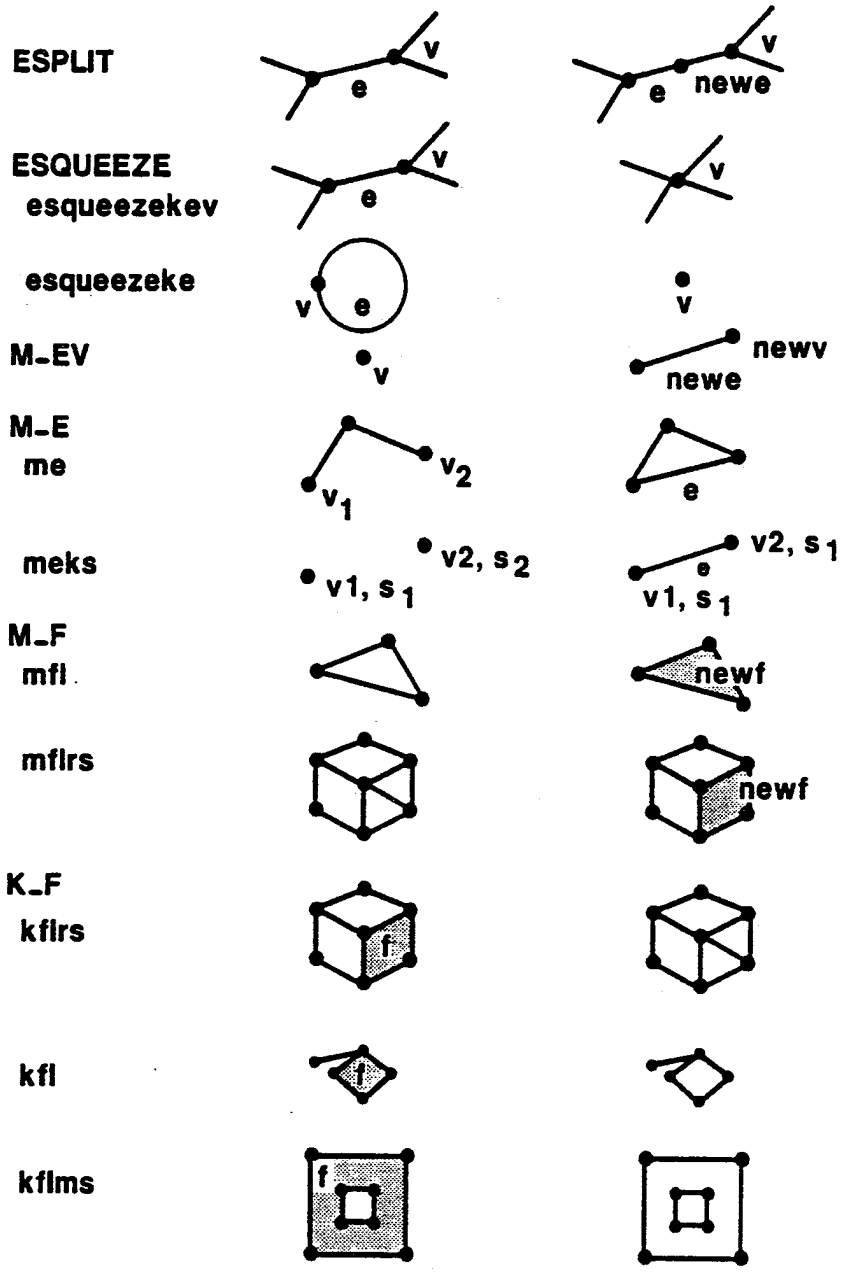


Figure 19 – 4. Action of the non-manifold topology operators

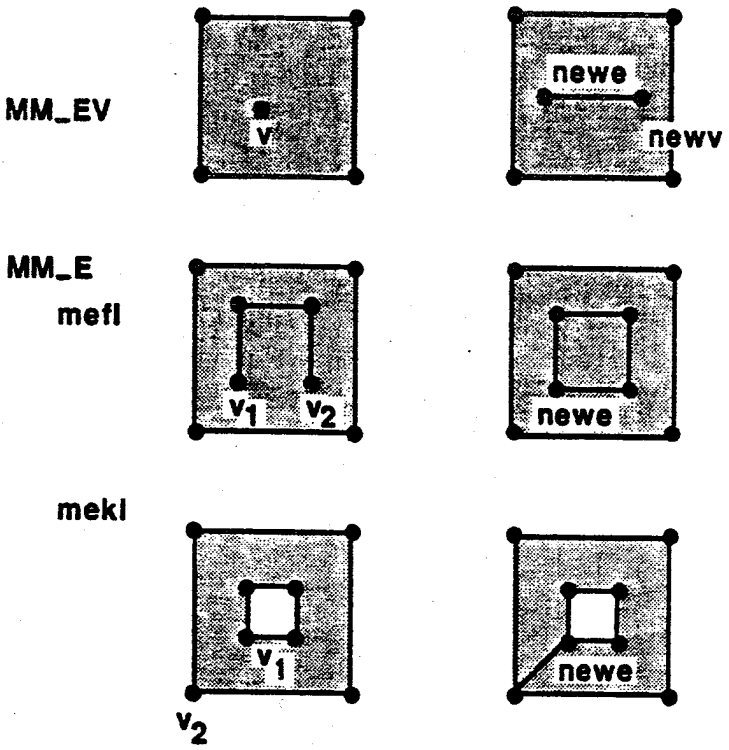


Figure 19 – 5. Action of the non-manifold topology operators

The set of operators shown in Figure 19 – 6 is one such minimal set. It was chosen for simplicity of the functionality of the operators as well as a minimal number of operators. While any possible model may be constructed with them, it is not a particularly convenient set to manipulate existing models (without additional operators).

To construct any model with this set of operators from some existing specification of the final boundary model, the following algorithm may be used:

1. Do a *M_MR* to create the model and initial region.
2. Do a *M_SV* for every vertex to be in the finished model.
3. Do a *M_E* between appropriate vertices for every edge to be in the finished model.
4. Do a *M_E* to connect together loops which share the same face. This includes "hole" loops in faces and vertices which will be a single loop vertex in the finished model. Thus all edges and vertices sharing a common face are connected, and a face with n loops (where $n \geq 2$) in the finished model requires $n-1$ *M_E* operations to be performed. Call these additional edges *loop edges*.
5. Do a *M_F* for every face in the finished model (excepting faces with a single vertex as the sole boundary), being sure to appropriately include *loop edges* into the boundary descriptions.
6. Do a *M_E* to create a self loop edge for every vertex which will be the sole boundary of a face in the finished model, making sure they will be created in the appropriate regions. Call these additional edges *face edges*.
7. Do two *M_F's* for each self loop edge in the *face edge* list, creating the new regions.
8. Do a *K_E* to eliminate each *loop edge* (causing a *keml* to be performed) and *face edge* (causing a *kefl* to be performed).

While this algorithm is also not particularly efficient, it is conjectured to be a

---

M_MR
M_SV
M_E
M_F
K_E

Figure 19 – 6. A minimal sufficient set of operators to construct any model

---

minimally sufficient set for the job for the following reasons. First, *M_MR* and *M_SV* are required to start any model. Second, the only useful alternative besides *M_E* for creating edges is *M_EV*, but in that case *M_E* would still be required to close a circuit of edges. Third, the only useful alternative besides *M_F* for creating faces, and in particular lamina faces, is *M_RSFL* combined with *K_F*, but that alternative uses two operators instead of one. Fourth, *K_E* is required to create disconnected graphs in a manifold surface and, with this set of operators, to create faces with only a single vertex loop for a face boundary.

For efficiency and convenience, a practical modeling system would offer more than this minimal set, however. Adding more operators to the minimal set described removes inconvenient restrictions on the order of operations necessary to construct and modify objects. A reasonable order of usefulness for adding more operators to this minimal set might be first *M_EV* to complete basic wireframe construction capabilities, and then *M_RSFL*, *MM_EV*, and *MM_E* for convenience in manifold modeling situations, followed by the others for more specialized situations and destructive operators for convenience in modification.

19.3. Examples of Use of the Non-Manifold Operators

A short example of applying the non-manifold topology operators to build the object shown in Figure 16 – 2 is now given.

For brevity, a list notation is used to describe the edge lists for the *M_F* operator. The face orientation initially chosen for the manifold operations, shown here as *outside* is arbitrary but must be used consistently.

```
                        "start the model"
M_MR(m,r1);

                        "create the tetrahedron"
M_SV(r1,s1,v1);
M_RSFL(v1,r1,r2,s2,f1,l1);                          "create face f1"
MM_EV(v1,nil,unspecified,f1,e1,v4);
MM_EV(v4,e1,CCW,f1,outside,e4,v2);
MM_E(v1,e1,CCW,v2,e4,CCW,f1,orient,e3,f2,l2);       "close face f2"
MM_EV(v2,e3,CCW,f1,outside,e5,v3);
MM_E(v3,e5,CW,v1,e3,CW,f1,outside,e2,f3,l3);        "close face f3"
MM_E(v3,e5,CW,v4,e4,CCW,f1,outside,e6,f4,l4);       "close face f4"

                        "create the lamina face"
M_EV(v3,r1,e7,v5);
M_E(v5,v2,r1,e8);
M_F(< e5,e8,e7> ,< f3,nil,nil> ,< outside,unspecified,unspecified> ,
        < unspecified,unspecified,unspecified> ,f5,l5,r_dummy,s_dummy);

                        "create the single vertex loop"
MM_EV(v5,e8,CCW,f5,outside,e10,v7);
K_E(e10,nil,nil,l6,s_dummy);

                        "create the wire"
M_EV(v1,r1,e9,v6);

                        "create the single vertex shell"
M_SV(r1,s3,v8);
```

19.4. Specification of the Access Operators

Operators to access data must be specified as carefully as the manipulation operators in order to maintain the major advantage of being able to layer application code on top of the topology implementation in a manner independent of the actual data structures utilized. In the past, efficiency constraints have prevented access operator specifications being made for manifold edge based boundary representations, but the wide availability of macro processors for a variety of languages largely nullifies this concern.

The topological adjacency information stored in any non-manifold model consists of the existence and adjacencies of the six topological element types. Queries and traversals are related to accessing this adjacency information.

Queries are single accesses to adjacency relationship information to determine a single element of the adjacent group of a specified element, which perhaps also meets some additional set of characteristics. When the adjacency relationship is ordered, the additional characteristic specified might be that the adjacent group element being sought directly follows or precedes a specified adjacent group element in the ordering.

Traversals are repeated queries to determine all members in an adjacent group of the adjacency relationship, even if none are originally known. Traversals may therefore be constructed by repeated queries, as long as termination and status information is available.

As an example of the need for status information during traversal, in a manifold topology situation involving strut edges (manifold edges which have the same face on either side), a traversal to find all edges around a loop could not simply terminate on encountering a given edge a second time, since it might be used either once or twice in the loop; some other criteria is required (usually an edge and a vertex is kept for status if self loops are disallowed, or a marking scheme is used). In a non-manifold environment a similar situation exists in the radial traversal of faces around an edge; a face is not sufficient status information since a face may be bounded by the edge multiple times. In this case a specific edge use is required for non-ambiguity in cases involving faces which would be non-manifold if their boundaries were included, such as the example in Figure 19 – 7 where the same face meets itself three times along an edge.

If the adjacent group of the adjacency relationship is ordered, then the traversal accesses are also ordered.

Following is a specification of access and traversal operators, which are independent of the underlying data structure, and for simplicity, many of which are independent of the concept of *uses* of topological elements. As seen with the *M_F*, *G_F*, and *G_E* operators, the element use concept is a natural and necessary one in some of the more complex non-manifold cases. For this reason, additional traversal operations which utilize the notion of element uses are also included.

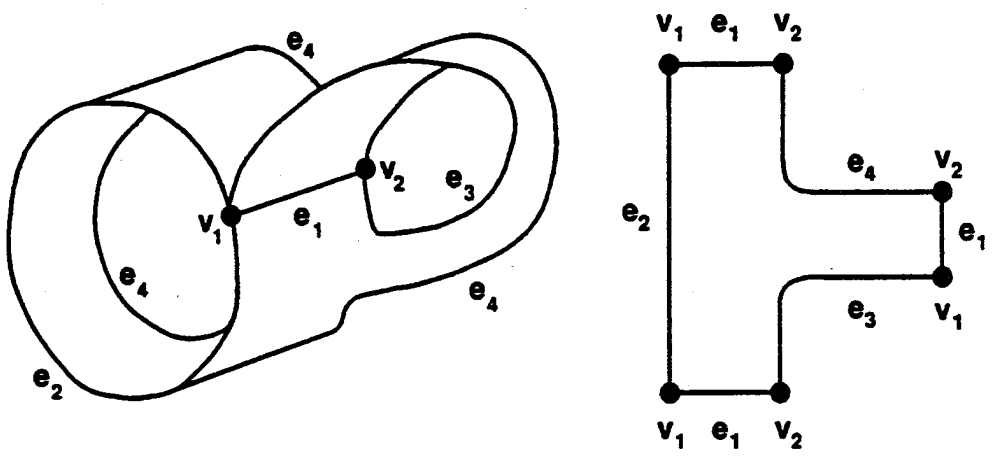


Figure 19 – 7. A non-manifold face using an edge three times

For uniformity of syntax, all of the traversal and relevant access operators include status variable parameters, even though they are not required in some implementations of some operators.

19.4.1. Query Operators

Single queries to adjacency relationship information can be classified according to whether the requested information involves downward or upward hierarchical adjacency relationships. Repeated queries can form the basis for full traversals.

