A COMPLETE GEOMETRIC AND TOPOLOGIC FILE STRUCTURE

by

Rajendra V Auradkar

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in the partial fulfillment of the requirements of the degree of

Master of Science

in

Industrial and Systems Engineering

Approved:

Dr. Osama K. Eyada

Dr. Michael P. Deisenroth

Dr. Sankar Jayaram

November 1992
Blacksburg, Virginia
A Complete Geometric and Topological File Structure

by

Rajendra V Auradkar

Dr. Osama K. Eyada, Chairman

Industrial and Systems Engineering

(ABSTRACT)

Automation of the process planning function is necessary for the effective linking of Computer-Aided Design (CAD) and Computer-Aided Manufacturing (CAM). Before automated process planning is performed, primitive geometric data from CAD databases needs to be converted to higher-level feature information required by the CAM systems. Many feature recognition algorithms have been developed to extract geometric features from different CAD representation schemes. However, the algorithms are CAD modeler dependent. This research addresses the problem of developing a standardized file format, Complete Geometric and Topological File Structure (CGTS), for feature recognition purposes. Algorithms for conversion of 3-D wire-frame data to CGTS format have been developed, using C++, and the process of feature recognition demonstrated using the CGTS file format.
Acknowledgments

I would like to express sincere thanks and deep gratitude to my major advisor and chairman, Dr. Osama K. Eyada, for his endless patience and encouragement.

Special thanks are due to Dr. Michael P. Deisenroth and Dr. Sankar Jayaram for serving on my committee and for their constant encouragement.

I would like to express great gratitude to Dr. Michael P. Deisenroth for providing me with the unbridled use of the laboratory facilities. I would also like to thank Krishna Kumar Krishnan and Arun Jayaraman for providing help in the preparation of this document.

I would like to thank Joni and Lavonda for their constant support and for the administrative assistance during the writing of my thesis. I also would like to thank Mrs. Lovedia Cole for her help during the past three years.
# TABLE OF CONTENTS

Acknowledgment v  
List of Figures vi  
List of Tables vii

## Chapter 1: Introduction

1.1 CAD Representation schemes 4  
1.2 Problem Definition 8  
1.3 Research Objectives 10  
1.4 Thesis Outline 11

## Chapter 2: Literature Review

2.1 Pattern Recognition Approach 12  
2.2 Graph Theory Approach 16  
2.3 Expert System Approach 22  
2.4 Summary of Research Efforts 26

## Chapter 3: CGTS Based Feature Recognition System

3.1 CGTS file structure 32  
3.1.1 The Header Section 33  
3.1.2 The Vertex Description Section 36  
3.1.3 The Edge Description Section 38  
3.1.4 The Face Description Section 40  
3.2 Methodology for conversion of wire-frame data to CGTS format 42  
3.3 Rules for feature recognition and extraction 56

## Chapter 4: System Demonstration

4.1 The Input Stage 60  
4.2 The CAD Data Processing and the CGTS output Stage 62  
4.3 The Display Stage 62
Chapter 5: Conclusions

5.1 Enhancements to CGTS 65
5.2 Conversion of CAD data to CGTS 66

References 69

Appendix 1: Source code listing 71

Appendix 2: Example CGTS output 129

Vita 141
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Process-planning modules and databases</td>
<td>3</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>CSG modeling and its data structure</td>
<td>6</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Alternate CSG representation of drawing in Figure 1.2</td>
<td>7</td>
</tr>
<tr>
<td>Figure 1.4</td>
<td>B-rep solid model and its data structure</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Pattern recognition technique</td>
<td>14</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Graph theoretic terminology</td>
<td>17</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Graph theoretic terminology</td>
<td>19</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Data modeler dependent feature recognition</td>
<td>29</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>Data modeler independent feature recognition</td>
<td>30</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>The Header Section of CGTS</td>
<td>34</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>The Vertex Description Section</td>
<td>37</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>The Edge Description Section</td>
<td>39</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>The Face Description Section</td>
<td>41</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Conversion of wire-frame data to CGTS format</td>
<td>43</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>System classes and their interaction</td>
<td>45</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>Example of Entities Section of DXF file</td>
<td>47</td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>Demonstration of topology construction</td>
<td>48</td>
</tr>
<tr>
<td>Figure 3.9</td>
<td>Class edge</td>
<td>49</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>Class vertex</td>
<td>51</td>
</tr>
<tr>
<td>Figure 3.11</td>
<td>Class face</td>
<td>53</td>
</tr>
<tr>
<td>Figure 3.12</td>
<td>Class toponode</td>
<td>57</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>The Input Stage</td>
<td>61</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>The CAD Data Processing and the CGTS output Stage</td>
<td>63</td>
</tr>
</tbody>
</table>
Figure 4.3  The Display Stage  64
Figure 5.1  Alternate ways of CAD data conversion to CGTS  68
LIST OF TABLES

Table 1    A summary of the feature recognition approaches  27
CHAPTER 1

INTRODUCTION

Process planning is a manufacturing activity in which design specifications are translated to manufacturing requirements: processes, equipment, tools and sequence. Selection of manufacturing processes is based on matching design requirements with processing capabilities. For example, the shape, size and tolerance requirements of a geometric feature, such as an internal cylinder, is matched with the capabilities of the processes which can generate that feature. A process/operation can be performed on different equipment. An internal cylinder can be generated on a lathe, a drill press, or a machining center. Factors such as required production rate and tooling influence equipment selection, while design specifications, production rate and cost determine the manufacturing sequence.

Manual process planning requires experienced process planners. Experience requires time to accumulate, is not an exact knowledge, and may not apply to new processes. Because of this and the necessity to bridge the gap between Computer-Aided Design (CAD) and Computer-Aided Manufacturing (CAM), many research efforts have been directed towards automating the manual process planning activities. Automated process planning or Computer-Aided Process Planning (CAPP) systems can be classified into two main groups: variant and generative. Variant CAPP systems are nothing more than an organized approach for retrieving existing process planning information based on
the concept of Group Technology (GT). GT is a philosophy which aims at taking advantage of similarities. Parts are grouped into families based on their design and manufacturing attributes. For each family, a standard process plan is manually developed and stored in the computer. When developing a process plan for a component, the design and manufacturing attributes of the component are analyzed to determine the family to which the part should belong. Once the family is determined, the standard plan is retrieved and edited to suite the part's manufacturing requirements.

Generative CAPP systems do not retrieve existing process plans but rather generate a process plan for each component automatically. Figure 1.1 shows a modular structure for a generative CAPP system [1]. At the top of the structure is the surface identification module which should automatically determine the surfaces to be machined based on the geometric information provided by the CAD database. To determine the surfaces to be machined, the manufacturing features of the part must first be identified. This requires manipulating the geometric data provided by the CAD database to determine the geometric and topological relationships among the data. This information would then serve as input to a feature recognition algorithm. Several feature recognition algorithms have been developed for different CAD modeling techniques: wire-frame, surface representation, and solid models. Section 1.1 provides a brief description of these CAD representation schemes in relation to feature recognition. Section 1.2 outlines the problem statement, followed by the research objectives in Section 1.3, and the thesis outline in Section 1.4.
Figure 1.1 Process-planning modules and databases
1.1 CAD Representation schemes

Feature recognition algorithms are heavily dependent on the type of data provided by the geometric modeler of the CAD system. The geometric modeler determines the method by which the user can create design drawings and the data structure for storing the CAD model in the computer memory. The data structure is determined by the type of information to be stored in the CAD database. The major geometric modelers are the wire-frame, surface representation, and solid models.

The wire-frame technique is the most widely used method for CAD representation because of its simplicity in development and usage. Under this technique, the object is represented in terms of edges, vertices and curves. This approach suffers from the problem of creating ambiguous objects. Visually a wire-frame object is confusing to the human eye. Additionally, complex objects cannot be very easily modeled. Information on the relationships between the basic geometric entities is not maintained, and in turn obtaining surface information requires some processing. It is possible to construct the topology for wire frame models, and thus most of the aforementioned problems could be resolved. This would combine the advantage of ease of development and use of the CAD system with the advantage of accessing topological information for computer-aided processing planning.

Surface representation models are created by connecting different types of geometric elements of a surface to line segments or boundaries. The advantage of this approach over wire-frame modeling is the availability of surface information. This helps in hidden-surface removal and overcomes the ambiguities of wire-frame models. Also,
because of the availability of the structure boundaries, surface representation can be used for NC programming. However, a great deal of processing is still required for building topological information.

The solid models technique is better than the above two methods in terms of the level of geometrical and topological relationships provided by its data structure. Surface information required for feature recognition is therefore easier to obtain. However, object modeling requires the user to think at a higher level of geometric abstractions. The most popular solid-modeling techniques are the Constructive Solid-Geometry (CSG) and the Boundary-Representation (B-rep). In CSG technique, the solid model is constructed by the combination of different solid primitives, such as cuboids, cylinders, spheres, and torus. The primitives are combined by the application of Boolean operations including union, difference and intersection. Therefore, the feature information is explicit but the geometric model is implicit. The user can change the size of the primitives and their location and orientation in space.

Figure 1.2 illustrates the creation of a CSG component using primitives. The model can be represented by a binary tree structure in which the terminal nodes are the primitives or a combination of primitives and the non-terminal nodes are the Boolean operators. A main drawback of CSG modeling is its non-uniqueness. The same model of Figure 1.2 can be represented in a different way as shown in Figure 1.3. This creates a problem for feature recognition since conversion of the CSG model to its boundary representation would be required.
Figure 1.2 CSG modeling and its data structure
Figure 1.3 Alternate CSG representation of drawing in Figure 1.2
The boundary representation or B-rep is a technique in which the solid model is built in terms of its faces, edges and vertices. Topological relationships, which are maintained in the data structure, provide the adjacency relationships between the geometric primitives. The data structure also gives surface information which is essential for feature recognition. Figure 1.4 illustrates a B-rep model along with its data structure. The feature information in B-Rep is implicit and the geometric model is explicit, which is the reverse of CSG representation. Part modeling is not easy, since the user again has to think at a higher level of geometric abstractions. In addition, the adjacency relationships between the geometric primitives must be evaluated for feature recognition purposes.

1.2 Problem Definition

In spite of the advancement in the applications of computers in the area of CAD and CAM these two areas are still "islands of automation". The translation of data from CAD systems to the information needed by CAM systems is mainly done manually. This represents a bottleneck to productivity improvement in the overall design/production cycle. The integration of these two systems is thus an important goal.

CAM systems require feature information for most of their functions. Unfortunately, current CAD systems can only provide primitive geometric entities to the CAM systems. This necessitates an intermediate stage of conversion of these low-level geometric primitives to higher-level feature information for achieving a true CAD/CAM integration. The process of conversion of low-level data to higher-level feature
Figure 1.4 B-rep solid model and its data structure
information involves feature recognition and extraction. Feature recognition involves building up the geometric and topological relationships of the geometric model from a CAD data file and isolating the manufacturing feature information. The process is further complicated with the presence of different CAD representation schemes, where each scheme provides a different geometric information to the feature recognition module.

The variation among the geometric information provided by the CAD systems could be resolved by having only one method of CAD model representation. However, this is a serious restriction, since different representation schemes serve different objectives. Although some research work is presently directed towards developing a feature-based CAD system, such systems are not available yet. Therefore, there is a need for a file format that would: (1) include geometric as well as topological information needed for feature recognition and extraction, and (2) be independent of the different CAD representation schemes. Such a file can serve as a pre-processor file for the feature recognition module.

1.3 Research Objectives

The objective of this research is three-fold:

(1) To develop a neutral file format for feature recognition modules needed for generative CAPP systems. The developed file format is called the Complete Geometric and Topological File Structure (CGTS).

(2) To develop algorithms for the conversion of wire-frame data to the CGTS file format.
(3) To demonstrate the applicability of CGTS for feature recognition purposes.

CGTS acts as an input file to any software module dealing with feature recognition. The geometric and topological information in CGTS is maintained at a level suitable for feature recognition and extraction. The file is neutral and thus should be the same irrespective of the CAD modeler used.

1.4 Thesis Outline

The literature survey of related work is provided in Chapter 2. The details of the Complete Geometric and Topological File Structure along with the algorithms adopted in converting the wire-frame data to the CGTS format are described in Chapter 3. The rules developed for feature recognition using the CGTS format are also provided in this chapter. Test runs and results obtained are documented in Chapter 4. Chapter 5 summarizes the research findings and provides possible future work.
CHAPTER 2

LITERATURE REVIEW

The majority of feature recognition algorithms can be classified based on the: (1) technique used - syntactic pattern recognition, graph theory or expert systems, (2) type of features being addressed - rotational and/or prismatic, or (3) the type of information provided by the CAD modeler - wire-frame, surface or solid model representation. This chapter discusses the relevant publications of feature recognition and extraction in terms of the technique used along with the type of features being addressed and the utilized CAD modeler. Section 2.1 describes the syntactic pattern recognition algorithms, Section 2.2 provides the graph theory techniques, and Section 2.3 provides the expert system techniques. Section 2.4 provides a summary of these research efforts.

2.1. Pattern Recognition Approach

In pattern recognition, the geometric model is decomposed into pattern primitives. A parser is used to check the pattern primitives to determine if a set of these primitives can form a pattern class. Pattern classes are defined with rules or grammars based on the type of feature: rotational or prismatic. If a set of pattern primitives match a pattern class, a
feature is recognized. A pattern recognition system is, therefore, composed of the input string obtained from the geometric model, pattern grammars and a parser [2].

Normally, pattern primitives are symbolically defined for each type of basic geometric entity. For each of the geometric entities encountered during the decomposition of the geometric model into pattern primitives, a corresponding pattern primitive is assigned and concatenated into a string. In the example given in Figure 2.1, pattern primitives A, B, and C are defined for the corresponding geometric entities. Pattern primitives can have associated lengths. The string is constructed by analyzing the geometric model, say, from the left to the right. The string formed out of this example is thus A(a)B(b)A(c)C(d)A(e), where the symbols in capitals refer to primitives and the small letters in brackets refer to the lengths of the primitive. This type of a string represents a slot and thus a feature is recognized.

Staley, Henderson and Anderson [3] described a system for recognizing holes from a 3-D solid geometric data base. Once the user selects a hole in an interactive fashion, syntactic pattern recognition techniques are applied to convert the data into a string of basic geometric primitives that form the hole. The string is a 2-D representation of the 3-D data. Earley's parsing algorithm is used to check if the input string is a part of the pre-defined hole grammars. Checking is done from the simplest hole pattern to the most complex one. If the input string matches any of the grammars, the system returns a value 'true' and specifies the type of hole that is recognized. If the string is not part of the grammar then a value of 'false' is returned. The 3-D solid objects were created using the solid modeller Romulus. This system was implemented in Lisp on a VAX 11/780 with a UNIX operating system.
Liu and Srinivasan [4] used syntactic pattern recognition techniques to determine the feasibility of producing a geometric feature on a machining center. The set of machining surfaces that the machining center is capable of generating is pre-defined. The surfaces are represented by a pattern string. A parser parses this string to determine if it is part of the machining center's capability. If the pattern matches a particular grammar, then the string is classified as machinable. The approach to feature recognition and extraction has not been specified, nor has the algorithm been implemented.