19.4.1.1. Downward Hierarchical Accesses

```
E(V)
    get_vertex_of_edge(var vertex_ptr: v; edge_ptr: e)
    get_other_vertex_of_edge(var vertex_ptr: v; vertex_ptr: v_existing; edge_ptr: e)
L< E>
    get_edge_in_loop(var edge_ptr: e; loop_ptr: l; var status_type: status)
    get_next_edge_around_loop(var edge_ptr: e; var status_type: status)
    get_last_edge_around_loop(var edge_ptr: e; var status_type: status)
F{L}
    get_loop_in_face(var loop_ptr: l; face_ptr: f; var status_type: status)
    get_next_loop_in_face(var loop_ptr: l; var status_type: status)
S{F}
    get_face_in_shell(var face_ptr: f; shell_ptr: s; var status_type: status)
    get_next_face_in_shell(var face_ptr: f; var status_type: status)
R{S}
    get_shell_in_region(var shell_ptr: s; region_ptr: r; var status_type: status)
    get_next_shell_in_region(var shell_ptr: s; var status_type: status)
```

19.4.1.2. Upward Hierarchical Accesses

```
V{E}
    get_edge_using_vertex(var edge_ptr: e; vertex_ptr: v; var status_type: status)
    get_next_edge_using_vertex(var edge_ptr: e; var status_type: status)
E< L>
    get_loop_using_edge(var loop_ptr: l; edge_ptr: e; var status_type: status)
    get_next_loop_around_edge(var loop_ptr: l; var status_type: status)
    get_last_loop_around_edge(var loop_ptr: l; var status_type: status)
L{F}
    get_face_using_loop(var face_ptr: f; loop_ptr: l; var status_type: status)
F{S}
    get_shell_using_face(var shell_ptr: s; face_ptr: f; var status_type: status)
S{R}
    get_region_using_shell(var region_ptr: r; shell_ptr: s; var status_type: status)
```

19.4.2. Traversal Operators

Traversal operators also need status variables for the same reasons as the query operators.

Traversals can be implemented in common procedural languages such as *Ada*, *C*, *Modula*, or *Pascal* in at least two ways. The first involves explicitly utilizing the native control structures of the language along with the query operators previously given to produce the traversal. The second technique utilizes macros to provide syntactically new traversal control structures which are simpler to use than the explicit technique.

Examples of each technique are shown below.

Pascal version

```
var     loop_ptr: l;
        edge_ptr: e;
        status_type: status;

get_edge_in_loop(e,l,status);
while (e < > NIL) do begin
        .
        .
        .
        get_next_edge_in_loop(e,status)
        end;
```

macro version

```
var     loop_ptr: l;
        edge_ptr: e;
        status_type: status;

foreach_edge_in_loop(e,l,status)
        .
        .
        .
        end_foreach;
```

The traversal operators are given in four groups, utilizing the macro style specification shown above, but the output parameters are marked as Pascal *var* parameters. First, *global traversals* involve enumerating all topological elements of a given type, regardless of their positioning in the topology. Second, *downward hierarchical traversals* enumerate all elements in the lower dimension adjacent group of a reference element of higher dimension, such as the enumeration of all faces in a shell, $S\{F\}$. Third, *upward hierarchical traversals* enumerate all elements in the higher dimension adjacent group of a reference element of lower dimension, such as the enumeration of all edges around a vertex, $V\{E\}$. Fourth, *element use traversals*, relevant to systems implementing the element use concepts such as those of the Radial Edge structure, enumerate all uses of a specific element. Implementations of many of these traversals for the Radial Edge structure can be found in Appendix C.

19.4.2.1. Global Traversals

```
foreach_region_in_model(var region_ptr: r; model: m; var status_type: status)
foreach_shell_in_model(var shell_ptr: s; model: m; var status_type: status)
```

```
foreach_face_in_model(var face_ptr: f; model: m; var status_type: status)
foreach_loop_in_model(var loop_ptr: l; model: m; var status_type: status)
foreach_edge_in_model(var edge_ptr: e; model: m; var status_type: status)
foreach_vertex_in_model(var vertex_ptr: v; model: m; var status_type: status)
```

19.4.2.2. Downward Hierarchical Traversals

```
R{S}
    foreach_region_in_model(var region_ptr: r; model: m; var status_type: status)
R{S}
    foreach_shell_in_region(var shell_ptr: s; region_ptr: r; var status_type: status)
S{F}
    foreach_face_in_shell(var face_ptr: f; shell_ptr: s; var status_type: status)
F{L}
    foreach_loop_in_face(var loop_ptr: l; face_ptr: f; var status_type: status)
L< E>
    foreach_edge_in_loop(var edge_ptr: e; loop_ptr: l; var status_type: status)
E{V}
    foreach_vertex_in_edge(var vertex_ptr: v; edge_ptr: e; var status_type: status)
```

19.4.2.3. Upward Hierarchical Traversals

```
V{E}
    foreach_edge_using_vertex(var edge_ptr: e; vertex_ptr: v; var status_type: status)
E< L>
    foreach_loop_using_edge(var loop_ptr: l; edge_ptr: e; var status_type: status)
L{F}
    foreach_face_using_loop(var face_ptr: f; loop_ptr: l; var status_type: status)
F{S}
    foreach_shell_using_face(var shell_ptr: s; face_ptr: f; var status_type: status)
S{R}
    foreach_region_using_shell(var region_ptr: r; shell_ptr: s; var status_type: status)
```

19.4.2.4. Element Use Traversals

```
foreach_faceuse_in_face(var faceuse_ptr: fu; face_ptr: f; var status_type: status)
foreach_loopuse_in_loop(var loopuse_ptr: lu; loop_ptr: l; var status_type: status)
foreach_edgeuse_in_edge(var edgeuse_ptr: eu; edge_ptr: e; var status_type: status)
foreach_vertexuse_in_vertex(var vertexuse_ptr: vu; vertex_ptr: v;
                var status_type: status)
```

19.5. Building on Low Level Non-Manifold Operators

Much like the manifold Euler operators, the non-manifold operators can be used as a low level base upon which to build more complex higher level modeling operators, while insulating those new operators from the details and complexities of the actual

data structures utilized.

The same considerations apply in designing higher level non-manifold operators except that the domain is far more flexible than the manifold environment.

An example of how a geometric modeling system can be built using a layered approach is shown in Figure 19 – 8.

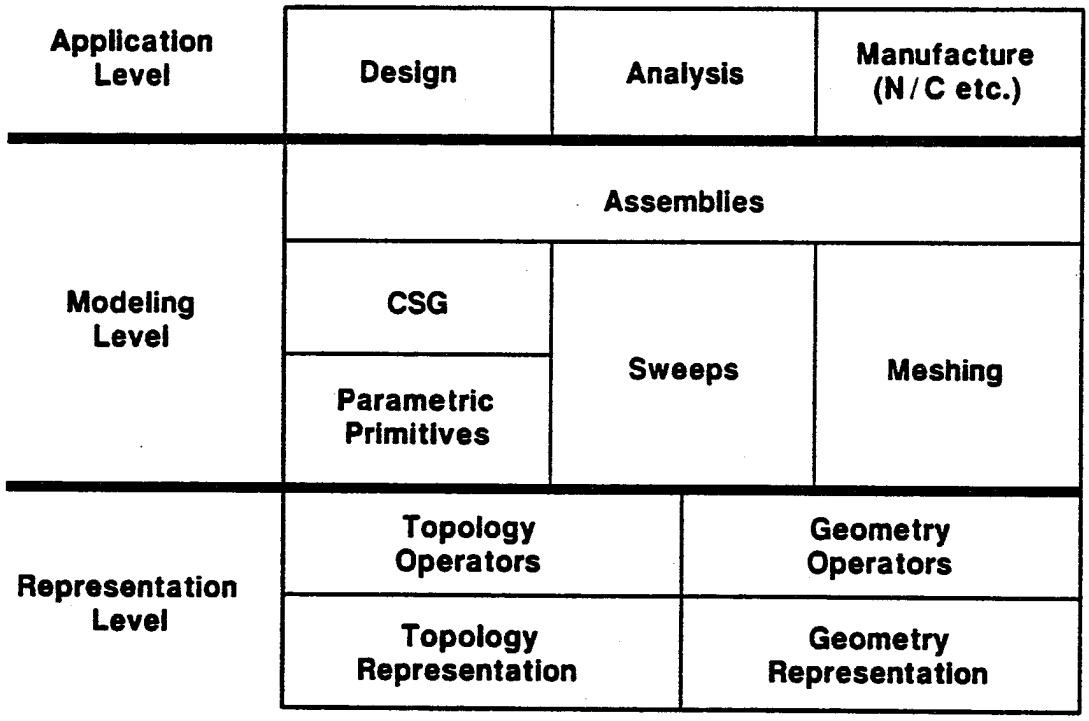


Figure 19 – 8. A layered approach to building a geometric modeling system

The goal is to have each layer build on top of lower layers, providing an increasing level of functionality with each new layer. The interface between each layer should be well specified and as independent of implementation details as possible. The same independence should be true for separate modules within each layer, although generally this is more difficult to achieve. The benefit of this approach is that such modularization, as applied to any system implementation, makes it cheaper and easier to build and maintain the system.

Figure 19 – 8 shows such ideas applied to a boundary based geometric modeling system. The lowest level, the *representation level* contains the basic representation functionality to describe the topology and geometry of the geometric shape. It is possible to interface to both topology and geometry representations through a set of interface operators, which are best designed to hide the implementation details of the actual representation as much as possible. The non-manifold topology operators are an example of such interface operators for a topology representation. The level directly above the representation level, the *modeling level,* provides the generic geometric shape manipulation facilities commonly found in current geometric modeling systems. Even within this layer some functions build on top of others. For example, CSG (Constructive Solid Geometry) operators utilize shapes created through parametric primitive and other functions. At the highest level, the *application level,* specific application implementations build on top of the generic modeling functionality provided by the modeling level.

A few additional issues concerning non-manifold topology representations and typical geometric modeling operations are also worthy of note.

A particular problem which has caused a great deal of complexity in implementations of the Boolean set operations for boundary implementations utilizing a manifold representational base is how to maintain the manifold state of each of the operands while still keeping track of how and where they intersect. A non-manifold implementation can directly intersect the objects in the same representational space and then prune away the undesired portions of the result, greatly simplifying the process.

Propagation of intersection information to adjacent elements would be automatically performed by the low level operators.

A closed form version of the Boolean set operations is feasible in the non-manifold environment. For applications desiring regularized set operations [Requicha & Voelcker 77], a regularizing function can be applied to the non-regular output. Construction of a regularizing function for the Radial Edge structure in particular is a simple task.

Similarly, construction of a function to keep only manifold parts of a model, if any, is a simple task when the Radial Edge structure is the representational base.

Chapter 20

THE INTERFACE BETWEEN TOPOLOGY AND GEOMETRY

This thesis concentrates on the use of topological information as a framework for geometric modeling representations, and therefore it treats topological issues in detail. Nevertheless, boundary representation geometric modeling *systems* must combine both topological and geometric information together to form a complete and cohesive representation of the shape of a three-dimensional model.

In the context of the types of geometric modeling systems of interest in this thesis, topology refers to the explicit storage of topological boundary adjacency relationship information, and geometry refers to the geometric surface, curve, and point definitions.

In general, geometric surface representations which are restricted to planar surfaces, and therefore restricted to straight line edges, have some extremely convenient properties; one of the most important is that curves of intersection between planar surfaces are also straight lines. In most curved surface representations, in general, intersections between surfaces create higher order intersection curves than those originally in the model. This causes a great deal of complexity in curved surface geometry systems. Nonetheless, there is a great need for curved surface representations which has stimulated much research in this area.

The intention of this chapter is to briefly point out a few of the requirements and problems in coordinating topology and curved surface geometry that currently appear to be neither trivial nor well understood. Additionally, the natural correspondence between parametric space description of parametric curved surface intersections and the direct representation of the *uses* of topological edge and vertex elements in

SECTION IV

TOPOLOGY AND GEOMETRY INTERFACE

adjacency topology representations is briefly discussed.

20.1. Problems in Coordinating Topological and Geometric Information

Combining explicit topology with geometry can help in the geometric modeling process, but it does not solve all geometric problems, since the topology is not independent information but is a reflection of the geometric information. In fact, "evaluating" a boundary representation from a procedural representation (such as CSG) is often a difficult, complex geometric task.

In general, for a representation involving an object based evaluated boundary model which explicitly stores topological adjacency information, geometric modeling operators creating and manipulating models in a complete modeling system must:

- determine the topology (topological boundary descriptions) of the result
- determine the geometry (geometric surface descriptions) of the result
- ensure that the geometry corresponds unambiguously to the topology

There does not currently appear to be a best ordering to these tasks, and sometimes they are most conveniently accomplished simultaneously.

One set of difficulties encountered in guaranteeing the correspondence between topology and geometry stems from surface intersection operations, where curves of intersection with singular points can occur at self intersections and cusps [Farouki 86] (see Figure 20 – 1). The surface intersection problem is a difficult one and current solutions are not necessarily trivial or robust processes. To maintain the topological domain restriction of non-intersection, the intersection curve, once found, must be segmented at all self-intersections. Once this has been done, the curve geometry segments of the original intersection curve geometry must then be appropriately associated with corresponding topology edges. This means that some geometric technique must be provided that uniquely specifies which of the various curve geometry segments and their orientations should be used at any given point on the topology boun-

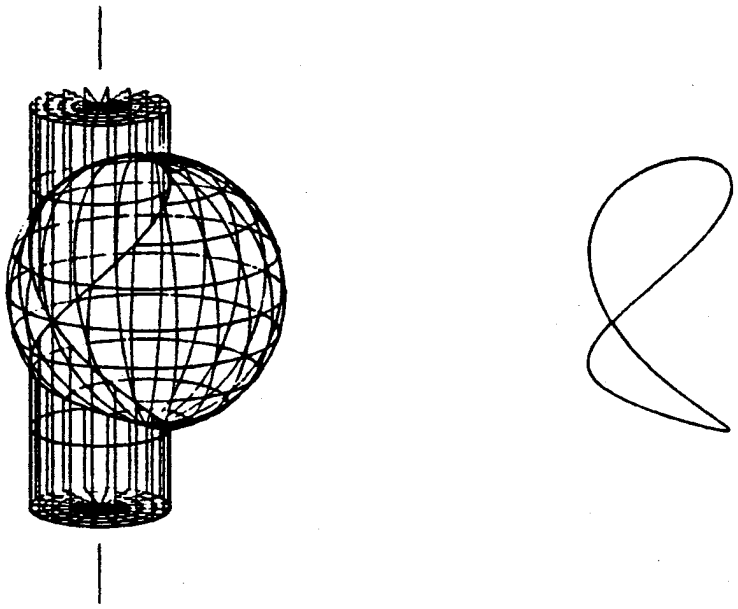


Figure 20 – 1. Example of a self-intersecting curve of intersection

dary graph.

The next two subsections discuss this last problem with respect to implicit and parametric surface geometry formulations.

20.1.1. Implicit Formulations

For implicit geometric surface representations in particular, geometrically differentiating the various geometric curve segments and uniquely identifying them so they can be associated with corresponding topology can be non-trivial [Hoffman & Hopcroft 86]. Two solutions to this problem have been proposed:

1. When referring to a specific curve segment, refer to the original curve geometry *and* a point on the *interior* of the curve segment to indicate which segment

is intended. Since all self-intersections have already been identified and therefore differentiated from the interiors of all new edge segments by the segmentation process, this uniquely identifies the proper segment [Requicha 80] (see Figure 20 – 2a).

2. Refer to the original curve geometry, a curve segment endpoint, and a tangent from that endpoint to indicate which curve segment is intended out of the several incident to the specified point [Hoffman & Hopcroft 86] (see Figure 20 – 2b).

The first method will work all of the time, but does not appear to be a convenient formulation and can be computationally intensive. The second method is computationally more convenient, but is admittedly not guaranteed to work all of the time, as

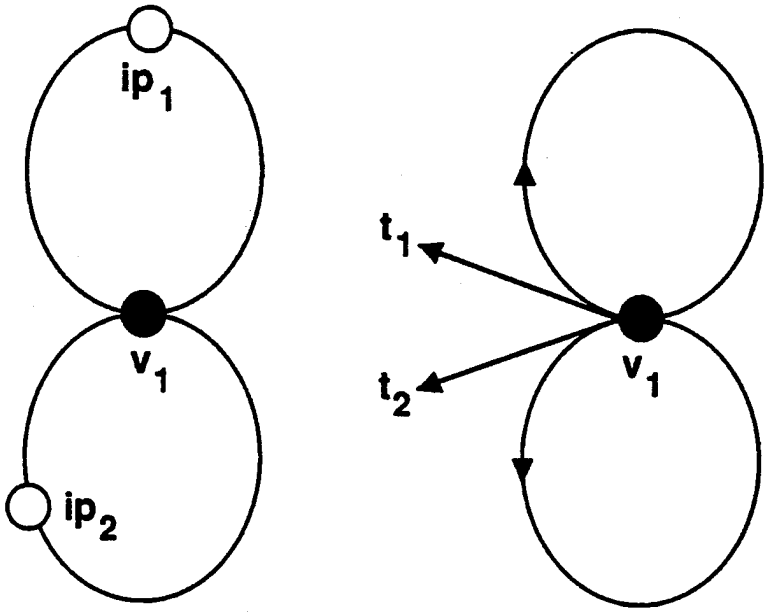


Figure 20 – 2. Techniques to uniquely identify implicit geometry curve segments

in the case of cusps at the curve segment endpoint, where higher order derivative information is required to distinguish the segments.

From a practical standpoint, unless the interior points are carefully selected, even the first method presented can fail to give unique results in some situations due to numerical precision problems (see Figure 20 – 3).

Determining the theoretical minimal information required to differentiate between the curve segments in an implicit geometric representation is an important part of understanding the problem. In the final analysis, however, it is the computational convenience of a differentiation technique that will be most important in determining the usefulness of a specific geometric surface representation in a topological framework based

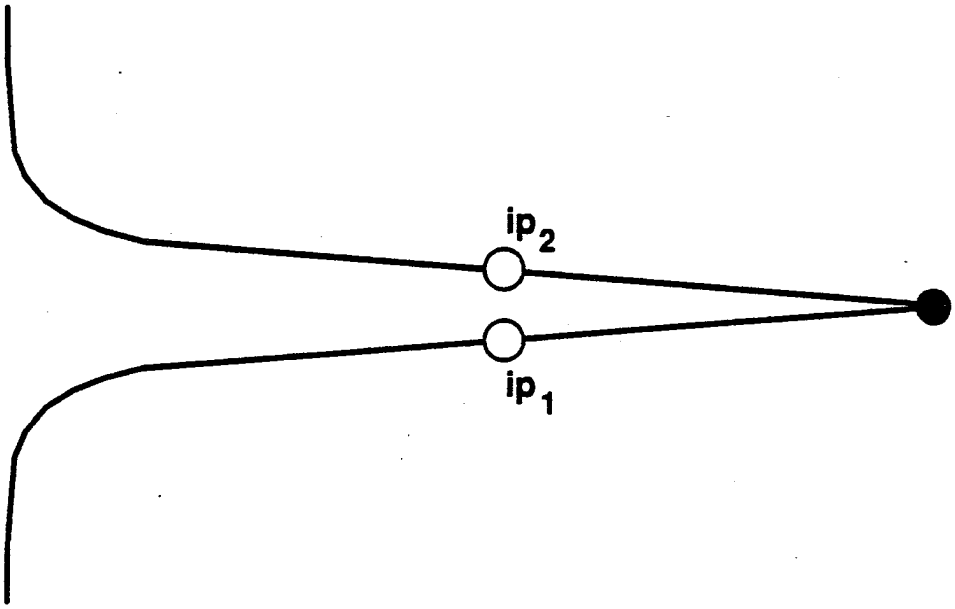


Figure 20 – 3. Curve segment specification prone to numerical precision problems

geometric modeling representation.

20.1.2. Parametric Formulations

There appear to be more options available to address the differentiation problem for parametric geometric representations, since the three-dimensional surface boundary curve segments can be described as separate formulations of two dimensional geometric curves in parametric space. In this way curve segments can be associated with unique geometry, which happens to be in parametric space. Naturally, accuracy problems can also occur in differentiating geometry in parametric space.

20.2. Representation of Intersection Curves with Parametric Geometry

There are often no analytic formulations for intersections of parametric surfaces; the most accurate representation of the curve of intersection remains a description of the intersection in the parametric spaces of *both* geometric surface definitions.

A problem with many current manifold boundary topology representations is that trying to maintain correspondence between the topology and geometry is difficult if one has only one place (a single topology edge record) to associate all curve geometry related to each of the originating intersecting parametric surfaces. In this case, a procedural determination of which curve geometry (including curve orientation) from which surface parameter space is associated with each use of the topological edge must be made whenever the geometry is referenced through the topology. The edge-use structure approach of the F-E data structure (Section 12.6) solves this problem in that the *use* of each edge is associated with each surface (in fact, with a specific orientation of the edge bounding that specific surface), and this is therefore the appropriate place to store a specific reference to the parametric geometry information (see Figure 20 – 4).

It is also possible to maintain exact correspondence between non-manifold topologies

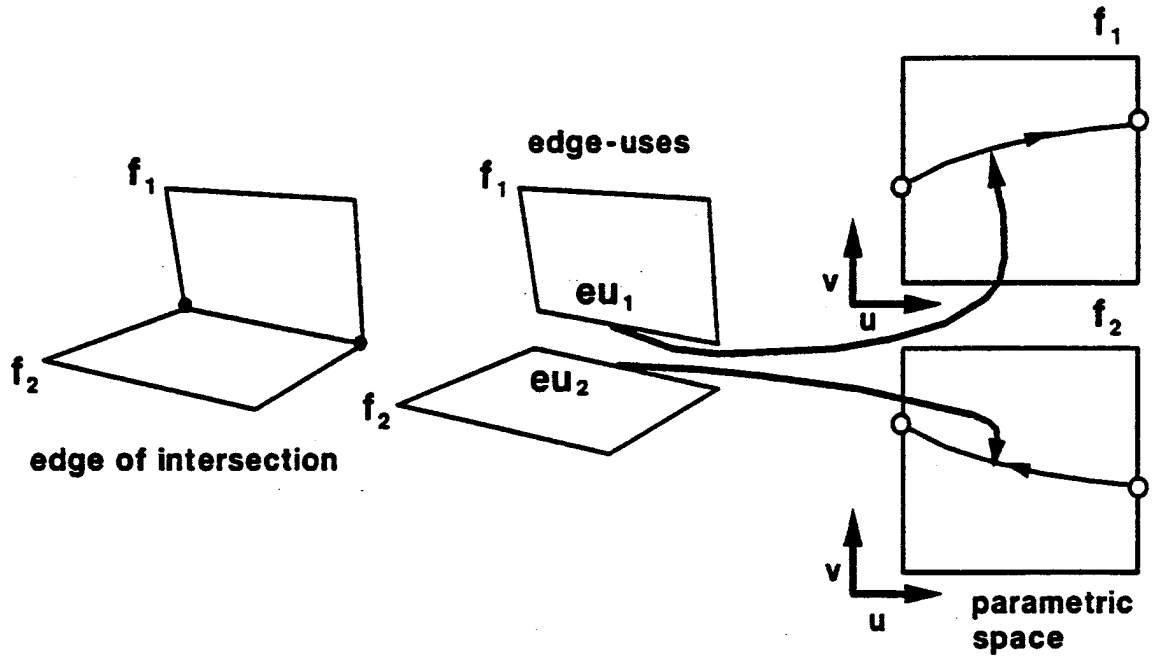


Figure 20 – 4. Correspondence between parametric geometry and edge uses in a manifold environment

and parametric geometry in a similar fashion by utilizing the edge-use structures of the Radial Edge non-manifold topology [Farouki & Weiler 86]. The difference of the non-manifold environment from a manifold environment is that both sides of each surface must be considered rather than just the one side necessary for manifold topologies. Therefore each use of the edge by a surface has two orientations. The Radial Edge structure has an edge-use structure for each of the orientations of the edge for each surface using the edge. Thus there is a unique place to put references to each parametric space description of the curve segments because there is a one-to-one correspondence of the topology with the geometry (see Figure 20 – 5).

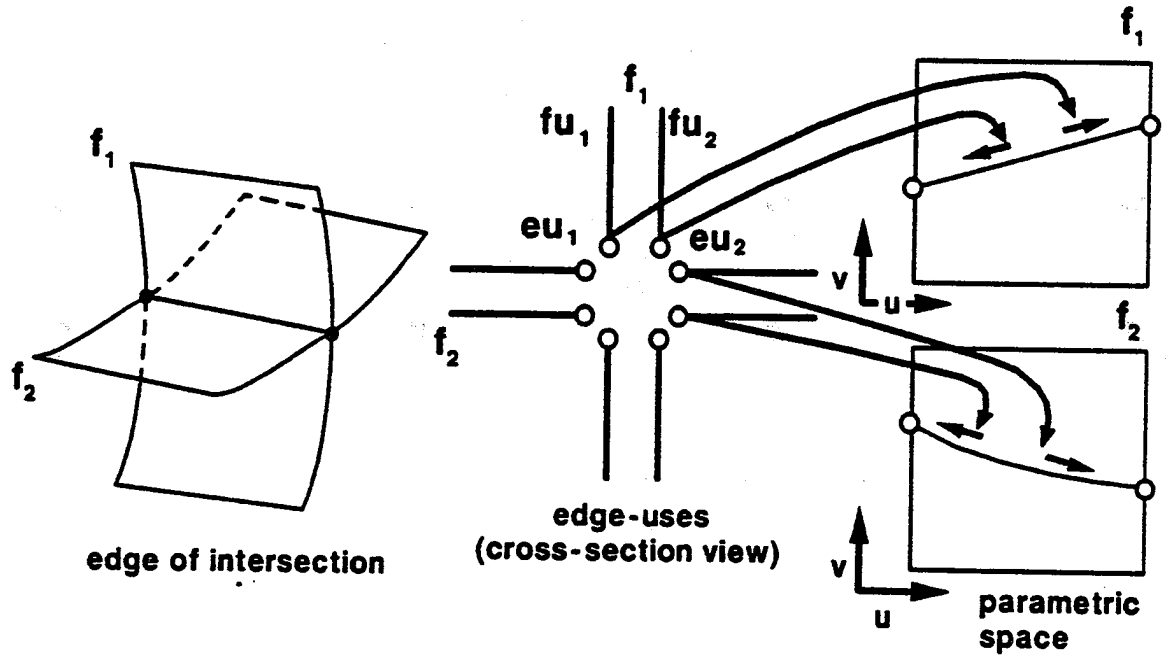


Figure 20 – 5. Correspondence between parametric geometry and edge uses in a non-manifold environment

Again, numerical accuracy problems can arise in parametric space as easily as in geometric space, but these should be detected and resolved in parametric space when surface and intersection calculations are made.

SECTION V

CONCLUSION

Chapter 21

CONCLUSION

This thesis has provided a detailed look at the topological aspects of geometric modeling boundary representations, from both a theoretical and practical viewpoint.

This chapter notes what I believe are the new and original contributions of this work to the geometric modeling field, as well as some related areas for future exploration.

21.1. Contributions

Probably the three most significant contributions of this thesis are the development of a theoretical foundation for manifold topology boundary modeling representations, the development of the non-manifold Radial Edge topology structure, and the development of the non-manifold topology operators.

Some general contributions include:

- renewed emphasis on the use of topology as a framework for modeling system design with stress on the consideration of both theoretical and practical concerns.
- development of a new geometric modeling representation classification system.
- development of a new comprehensive adjacency relationship terminology applicable in manifold and non-manifold domains which describes essential characteristics of topological adjacency relationships necessary for discussions of topological sufficiency.

Some contributions specifically related to manifold solid modeling are:

- development of a theoretical basis for object based evaluated manifold topology boundary modeling by establishing the theoretical minimal and practical minimal sufficient topological information.
- proof that the winged edge structure can be considered sufficient under the specified curved surface domain but requires additional complex procedures and storage space to be properly implemented.
- development of three new data structures for manifold solid modeling and proof of their topological sufficiency.

Contributions specifically related to non-manifold geometric modeling are:

- a new emphasis on non-manifold geometric modeling as a viable boundary modeling representation, with particular emphasis on its primary benefits of allowing a unified representation of wireframe, surface, and solid modeling forms simultaneously in the same environment, while increasing the representable range beyond what is achievable in any of the previous modeling forms.
- development of the first non-manifold geometric boundary modeling adjacency topology representation and proof of its completeness.
- development of the first non-manifold boundary topology modeling operators.

Contributions related to coordinating topological and geometric information in a modeling system are:

- discussion of how the direct representation of *uses* of topological edge and vertex elements in adjacency topology representations provide a natural and complete basis for coordinating multiple parametric descriptions of the same edge in situations like the intersection of multiple parametric surfaces.

21.2. Areas for Future Development

There is always room for further development and exploration of a topic. There are still several major issues in the geometric modeling field, including some important ones relating to adjacency topology boundary representations. Below are several open problems which are strongly related to the work described in this thesis.