A methodology was presented by Srinivasan, Liu and Fu [5] for the extraction of features from a 3-D solid model. The purpose is two-fold: part family formation and cutter path generation. A pattern string is created based on the 3-D solid model data input from the CAD system. The pattern string is represented as a tree structure which is similar to the tree structures used in the solid modeling systems. Parsing is done using the Earley's Parsing algorithm, and the coding is done using the C language.

Li and Bedsworth [6], developed a syntactic pattern recognition algorithm to detect rotational features from an Initial Graphics Exchange Specifications (IGES) file for CAPP. The input to this algorithm is the 2-D primitives from the IGES file. These primitives are converted into the pattern string, and the algorithm parses the string to extract the rotational features. The parts have to be symmetrical along the X-axis. The implementation was done using the CAD software CADAM.
2.2 Graph Theory Approach

Topological relationships between geometric entities of a CAD model can be represented using graphs. Topological data represented as graphs can be utilized to simplify the analysis of the geometric entities. Relevant definitions from graph theory with respect to the literature survey follow.

Graph: A graph $G$ is an ordered triple: a set of non-empty vertices, a set of edges and an incidence function that associates a pair of vertices for each edge of the graph.

Adjacency matrix: An adjacency matrix associated with a graph is a matrix $(n \times n)$, where $n$ is the number of vertices of the graph, each $(i,j)$ value of the matrix represents the number of edges joining vertex $i$ and vertex $j$.

Isomorphism: Two graphs $G$ and $H$ are said to be isomorphic, if there is a one-to-one correspondence between their vertices and edges, and the incidence relationships remain unchanged. Figure 2.2 explains isomorphism with an example[7].

Subgraph: A graph $H$ is a subgraph of $G$ if the number of vertices and edges of $H$ is a subset of the number of vertices and edges of $G$ and the incidence relationships remain the same for the subset of vertices and edges. If $H$ is a proper subset of $G$, $H$ is called as a proper subgraph of $G$. Figure 2.2 explains a sub-graph.
A graph of a cube (8 vertices, 12 edges)

A sub-graph of the graph of the cube

Graph $G$

Graph $H$

Graph $G$ and graph $H$ are isomorphic

Figure 2.2 Graph theoretic terminology
**Paths and cycles:** A path is defined by a set of distinct consecutive vertices and edges. A closed path which has non-distinct beginning and end vertices is called as a cycle or a loop.

**Connected graphs:** A graph $G$ is said to be connected if for every $(u,v)$ pair of vertices there is at least one $(u,v)$ path. A connected graph always has only one component.

**Cut Set:** A cut set is a set which consists of a minimal set of edges or vertices whose removal separates the connected graph into two or more components. If the cut set has $k$ elements, all of which are edges, the graph is said to have an edge connectivity of $k$, and if the elements are vertices the graph is said to have a vertex connectivity of $k$ or simply the graph is $k$-connected. Figure 2.3 explains connected graphs and vertex connectivity.

With respect to the topological model, data elements consisting of vertices, edges and faces, can be structured either as an edge-vertex or an edge-face graph.

**Edge-Vertex graph:** In an edge-vertex graph, the vertices of the graph represent the vertices of the geometric models and the edges of the graph represent the edges of the geometric model. The adjacency of vertices in the model is mapped accordingly to the graph.

**Edge-Face graph:** In an edge-face graph, the nodes of the graph represent the faces of the geometric model and the arcs of the graph represent the edges of the geometric model.
A one-connected graph $G$ (one component) with the cut vertex

Component 1
Component 2

Graph $G$ has two components after the cut-vertex is removed
Each component is 2-connected

Figure 2.3 Graph theoretic terminology
Adjacent nodes represent adjacent faces with the arcs denoting the common geometric edge between them.

Floriani [8] proposed a generalized edge-face graph as the basis for the extraction of features such as protrusions, depressions, through holes and handles. A B-rep model is used as the input. The edges and faces extracted from the solid model form the edge-face graph. This graph represents the object globally and provides multiply-connected features. The feature extraction algorithm analyses the edge-face graph and creates an Object Decomposition Graph data structure. This data structure is a decomposition of the main edge-face graph into its components. Each component of the object decomposition graph represents a feature. Floriani and Bruzzone [9] described an algorithm which is an improvement over the previous algorithm in terms of its ability to recognize through-holes and handles. A graph called as the Symmetric Boundary Graph (SBG) is created out of the B-rep data. This graph describes the faces, edges, vertices and loops of the object. It gives the relationships between the faces and loops, between the edges and loops and between its edges and vertices. The rest of the procedure followed is similar to the previous approach. The algorithm has not been implemented.

An algorithm was developed by Falcidieno and Giannini [10] for the extraction of features of a prismatic part based on graph theory and syntactic pattern recognition techniques. The algorithm operates in three steps - feature recognition, feature extraction and feature organization. During the feature recognition stage protrusions and depressions are extracted by using syntactic pattern recognition techniques. In the next stage of feature extraction, the algorithm adds dummy surfaces to the faces recognized in the feature recognition stage to make the resulting object conform to Euler's formula.
This is done to determine the amount of material to be removed to create a feature. During the third stage the features are arranged in a graph in a hierarchical fashion with nodes representing the features and the arcs representing the relationships between the nodes (or features). The algorithm has not been implemented.

Joshi and Chang [11] developed a graph-based methodology for the extraction of features, such as slots and holes, from a B-rep model. A face-edge graph, called Attributed Adjacency Graph (AAG), is formulated based on the input from the B-rep data. The vertices of the graph represent the faces of the part under consideration and the edges represent the edges shared by the faces. The angle formed at an edge by adjacent faces is assigned as an attribute to the corresponding edge, convex = 1 and concave = 0. The graph is complete when all the information about the part is arranged in the graph. A feature is considered as a sub-graph of this main graph. The algorithm then proceeds by isolating sub-graphs and with the help of heuristic rules recognizes the type of feature the sub-graph represents. The implementation has been done in FORTRAN 77 on a Sun 3/50 work-station and using the solid modeler ROMULUS.

A methodology for the extraction of slot features is described by Chuang and Henderson [12]. The procedure consists of extracting edge-vertex information from a B-rep model and storing it as an edge-vertex graph. Patterns are defined for various feature types. These patterns are sub-graphs of the main edge-vertex graph. The sub-graphs are then extracted out of the main graph and these are matched against the pre-defined patterns. The matched pattern is the feature that the sub-graph represents. The system has been implemented using the solid modeler ROMULUS and the PROLOG language.
Gavankar and Henderson [13] presented an algorithm for the extraction of protrusions and depressions. The B-rep model is converted to an edge-face graph. The algorithm searches the edge-face graph for faces with multiple edge loops. An object having a face with multiple edge loops is a one-connected graph. The algorithm then decomposes the graph into its bi-connected components which represent features such as depressions and protrusions. The algorithm has not been implemented.

2.3 Expert System Approach

A computer program which captures heuristic knowledge is called as a Knowledge Based Expert System (KBES). In contrast to conventional programs which are controlled by algorithms, expert systems are controlled by an inference engine which determines the rules to be fired for an outcome. The heuristic knowledge can be represented either as rules or as frames. In rule-based systems, a set of production rules determine the search through a search space to trigger an action. In frame-based systems, knowledge is organized in frames or structures or classes. Knowledge is abstracted in a hierarchy of classes which provide for inheritance. Frames are defined with attributes that apply both to the frame as well as its descendant frames. The descendant frames are specialized by adding additional attributes. The two basic styles to define the direction of search for solutions are the data-driven reasoning or forward chaining, and the goal-driven or backward chaining. In forward chaining, data is matched against the rules, to check for rules that apply and fire them, and takes the actions as a result of the rules fired. Backward chaining follows an inverse procedure by starting with a goal and moving backwards to establish relevant facts.
A system for the extraction of form features such as holes and slots was developed by Henderson and Anderson [14]. This system, called FEATURES, consists of three modules: a feature recognition module, a feature extraction module and a feature graph construction module. Faces, edges, and vertices are extracted from the B-rep model and these are represented as facts. These facts are searched and rules of logic are applied to detect the presence of features. The next step is the extraction of the features recognized in the previous stage. The last stage is the construction of the feature graph. This graph is linked according to the adjacency of the features. Features which have a common entrance face are linked directly to the main node whereas the other features are linked according to their adjacency with the features already linked to the main node. The algorithm is implemented in PROLOG and uses the solid modeler ROMULUS.

A system called turbo-CAPP was developed by Wang and Wysk [15] for process planning. The system is capable of recognizing only rotational features. Geometric entities from a 2-1/2 D CAD system are used as input to the system, which determines the machining surfaces of the part. The turbo-CAPP system has the capability of generating alternate process plans based on the type of machine. A manufacturing database is maintained, which is capable of providing machining parameters for the various operations needed to machine the part. An expert system is used for the generation of machining parameters. It is a combination of a rule-based as well as a frame-based system. It is implemented using AutoCAD.

Henderson and Chang [16] described a research project where an automatic process planning system has been developed which uses a feature extraction system called
Feature Recognizer and Process Planner (FRAPP). The solid model input can be either of a CSG or B-rep type. The knowledge-based system is capable of extracting features such as slots and holes. The knowledge-based system is built using an expert system tool KEE and the LISP language. The knowledge regarding the features, machine description and manufacturing criteria are stored as frames. The procedure consists of two basic steps: Feature Recognition and Process Planning. Feature recognition consists of recognition, extraction, analysis, computation and compilation. Recognition and computation deals with extraction of slots and holes on the part under consideration. Macro-features are constructed if there is a necessity to maintain constraints for the sequence in which the features are to be machined. Analysis and computation deals with deduction of the parameters of the features and compilation finally arranges all this detail in a features graph which is nothing but the frame-based representation of the extracted features. The next step is Process Planning. The Process Planner automatically determines the machining sequence required and also the machines needed from its database. The solid modeling system used is ROMULUS. The system is implemented on a TI-explorer workstation.

In the literature surveyed, some of the work does not belong to any of these three major categories. These are discussed in the following section. The discussion includes the semi-generative approach for process planning and the usage of CSG in the feature extraction approach. Chang and Wysk [17] described a system called Totally Integrated Process Planning System - TIPPS, which is a semi-generative process planning system. The user selects the surfaces that are to be machined. The system uses this information to extract the geometric entities that form the surface. By the application of an algorithm it
detects the type of feature that is formed out of the surfaces selected. The system is capable of extracting both prismatic as well as rotational features.

An algorithm was developed by Ferreira and Hinduja [18] for the recognition of slots, holes and protrusions, from a 2-1/2-D model. The algorithm works in three stages. During the first stage called as the convex-hull method, features are extracted from the convex faces of the object. The second stage of the algorithm extracts features which originate from the inner loops of the edges. Also, any face belonging to the extracted features, but not extracted during the previous stage, are extracted to make the feature description complete. During the third stage, features which have common geometric characteristics are glued together. This helps in manufacturing the glued features with a single tool. The algorithm is implemented using FORTRAN IV on the solid modeler GPM, on a Sun 3/50 work-station.

The problems related to the non-uniqueness of the CSG representation of solid models is addressed by Lee and Fu [19]. As the CSG model of a particular feature can be developed in a number of ways, feature recognition is severely affected. Their research proposes a two-step procedure, for feature recognition from a CSG representation of the model, namely feature extraction and unification. During the extraction process the CSG tree is traversed and all the principal axes of the primitives are extracted. Next, based on certain pre-conditions for the definitions of a feature, the principal axes are clustered together. The next stage in the process deals with the unification of the features and the re-construction of the CSG tree based on this unification. This attempts to establish a unique representation of the feature within the CSG tree, to avoid the ambiguity of CSG modeling. The algorithm is not implemented.
Perng, Chen and Li [20] developed a procedure for the extraction of machining features like slots and holes from a 3-D CSG input. The input to the system is the CSG model of the part. The algorithm converts this into a destructive solid geometry of the part. CSG of a part is a combination of union and difference operation of different primitives. Destructive solid geometry converts all the union operations into difference type of operations. Thus the extracted features represent machining volumes. The extracted features are arranged in a tree structure of machining volumes with all geometric operations of the difference type. The implementation is in the C programming language on a Masscomp 5500 computer.

2.4 Summary of Research Efforts

Table 2.1 provides a summary of the feature recognition approaches, categorized by - author, type of input, feature types, implementation, and the technique used. Among the research efforts that have used pattern recognition, [3,5,6], the input has been either solid model data or wire-frame data. But in all the cases the pattern representation has been a 2-D representation of the 3-D data. This leads to loss in depth information, and is not suitable for CAPP purposes. Moreover, pattern representation of geometries which are not symmetric around an axis is very complex.

In the expert systems approach [14, 16, 20], the object geometries are represented as facts, and rules are utilized to determine the presence of features. The complexity of the rules is dependent on the level of geometric data. Geometric data at a low
Table 1  A summary of the feature recognition approaches.

<table>
<thead>
<tr>
<th>Author</th>
<th>Input</th>
<th>Features</th>
<th>Implementation</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Li, Bedworth</td>
<td>Wire frame</td>
<td>Rotational</td>
<td>Yes</td>
<td>Pat. Rec.</td>
</tr>
<tr>
<td>Staley, Henderson</td>
<td>B-rep</td>
<td>Prismatic</td>
<td>Yes</td>
<td>Pat. Rec.</td>
</tr>
<tr>
<td>Anderson</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Liu, Srinivasan Fu</td>
<td>Wire-frame</td>
<td>Pris./Rot.</td>
<td>Yes</td>
<td>Pat. Rec.</td>
</tr>
<tr>
<td>Chang, Wysk</td>
<td>Wire frame</td>
<td>Pris./Rot.</td>
<td>Yes</td>
<td>Rules</td>
</tr>
<tr>
<td>Gavankar Henderson</td>
<td>B-rep</td>
<td>Prismatic</td>
<td>No</td>
<td>Graphs</td>
</tr>
<tr>
<td>Floriani</td>
<td>B-rep</td>
<td>Prismatic</td>
<td>No</td>
<td>Graphs</td>
</tr>
<tr>
<td>Joshi, Chang</td>
<td>B-rep</td>
<td>Prismatic</td>
<td>Yes</td>
<td>Graphs</td>
</tr>
<tr>
<td>Chuang, Henderson</td>
<td>B-rep</td>
<td>Prismatic</td>
<td>Yes</td>
<td>Graphs</td>
</tr>
<tr>
<td>Falcidieno, Giannini</td>
<td>B-rep</td>
<td>Prismatic</td>
<td>No</td>
<td>Graphs</td>
</tr>
<tr>
<td>Henderson, Anderson</td>
<td>B-rep</td>
<td>Prisntatic</td>
<td>Yes</td>
<td>Exp. Sys.</td>
</tr>
<tr>
<td>Floriani, Bruzzone</td>
<td>B-rep</td>
<td>Prismatic</td>
<td>No</td>
<td>Graphs</td>
</tr>
<tr>
<td>Perng, Cheng, Li</td>
<td>CSG</td>
<td>Prismatic</td>
<td>No</td>
<td>Rules</td>
</tr>
<tr>
<td>Lee, Fu</td>
<td>CSG</td>
<td>Prismatic</td>
<td>Yes</td>
<td>Rules</td>
</tr>
<tr>
<td>Henderson, Chang</td>
<td>B-rep/CSG</td>
<td>Prismatic</td>
<td>Yes</td>
<td>Exp Sys.</td>
</tr>
<tr>
<td>Wang, Wysk</td>
<td>Wire-frame</td>
<td>Rotational</td>
<td>Yes</td>
<td>Exp. Sys.</td>
</tr>
<tr>
<td>Ferreira, Hinduja</td>
<td>Wire-frame</td>
<td>Prismatic</td>
<td>Yes</td>
<td>Rules</td>
</tr>
</tbody>
</table>
level of abstraction may lead to complex rules. CSG models provide geometric data at a higher level of abstraction, but suffer from the problem of ambiguities and cannot be used with ease in rules. Reference [19] addresses the problem of ambiguities in detail. For example, in the work by Perng, Chen and Li [20], the CSG data model is first converted to a destructive solid geometry, to make the analysis easy. This is due to the fact that CSG data must be converted to its boundary representation for a detailed analysis.

Graph theory technique has good potential in feature recognition problems when used in conjunction with B-rep and related kind of data [8, 19, 10, 11, 12, 13]. One advantage is that the analysis can be done independent of the geometry. At the same time, as a graph models the topology in terms of the geometric primitives, geometric information can be explicitly obtained.

From the above discussion and Figure 2.4, it is evident that for different CAD modeling techniques, different feature extraction algorithms are needed. Due to this, feature recognizers have no consistent interface to CAD data. Feature recognizers would be vastly aided by a file format containing CAD data meant specifically for feature extraction purposes. This file format would remain consistent irrespective of the modeling system used as depicted in Figure 2.5. A file format meant specifically for this purpose is developed in this research (the Complete Geometric and Topological File Structure, CGTS). This results in shielding feature recognizers from the complexities of CAD
Figure 2.4 Data modeler dependant feature recognition
Figure 2.5 Data modeler independant feature recognition
implementations. Feature analysis will be done more efficiently by this consistent interface. Though Product Data Exchange Specifications (PDES) addresses the problem of representing data independent of the CAD implementation, there is no formal approach to develop a file format for process planning purposes [21].
CHAPTER 3

CGTS BASED FEATURE RECOGNITION SYSTEM

This chapter describes the feature recognition system developed that uses the CGTS file format. This is in terms of the design and implementation details of the CGTS file structure, the algorithms developed for the conversion of wire-frame data to the CGTS format, and the rules developed for feature recognition from this file format. Section 3.1 discusses the CGTS file structure. Section 3.2 discusses the methodology adopted in exporting wire-frame data from the CAD data file to the CGTS format. Section 3.3 discusses the rules developed for the recognition of features from the CGTS file format.

3.1 CGTS file structure

Spatial attributes of an object can be classified into two basic categories:

1. **Geometry**: Geometry gives information on the location of elements in space. It is useful for feature recognition, as it helps in determining the spatial orientation, and in obtaining additional information like length, area and volume.
2. **Topology**: In addition to geometry, feature recognition requires the spatial relationships between the geometric entities, given by the topology of the part. The topology of an object provides the spatial connectivity and contiguity.

The design of the CGTS file is based on these considerations. The CGTS file is divided into four sections: Header Section and Vertex Description Section, which together provide the geometric information, the Edge Description Section and the Face Description Section, which together provide the topological information.

### 3.1.1 The Header Section

Figure 3.1 shows an example of the Header Section of the CGTS file. The information in this section is meant to be documentary in nature as well as a description of the object in a global fashion. The first record is always reserved for title information. There are a total of eight records in the Header Section, which flag some attribute of the section. Flags of this kind, are used in all the four sections, and they either indicate the presence of the attribute value in the same record as they appear, or in the following record. The attributes of the Header Section are:

1. "**HEADER SECTION**": Flags the beginning of the Header Section.

2. "**END_HEADER_SECTION**": Flags the end of the Header Section.
THE GEOMETRIC AND TOPOLOGICAL FILE
HEADER SECTION
X_RANGE
5 15
Y_RANGE
5 15
Z_RANGE
5 10
N_VERTICES
24
N_EDGES
36
N_FACES
16
END_HEADER_SECTION

Figure 3.1 The Header Section of CGTS
3. "X_RANGE": The record following this flag provides the range of X-values, in an integer format and separated by spaces.

4. "Y_RANGE": The record following this flag provides the range of Y-values, in an integer format and separated by spaces.

5. "Z_RANGE": The record following this flag provides the range of Z-values, in an integer format and separated by spaces.

6. "N_VERTICES": The record following this gives the number of vertices of the object.

7. "N_EDGES": The record following this gives the number of edges of the object.

8. "N_FACES": The record following this gives the number of faces of the object.

The X, Y, and Z coordinate ranges can be used to calculate the enclosing dimensions of the object, and the total volume of the raw stock to be used for machining the object.
3.1.2 The Vertex Description Section

This section provides detailed data on the vertices of the object, with topological references to edges associated with this vertex. Figure 3.2 gives an example of the Vertex Description Section. This section is described by four attributes. The attributes are:

1. "VERTEXDESCRIPTIONSECTION": Flags the beginning of the section.

2. "ENDEVENTDESCRIPTIONSECTION": Flags the end of the section.

3. "V": This flags a vertex, and the integer value in the record is the identification number for the vertex. The coordinate values of the vertex are given in the next record. This is the only place in the file where any coordinate values are stored. The vertex identification number acts as a pointer to coordinate information.

4. "EDGEN": The record following this flag, gives the numerical identification of the edges generated by the referenced vertex. This information is useful in determining the location and orientation of the vertex, for edge traversal problems.
VERTEX_DESCRIPTION_SECTION
V  1
15  5  10
EDGEN
1  8  33
V  2
5  5  10
EDGEN
1  2  34
V  3
5  15  10
EDGEN
2  3  35
V  4
15  15  10
EDGEN
3  4  36
END_VERTEX_DESCRIPTION_SECTION

Figure 3.2 The Vertex Description Section
3.1.3 The Edge Description Section

The Edge Description Section, which is topological in nature, provides detailed information on the edges that form the object. The Edge Description Section, given in Figure 3.3, is described with the help of five attributes. They are:

1. "EDGE_DESCRIPTION SECTION": Flags the beginning of the section.

2. "END_EDGE_DESCRIPTION_SECTION": Flags the end of the section.

3. "E": Flags the edge, followed by the numerical identification of the edge on the same record.

4. "VIDS": The record following this flag provides the numerical identification of the vertices on which the edge is incident.

5. "PATHS": This flag is for the edges that originate from the end-point of the present edge. The end-point is defined as the second of the two vertices listed for the referenced edge. The record following this, gives the actual numerical identifications for the edges. This information could be used for traversal purposes.
EDGE_DESCRIPTION SECTION
E  1
VIDS
1  2
PATHS
2  34
E  2
VIDS
2  3
PATHS
3  35
E  3
VIDS
3  4
PATHS
4  36
E  4
VIDS
4  5
PATHS
5  25
END_EDGE_DESCRIPTION_SECTION

Figure 3.3 The Edge Description Section
3.1.4 The Face Description Section

The Face Description Section, which is topological in nature, gives detailed information on the faces that form the object. Adjacency relationships are provided to aid the process of feature recognition. Figure 3.4 gives an example the Face Description Section. This section is described by six attributes. They are:

1. "FACE_DESCRIPTION_SECTION": Flags the beginning of the section.

2. "END_FACE_DESCRIPTION_SECTION": Flags the end of the section.

3. "FACE": This flags the beginning of a face description, and it records five integer valued attributes. They are in the order:

   a) Identification number for the faces.
   b) Number of edges of the face.
   c) Number of adjacent faces.
   d) Flag, if face is circular or not: 1 - Yes, 0 - No.
   e) Flag, if loop or not: 1 - Yes, 0 - No.

4. "EDGE_NUMBERS": The record with this flag contains integer values, equal in number to the number of edges of the face. The integer values give an ordered listing of the edge identification numbers of the face.
FACE_DESCRIPTION_SECTION
FACE    1 8 8 0 0
EDGE_NUMBERS  1 8 7 6 5 4 3 2
ADJACENT_FACES
  3 4 4 6 10 11 14 16
ADJACENCY_RELATIONSHIP
  4 1 2 3 8
  4 1 2 3 4
  6 1 2 3 3
  10 1 2 3 7
  11 1 2 2 5
  14 1 2 3 6
  16 1 2 2 2
FACE    2 4 4 0 1
EDGE_NUMBERS  16 13 14 15
ADJACENT_FACES
  8 9 12 15
ADJACENCY_RELATIONSHIP
  8 1 2 2 14
  9 1 2 2 13
  12 1 2 2 15
  15 1 2 2 16
END_FACE_DESCRIPTION_SECTION

Figure 3.4 The Face Description Section
5. "ADJACENT_FACES": The record following this contains a list of numbers, equal in number to the number of adjacent faces. This list identifies the adjacent faces by their numerical identification.

6. "ADJACENCY_RELATIONSHIP": This flags the adjacency relationship values, with one record for each adjacent face. Each record has five values. The values correspond to adjacent face identification number, relationship flag value for X, Y and Z ranges (discussed later) and the identification number for the common edge between the two adjacent faces.

3.2 Methodology for conversion of wire-frame data to CGTS format

The block diagram in Figure 3.5 is used to explain the CGTS based feature recognition system, starting from the CAD model creation to the detection of features with the help of data available from the CGTS file. The CAD model is created, followed by the input of data from a given modeler. This data is processed to detect the faces that form the object, and determine the face linkages for topological information. The topological relationships are further qualified to make it suitable for the CGTS format. At this stage, the CGTS file is created. Finally, the feature recognition and extraction process is initiated.

The system has been implemented on an IBM PC, with a 386SX processor. The problem domain has been broken down into five logical classes. The data in each instance of these classes is complex in nature, and it would be desirable to keep it safe from being
Figure 3.5 Conversion of wire frame data to CGTS format
corrupted. It is also desirable that these classes be extendible for any further additions made to the CGTS file format, and also encapsulate unnecessary details from the extension library. Due to these considerations, the C++ language has been chosen, using the Borland C++ 2.0 compiler. The five classes of the domain are:

1. dxf class: for input data from the DXF file.
2. vertex class: for vertex entities.
3. edge class: for edge entities.
4. face class: for face entities.
5. topofile class: for the output of the CGTS file.

The interaction between the classes is described in Figure 3.6. A discussion on the methodology adopted follows.

1. **Model creation:** The CAD model is constructed with the help of AutoCAD using wire frame modeling. Wire frame modeling was chosen for two reasons: 1) due to the vast number of existing drawings in wire frame format, and 2) the need to address the inherent complexity of converting wire frame data to the CGTS format. The model data is stored by AutoCAD in a binary format which is not suitable for processing. AutoCAD provides an ASCII file called as the Drawing Interchange File (DXF) format, which stores CAD data in ASCII format. The model creation is followed by the conversion of the binary CAD data file to the DXF format.

2. **CAD data input:** Geometric CAD data is input from the DXF file for analysis. The file has five sections: Header Section, Tables Section, Blocks Section, Entities Section
Figure 3.6 System Classes and their interaction
and the End of File Section. The Entities Section provides the geometric data needed for the analysis. Figure 3.7 gives an example of the Entities Section. The line is given by the end-points, the records with values 10, 20 and 30 give the X, Y and Z values respectively of one end of the line in the following record, and the records with values 11, 21, and 31 give the X, Y and Z values respectively of the other end of the line in the following record.

3. **Building topology [Spatial connectivity and contiguity]:** The data obtained from the DXF file provides only edge-vertex information. Information on individual faces and the relationships among the faces is vital to the process of feature recognition, providing the ability to determine face linkages/adjacencies. In this phase the spatial connectivity and contiguity of the CAD model is determined. It is accomplished in three phases: Lower-level topology, Higher-level topology, and the detection of loop faces.

**Lower-level topology:** Lower-level topology deals with the detection of face information. Spatial data handling deals frequently with geometric entities at varying levels. References to these entities are made convenient with the help of numerical identification numbers. In this step, the edges obtained from the DXF file are given unique identification numbers. Figure 3.8 illustrates the numbering process. The number appearing beside each edge in the figure is the identification number given to the edge. This example assigns numbers to nine edges, the other edges can be processed in a similar fashion. The order in which the edges actually appear in the DXF file could be different, but the numbering scheme and the processing will remain the same. Figure 3.9 shows the *edge* class data structure used. The identification
class edge
{
    int idno;
    vertex *arr1;
    int vertexid[2];
    int adjedges[2];
    int traversecount;
    int pathflags[2];
    int featureflag;
    int temp, temp1;
public:
    edge( vertex *vt, int n);
    edge() { idno = 0; temp = 0; traversecount = 0;
    pathflags[0] = 0; pathflags[1] = 0; temp1 = 0;
    featureflag = 0;};
    int retidno(void) { return idno; }; 
    void retvert(vertex *ver);
    void show(void);
    friend ostream &operator<<(ostream &stream, edge ed);
    void fileprint(FILE *fp);
    int operator==(edge ed);
    int operator<=(edge ed);
    void assignvertid(int id);
    void assignpaths(int *id1);
    void getendid(int *id2) { *id2 = vertexid[1];};
    void getotherendid(int *id2) { *id2 = vertexid[0];};
    void recordtraversals(int idn);
    int gettraversecount() { return traversecount; }
    int getpathnottaken();
    void getadjedges(int *tt);
    void switchver();
    void getvertid(int *ite);
    void draw(int color);
    void flagasfeature() { featureflag = 1; }
    int retflagasfeature() { return featureflag; };
};

Figure 3.9 Class Edge
number is stored in the variable idno. The identification is a sequential numbering of edges, starting from 1.