Related to adjacency topology representations:

- determination of the theoretical minimal information for non-manifold topological sufficiency. This is the most outstanding unsolved problem in non-manifold geometric modeling representations as they are defined here. A solution to this problem can lead to a better understanding of non-manifold representations and could lead to new data structures. Proof of sufficiency of the Radial Edge structure and the non-manifold topology operators are linked to this open problem.
- proof of sufficiency of the Radial Edge structure. Since completeness has been shown here, once theoretical sufficiency has been shown it should be a simple matter to prove sufficiency.
- proof of sufficiency of the non-manifold topology operators and determination of minimal sufficient sets of operators to cover the entire representational space.
- further study of the concept of *element uses* in adjacencies and of *correspondence* may yield a deeper understanding of some of the topological issues in both manifold and non-manifold domains.

Related to general geometric modeling issues:

- determination of techniques to obtain topologically consistent answers in surface to surface intersections, including across surface junctures. Techniques to determine the qualitative topological characteristics of geometric surface intersections without requiring absolute accuracy in calculations would go a long way toward this goal.
- geometric accuracy will be a long standing issue for geometric modeling.

Among others, a major problem is the use of approximate number representation schemes without closed form arithmetic operations while still having the expectation of consistent results. This and other major unsolved geometric calculation accuracy problems will continue to haunt the computing field until ameliorating approaches are codified or true solutions are developed.

- more formal analysis of the modeling problem needs to be done. There are still gaps between formal mathematical theory and current geometric modeling and computing technology that need to be bridged. The geometric modeling area is an interdisciplinary one, where practitioners from several fields must work together. Bridging the gaps between these fields is complicated more by the different needs and interests historically developed in these fields than by the different terminologies developed. More inter-field cross-fertilization and multi-disciplinary research teams will be required to improve in this area.

LITERATURE CITED

LITERATURE CITED

[Agoston 76] Agoston, M., *Algebraic Topology*, Marcel Dekker, NY 1976.

[Arnold 62] Arnold, B., *Intuitive Concepts in Elementary Topology*, Prentice-Hall, Englewood Cliffs, 1962.

[Baer et al 79] Baer, A., Eastman, C., and Henrion, M., "Geometric Modelling: a Survey," CAD, Vol. 11, No. 5, September, 1979, pg. 253-272 .

[Baumgart 72] Baumgart, B., "Winged-edge Polyhedron Representation," Stanford Artificial Intelligence Report No. CS-320, October 1972.

[Baumgart 74] Baumgart, B., "GEOMED — a Geometric Editor," Stanford Artificial Intelligence Laboratory, AIM-232, May, 1974.

[Baumgart 75] Baumgart, B., "A Polyhedron Representation for Computer Vision," Proceedings of the National Computer Conference, 1975.

[Braid et al 78] Braid, I., Hillyard, R., and Stroud, I., "Stepwise Construction of Polyhedron in Geometric Modelling," CAD Group Document No. 100, University of Cambridge Computer Laboratory, October 1978.

[Braid 79] Braid, I., "Notes on a Geometric Modeller," CAD Group Document No. 101, University of Cambridge Computer Laboratory, June 1979.

[Eastman & Henrion 77] Eastman, C., and Henrion, M., "GLIDE: A Language for Design Information Systems," Computer Graphics, Vol. 11, No. 2, July 1977, pg. 24-33.

[Eastman & Thornton 79] Eastman, C., and Thornton, R., "A Report on the GLIDE2 Language Definition," CAD Group, Institute of Physical Planning, Carnegie-Mellon University, March 1979.

[Eastman & Weiler 79] Eastman, C., and Weiler, K., "Geometric Modeling Using the Euler Operators," Conference on Computer Graphics in CAD/CAM Systems, May 1979, pg. 248-259.

[Edmonds 60] Edmonds, J., "A Combinatorial Representation for Polyhedral Surfaces," American Mathematical Society Notices, Vol. 7, October 1960, pg. 646.

[Farouki 86] Farouki, R., "A Characterization of Parametric Surface Sections," Computer Vision, Graphics, and Image Processing, February, 1986.

[Farouki & Weiler 86] Farouki, R., and Weiler, K., "Proposal: A Boundary Modeler Utilizing Non-Manifold Topology and Analytic Trimmed Surface Geometry," CAD Branch, GE Corporate Research and Development, Internal Document.

[Graver & Watkins 77] Graver, J., and Watkins, M., *Combinatorics with Emphasis on the Theory of Graphs*, Springer-Verlag, N.Y., 1977.

[Hanrahan 82] Hanrahan, P., "Creating Volume Models from Edge-Vertex Graphs," Computer Graphics, Vol. 16, No. 3, July, 1982, pg. 77-84.

[Harary 72] Harary, F., *Graph Theory*, Addison-Wesley, Reading, MA., 1972.

[Hoffman & Hopcroft 86] Hoffman, C., and Hopcroft, J., "Geometric Ambiguities in Boundary Representations," TR 86-725, Department of Computer Science, Cornell University, January 1986.

[Mantyla 81] Mantyla, M., "Methodological Background of the Geometric Work-

bench," Report-HTKK-TKO-B30, Laboratory of Information Processing Science, Helsinki University of Technology, 1981.

[Mantyla 84] Mantyla, M., "A Note on the Modeling Space of Euler Operators," Computer Vision, Graphics and Image Processing, 26, 1984, pg. 45-60.

[Mantyla & Sulonen 82] Mantyla, M., and Sulonen, R., "GWB: A Solid Modeler with the Euler Operators," IEEE Computer Graphics, Vol. 2, No. 7, Sept. 1982, pg. 17-31.

[Markowsky & Wesley 80] Markowsky, G., and Wesley, M., "Fleshing Out Wireframes," IBM Journal of Research and Development, Vol. 24, No. 5, Sept. 1980, pg. 582-597.

[Nordhaus 72] Nordhaus, E., "On the Girth and Genus of a Graph," Graph Theory and Applications, Lecture Notes in Mathematics, Springer-Verlag, 1972, pg. 207-214.

[Requicha 77] Requicha, A., "Mathematical Models of Rigid Solid Objects," Production Automation Project Tech. Memo 28, Univ. Rochester, Nov. 1977.

[Requicha 80a] Requicha, A., "Representations of Rigid Solids - Theory, Methods, and Systems," ACM Computing Surveys, Vol. 12, No. 4,1980.

[Requicha 80b] Requicha, A., "Representations of Rigid Solid Objects," in *Computer Aided Design*, Lecture Notes in Computer Science, No. 89, Encarnacao, J., (Ed.), Springer Verlag, New York, 1980, pg. 2-78.

[Requicha & Tilove 78] Requicha, A., and Tilove, R., "Mathematical Foundations of CSG: General Topology of Regular Closed Sets," Production Automation Project Tech. Memo 27, Univ. Rochester, Mar. 1978.

[Requicha & Voelcker 77] Requicha, A., and Voelcker, H. "Constructive Solid Geometry," Production Automation Project Tech. Memo 25, Univ. Rochester, Nov. 1977.

[Requicha & Voelcker 83] Requicha, A., and Voelcker, H. "Solid Modeling: Current Status and Research Directions," IEEE Computer Graphics and Applications, Vol. 3. No. 7, October, 1983.

[Stoker 74] Stoker, D. "CRIPL-Edge Data Structure," unpublished, Carnegie-Mellon Univ., May 1974.

[Weiler 83] Weiler, K. "Adjacency Relationships in Boundary Graph Based Solid Models," June 1983, General Electric internal report (to be submitted for publication).

[Weiler 84] Weiler, K. "Topology as a Framework for Solid Modeling," Proceedings, Graphics Interface '84, Ottawa, Ontario, May 1984 (extended abstract).

[Weiler 85a] Weiler, K. "Edge Based Data Structures for Solid Modeling in Curved-Surface Environments," IEEE Computer Graphics and Applications, Vol. 5. No. 1, January, 1985.

[Weiler 85b] Weiler, K. "The Radial Edge Structure: a Topological Representation for Non-Manifold Geometric Modeling," January, 1985, General Electric internal report (to be submitted for publication).

[Weiler 85c] Weiler, K. "Boundary Graph Operators for Non-Manifold Geometric Modeling Representations," October, 1985, General Electric internal report (to be submitted for publication).

[White 73] White, A., *Graphs, Groups, and Surfaces,* Mathematical Studies 8, North Holland, Amsterdam, 1973.

[Whitney 32] Whitney, H., "Congruent Graphs and the Connectivity of Graphs,"

American Journal of Mathematics, No. 54, 1932, pg. 150-168.

[Woo 84] Woo, T., "A Combinatorial Analysis of Boundary Data Structure Schema," Dept. of Industrial & Operations Engineering, Tech. Report 84-12, Univ. of Michigan, Apr. 1984.

[Young 63] Young, J., "Minimal Imbeddings and the Genus of a Graph," Journal of Mathematics and Mechanics, Vol 12, No. 2, 1963, pg. 303-315.

APPENDICES

APPENDIX A

Appendix A

TOPOLOGICAL SUFFICIENCY UNDER CONSTRAINTS

This appendix considers topological sufficiency for manifold topologies with more restricted domains than the one described in Section II.

1. Sufficiency Under Constraints

It is interesting to consider a domain slightly different than that initially specified in Chapter 9 in Section II to discover if more convenient or simpler representations or input forms exist for special situations. Under certain constraints even unordered adjacency relationships can be used to form complete topology models for connected topologies. This is equivalent to finding transforms which convert data from the form of unordered element adjacency relationships into data in the form of ordered element adjacency relationships (which are sufficient to describe polyhedron topologies) but which can operate only if certain constraints are met. Finding such constraints is useful for situations which benefit from minimal input or partial information, such as in interactive CAD input of solid models of mechanical parts using topological techniques. Determining *all* of the constraints which apply to a given situation is vital, however, to ensure the correctness and unambiguity of the resulting model.

The discussion here only deals with purely topological techniques, leaving out hybrid approaches such as [Markowsky & Wesley 80] which utilize geometric as well as topological information. In the absence of topological information other than simple connectivity, it is obvious that additional information is necessary for handling disconnected graphs. However, the purely topological discussion provided here may also provide a basis for other hybrid techniques which utilize other information sources

only as absolutely necessary.

2. Disallowing Multigraphs and Self Loops

As mentioned in Chapter 11, The $V<V>$ and $V<F>$ element adjacency relationships, while in general individually insufficient for unambiguously representing the topologies of polyhedra, are sufficient if multigraphs and self loops are disallowed from the boundary graph representations (see Figure 11 – 13). This constraint is equivalent to requiring that the adjacent groups of the $E\{V\}$ element adjacency relationship uniquely specify their reference edge element. These restrictions are by definition satisfied for systems which model only planar faced polyhedra and disallow curved surfaces; this constraint is therefore an interesting one for planar faced polyhedra representation systems.

Note that the ability of the adjacent groups of $E\{V\}$ to uniquely identify its edge reference element does not imply the sufficiency of $E\{V\}$ to represent polyhedral topologies. Ordering information necessary for sufficiency is still absent.

Requiring the identity of an edge to be uniquely identifiable from its endpoints is equivalent to disallowing multigraphs and self loops. Under these constraints $V<V>$ information is equivalent to $V<E>$ information, and $F<V>$ information is equivalent to $F<E>$ information. $V<E>$ and $F<E>$ were already proven to be sufficient under all of the conditions identified in Chapter 9. The proofs follow.

Theorem A-1: When an edge is constrained to be uniquely identifiable from its $E\{V\}$ adjacent group information, then the $V<V>$ adjacency relationship is sufficient to unambiguously represent the adjacency topologies of curved surface polyhedra.

proof: Using the endpoints of an edge to uniquely identify an edge, a simple algorithm can be constructed to label the edges of the embedded graph from the $V<V>$ information by first constructing the $E\{V\}$ relationship from $V<V>$. Once this has been done, it is a simple matter to construct the

$V< E>$ information for each vertex v_i by placing in order in a new $v_i< E>$ adjacent group the edges identified by the unordered pair $\{v_i\ v_i< V>_j\}$ for $j \leftarrow 1..|v_i< V>|$. The identity of the edge can be found by searching the $E\{V\}$ relationship for an adjacent group matching the $\{v_i\ v_i< V>_j\}$ information since only one edge will have the matching group. When all members of the adjacent groups of the $V< V>$ information for all of the vertices have been used in this way, then the $V< E>$ information for the embedded graph has been produced. By the Edmonds theorem, this is sufficient to unambiguously represent the topologies of polyhedra.

Theorem A-2: When an edge is constrained to be uniquely identifiable from its $E\{V\}$ adjacent group information, then the $F< V>$ adjacency relationship is sufficient to unambiguously represent the adjacency topologies of curved surface polyhedra.

proof: Similar to the proof directly above, an algorithm can be constructed to first label the edges of the embedded graph from the $F< V>$ information alone by labeling an edge for every two consecutive vertices $\{f_i< V>_j\ f_i< V>_{j+1}\}$ in the cyclic lists of vertices of the adjacent groups of the $F< V>$ relationship. Using this technique one would first create the $E\{V\}$ relationship. Once the edges have been labeled in this way, using the $E\{V\}$ information, $F< V>$ can be easily converted into the $F< E>$ relationship which by Theorem 11-2 is sufficient.

3. Unique E{F} Adjacent Groups

The $V< F>$ and $F< F>$ element adjacency relationships, normally individually insufficient for the representation of polyhedral topologies under the constraints identified in Chapter 9, become sufficient if they are additionally constrained so that the $E\{F\}$ adjacency relationship of the boundary graph can be guaranteed to uniquely identify the reference edge.

Theorem A-3: When an edge is uniquely identifiable from its $E\{F\}$ adjacent group, then the $V<F>$ adjacency relationship is sufficient to unambiguously represent the adjacency topologies of curved surface polyhedra.

proof: Since the identity of an edge is constrained to be uniquely identifiable from its two adjacent faces, a simple algorithm can be constructed to label the edges of the embedded graph from the $V<F>$ information by constructing the $E\{F\}$ relationship from $V<F>$. Once this has been done, it is a simple matter to construct the $V<E>$ information for each vertex v_i by placing in order in a new $v_i<E>$ adjacent group the edges identified by the unordered face pair $\{v_i<F>_j\ v_i<F>_{j+1}\}$ found in the cyclic ordered adjacent group of the $V<F>$ information for v_i. The identity of the edge can be found by searching the $E\{F\}$ relationship for an adjacent group matching the $\{v_i<F>_j\ v_i<F>_{j+1}\}$ information since only one edge will have the matching group. When all members of the adjacent groups of the $V<F>$ information for all of the vertices have been used in this way, then the $V<E>$ information for the embedded graph has been produced. By the Edmonds theorem, this is sufficient to unambiguously represent the topologies of polyhedra.