Figure 3.8 shows that as vertex number 1 is common to edge 1, 2 and 9, it is repeated in the vertex list. Similarly, other vertices are repeated, and it is important to resolve duplicates and assign unique identification numbers to the vertices. Figure 3.8 illustrates the vertex numbering process. The number appearing at each vertex in the figure is the identification number given to the vertex. Figure 3.10 shows the vertex class data structure. The duplicate vertices are discarded from the list, and a unique identification number is given to each vertex. This example assigns numbers to eight vertices, the other vertices can be processed in a similar fashion. The identification numbers are maintained in the variable idno of the vertex data structure. As and when new information is detected, it is immediately recorded in the respective data structures.

At this stage, the edges and vertices have been assigned unique identification numbers. Face information is still not available. For determination of face information, the paths or edges that each vertex generates is determined. In Figure 3.8, vertex 1 generates edges 1, 2 and 9. This information is recorded in the variable edgenerated of the vertex data structure. Using this information, the radiating edges from the end-point of each edge is calculated, and stored in the variable adjedges of the edge data structure. Edge 2 in Figure 3.8 generates edge 3 and edge 5. This will help in resolving the problem of back-tracking along edges during the construction of faces.
class vertex
{
    int x, y, z;
    int idno;
    int edgenerated[3];
    int flag;
public:
    vertex() { x = 0; y = 0; z = 0; flag = 0; };
    vertex(int a, int b, int c);
    void assignid(int id);
    vertex operator=(vertex vt);
    int operator==(vertex vt);
    int retzval() { return z; };
    int retyval() { return y; };
    int retxval() { return x; };
    int retflag() { return flag; };
    void setflag() { flag = 1; }
    int getref(int flag);
    void show(void);
    friend ostream &operator<<(ostream &stream, vertex vt);
    void fileprint(FILE *fp);
    void retvertex(vertex vt);
    int retidno() { return idno; }
    void assignedgid(int *id1);
    void getedgeids(int *id2);
};
To begin processing the edges, any one edge can be selected from the edge list. Associated with each edge is a counter variable called \textit{traversecount}. This keeps track of the number of times an edge has been traversed. Each edge has to be traversed exactly two times before information on all the faces is available. This is due to the fact that any edge can be common to two and only two faces, with different face equations.

Edge 1 is chosen and the \textit{traversecount} for the edge is checked. If the count is lesser than two, then the edge is considered. Edge 4, which is the first of the radiating edges from the end-point of edge 1 is now considered. A variable called \textit{pathflags} maintains information on edge traversals of radiating edges. If edge 4 was traversed from edge 1, then this direction is ignored, else this direction is taken. If the path was taken, the variable \textit{pathflags} associated with edge 4 is flagged and the \textit{traversecount} for both the edges is incremented. A path is now defined by edges 1 and 4.

The next step is the collection of vertices along the current path. Three unique vertices 1, 4 and 3 are available, in that order. These three vertices are used to compute the equation of the plane passing through them. The list of vertices is then searched to extract all the points that satisfy this equation. The only other vertex that satisfies this equation is vertex 2, completing a list of all points that constitute the face. From this list, the list of ordered edges 1, 2, 3 and 4, comprising this face is computed and pointers to these edges are stored in the variable \textit{edgeidnos} of the face data structure, given in Figure 3.11. A unique identification is given to the face and stored in the variable \textit{idno} of the face data structure. Now that edge 4 has been traversed, edge 9 remains to be traversed. This traversal provides a similar path defined by edges
class face
{
    int numvertex;
    int idno;
    vertex *arr;
    int *edgeidnos;
    int numadjfaces;
    int *adjfaceids;
    int *adjrel;
    int *commonedgeid;
    int loop;
    char feature[80];
    int featureflag;
public:
    face( vertex *vt, int n ,int m),
    face(int *idnos);
    int retidno(void) { return idno; };
    void retedgeids(int *ids);
    void adjacency(face *arrayoffaces, int facecount);
    void retadjfaceids(int *addids);
    void retvert(vertex *ver);
    int retnoedges(void) { return numvertex; };
    int rtnovers(void) { return numvertex; };
    int retnumadjfaces(void) { return numadjfaces; };
    face() { numvertex = 0; idno = 0, loop = 0;};
friend ostream &operator<<(ostream &stream, face tdp);
    void fileprint(FILE *fp);
    void retrzrange(int *zval);
    void retyrange(int *yval);
    void retxrange(int *xval);
    void calcadjrel(face *arrayoffaces);
    void determinefeature(int *xrange, int *yrange,
    int *zrange);
    int operator==(face fc);
    void changevert(vertex *vert);
    void flagasloop() { loop = 1; };
    void detectloops(face *objface, int num);
    void setflag() { loop = 0; };
    int getfeatureflag() { return featureflag; };
};

Figure 3.11 Class Face
1 and 9, and a set of three vertices that define a plane equation. The rest of the processing is as before. This process of traversing the edge list is continued in a similar fashion. At each pass, a check is placed on the traverse count of all the edges. When this traverse count is exactly two for all the edges, the face construction is complete. Each time a face is detected, a unique numerical identification number is given.

**Higher-level topology:** Higher-level topology establishes face linkages/adjacencies information for each face. At this stage, the face list is complete. Feature recognition algorithms need to know the relationships of one face to another in the face list. The first step is to detect all the faces linked to each face. This is accomplished by checking the edge list for each face with the edge list of every other face in the face list. Faces that share a common edge with the current face are linked to it. Pointers to the linked faces are stored in the variable *adjfaceids* of the face data structure.

**Detection of loop faces:** A loop face is a face which is totally contained in another face, and both faces share the same face equation. No geometric elements are shared by these two faces. In Figure 3.8, a face is detected when edges 3 and 5 are traversed. However, when the list of vertices which satisfy this equation is built, this will include the vertices 6, 7, 8 and 9 with edges 6, 7, 8, and 9. When the face list is being constructed, the edge list is divided into groups based on the connectivities. A group is defined by a set of edges that form a cycle. If more than one group exists for an edge list, then a loop(s) exists. These loops are stored as ordinary faces in the face list. Feature extraction is not possible until this special case is resolved. Thus, each face is picked from the face list and compared with every other face. If all the edges of
a face are totally enclosed by another face then this face is marked as a loop, and the 
information is recorded in the *loop* variable of the face data structure.

At this point, the objective of conversion of low-level geometric entities to a 
higher-level is complete. Topological relationships regarding the surfaces is now 
available. The next logical step is to qualify the relationships between the surfaces and 
export them to the Complete Geometric and Topological File Structure format.

4. **Qualifying linkage (or adjacency) relationships:** The adjacency information on all 
faces has been determined, and this information is further processed to qualify the 
adjacency relationships. This information is stored in the variable *adjrel* of the face 
data structure. To qualify the adjacency relationships, five flags numbering from 1 to 
5 are given for each relationship. The X, Y and Z ranges of each face is compared 
against the adjacent faces. For sake of convenience, the face being referenced is called 
as the host face and the adjacent faces are called as the guest faces. Based on these 
comparisons, the flags are assigned in the following order:

a. If the lower and upper bound of the host range is greater than the lower and upper 
   bound of the guest range, respectively, the flag value is 5.

b. If the lower and upper bound of the host range is lesser than the lower and upper 
   bound of the guest range, respectively, the flag value is 4.

c. If the lower and upper bound of the host range is equal to the lower and upper 
   bound of the guest range, respectively, the flag value is 3
d. If either the lower or upper bound of the host range is greater than the lower or upper bound of the guest range, respectively, the flag value is 2.

e. If either the lower or upper bound of the host range is lesser than the lower or upper bound of the guest range, respectively, the flag value is 1.

5. Creation of the CGTS file: The data from the wire-frame model has been processed, it is now at a level that makes it suitable to be exported to the CGTS file. Figure 3.12 shows the topofile class that has been created to take care of this function. The four functions `wrtbdrinfo`, `wrverterxinfo`, `wrtedgeinfo` and `wrtfaceinfo`, respectively output the Header, Vertex, Edge and Face Description respectively. The next stage is the extraction of feature information from this data format.

6. Demonstrate feature recognition: As the CGTS file has been created, the process of feature recognition can now be initiated. Rules have been developed based on the CGTS file format, which determine the presence of features. The following section discusses these rules in detail.

3.3 Rules for feature recognition and extraction

There has been no formal and complete effort to classify features as yet. Thus, most of the features that have been addressed in the research papers fall in the general category of either protrusions, depressions, slots, blind slots, through slots, holes, blind

56
class topofile
{
    FILE *fp;

public:

    void assign(FILE *fp1) { fp = fp1; }

    void wrthdrinfo(int *xrange, int *yrange, int *zrange,
                     int nvert, int nedg, int nfac);

    void wrtvertexinfo(vertex *arrayofvertices, int vercount);

    void wrtedgeinfo(edge *arrayofedges, int edgecount);

    void wrtfacinfo(face *arrayoffaces, int facecount);
};

Figure 3.12 Class Topofile
holes, steps and a combination of these formed by intersections. Among the feature types that have been considered in this research work are simple depressions, protrusions, blind slots, through slots, steps and holes. Combination type of features are not considered. Also, it is assumed that the faces that form the object are parallel to the principal axes. Curved edges are not considered. Curved edges are a special case of edges, and could be implemented by creating a curve class which inherits the properties of the edge class, and specializes it with the addition of a data element for curvature.

This section enumerates some of the rules used in recognizing features from the CGTS file format. For the detection of features, the check starts with each face. The relationship values associated with each of the adjacent faces is counted, and based on this count the type of feature is determined. Each feature type has certain defined rules. They are:

1. **Depressions**
   
a. If face is a loop. This information is given by the record which flags the beginning of a face. The 5th integer in this record flags a loop with a value of 1.

   b. Count number of adjacent faces with relationship value of 1, along X, Y, or Z. If any one of them has a total equal to the number of adjacent faces, a depression is found.
2. **Protrusions**
   a. If face is a loop. This information is given by the record which flags the beginning of a face. The 5th integer in this record flags a loop with a value of 2.
   b. Count number of adjacent faces with relationship value of 1, along X, Y, or Z. If any one of them has a total equal to the number of adjacent faces, a protrusion is found.

3. **Cylindrical holes**
   a. If the fourth integer values on the record that flags a hole is set to one, this is a loop face which is cylindrical. This signifies a cylindrical hole feature.

4. **Blind slots**
   a. If the count of relationships with a value of 4 is 1 and with a value of 2 is 1 less than the number of adjacent faces, this signifies blind slots.

5. **Steps**
   a. If the count of relationships with a value of 1 is 1, and with a value of 2 is 1, and with a value of 4 is 2 a step is discovered.

6. **Through slot**
   a. If the count of relationships with a value of 2 is 2, and with a value of 4 is 2, a through slot is discovered.
CHAPTER 4

SYSTEM DEMONSTRATION

This chapter demonstrates the CGTS based feature recognition system with the help of a test run. The system runs in three stages: - the Input Stage, the CAD Data Processing and CGTS output Stage, and the Display Stage. Section 4.1 discusses the Input Stage, section 4.2 discusses the CAD Data Processing and CGTS output Stage and section 4.3 discusses the Display Stage.

4.1 The Input Stage

During this stage, the system software is run and the input is obtained from the user. The name of the executable program is FEREC, an acronym for Feature Recognizer. The program runs under the MS-DOS environment, and is executed from the DOS prompt. Figure 4.1 shows the preliminary screen that appears when the program is run. This screen informs the user name of the system, and instructs the user to proceed by pressing any key. At the press of a key, a pop-up menu appears on the right hand side of the screen. The user needs to choose "Start" from the menu to initiate the process. As the system was designed to be generative in nature, the input from the user is kept as minimal as possible. The program assumes that the CAD model has already been created on a CAD system and converted to the DXF file format. The program thus prompts
WELCOME TO THE
Feature Recognition System

Developed By

Dr Osama Eyada and
Rajendra V Auradkar

Industrial and Systems Engineering
Virginia Polytechnic and
State University
November 1992

Please enter the name of the input data file:

Figure 4.1 The Input Stage
the user to input the name of the DXF file that stores the CAD data. This is the only input needed by the system. The name of the output CGTS file is assigned internally, and is the same as the name of the DXF file, with the name of the extension changed to CGT.

4.2 CAD Data Processing and CGTS output Stage

During this stage, the wire frame data from the DXF file is processed to convert it to the CGTS format, and is output to disk. Figure 4.2 shows the screen that appears when the user has finished with the input of the DXF file name. This screen takes the user through the steps that the software is performing. When the CGTS file is output and feature recognition algorithms have been run using the CGTS file, the software instructs the user to press any key to move to the Display Stage.

4.3 Display Stage

During this stage, the CAD object is displayed to the user along with information on the type of features that were detected. Figure 4.3 shows the screen that appears during the Display Stage. The CAD object is redrawn on the screen by using the 2-D graphic capabilities provided by the Borland C++ compiler. Transformations are applied by the software to get the desired 3-D view for the object. The features that were detected are displayed to the user on the top left hand corner. At the press of any key, the run is terminated.
Reading geometric data from the DXF file......

Building topology......

Detecting loops......

Determining adjacent faces......

Calculating adjacency relationships......

Creating CGTS file......

Detecting features......

Press any key to move to the Display Stage......

Figure 4.2 The CAD Data Processing and output CGTS Stage
Protrusion discovered
Blind step discovered

Press any key to exit......

Figure 4.3 The Display Stage
CHAPTER 5

CONCLUSIONS

This thesis discusses the importance of feature recognition for the integration of CAD and CAM through CAPP, and the need for having a consistent interface to CAD data. To address this need, a file format (CGTS) independent of CAD modelers has been developed. Algorithms have also been developed to convert wire-frame data to the CGTS format, and used to demonstrate feature recognition. A discussion on further improvements to this work is the focus of this chapter. Suggestions for improvement and future research work is discussed with respect to enhancements to the CGTS file format, and alternate ways for translation of CAD data to the CGTS format.

5.1 Enhancements to CGTS

The development of any file format structure is always a process of continuous improvement. CGTS also has a scope for further improvement in the level of information that is available to the feature recognition algorithms. Suggestions for further improvements are:

a. In the Face Description Section, the record that flags a face contains five integer values in the same record. The fifth integer flags a loop face, immediately suggesting either a protrusion, depression or a hole feature. Further improvements
in CGTS can be made in a similar fashion, so that other feature types are also explicitly flagged. Apart from providing feature level information, this would help in reducing the search space of the feature recognition algorithms.

b. Gavankar and Henderson [13] use graph theory in analyzing B-rep data, and observe that features like protrusions and depressions appear as one-connected graphs in an edge-face graph. Similarly, other properties of graphs are worthy of investigation for feature analysis. Future work could also address the probability of representing the object as graph theoretic objects in the CGTS file, with specific information highlighting any graph-theoretic property. This could further aid the process of developing standard algorithms for feature analysis.

c. An issue not addressed by the thesis, is the inclusion of tolerance information, which is essential for manufacturing. As tolerance data specifications vary according to the CAD model used, a standard CAD data interface could lead to a simplified and standard way of representing these specifications. Thus, additional structure needs to be given to the CGTS file format to include tolerance information.

5.2. Conversion of CAD data to CGTS

In this thesis, the DXF file format has been used as the source of information for CAD data. An intermediate file could have the disadvantage of loss of information in the process of conversion to this format, as well as adding an additional layer between
the conversion of CAD data to the CGTS format. This can be improved in either or both of the following ways:

a. **Built-in routines in the CAD system**: Figure 5.1 explains the process by including the routines in the CAD system. This conversion process can either be done in parallel with the CAD model creation, or after the CAD model is created, but before it is finally moved to disk from memory. One advantage of this could be the tagging of tolerance information, in an interactive fashion with the user, and writing this information out to the CGTS file.

b. **CAD for CAPP purposes**: CAD systems could be built which are specifically meant for CAPP purposes, and avoid the use of general-purpose CAD systems. Some of the functions that a system of this kind may perform is:

1. Provide multiple CAD modeling types.

2. Model evaluation in a periodic fashion.

3. Signal surface and feature detection to user, and get tolerance information and simultaneously tag it to the feature.

4. Continuously build and update information to suit CGTS requirements.
Figure 5.1 Alternate way of CAD data conversion to CGTS
REFERENCES


Appendix A - Source Code Listing
// A program for feature recognition

// Developed by Rajendar V Auradkar and Dr. Osama K. Eyada
// Master's Thesis, November 1992
// Virginia Polytechnic Institute and State University
// Main program file, file name - FEREC.CPP

#include "topocls.cpp"  /* system application elements */
#include "topofil.h"    /* output topofile */
#include "dxf.h"        /* input dxf file */
#include <stdlib.h>
#include <fstream.h>
#include <io.h>
#include <conio.h>
#include <string.h>
#include <dos.h>
#include <alloc.h>
#include "window.cpp"   /* windowing class */
#include "menu.h"       /* menuing class */

void initialize(void);
main(int argc, char *argv[])
{
    clrscr();
    // windows for user information
    Window w1(Point(20,5), Point(65,19));
    Window w2(Point(20,21), Point(65,23));
    highlightOn();
    textattr(WHITE | BLUE*16);
    w1.open();
    w1 << ("\n\r");
    w1 << (" WELCOME TO THE FEATURE RECOGNISER\n\r");
    w1 << ("\n\r");
    w1 << (" DEVELOPED BY \n\r");
    w1 << ("\n\r");
    w1 << (" DR. OSAMA EYADA AND \n\r");
    w1 << (" RAJENDRA V AURADKAR \n\r");
    w1 << ("\n\r");
    highlightOff();
    w2.open();
    textattr(WHITE | BLINK | RED*16);
    w2 << ("Press any key to continue......\n\r");
    getch();

    //-----build menu
    Menu m1(Point(66,5));
textattr(RED | LIGHTBLUE);
m1.add("Get slot");
m1.add("Get boss");
m1.add("Get hole");
m1.add("Get all");
m1.add("Start");
int done = 0;
while(!done)
{
    switch(m1.getChoice())
    {
    case 0 : highlightOn();
        w1 << ("Get Slots Selected---\n\r");
        highlightOff();
        break;
    case 1 : highlightOn();
        w1 << ("Get Boss Selected---\n\r");
        highlightOff();
        break;
    case 2 : highlightOn();
        w1 << ("Get hole ---\n\r");
        highlightOff();
        break;
    case 3 : highlightOn();
        w1 << ("Get All---\n\r");
        highlightOff();
        break;
    case -1:
        case 4 : done = 1; break;
    }
}
fullScreen();
gotoxy(1,24);

vertex *objvertex;
edge *objedge;
face *objface;
FILE *fp;
int xrange[2], yrange[2], zrange[2];
int vertexcount, edgecount, facecount;

char topnam[80], dxfname[80];
cout << "Please enter the name of the input model data file\n";
cin >> dxfname;
int a;
a = 0;
if(dxfname[0] == '3') a = 1;
// char *dxfname = argv[1];
cout << "Please enter the name of the CGTS file\n";
cin >> topnam;

c1rscr();
initialize();
cout << "Reading geometric data from the DXF file......\n";
//create instance of input dxf class
dx dxffile;
// initialize dxf class
dxfile.assign(dxfname);
// get aggregate data from the dxf file
dxfile.retag horr(data(&vertexcount, &edgecount, &facecount);
// allocate memory for face elements
objface = new face[edgecount],
// allocate memory for vertex elements
objvertex = new vertex[edgecount];
// allocate memory for edge elements
objedge = new edge[edgecount];
// check for memory errors
if(objedge || !objvertex || !objface)
{
    cout << "allocation failed";
    exit(0);
}

FILE *fp1;

if((fp1 = fopen(topnam,"w")) == NULL)
{
    printf("Cannot open file\n");
    exit(1);
}

cout << "Building topology......\n";
// get dxf data and process it
dxfile.getallgeom(objvertex, objedge, objface, &vertexcount,
&edgecount, &facecount, xrange, yrange, zrange);
// get aggregate data
dxfile.retaggdata(&vertexcount, &edgecount, &facecount);

setcolor(1);
settextstyle(GOTHIC_FONT, HORIZ_DIR, 4);

cout << "\\nDetecting loops......\\n";
// get information on loops
for(int i = 0; i < facecount-1; i++)
{
    for(int j = i+1; j < facecount; j++)
    {
        objface[i].detectloops(objface,j);
    }
}

cout << "\\nDetermining adjacent faces......\\n";
for(i = 0; i < facecount; i++)
{
    objface[i].adjacency(objface, facecount);
}

cout << "\\nCalculating adjacency relationships......\\n";
cout << "\\nCreating CGTS......\\n";
cout << "\\nDetecting features......\\n";
cout << "\\nPlease any key to display model......\\n";
getch();
c1rscr();
for(i = 0; i < facecount; i++)
{
    objface[i].calcadjrel(objface);

    objface[i].determinefeature(xrange,yrange,zrange,a);
}
// create instance of output toprofile
topofile topo;
// initialize output toprofile class
topo.assign(fp1);
// write the header info.
topo.wrthdrinfo(xrange, yrange, zrange, vertexcount, 
    edgecount, facecount);
// write vertex info.
topo.wrtvertexinfo(objvertex, vertexcount);
// write edge info.
topo.wrtedgeinfo(objedge, edgecount);
// write face info.
topo.wrtfaceinfo(objface, facecount);

// cout << coreleft() << "\n";

int ids[20],ids1[20], num = 0;
// determine feature details
for(i = 0; i < facecount; i++)
{
    if(objface[i].getfeatureflag() == 1)
    {
        objface[i].retadjfaceids(ids1);
        objface[i].retedgeids(ids);
        num = objface[i].retinvers();
        for(int i = 0; i < num; i++)
        {
            objedge[ids[i]].flagasfeature();
        }
        for(i = 0; i < objface[i].retnumdfaces(); i++)
        {
            objface[ids[i][i]].retedgeids(ids);
            num = objface[ids1[i][i]].retinvers();
            for(i = 0; i < num; i++)
            {
                objedge[ids[i]].flagasfeature();
            }
        }
    }
}
getch();

int color = 0;
setlinestyle(SOLID_LINE,0,THICK_WIDTH);
for(i = 0; i < edgecount; i++)
{
    color = 1;
    objedge[i].draw(color);
}
getch();
// back to text mode
restorecrtmode();

return(0);
}

// initialize graphics routines
void initialize(void)
{
    int GraphDriver;
    int GraphMode;
    int MaxX, MaxY;
    int MaxColors;
    int ErrorCode;
    int xasp, yasp;
    double AspectRatio;
    struct palettetype palette;

    GraphDriver = DETECT;
    initgraph( &GraphDriver, &GraphMode, "c:\borlandc\bgi" );
    ErrorCode = graphresult();
    if( ErrorCode != grOk )
    {
        printf(" Graphics System Error: %s\n", grapherrormsg( ErrorCode ) );
        exit( 1 );
    }
    getpalette( &palette );
    MaxColors = getmaxcolor() + 1;
    getaspectratio( &xasp, &yasp );
}

// Name of source file - dxf.h
// Purpose - for input dxf class
// should be included in main during compilation

#include <stdlib.h>
#include <fstream.h>
#include <io.h>
#include <string.h>
#include <dos.h>
#include <alloc.h>

#include "def.h"

// Function which calculates the ids for edges when input is polylines
void edgeidcalc(face *, edge *, vertex *, int, int *, int);

// Function which calculates ids for vertices when input is lines
void vertexidcalc(vertex *, edge *, int, int **);

// Function which calculates the edges that a vertex generates
// and the paths that an edge can take
void vertexedgeidcalc(vertex *, int, edge *, int);

// Function that detects the faces that form the object when input is lines
void detectfaces(vertex *, int, edge *, int, face *, int **);

// Function that calculates the equation to a plane to detect faces when
// input is in the form of lines
int planeequation(vertex *, int *, int *, int *, int *);

// Function to get all the points that satisfy the equation to the
// plane calculated in the planeequation function
void getotherpoints(int, vertex *, int, vertex *, int *);

// Function to calculate the edges that form a plane
int* getplanedges(int, vertex *, edge *, int, int *);
// calculate ordered vertices for face
void vertexcalc(int, vertex *, edge *, face *);

class dxf
{
// for input file name
char dxfname[80];
// for edge count
int edgcount;
// for face count
int facenumber;
// for vertex count
int vertexcount;
public:
// returns aggregate geometric information
void retaggdata(int *a, int *b, int *c);
// assign name to dxf file
void assign(char *name);
// get the edges from the dxf file
edge* getedges(int *ecount, int *xrange, int *yrange, int *zrange);
// get all geometry after the topology is built, when input is polylines
void getallgeom(face *objface, edge *objedge, int *fcount,int *ecount,
int *xrange, int *yrange, int *zrange);
// get all geometry after topology is built, when input is edges
void getallgeom(vertex *objvertex, edge *objedge, face *objface, int
*vercount, int *ecount, int *facecount, int *xrange,
int *yrange, int *zrange);
// display dxf information for debugging purposes
void show();
}

// client objects requesting dxf information
void dxf::retagdata(int *a,int *b,int *c)
{
    *a = vertexcount;
    *b = edgecount;
    *c = facecount;
}

// for debugging purposes
void dxf::show()
{
    cout << edgecount << "," << facecount << " ," << vertexcount;
}

void dxf::assign(char *name)
{
    strcpy(dxfname,name);

    char str[80];

    facecount = edgecount = 0;
// open input dxf file
    ifstream in(dxfname);

    if(!in) /* check for file errors */
    {
        cout << "cannot open file";
    }
exit(1);
}

// loop till end of file
while(strncmp(str,"EOF")) /* till end of file */
{
    in >> str;
    if(!strncmp(str,"POLYLINE")) /* if polyline is detected */
    {
        facecount += 1;
    }

    if(!strncmp(str,"LINE")) /* if line is detected */
    {
        edgecount += 1;
    }
}

in.close();

// get geometry infomation after topology is built, if input is edges
void dxf::getallgeom(vertex *objvertex, edge *objedge, face *objface,
    int *vercount, int *ecount, int *fcount, int *xrange,
    int *yrange, int *zrange)
{
    char str[80], str1[80];

    vertex tmp[2];

    *vercount = *ecount = *fcount = 0;

    xrange[0] = yrange[0] = zrange[0] = 100;


    int min = 0, max = 1;
    // open input dxf file
    ifstream in1(dxfname);
if(!in1)    /* check for file errors */
{
    cout << "cannot open file";
    exit(1);
}

int i = 0,x = 0,y = 0,z = 0;
float temp = 0;

while(strcmp(str1,"EOF") NE 0)    /* loop till end of file */
{
    in1 >> str1;
    // if line entity
    if(!strcmp(str1,"LINE"))
    {
        do
        {
            in1 >> temp;
            i = int(temp);
            switch(i)
            {
            case 10 :    /* start x */
                in1 >> x;
                if(x LT xrange[min]) xrange[min] = x;
                if(x GT xrange[max]) xrange[max] = x;
                break;
            case 20 :    /* start y */
                in1 >> y;
                if(y LT yrange[min]) yrange[min] = y;
                if(y GT yrange[max]) yrange[max] = y;
                break;
            case 30 :    /* start z */
                in1 >> z;
                if(z LT zrange[min]) zrange[min] = z;
                if(z GT zrange[max]) zrange[max] = z;
                vercount += 1;    /* increment vertices */
                tmp[0] = vertex(x,y,z);
                break;
            case 11 :
                in1 >> x;    /* end x */
if(x LT xrange[min]) xrange[min] = x;
if(x GT xrange[max]) xrange[max] = x;
break;
case 31:
    inl >> y;  /* end y */
    if(y LT yrange[min]) yrange[min] = y;
    if(y GT yrange[max]) yrange[max] = y;
    break;
case 31:
    /* end z */
    inl >> z;
    if(z LT zrange[min]) zrange[min] = z;
    if(z GT zrange[max]) zrange[max] = z;
    vrcount += 1;
    tmp[1] = vertex(x,y,z);
    *ecount += 1;
    // create an edge object
    objedge[*ecount-1] = edge(tmp,*ecount);
    break;
    default:
    break;
}
}while(i NE 31);
}

int *vrc = 0;
// calculate ids for vertices
vertexidcalc(objvertex, objedge, *ecount, &vrc);

vrcount = *vrc;

*verc = *vrc;

ecount = *ecount;
// calculate ids for edges, and determine path traversals
vertexedgeidcalc(objvertex, vrcount, objedge, edgenetcount);
// detect faces of object
detectfaces(objvertex, vrcount, objedge, edgenetcount,
ojface, &fcount);
facecount = *fcount;