Theorem A-4: When an edge is uniquely identifiable from its $E\{F\}$ adjacent group, then the $F<F>$ adjacency relationship (*FF* definition A) is sufficient to unambiguously represent the adjacency topologies of curved surface polyhedra.

proof: Similar to the proof directly above, since the unordered set of two faces adjacent to an edge are constrained to uniquely determine the identity of that edge, an algorithm can be constructed to label the edges of the embedded graph from the $F<F>$ information alone by labeling an edge for every set $\{f_i\ f_i<F>_j\}$, $i \leftarrow 1..n, j \leftarrow 1..n$, consisting of the reference face f_i and each adjacent face f_j in the cyclic list of faces in the adjacent group of the $f_i<F>_j$ relationship. Using this technique one would create the $E\{F\}$ relationship. Once the edges have been labeled in this way, using the $E\{F\}$

information, $F< F>$ can be easily converted into the $F< E>$ relationship which by Theorem 11-2 is sufficient.

The uniqueness of the $E\{F\}$ information of an edge might be enforceable by additional connectivity constraints, but this would limit the representational range of the modeling technique considerably, as 2-connected objects are not unusual in modeling applications (see Figure A – 1). Thus constraints to guarantee the uniqueness of the $E\{F\}$ information of an edge are less likely to be as workable in actual modeling systems as constraints to guarantee the uniqueness of the $E\{V\}$ information of an edge, since unique $E\{V\}$ information can be at least artificially maintained more easily than unique $E\{F\}$ information without reducing representational range.

4. Sufficiency with Connectivity Information

Simple connectivity information, which is equivalent to the information in an unmapped graph (and some of the unordered element adjacency relationships), is normally not sufficient to derive unique mappings. Under certain conditions, however, sufficiency can be achieved.

4.1. The Three-connected and Planar Constraint for Graphs

The $V\{V\}$ unordered element adjacency relationship and other members of its equivalence class (such as $E\{V\}$) are essentially just the connectivity information that is normally associated with unmapped graphs. A unique mapping of a graph to the surface of a sphere (note that the mapping information corresponds to the ordering information not present in $V\{V\}$) can be derived from $V\{V\}$ under the following constraints:

- the graph is not a pseudograph (no multiple edges or self loops)
- the graph is three-connected
- the graph is planar

Discovery of this relationship is attributed to Whitney [Whitney 32]. Under the constraints of being three-connected and not a pseudograph, there is only one embedding which can be constructed from the connectivity information without violating the planar constraint, thus providing a method of constructing the embedding. Note that three-connected here refers to vertex connectivity and not edge connectivity or vertex degree. This has been confused by some practitioners developing solid input algorithms. An algorithm which can find the unique mapping of a constrained graph from its connectivity information alone, or report that the graph is nonplanar or less than three connected has been described in [Hanrahan 82].

A few remarks are in order on the applicability towards solid modeling input of mapping techniques based solely on these constraints. The planarity constraint, while not desirable, is a concept that is readily understood by users developing solid models since it is a characteristic basic to the user concept of the shape of a solid. The connectivity constraint, however, restricts the class of solids that can be input by this technique in ways that are not always intuitively obvious to the user. Figure A – 1 shows an example of one such solid which would not be immediately perceived by many as violating the connectivity restriction.

Every two-connected subgraph in a graph has two possible mappings to a surface. The ambiguity caused by the two-connected subgraph in Figure A – 1 can usually be resolved by using geometric information normally available from the input, but discussion of such techniques is outside the scope of this appendix.

4.2. Removing Constraints

There are undoubtedly other restricted classes of graphs for which unique mappings can be derived from connectivity information alone. More interesting, however, would be the relaxation of the constraints, and the development of a practical general algorithm to derive valid mappings for any class of polyhedra boundary graphs.

Relaxing the connectivity constraint alone allows ambiguity to occur on exactly how a

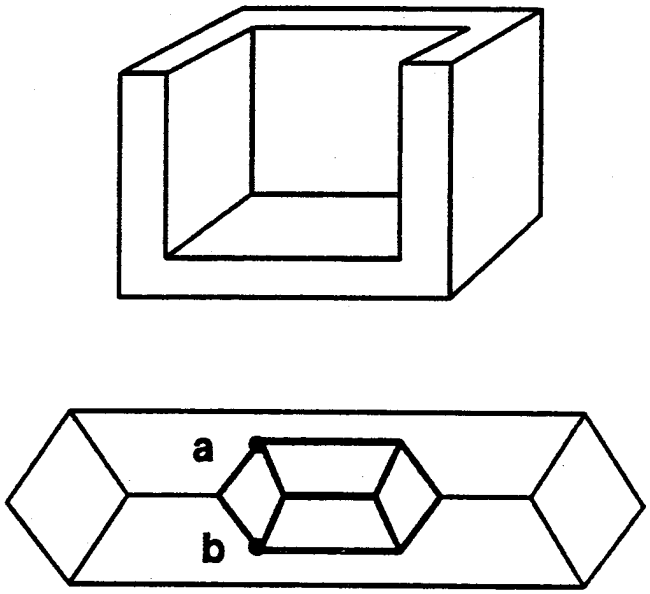


Figure A-1. An object and its 2-connected boundary graph

graph can be mapped to a surface. As seen previously, a two-connected subgraph of a graph can be mapped two different ways. A one-connected subgraph can be mapped in at most $n-1$ ways if n is the degree of the vertex at which the subgraph makes its connection. While these ambiguities cannot be resolved by purely topological techniques, the different mappings can be enumerated easily once the vertices which attach such subgraphs are known. Other techniques may then be used to select from the alternatives.

Relaxing the planarity (genus) constraint is a little less straightforward. As soon as the genus of the object being represented is unconstrained or unknown, two new

complications come into play.

First, the same graph may be mappable onto surfaces of different genus. Second, even within each genus there may be several alternative mappings. The exhaustive way of determining the desired genus and embedding of a graph is to start enumerating the alternate mappings until a valid mapping has been selected (by non-topological techniques) or all embeddings have been enumerated. The Edmonds embedding technique can be used to enumerate all embeddings for a given graph by performing permutations on the order in the adjacent groups of constructed $V< E>$ information. The embeddings with the maximum number of faces will have the minimum genus, and those with the minimum number of faces will have the maximum genus [Young 63]. Direct calculation of the lower and upper bound of the genus of graphs can be performed for some types of graphs from the graph characteristics (see [Nordhaus 72]), but in general these quantities are not easy to determine. It also does not appear to be well understood how to enumerate all of the possible embeddings within a given genus. The only cases in which the genus of a graph are known to be unique (the maximum genus equals the minimum genus) represent a small subset of the possible planar graphs [Nordhaus 72].

Possible relationships between the connectivity and unique mappings of non-planar graphs are also not well understood. Simply increasing the connectivity constraint in direct proportion to the increase in genus does not necessarily ensure a unique mapping. An example is the well known hypercube (Figure A – 2), in this case an object of genus one and connectivity of four which has three equally valid yet distinct mappings.

Relaxing the non-pseudograph constraint also causes problems when trying to determine graph embeddings. For example, there is no topological way to determine which of the several faces adjacent to a vertex should contain a self loop attached to that vertex given connectivity information alone (Figure A – 3). Determining the order of multiple edges between two vertices gives rise to similar problems. In this case it is reasonably easy to enumerate the possibilities however.

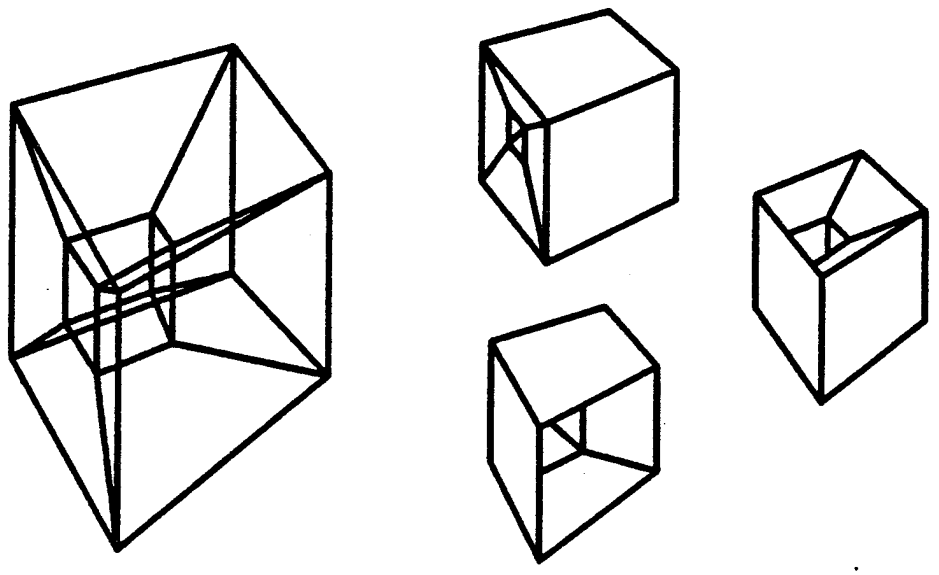


Figure A-2. The hypercube

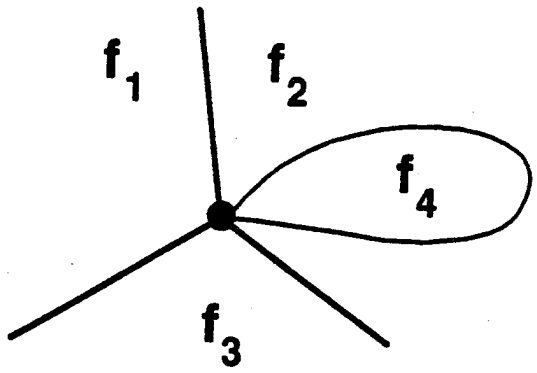


Figure 3. Self loop located at a vertex shared by several faces

Any system capable of defining unrestricted polyhedron definitions from connectivity information would need the capability of enumerating the topologically possible mappings, and allow choices between mappings based on non-topological information. Efficient systems would try to eliminate having to examine all of the alternatives by using heuristics based on knowledge of their probabilistic distribution functions of likely alternatives and the operating context of the application.

APPENDIX B

Appendix B

STORAGE AND ACCESSING EFFICIENCY COMPARISONS

This appendix compares the four manifold edge based data structures of Chapter 12 in terms of storage requirements, accessing efficiency, and algorithmic complexity.

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.

Even if the circumstances of comparison are as equivalent as possible for all representations, questions arise as to defining the "typical" applications, since optimal choices involve careful statistical analysis of actual usage patterns. This approach can yield overall better space and time performance, but is not foolproof, since usage patterns can change drastically based even on minor changes in heavily used application code.

At best, it's a tricky business.

The analysis here does not intend to be complete or rigorous in the sense described above, but does attempt to reflect some approximate measure of the time and space requirements and the complexity of access algorithms necessary to exercise the functionality of the data structures presented in a large model environment over the domain for which they are intended to provide information.

With these caveats, we proceed.

1. Space Requirements for the Manifold Data Structures

The number of pointers required for complete edge adjacency information in each of the four data structures is shown in Table B – 1. 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 B – 2. Assumptions made to allow comparison include that pointers are 32 bits in length and that an eight-bit byte is the minimum size storage unit. Sizes of face (17 bytes) and vertex (4 bytes) records were derived from the support record structures defined earlier in Chapter 12. No geometry or other attributes were assumed except that pointers to such attributes are included. These choices have a tendency to maximize the apparent difference between the alternative structures because geometry and other attributes are not included; the numbers presented are therefore worst case differences. In a more realistic situation additional information storage (such as face, edge, and vertex geometry) would reduce the percentage of space attributable to differences in the edge structures presented.

Table B-1. Representation Storage Requirements per Edge

Representation	Number of Pointer Fields
W-E	9
modified W-E	9
V-E	12
V-E	12

Table B-2. Typical Storage Requirements for Some Solid Objects

Object Type	*faces*	*edges*	*verts*	*total size in bytes W-E and modified* W-E	*total size in bytes V-E or F-E*	*% increase*
prism						
4 sided	6	12	8	622	754	21.22
4000 sided	4002	12000	8000	592030	724030	22.30
approximated sphere						
64 quad facets	32	56	26	5972	7292	22.10
64k quad facets	32768	65280	32514	6475548	7914524	22.22

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 are 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 structure exactly equivalent.

In general, for the types of objects considered, the V-E and F-E structures required about 20% more storage than the W-E and modified W-E structures.

As can be seen from the size requirements, none of the structures provides a minimal size representation for the objects considered compared to many procedural or other conceptualizations of the objects. In general, the four representation structures are not intended as minimal size storage formats but are intended 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 presented is their ability to maintain their validity without requiring a change of representational form regardless of the number of manipulations made to the model.

2. Time Requirements

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

- The operations should be as primitive as possible in the sense that all the information from the data structure can be reasonably extracted by one or more applications of these functions.
- The operations must not be at too low a level (involved in 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.

The functions chosen as meeting these criteria and still providing some insight into the alternative data structures are the functions to obtain 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. A similar approach is utilized in [Woo 84].

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 which access data a 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 because any I/O overhead due to page faults is related to field rather than record accesses as 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 *ee_cw_ptr* 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.

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.

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 due to I/O overhead.

3. Accessing Efficiency Comparison of the Manifold Data Structures

The accessing 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 B – 3.

The record accessing costs for each of the four data structures are shown in Table B – 4.

The W-E and modified W-E structures are superior with respect to record access

Table B-3. 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 chose to use other vertex in edge end representation.
[2] would be 3 if chose to use other face in edge side representation.

requirements. As seen in Table B – 4, 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 edge structures of the V-E and F-E structure accessing procedures cause two record accesses to be made. 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 equal or better performance than the modified W-E structure for adjacency relationship accesses where the edge is not the reference element. Efficiency in adjacency relationship accesses where the edge is not the reference element is often more important in practice since the field accessing costs given in Table B – 3 must

Table B-4. 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

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.

4. Accessing Algorithm Complexity of the Manifold Data Structures

The complexity of the algorithms necessary to manipulate the data structures is sometimes a more important evaluation criteria 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.