// function to detect faces
void detectfaces(vertex *objvertex, int vertexcount, edge *objedge, int edgecount, face *objface, int **facecount)
{
    int count = 0, flag = 0, numpoints = 0, nextedge = 0, count1 = 0;
    int a = 0, b = 0, c = 0, d = 0, tt = 0;
    **facecount = 0;
    vertex plane[20], temp[2], temp1[2], tver[25];
    int it1[25];
    face ftemp;
    // loop till number of edges
    for(int i = 0; i < edgecount; i++)
    {
        count1 = 0;
        // check if edges have been traversed LT 2
        if( (count1 = objedge[i].gettraversecount()) LT 2)
            do
            {
            // increment counter for edge traversals
                ++count1;
                objedge[i].retvert(temp);
                // get next edge, whose path is still not taken
                nextedge = objedge[i].getpathnottaken();
                // check if the edge has been traversed twice, if yes get next edge
                if( (count = objedge[nextedge-1].gettraversecount()) GE 2)
                    nextedge = objedge[i].getpathnottaken();
                // if LT 2
                if( (count = objedge[nextedge-1].gettraversecount()) LT 2)
                {
                    count = 0;
                    // get vertices of nextedge
                    objedge[nextedge-1].retvert(temp1);
                    plane[++count-1] = temp[0];
                    plane[++count-1] = temp[1];
                    for(int h = 0; h < 2; h++)
                    {
                        if( !(plane[0] EQ temp1[h]) AND !(plane[1] EQ temp1[h]) )
                        {
                            plane[++count-1] = temp1[h];
                        }
                    }
                    // calculate plane equation for the two edges
                    flag = planeequation(plane, &a, &b, &c, &d);
                    // get other points that satisfy the equation, to complete face
if(flag) getterpoints(flag, objvertex, vertexcount, plane, &numpoints);
    int *edgeids = new int[numpoints];
    // calculate the edges that form the face
    edgeids = getplanedges(numpoints, plane, objedge, edgecount, &tt);
    // get the edge list for face
    for(int ii = 0; ii LT tt; ii++)
    {
        int yt = 0;
        for(int jj = numinfo[ii].start-1; jj LT numinfo[ii]
            .end; jj++)
        {
            it1[yt] = edgeids[jj];
            yt += 1;
        }
        int ite = numinfo[ii].end - numinfo[ii].start + 1;
        ftemp = face(plane,ite,1);
        ftemp = face(it1);
        int f = 0;
        // check if face is already been constructed
        for(h = 0; h LT **facecount; h++)
        {
            if(objface[h] EQ ftemp)
                f = 1;
        }
        // if not then construct face
        if(!f)
        {
            int id = objedge[i].retidno();
            for(h = 0; h LT ite; h++)
            {
                objedge[it1[h-1]].recordtraversals(id);
            }
            // construct face by calling constructor 1
            objface[++**facecount-1] = face(plane,ite,
                **facecount);
            // construct face by calling constructor 2
            objface[**facecount-1] = face(it1);
        }
    }
} while(count1 LT 2); /* end of while loop when edge traversals are 2
cout << "\n";
// calculate ordered list of vertices for face
vertexcalc(**facecount, objvertex, objedge, objface);

int* getplanedges(int numpoints, vertex *plane,
edge *objedge, int edgecount, int *tt1)
{
    int *facedges, *fed, *pointer, count = 0,
    flag = 0, flag1 = 0, temp = 0, fc = 0;

    vertex tmp[2], *plane1;
    edge etemp;

    // allocate space for a new face
    facedges = new int[numpoints];
    pointer = new int[numpoints];
    fed = new int[numpoints];
    plane1 = new vertex[numpoints];
    // chack if allocation succeeded
    if(!facedges OR !fed OR !plane1 OR !pointer)
    {
        cout << "Allocation failed";
        exit(1);
    }

    // initialize values
    for(int i = 0; i LT numpoints; i++)
    {
        plane1[i] = plane[i];
        pointer[i] = 0;
    }

    // loop till number of points of face, to get edge ids
    for(i = 0; i LT numpoints; i++)
    {
        for(int k = i+1; k LT numpoints; k++)
        {
            tmp[0] = plane[i];
            tmp[1] = plane[k];
            etemp = edge(tmp, 1);
            for(int j = 0; j LT edgecount; j++)
            {
                // if new edge found, record id in data structure
            }
        }
    }
}
if(etemp EQ objedge[j])
{
    facedges[++count-1] = objedge[j].retidno();
}
}
}
}
int nump = numpoints;
count = -1;
int cnt = -1;
int ftemp = 0;
int itt = 0;
// loop till list is ordered
do
{
    ++count;
    int ind = count;
    ++cnt;
    fed[count] = facedges[cnt];
    pointer[cnt] = 1;
    do
    {
        for(int i = 0; i LT numpoints; i++)
        {
            if( fed[count] NE facedges[i] AND !pointer[i])
            {
                if(objedge[fed[count]-1] LE objedge[facedges[i]-1])
                {
                    pointer[i] = 1;
                    if(count GT 0 AND objedge[fed[ind]-1]
                        LE objedge[facedges[i]-1]) flag = 1;
                    fed[+count] = facedges[i];
                }
            }
        }
    } while(!flag);
    ftemp++;
    ftemp++;
    if(count +1 EQ nump) flag1 = 1;
    temp = 0;
    for(int p = 0; p LT numpoints; p++)
    {
        if(!pointer[p])
        {
            }
facedges[temp++] = facedges[p];
}
}
for(p = 0; p LT numpoints; p++) pointer[p] = 0;
numpoints = temp;
numinfo[fc].start = itt + 1;
itt = count + 1;
uminfo[fc].end = itt;
++fc;
} while(!flag1);
*tt1 = ftemp;
int id = 0;
vertex tt[2];
objedge[fed[0]-1].retvert(tt);
plane[++id-1] = tt[0];
plane[++id-1] = tt[1];
int ff = 0;
for(i = 1; i LT nump; i++)
{
    objedge[fed[i]-1].retvert(tt);
    for(int n = 0; n LT 2; n++)
    {
        for(int j = 0; j LT id; j++)
        {
            if(plane[j] EQ tt[n]) ff = 1;
        }
        if(!ff)
        {
            plane[++id] = tt[n];
        }
    }
}
// release memory
delete [nump] plane1;
delete [nump] facedges;
return fed;
}

// get other points of the plane
void getotherpoints(int flag, vertex *objvertex, int vertexcount,
                     vertex *plane, int *numpoints)
* numpoints = 3;
int ver1 = plane[0].retidno();
int ver2 = plane[1].retidno();
int ver3 = plane[2].retidno();
int refval = plane[0].getref(flag);
for(int i = 0; i LT vertexcount; i++)
{
    int j = objvertex[i].retidno();
    if(j NE ver1 AND j NE ver2 AND j NE ver3)
    {
        if( refval EQ objvertex[i].getref(flag) )
        {
            plane[++]numpoints-1] = objvertex[i];
        }
    }
}

// calculate plane equation
int planeequation(vertex *plane, int *a, int *b, int *c, int *d)
{
    int flag = 0;
    int x1 = plane[0].retxval(), y1 = plane[0].retyval(), z1 = plane[0].retzval();
    int x2 = plane[1].retxval(), y2 = plane[1].retyval(), z2 = plane[1].retzval();
    int x3 = plane[2].retxval(), y3 = plane[2].retyval(), z3 = plane[2].retzval();

    *a = *b = *c = *d = 0;

    if(x1 EQ x2 AND x2 EQ x3) return flag = 1;
    if(y1 EQ y2 AND y2 EQ y3) return flag = 2;
    if(z1 EQ z2 AND z2 EQ z3) return flag = 3;
    return flag = 0;
}

// calculate ids for edges and vertices
void vertexedgeidcalc(vertex *objvertex, int vertexcount, edge *objedge,
                        int edgecount)
{
    int tmp[3], count = 0, ted = 0;

    vertex temp[2];
// loop till end of vertices
for(int i = 0; i < vertexcount; i++)
{
    // loop till end of edges
    for(int j = 0; j < edgecount; j++)
    {
        // get vertices od edges
        objedge[j].retvert(temp);
        // resolve duplicate occurrences of vertices
        if(temp[0] == objvertex[i] OR temp[1] == objvertex[i])
        {
            ted = objedge[j].retidno();
            tmp[count++] = ted;
        }
    }
    // assign id for vertex
    objvertex[i].assigngid(tmp);
    count = 0;
}

int ids[3];
// get the traversal paths for vertices and edges
for(i = 0; i < edgecount; i++)
{
    int flag = 0;
    // get end id foe degs
    objedge[i].getendid(&ted);
    // get flag for vertex, for traversal
    flag = objvertex[ted-1].retflag();
    // if not assigned
    if(!flag)
    {
        // assign flag
        objvertex[ted-1].setflag();
        // get ids of edges radiating from vertex
        objvertex[ted-1].getedgeids(ids);
        // assign paths to edges
        objedge[i].assignpaths(ids);
    }
// if assigned
    if(flag)
    {
    // get the other end
        objedge[i].getotherendid(&ted);
    // get radiating edges
        objvertex[ted-1].getedgeids(ids);
    // assign traversal paths
        objedge[i].assignpaths(ids);
    }
}

// calculate ids for vertices when input is vertices

void vertexidcalc(vertex *objvertex, edge *objedge, int nedges, int **verc)
{
    int flag = 0;
    vertex tmp[2];
    **verc = 0;
    // loop till end of edges
        for(int i = 0; i LT nedges; i++)
        {
        // if first edge
            if(i)
            {
            // return vertices
                objedge[i].retvert(tmp);
                objvertex[+++verc-1] = tmp[0];
            // assign ids
                objvertex[**verc-1].assignid(**verc);
            // assign vertex ids for edges
                objedge[i].assignvertid(**verc);
                objvertex[+++verc-1] = tmp[1];
            // assign ids
                objvertex[**verc-1].assignid(**verc);
            // assign vertex ids for edges
                objedge[i].assignvertid(**verc);
            }
            // if not the first vertex
            if(i GT 0)
{ 
  // get edge verteces
  objedge[i].retvert(tmp);
  int temp = **verc;
  for(int l = 0; l LT 2; l++)
  {
    flag = 0;
    for(int k = 0; k LT temp; k++)
    {
      // if vertex detected before
      if(objvertex[k] EQ tmp[l])
      {
        flag = 1;
        int id = objvertex[k] retidno();
        objedge[i].assignvertid(id);
      }
    }
    if(!flag)
    {
      objvertex[++**verc-1] = tmp[l];
      objvertex[**verc-1].assignid(**verc);
      objedge[i].assignvertid(**verc);
    }
  }
  // If nedges GT 0
} // Loop for number of edges

// get all the geometry when the input is polylines
void dxf::getallgeom(face *objface, edge *objedge, int *fcount, 
  int *ecount, int *xrange, int *yrange, int *zrange)
{
  vertex *objvertex;

  char str[80], str1[80];

  int vercount = 0, nedges = 0;

  *fcount = facecount;
  // allocate memory for vertices
  objvertex = new vertex[25];
  // check for memory failures
  if(!objvertex)
{  
cout << "Allocation failed 1 ";  
exit(1);  
}

xrange[0] = yrange[0] = zrange[0] = 100;  
int min = 0, max = 1;  
// open input dxf stream  
ifstream in1(dxfname);  
if(!in1)  
{  
cout << "cannot open file";  
exit(1);  
}

int i = 0,x = 0,y = 0,z = 0, fc = 0;  
float temp = 0;  
// loop till end of file  
while(strcmp(str, "EOF")!=0)  
{  
in1 >> str1;  
// if polyline detected  
if(!strcmp(str1,"POLYLINE"))  
{  
while(strcmp(str, "SEQEND")!=0)  
{  
in1 >> str;  

if(!strcmp(str,"ENDSEC")) break;  

if(!strcmp(str,"SEQEND"))  
{  
in1 >> str;  

fc += 1;  

objface[fc-1]=face(objvertex,vercount,fc);  
}
edgeidcalc(objface,objedge,objvertex,vercount,&nedges,fc);

  vercount=0;
}
// if vertex detected
  if(!strcmp(str,"VERTEX"))
  {
    do
      {
        in1 >>= temp;
        i = int(temp);
        switch(i)
          {
            case 10:    /* for x values */
              in1 >>= x;
              if(x LT xrange[min]) xrange[min] = x;
              if(x GT xrange[max]) xrange[max] = x;
              break;
            case 20:    /* for y values */
              in1 >>= y;
              if(y LT yrange[min]) yrange[min] = y;
              if(y GT yrange[max]) yrange[max] = y;
              break;
            case 30:    /* for z values */
              in1 >>= z;
              if(z LT zrange[min]) zrange[min] = z;
              if(z GT zrange[max]) zrange[max] = z;
              vercount += 1;    /* increment vertices */
              objvertex[vercount-1] = vertex(x,y,z);
              break;
            default:
              break;
          }
    }while(i NE 30);
  }
}

*ecount = nedges;
/* The function edgeidcalc() calculates the 3d lines that form each face of the part under consideration. This information is not given by the DXF file. The only information available is the vertex co-ordinates of each face. Edge information is extracted from each of these faces and an unique identification is given to each of these lines. This information is essential for the building of topological information of the part. Adjacencies are calculated from this information. */

void edgeidcalc(face *objface, edge *objedge, vertex *objvertex,
             int ntd, int *nedges, int fc)
{
    vertex *tmp;
    int res = *nedges, flag = 0;
    int *idnos;
    tmp = new vertex[2];
    idnos = new int[ntd];

    if(!tmp OR !idnos)
    {
        cout "Allocation failed 2"
        exit(1);
    }

    if(*nedges EQ 0)
    {
        *nedges = ntd;
        for(int i = 0; i LT ntd; i++)
        {
            tmp[0] = objvertex[i];
            tmp[1] = objvertex[i+1];
            if(i EQ ntd-1)
            {
                
            
        
    
94
tmp[0] = objvertex[i];
tmp[1] = objvertex[0];

objedge[i] = edge(tmp, i+1);
idnos[i] = i+1;

objface[fc-1] = face(idnos);
return;

for(int i = 0; i LT ntd; i++)
{
    tmp[0] = objvertex[i];
tmp[1] = objvertex[i+1];
    if(i EQ ntd-1)
    {
        tmp[0] = objvertex[0];
tmp[1] = objvertex[i];
    }

    flag = 0;
    for(int k = 0; k LT *nedges; k++)
    {
        if(edge(tmp,1) EQ objedge[k])
        {
            idnos[i] = objedge[k].retidno();
            flag = 1;
        }
    }

    if(!flag)
    {
        res += 1;
        idnos[i] = res;
    }
objedge[res-1] = edge(tmp,res);
}
}

*nedges = res;
objface[fc-1] = face(idnos);
return;
}

// calculate vertices of faces
void vertexcalc(int facenct, vertex *objvertex, edge * objedge,
face *objface)
{
    int edges[20], vertid[2], numed;
    vertex vert[20];
    // loop till face count
    for(int i = 0; i LT facenct; i++)
    {
        // return number of edge for each face
        numed = objface[i].retnoedges();
        // return the edge ids for the face
        objface[i].retedgeids(edges);
        // return the vertex ids for the face
        objedge[edges[0]-1].getvertid(vertid);
        vert[0] = objvertex[vertid[0]-1];
        vert[1] = objvertex[vertid[1]-1];
        int count = 2;
        // loop till number of edges
        for(int j = 1; j LT numed; j++)
        {
            // get the vertex id for each edge
            objedge[edges[j]-1].getvertid(vertid);
            vertex tt1 = objvertex[vertid[0] - 1];
            vertex tt2 = objvertex[vertid[1] - 1];
            if(tt1 EQ vert[count-1] OR tt1 EQ vert[count-2])
            vert[count] = tt2;
            if(tt2 EQ vert[count-1] OR tt2 EQ vert[count-2])
            vert[count] = tt1;
            count++;
        }
        // assign the vertex ids for the face
        objface[i].changevert(vert);
// Name of source file - topocl.cpp
// Should be included in main
// Declares and defines the classes for implementation - vertex, edge and face

#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <graphics.h>
#define SC 0.707
#define SCALE 15

/* Following is the class declaration for three dimensional 
points. The function declaration and definitions are 
present in the same file. There are two initialiser 
functions and two functions for display of the 3-d 
points. It is advised to use the overloaded friend 
insertor function for displaying for convenience */
class vertex
{
   // coordinate values
   int x, y, z;
   // id
   int idno;
   // edges generated by vertex
   int edgenumber[3];
   // if vertex flagged, for internal purposes
   int flag;

   public:
   // constructor
   vertex() { x = 0; y = 0; z = 0; flag = 0; };

   // ~vertex();
   // constructor
   vertex(int a, int b, int c);
   // assign id
   void assignid(int id);
   // assignment overload
   vertex operator=(vertex vt);
   // equality overload
   int operator==(vertex vt);
   // get the x, y and z value of vertex
   int retzval() { return z; };
int retryval() { return y; };

int retxval() { return x; };

// get flag value
int retflag() { return flag; };

// set the flag
void setflag() { flag = 1; }

int getref(int flag);
// debugging information
void show(void);

friend ostream &operator<<(ostream &stream, vertex vt);
// print vertex info. to file
void fileprint(FILE *fp);
// get vertex info.
void retvertex(vertex vt);
// get idno
int retidno() { return idno; };
// assign edge radiating info.
void assignedgid(int *id1);
// get edge radiating info.
void getedgeids(int *id2);

int vertex::getref(int flag)
{
    if(flag == 1) return x;
    if(flag == 2) return y;
    if(flag == 3) return z;
}

// get the edges generated information
void vertex::getedgeids(int *id2)
{
    for(int i = 0; i < 3; i++)
    {
        id2[i] = edgegenerated[i];
    }


// assign the edge information
void vertex::assignedgid(int *id1)
{
    for(int i = 0; i < 3; i++)
    {
        edgenerated[i] = id1[i];
    }
}

ostream &operator<<(ostream &stream, vertex vt)
{
    stream << vt.x << " ";
    stream << vt.y << " ";
    stream << vt.z << "\n";
    stream << vt.idno << "\n";
    return stream;
}

// constructor
vertex::vertex(int a, int b, int c)
{
    //cout << "constructing\n";
    x = a;
    y = b;
    z = c;
}

// assign id to vertex
void vertex::assignid(int a)
{
    //cout << "constructing\n";
    idno = a;
}

// for debugging purposes
void vertex::show(void)
{
    cout << x << ", ";
}
cout << y << ",";
cout << z << "\n";
}

// assignment overload
vertex vertex::operator=(vertex vt)
{
    x = vt.x;
y = vt.y;
z = vt.z;
idno = vt.idno;
edgenerated[0] = vt.edgenerated[0];
edgenerated[1] = vt.edgenerated[1];
return *this;
}

// equality overload
int vertex::operator==(vertex vt)
{
    if( x == vt.x && y == vt.y && z == vt.z)
        return 1;
    return 0;
}

// print to file
void vertex::fileprint(FILE *fp)
{
    fprintf(fp, "%-7s %d\n", "V", idno);
    fprintf(fp, "%-10d %-10d %-10d\n", x, y, z);
    fprintf(fp, "EDGEN\n");
    fprintf(fp, "%-6d%-6d%-6d", edgenerated[0], edgenerated[1], edgenerated[2]);
    fprintf(fp, "\n");
}

// class declarataion for edges
class edge
{
    // id no
    int idno;
    // vertices of edge
    vertex *arr1;
// id of vertices
    int vertexid[2];
// adjacent edges to this edge
    int adjedges[2];
// count for traversals
    int traversecount;
// flags to flag paths
    int pathflags[2];
// if edge forms feature
    int featureflag;

    int temp, temp1;

public:
// constructor
    edge( vertex *vt, int n);

// ~edge() { };
// constructor
    edge() { idno = 0; temp = 0; traversecount = 0; pathflags[0] = 0;
         pathflags[1] = 0; temp1 = 0; featureflag = 0;};
// return id
    int retidno(void) { return idno; }
// return vertices
    void retvert(vertex *ver);
// debugging purposes
    void show(void);
// debugging purposes
    friend ofstream &operator<<(ostream &stream, edge ed);
// print to file
    void fileprint(FILE *fp);
// overload equality operator
    int operator==(edge ed);
// overload <= operator
    int operator<=(edge ed);
// assign vertex ids of edge
    void assignvertid(int id);
// assign paths for edge
    void assignpaths(int *id1);
// get the end id of edge
    void getendid(int *id2) { *id2 = vertexid[1]; }
// get the other end of edge
    void getotherendid(int *id2) { *id2 = vertexid[0]; };

102
// record traversals od edges
void recordtraversals(int idn);
// return the traverse count of edge
int gettraversecount() { return traversecount; }
// get the path not taken by the edge
int getpathnottaken();
// get the adjacent edges
void getadjedges(int *tt);
// switch vertices to order
void switchver();
// get the vertex ids
void getvertid(int *tie);
// draw edges in graphics mode
void draw(int color);
// flag edge as belonging to feature
void flagasfeature() { featureflag = 1; }
// get if edge belongs to feature
int retflagfeature() { return featureflag; }

);

// draw edge by application of transformations to draw in 3-D
void edge::draw(int color)
{
    setcolor(color);

    int x = arr1[0].retxval()*SCALE;
    int y = arr1[0].retyval()*SCALE;
    int z = arr1[0].retzval()*SCALE;
    int x1 = arr1[1].retxval()*SCALE;
    int y1 = arr1[1].retyval()*SCALE;
    int z1 = arr1[1].retzval()*SCALE;

    int x2 = (x-y)*SC + 200;
    int y2 = (x+y)*SC + 200;
    int x3 = (x1-y1)*SC + 200;
    int y3 = (x1+y1)*SC + 200;

    line( x2 , y2 - z , x3 , y3 - z1 );
}
// return the vertex ids of edge
void edge::getvertid(int *ite)
{
    ite[0] = vertexid[0];

    ite[1] = vertexid[1];
}

void edge::switchver()
{
    vertex temp = arr1[0],
    arr1[0] = arr1[1],
    arr1[1] = temp;
    int temp1 = vertexid[0],
    vertexid[0] = vertexid[1],
    vertexid[1] = temp1;
}

// return adjacent edges for edge
void edge::getadjedges(int *tt)
{
    tt[0] = adjedges[0],
    tt[1] = adjedges[1],
}

// record traversals for the edge
void edge::recordtraversals(int idn)
{
    ++traversecount;
    if(idn == idno) return;
    if(idn == adjedges[0]) return;
    if(idn == adjedges[1])
    {
        int temp = adjedges[0],
        adjedges[0] = adjedges[1],
        adjedges[1] = temp;
        return;
    }
}

// get the path not taken by the edge end point
int edge::getpathnottaken()
{ if(temp1 == 1) return adjedges[1];
  ++temp1;
  return adjedges[0];
  for(int i = 0; i < 2; i++)
  {
    if(pathflags[i] != 1)
    {
      return adjedges[i];
    }
  }
}

// assign paths for edges ends
void edge::assignpaths(int *id1)
{
  int count = 0;
  for(int i = 0; i < 3; i++)
  {
    if(id1[i] != idno)
    {
      adjedges[count++] = id1[i];
    }
  }
}

// assign vertex ids for edge vertices
void edge::assignvertid(int id)
{
  vertexid[temp] = id;
  arr1[temp].assignid(id);
  ++temp;
}

// return vertex information
void edge::retvert(vertex *ver)
{
  for(int i = 0; i < 2; i++)
  {
    ver[i] = arr1[i];
  }
}
// constructor for edge, takes a list of vertices and an id no
edge::edge(vertex *vt, int n)
{
    arr1 = new vertex[2];
    arr1[0] = vt[0];
    arr1[1] = vt[1];
    idno = n;
}

// for debugging information
void edge::show(void)
{
    cout << idno << "\n";
    for(int i = 0; i < 2; i++)
    {
        arr1[i].show();
    }
}

// overloaded == operator function
int edge::operator==(edge ed1)
{
    if(arr1[0] == ed1.arr1[0] && arr1[1] == ed1.arr1[1])
        return 1;
    if(arr1[0] == ed1.arr1[1] && arr1[1] == ed1.arr1[0])
        return 1;
    return 0;
}

// overloaded <= operator function
int edge::operator<=(edge ed2)
{
    if(arr1[0] == ed2.arr1[0] || arr1[1] == ed2.arr1[1]
        || arr1[0] == ed2.arr1[1] || arr1[1] == ed2.arr1[0])
        return 1;
    return 0;
}

// for debugging purposes
ostream &operator<<(ostream &stream, edge ed)
{
}
stream << ed.idno << "\n";
stream << ed.arr1[0] << ed.arr1[1] << "\n";
return stream;
}

// function to print vertex information to file
void edge::fileprint(FILE *fp)
{
    fprintf(fp, "%7s %5d\n", "E", idno);
    fprintf(fp, "%20s \n", "VIDS");
    fprintf(fp, "%10d %10d\n", vertexid[0], vertexid[1]);
    fprintf(fp, "PATHS\n");
    fprintf(fp, "%10d %10d\n", adjedges[0], adjedges[1]);
}

/* Following is the class definition and declaration for the
 faces. The private section of the class consists of an integer
 variable num which is the number of 3-d points or vertices
 that makes up the face. A 3-d pointer is declared which
 is meant for an array of 3-d points that make up the face.
 There is one constructor function which takes as input the
 number of 3-d points and a pointer of type vertex which holds
 the address of the 3-d elements that make up the face */

class face
{
    int numvertex;    // No of vertices that form the face
    int idno;        // Internal identification given to the face
    vertex *arr;     // An array of vertices
    int *edgeidnos;  // The edge id nos that form the face
    int numadjfaces; // The number of adjacent faces
    int *adjfaceids; // The id nos of the adjacent faces

    107
int *adjrel;       // The relationships of this face with adj faces
int *commonedgeid; // The common edge between the adjacent faces
int loop;         // Flag to indicate if face loops in another
char feature[80]; // The name of the feature
int featureflag;

public:

/*! Constructor function for face. This function assigns the vertex values for the face, gives it an identification number and the number of vertices that form the face */

face( vertex *vt, int n, int m);

// ~face();

/*! This is also a constructor function. This function assigns numerical id nos for the edges that form the face */

face(int *idnos);

/*! This function returns the id no for a face */

int retidno(void) { return idno; };

/*! This function makes a copy of the edge ids that form the face */

void retedgeids(int *ids);

/*! This function calculates the faces that are adjacent to a face. The input to this is a pointer to the array of faces that form the object and the number of faces that form the object */

void adjacency(face *arrayoffaces, int facecount);

/*! This function makes a copy of the adjacent face ids */
void retadjfaceids(int *addids);

/* This function makes a copy of the vertices that form the face */
void retvert(vertex *ver);

/* This returns the number of edges that form the face */
int retnoedges(void) { return numvertex; };

/* This returns the number of vertices that form the face */
int retnumvertices(void) { return numvertex; };

/* This returns the number of adjacent faces */
int retnumadjfaces(void) { return numadjfaces; };

/* A constructor function that initialises numvertex and idno to zero */
face() { numvertex = 0; idno = 0; loop = 0; };

/* A function that outputs the face characteristics to screen */
friend ostream &operator<<(ostream &stream, face tdp);

/* This function writes the face information out to the Geometric and Topological File in the specified format */
void fileprint(FILE *fp);

/* This function returns the z-value range for the face */
void retzrange(int *zval);

/* This function returns the y-value range for the face */
void retyrange(int *yval);

/* This function returns the x-value range for the face */
void retxrange(int *xval);
/* This function calculates the adjacency relationships of a face with its adjacent faces. The input to this is a pointer to a list of faces that make up the face */

void calcadjrel(face *arrayoffaces);

/* This function includes all the rules to determine if a face is a candidate for a feature, if it is then the adjacent faces are checked, if they satisfy the conditions for which they are checked the face is given the name of the feature that has been recognised */

void determinefeature(int *xrange, int *yrange, int *zrange, int a);

int operator==(face fc);

void changevert(vertex *vert);

void flagasloop() { loop = 1; };

void detectloops(face *objface, int num);

void setflag() { loop = 0; };

int getfeatureflag() { return featureflag; };

};

// function to detect if face is loop face
void face::detectloops(face *objface, int num)
{
    // face *objface - pointer to face to be compared
    // id number of the face
    int xrange1[2], xrange2[2], yrange1[2], yrange2[2],
        zrange1[2], zrange2[2];
    int flag1 = 0, flag2 = 0, flag3 = 0, count = 0;
    int min = 0, max = 1;

    // return x, y, z ranges of face
    retxrange(xrange1);
    retyrange(yrange1);
    retzrange(zrange1);
// get ranges, low and high
int xmin1 = xrange1[min];
int xmax1 = xrange1[max];
int ymin1 = yrange1[min];
int ymax1 = yrange1[max];
int zmin1 = zrange1[min];
int zmax1 = zrange1[max];

// get x, y, and z ranges of face to be compared
objface[num].retxrange(xrange2);
objface[num].retyrange(yrange2);
objface[num].rezrange(zrange2);

// get ranges of face to be compared, low and high
int xmin2 = xrange2[min];
int xmax2 = xrange2[max];
int ymin2 = yrange2[min];
int ymax2 = yrange2[max];
int zmin2 = zrange2[min];
int zmax2 = zrange2[max];

if(xmin1 == xmin2 && xmax1 == xmax2) flag1 = 1;
if(ymin1 == ymin2 && ymax1 == ymax2) flag2 = 1;
if(zmin1 == zmin2 && zmax1 == zmax2) flag3 = 1;

if(flag1) ++count;
if(flag2) ++count;
if(flag3) ++count;

if(count != 1) return; /* face is not loop */

if(flag1)
{
  if(ymin1 < ymin2 && ymax1 > ymax2 && zmin1 < zmin2 &&
     zmax1 > zmax2)
  {
    objface[num].flagasloop(); /* loop, flag as such */
    return;
  }

  if(ymin1 > ymin2 && ymax1 < ymax2 && zmin1 < zmin2 &&
     zmax1 < zmax2)
  {
flagasloop(); /* loop, flag as such */
return;
}
}

if(flag2)
{
if(xmin1 < xmin2 && xmax1 > xmax2 && zmin1 < zmin2 &&
zmax1 > zmax2)
{
objface[num].flagasloop(); /* loop, flag as such */
return;
}
if(xmin1 > xmin2 && xmax1 < xmax2 && zmin1 > zmin2 &&
zmax1 < zmax2)
{
flagasloop(); /* loop, flag as such */
return;
}
}

if(flag3)
{
if(xmin1 < xmin2 && xmax1 > xmax2 && ymin1 < ymin2 &&
ymax1 > ymax2)
{
objface[num].flagasloop(); /* loop, flag as such */
return;
}
if(xmin1 > xmin2 && xmax1 < xmax2 && ymin1 > ymin2 &&
ymax1 < ymax2)
{
flagasloop(); /* loop, flag as such */
return;
}
}