The accessing procedures used for determining the number of field and record accesses necessary as well as for comparing complexity are given in detail in Figure B – 1. The figure has six sections. The first section 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 third through sixth sections describe the actual accessing procedures for the W-E, modified W-E, V-E, and F-E structures respectively.

The procedures utilize the edge data structures as described in Chapter 9, "Manifold Data Structures". Where necessary, the accessing procedures presented perform extra assignments to reduce the number of field accesses necessary, utilizing temporary variables. The procedures listed in Figure B – 1 can be used to examine the number of temporary variable assignments performed to obtain these numbers.

Since adjacent groups of elements in ordered adjacency relationships are usually circular lists of elements, the accessing procedures to find an adjacent element as shown in Figure B – 1 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 to enumerate 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. The initial and terminating loop conditions are described in Figure B – 1b.

As seen in the accessing procedures of Figure B – 1c, 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 needs to continually determine which edge side or end was intended every time an edge pointer appears in a field used to find an adjacent element. In the planar polyhedral environment implementation described in Figure B – 1c, this is done by carrying along a vertex as well as edge pointer during traversals of adjacent groups, using the vertex pointer information to determine which side is indicated. Note the effect of this particular scheme on the field accessing costs of the $F< F>$ relationship (see Table B – 3). In most implementations of the W-E structure this determination involves either carrying along additional information as done in the procedures presented, or by using marker bit fields; both techniques require additional processing during traversals. Note that the accessing procedures described for the W-E structure are only valid 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 each vary.

$v<V>$ - 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

Note: 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 towards the surface.

Figure B-1a: Adjacency Relationship Queries

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

W-E: given vertex structure $v = vfirst$ and edge structure $e = efirst$ for which $e.ev\_ptr[1] = v$ or $e.ev\_ptr[2] = v$ ($vfirst$, $efirst$ 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 $e.ev\_ptr[h] = v$ ($efirst$, $hfirst$ is used to detect end of loop)

V-E,F-E: given vertex structure v and edge half structure $e = efirst$ such that $e.ee\_mate\_ptr.ev\_ptr = v$ ($efirst$ 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 $e.ev\_ptr[1] = v$ or $e.ev\_ptr[2] = v$

modified W-E: given edge structure e and side indicator h to determine ordering such that $e.ev\_ptr[h] = v$

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

$f<V>, f<E>, f<F>$

W-E: given face structure f and an edge structure $e = efirst$ such that $e.ef\_ptr[1] = f$ or $e.ef\_ptr[2] = f$, 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 $e.ef\_ptr[flip(h)] = f$, 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 $e.ee\_mate\_ptr.ef\_ptr = f$ ($efirst$ is used to detect end of loop)

Figure B-1b: Initial Conditions for Queries

```
V< V>  - three field accesses / one record access
      if (v =  e^.ev_ptr[1])
               then half ← 1; otherhalf ← 2
               else half ← 2; otherhalf ← 1
      adjacentv ← e^.ev_ptr[otherhalf]
      e ← e^.ee_ccw_ptr[half]

V< E>  - two field accesses / one record access
      if (v =  e^.ev_ptr[1])
               then half ← 1
               else half ← 2
      e ← e^.ee_ccw_ptr[half]

V< F>  - three field accesses / one record access
      if (v =  e^.ev_ptr[1])
               then half ← 1
               else half ← 2
      f ← e^.ef_ptr[half]
      e ← e^.ee_ccw_ptr[half]

E[V] - three field accesses / one record access
      if (v =  e^.ev_ptr[1])
               then half ← 1; otherhalf ← 2
               else half ← 2; otherhalf ← 1
      v1 ← e^.ev_ptr[half]
      v2 ← e^.ev_ptr[otherhalf]

E[[E][E]] - five field accesses / one record access
      if (v =  e^.ev_ptr[1])
               then half ← 1; otherhalf ← 2
               else half ← 2; otherhalf ← 1
      e1 ← e^.ee_ccw_ptr[otherhalf]
      e2 ← e^.ee_cw_ptr[half]
      e3 ← e^.ee_ccw_ptr[half]
      e4 ← e^.ee_cw_ptr[otherhalf]

E[F] - three field accesses / one record access
      if (v =  e^.ev_ptr[1])
               then half ← 1; otherhalf ← 2
               else half ← 2; otherhalf ← 1
      f1 ← e^.ef_ptr[half]
      f2 ← e^.ef_ptr[otherhalf]

F< V>  - three field accesses / one record access
      if (v =  e^.ev_ptr[1])
               then half ← 1; otherhalf ← 2
               else half ← 2; otherhalf ← 1
      v ← e^.ev_ptr[otherhalf]
      e ← e^.ee_cw_ptr[half]

F< E>  - three field accesses / one record access
      if (v =  e^.ev_ptr[1])
               then half ← 1; otherhalf ← 2
               else half ← 2; otherhalf ← 1
      v ← e^.ev_ptr[otherhalf]
      e ← e^.ee_cw_ptr[half]

F< F>  - four field accesses / one record access
      if (v =  e^.ev_ptr[1])
               then half ← 1; otherhalf ← 2
               else half ← 2; otherhalf ← 1
      f ← e^.ef_ptr[half]
      v ← e^.ev_ptr[otherhalf]
      e ← e^.ee_cw_ptr[half]
```

Figure B-1c: W-E Structure Adjacency Relationship Accessing Procedures (for connected graph planar polyhedral environments only)

```
V< V>  - three field accesses / one record access
       adjacentv ← e^.ev_ptr[flip(h)]      {where flip(n) gives opposite of n}
       olde ← e
       e ← e^.ee_ccw_ptr[h]
       h ← olde^.ee_ccw_half[h]

V< E>  - two field accesses / one record access
       olde ← e
       e ← e^.ee_ccw_ptr[h]
       h ← olde^.ee_ccw_half[h]

V< F>  - three field accesses / one record access
       f ← e^.ef_ptr[h]
       olde ← e
       e ← e^.ee_ccw_ptr[h]
       h ← olde^.ee_ccw_half[h]

E[V] - two field accesses / one record access
       v1 ← e^.ev_ptr[h]
       v2 ← e^.ev_ptr[flip(h)]

E[[E][E]] - four field accesses / one record access
       e1 ← e^.ee_ccw_ptr[flip(h)]
       e2 ← e^.ee_cw_ptr[h]
       e3 ← e^.ee_ccw_ptr[h]
       e4 ← e^.ee_cw_ptr[flip(h)]

E[F] - two field accesses / one record access
       f1 ← e^.ef_ptr[h]
       f2 ← e^.ef_ptr[flip(h)]

F< V>  - three field accesses / one record access
       v ← e^.ev_ptr[h]
       olde ← e
       e ← e^.ee_cw_ptr[h]
       h ← olde^.ee_cw_half[h]

F< E>  - two field accesses / one record access
       olde ← e
       e ← e^.ee_cw_ptr[h]
       h ← olde^.ee_cw_half[h]

F< F>  - three field accesses / one record access
       f ← e^.ef_ptr[h]
       olde ← e
       e ← e^.ee_cw_ptr[h]
       h ← olde^.ee_cw_half[h]
```

Figure B-1d: Modified W-E Structure Adjacency Relationship Accessing Procedures (for all connected graph environments)

```
V< V>  - two field accesses / one record access
      e ← e^.ee_cw_ptr
      v ← e^.ev_ptr

V< E>  - one field access / one record access
      e ← e^.ee_cw_ptr

V< F>  - two field accesses / one record access
      e ← e^.ee_cw_ptr
      f ← e^.ef_ptr

E[V] - three field accesses / two record accesses
      v1 ← e^.ev_ptr
      v2 ← e^.ee_mate_ptr^.ev_ptr

E[[E][E]] - five field accesses / two record accesses
      e1 ← e^.ee_cw_ptr
      e2 ← e^.ee_ccw_ptr
      etemp ← e^.ee_mate_ptr
      e3 ← etemp^.ee_cw_ptr
      e4 ← etemp^.ee_ccw_ptr

E[F] - three field accesses / two record accesses
      f1 ← e^.ef_ptr
      f2 ← e^.ee_mate_ptr^.ef_ptr

F< V>  - three field accesses / two record accesses
      e ← e^.ee_ccw_ptr^.ee_mate_ptr
      v ← e^.ev_ptr

F< E>  - two field accesses / two record accesses
      e ← e^.ee_ccw_ptr^.ee_mate_ptr

F< F>  - three field accesses / two record accesses
      e ← e^.ee_ccw_ptr^.ee_mate_ptr
      f ← e^.ef_ptr
```

Figure B-1e: V-E Structure Adjacency Relationship Accessing Procedures (all connected graph environments)

```
V< V>  - three field accesses / two record accesses
      e ← e^.ee_mate_ptr^.ee_ccw_ptr
      v ← e^.ev_ptr

V< E>  - two field accesses / two record accesses
      e ← e^.ee_mate_ptr^.ee_ccw_ptr

V< F>  - three field accesses / two record accesses
      e ← e^.ee_mate_ptr^.ee_ccw_ptr
      f ← e^.ef_ptr

E[V] - three field accesses / two record accesses
      v1 ← e^.ev_ptr
      v2 ← e^.ee_mate_ptr^.ev_ptr

E[[E][E]] - five field accesses / two record accesses
      etemp ← e^.ee_mate_ptr
      e1 ← etemp^.ee_ccw_ptr
      e2 ← e^.ee_cw_ptr
      e3 ← e^.ee_ccw_ptr
      e4 ← etemp^.ee_cw_ptr

E[F] - three field accesses / two record accesses
      f1 ← e^.ef_ptr
      f2 ← e^.ee_mate_ptr^.ef_ptr

F< V>  - two field accesses / one record access
      e ← e^.ee_cw_ptr
      v ← e^.ev_ptr

F< E>  - one field access / one record access
      e ← e^.ee_cw_ptr

F< F>  - two field accesses / one record access
      e ← e^.ee_cw_ptr
      f ← e^.ef_ptr
```

Figure B-1f: F-E Structure Adjacency Relationship Accessing Procedures (all connected graph environments)

Appendix C

TRAVERSALS OF THE RADIAL EDGE STRUCTURE

Detailed definitions of several traversal functions for the Radial Edge structure are now described to provide a better understanding of the semantics of the Radial Edge structure and to provide support for the the adjacency relationship derivation algorithms presented in Appendix D.

The traversal algorithms are interdependent and freely make use of other traversal algorithms. For simplicity of description, they are assumed to be non-destructive traversals where the actions performed against each element do not change the topology of the model. Destructive traversals are also possible.

Four types of traversals are described: *general traversals* which visit each element in the model, *global traversals* which visit every element of a given type in a model, *downward hierarchical traversals* which visit all lower dimensional elements adjacent to a specific higher dimensional element type, and *use-component traversals* which visit all use elements related to a specific basic topological element.

The traversal algorithms are described in three forms. The first form is basically a description of loop control mechanisms which identify actions required for loop initialization, the increment after the loop body is performed, and the continuance test to be performed after the increment is done. The second form defines the traversal in terms of other traversals. The third form is that used by the general traversal, which assigns the body of the loop to a procedure which is called during traversal as a *visit* operation on each applicable element.

It is simple to transform traversals involving element uses, such as those of the

downward hierarchical traversal, into basic element traversals by accessing the relevant upward hierarchical pointers available in each of the use element structures in the Radial Edge structure.

The general traversal mechanism presented, while simplified and unoptimized, does allow traversal of all components of the entire model. The Radial Edge structure has hierarchical groups of circular linked lists of the elements for just about everything except wireframes. For traversal of wireframes a marking scheme is used to assist heuristic search of all connected edge and vertex components of each wireframe sub-graph in the model. In the traversal description, a marking scheme is also used with face and loop element types to indicate when a topological element has been visited. Minor modifications to the general traversal algorithms presented allow the correct traversal of elements such as shells and faces rather than the entire model as well as traversal of element uses.

1. Generalized Traversal

```
traverse_model(m)
  pre_visit_model(m)
  foreach_region_in_model(r,m,status)
    traverse_region(r)
  post_visit_model(m)

traverse_region(r)
  pre_visit_region(r)
  foreach_shell_in_region(s,r,status)
    traverse_shell(s)
  post_visit_region(r)

traverse_shell(s)
  pre_visit_shell(s)
  case s^.downptr of
    FACEUSEptr:
      foreach_faceuse_in_shell(fu,s,status)
        traverse_faceuse(fu)
    EDGEUSEptr:
      traverse_wire(s^.seu_ptr^.euvu_ptr)
    VERTEXUSEptr:
      visit_vertex(s^.svu_ptr^.vuv_ptr)
    end {case}
  post_visit_shell(s)

traverse_faceuse(fu)
  if (face_is_not_marked(fu^.fuf_ptr)) then begin
    mark_face(fu^.fuf_ptr)
    pre_visit_face(fu^.fuf_ptr)
    foreach_loopuse_in_faceuse(lu,fu,status1)
      traverse_loopuse(lu)
    post_visit_face(fu^.fuf_ptr)
    end {then}

traverse_loopuse(lu)
  if (loop_is_not_marked(lu^.lul_ptr)) then begin
    mark_loop(lu^.lul_ptr)
    pre_visit_loop(lu^.lul_ptr)
    case lu^.downptr of
      EDGEUSEptr:
        foreach_edgeuse_in_loopuse(eu,lu,status)
          if (edge_is_not_marked(eu^.eue_ptr))
                                then begin
            mark_edge(eu^.eue_ptr)
            visit_edge(eu^.eue_ptr)
            traverse_wire(eu^.euvu_ptr)
            end {then}
      VERTEXUSEptr:
        traverse_wire(lu^.luvu_ptr)
      end {case}
    post_visit_loop(lu^.lul_ptr)
    end {then}

traverse_wire(vu)
  if (vertex_is_not_marked(vu^.vuv_ptr))
                              then begin
    mark_vertex(vu^.vuv_ptr)
    visit_vertex(vu^.vuv_ptr)
    foreach_vertexuse_in_vertex(vu,vu^.vuv_ptr,
                              status)
      if vu^.upptr = EDGEUSEptr then
        if vu^.vueu_ptr^.upptr = SHELLptr then
          if (edge_is_not_marked(vu^.vueu_ptr^.
                .eue_ptr)) then begin
            mark_edge(vu^.vueu_ptr^.eue_ptr)
            visit_edge(vu^.vueu_ptr^.eue_ptr)
            traverse_wire(vu_vueu_ptr^.
                eueu_mate_ptr^.euvu_ptr)
            end {then}
    end {then}
```

2. Global Model Traversals

```
foreach_region_in_model(r,m,status)
  loop initialization: r ← m^.mr_ptr
  bottom of loop increment: r ← r^.mrnext
  bottom of loop test: until r = m^.mr_ptr

foreach_shell_in_model(s,m,status)
  foreach_region_in_model(r,m,status)
    foreach_shell_in_region(s,r,status)
      execute loop body

foreach_face_in_model(f,m,status)
  pre_visit_face ← loop body
  traverse_model(m)

foreach_loop_in_model(l,m,status)
  foreach_face_in_model(f,m,status)
    foreach_loopuse_in_faceuse(lu,f^.ffu_ptr,status)
      l ← lu^.lul_ptr
      execute loop body

foreach_edge_in_model(e,m,status)
  visit_edge(e) ← loop body
  traverse_model(m)

foreach_vertex_in_model(v,m,status)
  visit_vertex(v) ← loop body
  traverse_model(m)
```

3. Downward Hierarchical Traversals

```
foreach_shell_in_region(s,r,status)
  loop initialization: s ← r^.rs_ptr
  bottom of loop increment: s ← s^.rs_next
  bottom of loop test: until s = r^.rs_ptr

foreach_faceuse_in_shell(fu,s,status)
  loop initialization: fu ← s^.sfu_ptr
  bottom of loop increment: fu ← fu^.sfu_next
  bottom of loop test: until fu = s^.sfu_ptr

foreach_loopuse_in_faceuse(lu,fu,status)
  loop initialization: lu ← fu^.fulu_ptr
  bottom of loop increment: lu ← lu^.fulu_next
  bottom of loop test: until lu = fu^.fulu_ptr

foreach_edgeuse_in_loopuse(eu,lu,status)
  loop initialization:
    if not (lu^.downptr = EDGEUSEptr)
      then skip loop
      else eu ← lu^.lueu_ptr
  bottom of loop increment: eu ← eu^.eueu_cw_ptr
  bottom of loop test: until eu = lu^.lueu_ptr

foreach_vertexuse_in_edgeuse(vu,eu,status)
  loop initialization: vu ← eu^.euvu_ptr
  bottom of loop increment:
    vu ← eu^.eueu_mate_ptr^.euvu_ptr
  bottom of loop test:
    until vu = vu ← eu^.euvu_ptr
```

4. Radial Edge Use-Component Traversals

```
foreach_faceuse_in_face(fu,f,status)
    "will always produce two iterations"
  loop initialization: fu ← f^.ffu_ptr
  bottom of loop increment:
    fu ← fu^.fufu_mate_ptr
  bottom of loop test: until fu = f^.ffu_ptr

foreach_loopuse_in_loop(lu,l,status)
    "will always produce two iterations"
  loop initialization: lu ← l^.llu_ptr
  bottom of loop increment:
    lu ← lu^.lulu_mate_ptr
  bottom of loop test: until lu = l^.lu_ptr

foreach_edgeuse_in_edge(eu,e,status)
  loop initialization:
    eu ← e^.eeu_ptr
    i ← 1
  bottom of loop increment:
    if odd(i)
      then eu ← eu^.eueu_mate_ptr
      else eu ← eu^.eueu_radial_ptr
    i ← i + 1
  bottom of loop test: until eu = e^.eeu_ptr

foreach_vertexuse_in_vertex(vu,v,status)
  loop initialization: vu ← v^.vvu_ptr
  bottom of loop increment: vu ← vu^.vuvu_next
  bottom of loop test: until vu = v^.vvu_ptr
```

APPENDIX D

Appendix D

COMPLETENESS OF THE RADIAL EDGE STRUCTURE

This appendix describes the algorithms for derivation of the thirty-six non-manifold adjacency relationships from the Radial Edge structure in a Pascal-like programming notation consistent with the Radial Edge data structure descriptions in Chapter 17. The sequence of derivation algorithms show how all adjacency information can be obtained from the Radial Edge data structure non-manifold geometric modeling representation.

First several additional functions utilized in the derivation algorithms are described. The derivation algorithms themselves are then described. The derivation algorithms also utilize the traversal algorithms of Appendix C.

Further discussion of the derivation process can be found in Chapter 18.

It should be noted that the oppositely oriented adjacent groups of $E\{< [E]> \}$, $E\{< L > \}$, $E\{< F> \}$, $E\{< S> \}$, and $E\{< R > \}$ adjacency relationships are not shown here for brevity.

1. Additional Functions

A few additional functions are used to simplify the descriptions of the derivation of the non-manifold adjacency relationships from the Radial Edge structure information:

add *element* to *adjacency relationship unordered adjacent group*
adds *element* to the adjacent group *if* it does not already exist in the set.

append *element* to *adjacency relationship ordered adjacent group*
places *element* at the end of the ordered adjacent group list so that the order of entry is preserved. Duplicates are allowed.

odd(*integer*)
returns *TRUE* if the input *integer* is an odd number, and *FALSE* if it is an even number.

2. Algorithms to Derive the Adjacency Relationships

2.1. Upward Hierarchical Diagonal Adjacency Relationships

V [E]

```
foreach_vertex_in_model(v,m,status1) begin
 create adjacency relationship v[E]
        with empty adjacent group
 foreach_vertexuse_in_vertex(vu,v,status2)
  if vu^.upptr = EDGEUSEptr
    then append vu^.vueuptr^.eue_ptr to v[E]
 output v[E]
 end {foreach}
```

E< L >

```
foreach_edge_in_model(e,m,status1) begin
 create adjacency relationship e< L>
        with empty adjacent group
 i ← 1
 foreach_edgeuse_in_edge(eu,e,status2)
  if eu^.upptr = LOOPUSEptr then begin
        "this skips over the double facesides"
    if (odd(i)) append eu^.eulu_ptr^.
                  lul_ptr to e< L>
    i ← i + 1
    end {then}
 output e< L>
 end {foreach}
```

L [F][1]

```
foreach_loop_in_model(l,m,status1) begin
 create adjacency relationship l[F]
        with empty adjacent group
 append l^.llu_ptr^.lufu_ptr^.fuf_ptr to l[F]
 output l[F]
 end {foreach}
```

F [S][2]

```
foreach_face_in_model(f,m,status1) begin
 create adjacency relationship f[S]
        with empty adjacent group
 foreach_faceuse_in_face(fu,f,status2)
  append fu^.fus_ptr to f[S]
 output f[S]
 end {foreach}
```

S [R][1]

```
foreach_shell_in_model(s,m,status1) begin
 create adjacency relationship s[R]
        with empty adjacent group
 append s^.sr_ptr to s[R]
 output s[R]
 end {foreach}
```

2.2. Downward Hierarchical Diagonal Adjacency Relationships

E [V][2]

```
foreach_edge_in_model(e,m,status1) begin
 create adjacency relationship e[V]
        with empty adjacent group
 foreach_vertexuse_in_edgeuse(vu,e^.eeu_ptr,status)
  append e^.eeu_ptr^.euvu_ptr^.vuv_ptr to e[V]
 output e[V]
 end {foreach}
```

L [< E >]$^{|F[S]|}$

```
foreach_loop_in_model(l,m,status1) begin
 create adjacency relationship l[]
        with empty adjacent group
 foreach_loopuse_in_loop(lu,l,status2) begin
  create an empty adjacent group < E>
  if lu^.downptr = EDGEUSEptr then
   foreach_edgeuse_in_loopuse(eu,lu,status3)
       append eu^.eue_ptr to < E>
  append current < E> to l[]
  end {foreach}
 output l[< E> ]
 end {foreach}
```

F [L]

```
foreach_face_in_model(f,m,status1) begin
 create adjacency relationship f[L]
        with empty adjacent group
 foreach_loopuse_in_faceuse(lu,f^.ffu_ptr,status2)
  append lu^.lul_ptr to f[L]
 output f[L]
 end {foreach}
```

S {F}

```
foreach_shell_in_model(s,m,status1) begin
 create adjacency relationship s{F}
        with empty adjacent group
 foreach_faceuse_in_shell(fu,s,status2)
  add fu^.fuf_ptr to s{F}
 output s{F}
 end {foreach}
```

```
R {S}
foreach_region_in_model(r,m,status1) begin
 create adjacency relationship r{S}
        with empty adjacent group
 foreach_shell_in_region(s,r,status2)
  add s to r{S}
 output,r{S}
 end {foreach}
```

2.3. Upward Hierarchical Adjacency Relationships

```
V {L}
foreach_vertex_in_model(v,m,status1) begin
 create adjacency relationship v{L}
        with empty adjacent group
 foreach_vertexuse_in_vertex(vu,v,status2)
  case vu^.upptr of
   EDGEUSEptr:
    if vu^.vueu_ptr^.upptr = LOOPUSEptr
     then add vu^.vueu_ptr^.eulu_ptr^.
                lul_ptr to v{L}
   LOOPUSEptr:
    add vu^.vul_ptr to v{L}
   end {case}
 output v{L}
 end {foreach}

V {F}
foreach_vertex_in_model(v,m,status1) begin
 create adjacency relationship v{F}
        with empty adjacent group
 foreach_vertexuse_in_vertex(vu,v,status2)
  case vu^.upptr of
   EDGEUSEptr:
    if vu^.vueu_ptr^.upptr = LOOPUSEptr
     then add vu^.vueu_ptr^.eulu_ptr^.
                lufu_ptr^.fuf_ptr to v{F}
   LOOPUSEptr:
    add vu^.vul_ptr^.lufu_ptr^.fuf_ptr to v{F}
   end_case
 output v{F}
 end {foreach}
```

```
V {S}
foreach_vertex_in_model(v,m,status1) begin
 create adjacency relationship v{S}
        with empty adjacent group
 foreach_vertexuse_in_vertex(vu,v,status2)
  case vu^.upptr of
   EDGEUSEptr:
    if vu^.vueu_ptr^.upptr = SHELLptr
     then add vu^.vueu_ptr^.eus_ptr to v{S}
     else add vu^.vueu_ptr^.eulu_ptr^.
                lufu_ptr^.fus_ptr to v{S}
   LOOPUSEptr:
    add vu^.vul_ptr^.lufu_ptr^.fus_ptr to v{S}
   SHELLptr:
    add vu^.vus_ptr to v{S}
   end_case
 output v{S}
 end {foreach}

V {R}
foreach_vertex_in_model(v,m,status1) begin
 create adjacency relationship v{R}
        with empty adjacent group
 foreach_vertexuse_in_vertex(vu,v,status2)
  case vu^.upptr of
   EDGEUSEptr:
    if vu^.vueu_ptr^.upptr = SHELLptr
     then add vu^.vueu_ptr^.eus_ptr^.sr_ptr to v{R}
     else add vu^.vueu_ptr^.eulu_ptr^.lufu_ptr^.
                fus_ptr^.sr_ptr to v{R}
   LOOPUSEptr:
    add vu^.vul_ptr^.lufu_ptr^.fus_ptr^.sr_ptr to v{R}
   SHELLptr:
    add vu^.vus_ptr^.sr_ptr to v{R}
   end_case
 output v{R}
 end {foreach}
```

```
E< F> |E< L> |
foreach_edge_in_model(e,m,status1) begin
 create adjacency relationship e< F>
        with empty adjacent group
 i ← 1
 foreach_edgeuse_in_edge(eu,e,status2)
  if eu^.upptr = SHELLptr then begin
        "this skips over the double facesides"
    if ( odd(i)) append eu^.eulu_ptr^.
          lufu_ptr^.fuf_ptr to e< F>
    i ← i + 1
    end {then}
 output e< F>
 end {foreach}
```

```
E< S> |E< L> |
foreach_edge_in_model(e,m,status1) begin
 create adjacency relationship e< S>
        with empty adjacent group
 i ← 1
 foreach_edgeuse_in_edge(eu,e,status2)
  if eu^.upptr = SHELLptr
    then append eu^.eus_ptr to e< S>
    else begin
        "this skips over the double facesides"
      if ( odd(i)) append eu^.eulu_ptr^.
                lufu_ptr^.fus_ptr to e< S>
      i ← i + 1
      end {else}
 output e< S>
 end {foreach}
```

```
E< R> |E< L> |
foreach_edge_in_model(e,m,status1) begin
 create adjacency relationship e< R>
        with empty adjacent group
 i ← 1
 foreach_edgeuse_in_edge(eu,e,status2)
  if eu^.upptr = SHELLptr
    then append eu^.eus_ptr^.sr_ptr to e< R>
    else begin
        "this skips over the double facesides"
      if ( odd(i)) append eu^.eulu_ptr^.
          lufu_ptr^.fus_ptr^.sr_ptr to e< R>
      i ← i + 1
      end {else}
 output e< R>
 end {foreach}
```

```
L [S] |F[S]|
foreach_loop_in_model(l,m,status1) begin
 create adjacency relationship l[S]
        with empty adjacent group
 foreach_loopuse_in_loop(lu,l,status2)
  append lu^.lufu_ptr^.fus_ptr to l[S]
 output l[S]
 end {foreach}
```

```
L [R ] |F[S]|
foreach_loop_in_model(l,m,status1) begin
 create adjacency relationship l[R]
        with empty adjacent group
 foreach_loopuse_in_loop(lu,l,status2)
  append lu^.lufu_ptr^.fus_ptr^.sr_ptr to l[R]
 output l[R]
 end {foreach}
```

```
F [R ] |F[S]|
foreach_face_in_model(f,m,status1) begin
 create adjacency relationship f[R]
        with empty adjacent group
 foreach_faceuse_in_face(fu,f,status2)
  append fu^.fus_ptr^.sr_ptr to f[R]
 output f[R]
 end {foreach}
```

2.4. Downward Hierarchical Adjacency Relationships

$L[< V >]^{IF[S]}$

```
foreach_loop_in_model(l,m,status1) begin
 create adjacency relationship l[]
        with empty adjacent group
 foreach_loopuse_in_loop(lu,l,status2) begin
  create an empty adjacent group < V>
  case lu^.downptr of
   EDGEUSEptr:
    foreach_edgeuse_in_loopuse(eu,lu,status3)
     append eu^.euvu_ptr^.vuv_ptr to < V>
   VERTEXUSEptr:
    append lu^.luvu_ptr^.vuv_ptr to < V>
   end {case}
  append current < V>  to l[]
  end {foreach}
 output l[< V> ]
 end {foreach}
```

$F[[< V >]]^{IF[S]}$

```
foreach_face_in_model(f,m,status1)
 create adjacency relationship f[]
        with empty adjacent group
 foreach_faceuse_in_face(fu,f,status2) begin
  create empty adjacent group []
  foreach_loopuse_in_faceuse(lu,fu,status3) begin
   create an empty adjacent group < V>
   case lu^.downptr of
    EDGEUSEptr:
     foreach_edgeuse_in_loopuse(eu,lu,status4)
      append eu^.euvu_ptr^.vuv_ptr to < V>
    VERTEXUSEptr:
     append lu^.luvu_ptr^.vuv_ptr to < V>
    end {case}
   append current < V>  to []
   end {foreach}
  append current [< V> ] to f[]
  end {foreach}
 output f[[< V> ]]
 end {foreach}
```

$F[[< E >]]^{IF[S]}$

```
foreach_face_in_model(f,m,status1)
 create adjacency relationship f[]
        with empty adjacent group
 foreach_faceuse_in_face(fu,f,status2) begin
  create empty adjacent group []
  foreach_loopuse_in_faceuse(lu,fu,status3) begin
   create an empty adjacent group < E>
   case lu^.downptr of
    EDGEUSEptr:
     foreach_edgeuse_in_loopuse(eu,lu,status4)
      append eu^.eue_ptr to < E>
    VERTEXUSEptr:
     do nothing
    end {case}
   append current < E>  to []
   end {foreach}
  append current [< E> ] to f[]
  end {foreach}
 output f[[< E> ]]
 end {foreach}
```

$S\{V\}$

```
foreach_shell_in_model(s,m,status1) begin
 create adjacency relationship s{V}
        with empty adjacent group
 foreach_vertex_in_model(v,m,status2)
  foreach_vertexuse_in_vertex(vu,v,status3)
   case vu^.upptr of
    EDGEUSEptr:
     if vu^.vueu_ptr^.upptr = SHELLptr
      then begin
       if vu^.vueu_ptr^.eus_ptr = s then
        add vu^.vuv_ptr to s{V}
       end {then}
      else
       if vu^.vueu_ptr^.eulu_ptr^.
            lufu_ptr^.fus_ptr = s then
        add vu^.vuv_ptr to s{V}
    LOOPUSEptr:
     if vu^.vulu_ptr^.lufu_ptr^.fus_ptr = s then
      add vu^.vuv_ptr to s{V}
    SHELLptr:
     if vu^.vus_ptr = s then
      add vu^.vuv_ptr to s{V}
    end {case}
 output s{V}
 end {foreach}
```

```
S{E}
foreach_shell_in_model(s,m,status1) begin
 create adjacency relationship s{E}
        with empty adjacent group
 foreach_edge_in_model(e,m,status2)
  foreach_edgeuse_in_edge(eu,e,status3)
   if eu^.upptr = SHELLptr
    then begin
     if eu^.eus_ptr = s then
      add eu^.eue_ptr to s{E}
     end {then}
    else
     if eu^.eulu_ptr^.
          lufu_ptr^.fus_ptr = s then
      add eu^.eue_ptr to s{E}
 output s{E}
 end {foreach}

S{L}
foreach_shell_in_model(s,m,status1) begin
 create adjacency relationship s{L}
       with empty adjacent group
 foreach_faceuse_in_shell(fu,s,status2)
  foreach_loopuse_in_faceuse(lu,fu,status3)
   add lu^.lul_ptr to s{L}
 output s{L}
 end {foreach}

R{V}
foreach_region_in_model(s,m,status1) begin
 create adjacency relationship r{V}
       with empty adjacent group
 foreach_vertex_in_model(v,m,status2)
  foreach_vertexuse_in_vertex(vu,v,status3)
   case vu^.upptr of
    EDGEUSEptr:
     if vu^.vueu_ptr^.upptr = SHELLptr
      then begin
       if vu^.vueu_ptr^.eus_ptr^.
                        sr_ptr = r then
        add vu^.vuv_ptr to r{V}
       end {then}
      else
       if vu^.vueu_ptr^.eulu_ptr^.lufu_ptr^.
                fus_ptr^.sr_ptr = r then
        add vu^.vuv_ptr to r{V}
    LOOPUSEptr:
     if vu^.vulu_ptr^.lufu_ptr^.fus_ptr^.
                sr_ptr = r then
      add vu^.vuv_ptr to r{V}
    SHELLptr:
     if vu^.vus_ptr^.sr_ptr = r then
      add vu^.vuv_ptr to r{V}
   end {case}
 output r{V}
 end {foreach}
```

```
R{E}
foreach_region_in_model(s,m,status1) begin
 create adjacency relationship r{E}
       with empty adjacent group
 foreach_edge_in_model(e,m,status2)
  foreach_edgeuse_in_edge(eu,e,status3)
   if eu^.upptr = SHELLptr
    then begin
     if eu^.eus_ptr^.sr_ptr = r then
      add eu^.eue_ptr to r{E}
     end {then}
    else
     if eu^.eulu_ptr^.lufu_ptr^.
          fus_ptr^.sr_ptr = r then
      add eu^.eue_ptr to r{E}
 output r{E}
 end {foreach}

R{L}
foreach_region_in_model(r,m,status1) begin
 create adjacency relationship r{L}
       with empty adjacent group
 foreach_shell_in_region(s,r,status2)
  foreach_faceuse_in_shell(fu,s,status3)
   foreach_loopuse_in_faceuse(lu,fu,status4)
    add lu^.lul_ptr to r{L}
 output r{L}
 end {foreach}

R{F}
foreach_region_in_model(r,m,status1) begin
 create adjacency relationship r{F}
       with empty adjacent group
 foreach_shell_in_region(s,r,status2)
  foreach_faceuse_in_shell(fu,s,status3)
   add fu^.fuf_ptr to r{F}
 output r{F}
 end {foreach}
```

2.5. Main Diagonal Adjacency Relationships

$V[V]^{|V[E]|}$

```
foreach_vertex_in_model(v,m,status1) begin
 create adjacency relationship v[V]
        with empty adjacent group
 foreach_vertexuse_in_vertex(vu,v,status2)
  if vu^.upptr = EDGEUSEptr then
    add vu^.vueuptr^.eueu_mate_ptr^.
        euvu_ptr^.vuv_ptr to v[V]
 output v[V]
 end {foreach}
```

$E< [E]^{|E[V]|}>$

```
foreach_edge_in_model(e,m,status1) begin
 create adjacency relationship e< >
        with empty adjacent group
 foreach_edgeuse_in_edge(eu,e,status2) begin
  create adjacency relationship [E]
        with empty adjacent group
  if eu^.upptr = LOOPUSEptr then begin
    append eu^.eueu_cw_ptr^.eue_ptr to [E]
    append eu^.eueu_ccw_ptr^.eue_ptr to [E]
    end {then}
  append current [E] to e< >
  end {foreach}
 output e< [E]>
 end {foreach}
```

$L [< [L]^2>]$

```
foreach_loop_in_model(f,m,status1)
 create adjacency relationship l[]
        with empty adjacent group
 foreach_loopuse_in_loop(lu,l,status2) begin
  create an empty adjacent group < >
  if lu^.downptr = EDGEUSEptr then
    foreach_edgeuse_in_loopuse(eu,lu,status3) begin
      create an empty adjacent group [L]
      append eu^.eueu_radial_ptr^.eulu_ptr^.
                lufu_ptr^.fuf_ptr to [L]
      append eu^.eueu_mate_ptr^.eueu_radial_ptr^.
                eulu_ptr^.lufu_ptr^.fuf_ptr to [L]
      append current [L] to < >
      end {foreach}
  append current < [L]> to l[]
  end {foreach}
 output l[< [L]> ]
 end {foreach}
```

$F[[< [F]^{|V[L]|}>]]^2$

```
foreach_face_in_model(f,m,status1)
 create adjacency relationship f[]
        with empty adjacent group
 foreach_faceuse_in_face(fu,f,status2) begin
  create empty adjacent group []
  foreach_loopuse_in_faceuse(lu,fu,status3) begin
   create an empty adjacent group < >
   if lu^.downptr = EDGEUSEptr then
    foreach_edgeuse_in_loopuse(eu,lu,status4) begin
     create an empty adjacent group [F]
     append eu^.eueu_radial_ptr^.eulu_ptr^.
               lufu_ptr^.fuf_ptr to [F]
     append eu^.eueu_mate_ptr^.eueu_radial_ptr^.
               eulu_ptr^.lufu_ptr^.fuf_ptr to [F]
     append current [F] to < >
     end {foreach}
   append current < [F]> to []
   end {foreach}
  append current [< [F]> ] to f[]
  end {foreach}
 output f[[< [F]> ]]
 end {foreach}
```

$S\{S\}$

```
foreach_shell_in_model(s,m,status1) begin
 create adjacency relationship s{S}
        with empty adjacent group
 foreach_faceuse_in_shell(fu,s,status2)
  if not (fu^.fufu_mate_ptr^.fus_ptr = s)
   then add fu^.fufu_mate_ptr^.fus_ptr to s{S}
 output s{S}
 end {foreach}
```

$R\{R\}$

```
foreach_region_in_model(r,m,status1) begin
 create adjacency relationship r{R}
        with empty adjacent group
 foreach_shell_in_region(s,r,status2)
  foreach_faceuse_in_shell(fu,s,status3)
   if not (fu^.fufu_mate_ptr^.fus_ptr^.sr_ptr = r)
    then add fu^.fufu_mate_ptr^.fus_ptr^.
                sr_ptr to r{R}
 output r{R}
 end {foreach}
```

APPENDIX E

Appendix E

SELECTIVE QUERY AND TRAVERSAL

This appendix describes the rationale and implementation of a selective query mechanism for geometric modeling data structures. The selection is performed based on arbitrary specifications of binary attribute data usually assigned to components of the model at model creation time.

1. Rationale

The flexibility offered by the non-manifold modeling form increases the likelihood (and desirability) of maintaining geometric shape information for many different but related objectives, co-existing in the same model. It is important in these cases to keep the different classes of data differentiated from each other while still allowing them to be considered together or in any combination when desired. A general mechanism to provide this facility would be highly desirable. A modeler system implementation, rather than multiple application dependent implementations, would require less resources and be more efficient.

The class or attribute data which is used as a sele on key is normally determined by the application, not the modeler. For exam , in applications involved with manufacturing mechanical parts, it may be desir to compare center line or other manufacturing or tolerance datum point geomet formation with the the designed geometric shape of a mechanical part, and to kee merged data in the same model yet still differentiated (see Figure 3 – 2). Mesh r FEM (finite element method) analysis may also exist in the same model, as w possible approximate geometric shape information created to simplify FEM m g. It may also be desirable to

associate additional application dependent characteristics such as identification of purpose or functionality, sequence of manufacturing, and other characteristics with portions of the original model.

Selectively utilizing model information of this type requires the ability to perform selective query and traversal based on an arbitrary set of characteristics for the portions of the model currently under consideration. Proper design of this facility prevents specific applications from being inundated with data intended for other applications, while still allowing coordination of all data. For example, a single mechanical part model may contain primary shape information, FEM meshing or tolerance information, and other relevant geometric data. When the model is to be displayed it is desirable to have the ability to select what portions of the model should be displayed by specifying the characteristics desired (shape only, mesh only, shape and mesh together, etc.). Similarly, queries involving adjacency should only be concerned with elements matching the characteristics currently under consideration, giving the appearance that the model consists only of the elements which match the selection criteria.

2. Implementation

One implementation approach to allow selective query and traversal involves associating application controlled attribute values or identity values with topological elements and requiring all application code to individually perform checking and selection for every query and traversal, regardless of whether it really uses such capabilities. A more general approach, however, is to have the modeling system itself perform this kind of checking and selection, reducing application implementation complexity and replication of the same checking code. Applications that do not use these facilities would then not need to consider these facilities at all, even in models containing elements with selection attributes.

The approach taken here is to associate an attribute mask with each topological ele-

ment, in which a single bit is used to determine the presence or absence of an application designated characteristic. An attribute mask allows representation and selection based on multiple simultaneous characteristics. Other alternatives are equally possible; this particular one was felt to be useful because of its support of multiple independent attributes.

The interface to this capability is described below in a Pascal-like syntax. All individual applications actually using the capability must restore creation and selection masks to their previous values when finished or invoking other applications. Application programmers not utilizing attribute selection capabilities do not need to be concerned about them in their implementations. The size of the attribute mask is implementation dependent.

set_create_mask(masktype: mask)
All topological elements created after invocation of *set_create_mask* will have an attribute mask equivalent to *mask* until the next call to *set_create_mask*. A 1 in a bit position indicates presence of an attribute; a 0 in a bit position indicates absence of an attribute. The default value of the creation mask is all zeroes with a one in the least significant bit position. This corresponds with the default selection masks to allow use of the system without setting masks.

get_create_mask(): masktype
Returns the current creation mask value.

set_query_mask(masktype: must_have, must_not_have)
All queries and traversals will perform attribute checking to provide only the required elements which also meet the specified characteristics of having *all* attributes specified in *must_have*, and lacking *all* attributes specified by *must_not_have*. Specifically, in a Pascal like syntax, with *AND* symbolizing a bitwise "and", and *and* symbolizing the binary valued Boolean "and".

(((element's attribute mask) AND *must_have*) = *must_have*)
and
(((element's attribute mask) AND *must_not_have*) = 0))

must be true in order for a match to exist. These query masks will remain in effect until the next call to *set_query_mask*. A 1 in a bit position indicates presence of an attribute; a 0 in a bit position indicates absence of an attribute. The default value of the *must_have* mask is all zeroes with a one in the least significant bit position, and the default value of the *must_not_have* mask is all ones with a zero in the least significant bit position. This corresponds with the default creation mask to allow use of the system without setting masks. As an example, mask settings to include *all* elements in any queries regardless of their attribute values would be *must have* and *must_not_have* consisting of all zeroes.

get_query_mask(var masktype: must_have, must_not_have)
Returns the query mask values currently in effect.

The least significant bit position is used by the system to keep track of all data which has *not* had attributes set so that they may be retrieved by applications not using the facility via the default mask settings.

Note that this capability is used for query and traversal only, and does not affect the behavior of the construction operators as far as specification of adjacency and positioning are concerned, since all elements must be inserted into the model in relation to *all* existing model data.