// assign vertex changes for ordering
void face::changevert(vertex *vert)
{
for(int i = 0; i < numvertex; i++)
{

arr[i] = vert[i];
}
}

// overload equality operator
int face::operator==(face fc)
{
    int temp = 0;
    if( (temp = fc.retnovers()) != numvertex)
    {
        return 0;
    }
    vertex *ver;int *in;
    ver = new vertex[temp];
    in = new int[temp];
    if(!ver || !in)
    {
        cout << "Allocation failed 3\n";
        exit(1);
    }
    // get face information to check for equality
    fc.retnvert(ver);
    fc.retnedgeids(in);
    int count = 0;
    for(int i = 0; i < numvertex; i++)
    {
        for(int j = 0; j < temp; j++)
        {
            // if(ver[j] == arr[i])
            if(in[j] == edgeidnos[i])
            {
                ++count;
            }
        }
    }
    if(count == temp) /* faces are equal */
    {
        delete [temp] ver;
        delete [temp] in;
        return 1;
    }
    delete [temp] ver;
delete [temp] in;
return 0;
}

// function which determines the presence of features
void face::determinefeature(int *xrange, int *yrange, int *zrange, int a)
{
    int xval[2], yval[2], zval[2], count[6] = {0,0,0,0,0,0};
    retrangex(xval);
    retaryrange(yval);
    retzrange(zval);

    // condition for pocket
    if(xval[0] > xrange[0] && xval[1] < xrange[1] && yval[0] > yrange[0]
    {
        for(int i = 0; i < numadjfaces; i++)
        {
            if(adjrel[i] == 2) count[1] += 1;
        }

        if(count[1] == numadjfaces)
        {
            if(a == 0) cout << "Pocket discovered\n";
            featureflag = 1;
        }
        count[1] = 0;
    }

    // condition for protrusion
        yval[0] > yrange[0] && yval[1] < yrange[1])
    {
        for(int i = 0; i < numadjfaces; i++)
        {

if(adjrel[i] == 1) count[1] += 1;
}

if(count[1] == numadjfaces)
{
    if(a == 1) cout << "Protrusion discovered\n";
    featureflag = 1;
}
count[1] = 0;

// condition for blind slot
if(xval[0] == xrange[0] || xval[1] == xrange[1] ||
   yval[0] == yrange[0] || yval[1] == yrange[1])
{
    for(int i = 0; i < numadjfaces; i++)
    {
        if(adjrel[i] == 2) count[2] += 1;
        if(adjrel[i] == 4) count[4] += 1;
    }

    {
        cout << "Blind Slot discovered\n";
        featureflag = 1;
    }
    count[2] = 0;
    count[4] = 0;
}

// condition for step
if(xval[0] == xrange[0] || xval[1] == xrange[1] ||
   yval[0] == yrange[0] || yval[1] == yrange[1])
{
    for(int i = 0; i < numadjfaces; i++)
    {
        if(adjrel[i] == 1) count[1] += 1;
        if(adjrel[i] == 2) count[2] += 1;
        if(adjrel[i] == 4) count[4] += i;
    }
}
{
    cout << "Step discovered";
    featureflag = 1;
}
count[1] = 0;
count[2] = 0;
count[4] = 0;

// condition for through slot
if(xval[0] == xrange[0] || xval[1] == xrange[1] ||
    yval[0] == yrange[0] || yval[1] == yrange[1])
{
    for(int i = 0; i < numadjfaces; i++)
    {
        if(adjrel[i] == 2) count[2] += 1;
        if(adjrel[i] == 4) count[4] += 1;
    }
}

{
    cout << "Through Slot Discovered";
    featureflag = 1;
}
count[2] = 0;
count[4] = 0;

// calculate the adjacent relationships
void face::calcadjrel(face *arrayoffaces)
{
    int zhost[2], zguest[2], flag=0;

    // get z range of visited face
    retzrange(zhost),
for(int i = 0; i < numadjfaces; i++)
{
    // return the z range of each guest face
    arrayoffaces[(adjfaceids[i]-1)].ztrange(zguest);

    flag = 0;

    // give relationship value of 4
    if(zhost[0] > zguest[0] && zhost[1] < zguest[1])
    {
        adjrel[i] = 5;
        flag = 1;
    }

    // give relationship value of 4
    if(!flag)
    {
        if(zhost[0] < zguest[0] && zhost[1] > zguest[1])
        {
            adjrel[i] = 4;
            flag = 1;
        }
    }

    // give a relationship value of 3
    if(zhost[0] == zguest[0] && zhost[1] == zguest[1])
    {
        adjrel[i] = 3;
        flag = 1;
    }

    // give a relationship value of 1
    if(flag)
    {
        if(zhost[1] > zguest[1] || zhost[0] > zguest[0])
        {
            adjrel[i] = 1;

            flag = 1;
        }
    }
}
// give a relationship value of 2
if(flag)
{
    if(zhost[1] < zgues[1] || zhost[0] < zgues[0])
    {
        adjrel[i] = 2;
        flag = 1;
    }
}
}

// get the z range of face
void face::retzrange(int *zval)
{
    int z, ids[2];
    zval[0] = 100;
    zval[1] = -100;

    // for each vertex
    for(int i = 0; i < numvertex; i++)
    {
        // get z value
        z = arr[i].retzval();

        if( z <= zval[0]) zval[0] = z;

        if( z >= zval[1]) zval[1] = z;
    }
}

// get the y ranges of each face
void face::reyrange(int *yval)
{
    int y;
    yval[0] = 100;
    yval[1] = -100;

    // for number of vertices
    for(int i = 0; i < numvertex; i++)
    {
// get the y value
    y = arr[i].retYval();

    if( y <= yval[0]) yval[0] = y;
    if( y >= yval[1]) yval[1] = y;
}

// return x range of face
void face::retxrange(int *xval)
{
    int x;
    xval[0] = 100;
    xval[1] = -100;

    // for number of vertices
    for(int i = 0; i < numvertex; i++)
    {
        // get the x values for each vertex
        x = arr[i].retXval();

        if( x <= xval[0]) xval[0] = x,
        if( x >= xval[1]) xval[1] = x;
    }
}

/* This function builds the adjacency relationships between all the
different faces in the part. This is done by comparing the edges
of a face with the edges of all the remaining faces and if a
common edge is found between any of the faces these faces become
adjacent and the id no for the adjacent faces is recorded in the
face information for that face */

void face::adjacency(face *arrayoffaces, int facecount)
{
int *ids2, nids = 0;

ids2 = new int[20];

for(int i = 0; i < facecount; i++)
{
    if(i+1 != idno)
    {
        ids2 = new int[arrayoffaces[i].retnoedges()];
        arrayoffaces[i].retegeids(ids2);
        for(int k = 0; k < arrayoffaces[i].retnoedges(); k++)
        {
            for(int l = 0; l < numvertex; l++)
            {
                if(edgeidnos[l] == ids2[k])
                {
                    adjfaceids[nids] = arrayoffaces[i].retidno();
                    commonedgeid[nids] = ids2[k];
                    nids += 1;
                }
                // If the edge ids match
            }
            // Loop for comparision of edges of each face
        }
        // Loop for determining adjacencies
    }
    // Loop if the face ids do not match
}
// Loop ending for comparision of each face

numadjfaces = nids;
delete [20] ids2;
// return adjacent face information of face
void face::retadjfaceids(int *addids)
{
    for(int i = 0; i < numadjfaces; i++)
    {
        addids[i] = adjfaceids[i];
    }
}

// return vertex information for faces
void face::retvert(vertex *ver)
{
    for(int i = 0; i < numvertex; i++)
    {
        ver[i] = arr[i];
    }
}

// return the edge ids for face
void face::retedgeids(int *ids)
{
    for(int i = 0; i < numvertex; i++)
    {
        ids[i] = edgeidnos[i];
    }
}

// constructor for faces, assign vertex information
face::face(vertex *vt, int n, int m)
{
    // allocate memory
    arr = new vertex[n];

    edgeidnos = new int[n];

    adjfaceids = new int[n];

    adjrel = new int[n];

    commonedgeid = new int[n];
// check for memory failures
if(!arr || !edgeidnos || !adjfaceids || !adjrel || !commonedgeid)
{

    cout << "Allocation Failed 4\n",

    exit(1);

}

numvertex = n;

idno = m;

for(int i=0; i<n; i++)
{
    arr[i]=vt[i];
}

// constructor for face, assign edge ids
face::face(int *idnos)
{
    for(int i = 0; i < numvertex; i++)
    {
        edgeidnos[i] = idnos[i];
    }
}

// print to screen for debugging information
ostream &operator<<(ostream &stream, face tdp)
{
    stream << "Number of vertices -> " << tdp.numvertex << " , " << "Face Id No -> " << tdp.idno << "\n";
    stream << "Edge numbers -> ";
    for(int i = 0; i < tdp.numvertex; i++)
    {
        stream << tdp.edgeidnos[i] << " ";
    }
    stream << "\n";
    for(i = 0; i < tdp.numvertex; i++)
    {

stream << "Vertex Co-ordinates -> " << tdp.arr[i];
}
return stream;
}

// print face information to CGTS file
void face::fileprint(FILE *fp)
{
    fprintf(fp,"%s %5s %5d %5d %5d %5d %5d\n", "FACE ", " ", idno, numvertex, numadjfaces, 0, loop);
    // print edge number information
    fprintf(fp," EDGE_NUMBERS ");
    for(int i = 0; i < numvertex; i++)
    {
        fprintf(fp,"%5d", edgeidnos[i]);
    }
    // print adjacent face information
    fprintf(fp,"\n");
    fprintf(fp,"ADJACENT_FACES\n");
    for(i = 0; i < numadjfaces; i++)
    {
        fprintf(fp,"%5d", adjfaceids[i]);
    }
    fprintf(fp,"\n");
    // print adjacency relationships
    fprintf(fp,"ADJACENCY_RELATIONSHIP\n");
    for(i = 0; i < numadjfaces; i++)
    {
        // print adjacent face ids
        fprintf(fp,"%5d", adjfaceids[i]);
        // print adjacent relationships
        fprintf(fp,"%5d", adjrel[i]);
        // print common edge ids
        fprintf(fp,"%5d\n", commongedgeid[i]);
    }


```c
}
    fprintf(fp, "\n");
}
```
// Name of source file - topofil.h
// should be included with main file
// compile with main file
// class topofil declaration and definition for output CGTS file

#include <stdlib.h>
#include <fstream.h>
#include <io.h>
#include <string.h>
#include <dos.h>
#include <alloc.h>

class topofil
{
    // file name of CGTS
    FILE *fp;

    public:
    // assign the file name
    void assign(FILE *fp1) { fp = fp1; }
    // write the header info.
    void wrthdrinfo(int *xrange, int *yrange, int *zrange,
                      int nvert, int nedg, int nfac);
    // write the vertex info.
    void wrtvertexinfo(vertex *arrayofvertices, int vercount);
    // write the edge info.
    void wrtedgeinfo(edge *arrayofedges, int edgecount);
    // write the face info.
    void wrtfaceinfo(face *arrayoffaces, int facecount);

};

void topofil::wrtthdrinfo(int *xrange, int *yrange, int *zrange,
                            int nvert, int nedg, int nfac)
{

    // print title info.
    fprintf(fp,"%-30s", " THE GEOMETRIC AND TOPOLOGICAL FILE\n ");
    fprintf(fp,"\n");
    fprintf(fp,"HEADER_SECTION\n");
// print x ranges
fprintf(fp, "%s", " X_RANGE\n");

fprintf(fp, "%-5d %-10s %-5d\n", xrange[0], " ", xrange[1]);
// print y ranges
fprintf(fp, "%s", " Y_RANGE\n");

fprintf(fp, "%-5d %-10s %-5d\n", yrange[0], " ", yrange[1]);
// print z ranges
fprintf(fp, "%s", " Z_RANGE\n");

fprintf(fp, "%-5d %-10s %-5d\n", zrange[0], " ", zrange[1]);

fprintf(fp, "\n");
// print number of vertices
fprintf(fp, "%s", " N_VERTICES\n");

fprintf(fp, "%-5d\n", nvert);

fprintf(fp, "\n");
// print number of edges
fprintf(fp, "%s", " N_EDGES\n");

fprintf(fp, "%-5d\n", nedg);

fprintf(fp, "\n");
// print number of faces
fprintf(fp, "%s", " N_FACES\n");

fprintf(fp, "%-5d\n", nfac);

fprintf(fp, "\n");
// terminate header section
fprintf(fp, "END_HEADER_SECTION\n");

fprintf(fp, "\n");

fprintf(fp, "\n");

}
void tologfile::wrtvertexinfo(vertex *arrayofvertices, int vercount)
{
    // start vertex information
    fprintf(fp,"VERTEX_DESCRIPTION_SECTION\n");

    for(int i = 0; i < vercount; i++)
    {
        // print coordinate info.
        arrayofvertices[i].fileprint(fp);
        fprintf(fp,"\n");
    }

    // terminate vertex section
    fprintf(fp,"END_VERTEX_DESCRIPTION_SECTION\n");

    fprintf(fp,"\n");

    fprintf(fp,"\n");
}

void tologfile::wrtedgeinfo(edge *arrayofedges, int edgecount)
{
    // start edge information
    fprintf(fp,"EDGE_DESCRIPTION_SECTION\n");

    for(int i = 0; i < edgecount; i++)
    {
        // print edge information
        arrayofedges[i].fileprint(fp);
        fprintf(fp,"\n");
    }

    // terminate edge information
    fprintf(fp,"END_EDGE_DESCRIPTION_SECTION\n");

    fprintf(fp,"\n");

    fprintf(fp,"\n");
}

void tologfile::wrtfaceinfo(face *arrayoffaces, int facecount)
{

// start face information
fprintf(fp,"FACE_DESCRIPTION_SECTION\n");

fprintf(fp,"\n");
for(int i = 0; i < facecount; i++)
{
// print face information
arrayoffaces[i].fileprint(fp);
  fprintf(fp,"\n");
}
// terminate face information
fprintf(fp,"END_FACE_DESCRIPTION_SECTION\n");

fprintf(fp,"\n");}
Appendix B - Example CGTS output
THE GEOMETRIC AND TOPOLOGICAL FILE

HEADER_SECTION
X_RANGE
5    15
Y_RANGE
5    15
Z_RANGE
5    10

N_VERTICES
24

N_EDGES
36

N_FACES
18

END_HEADER_SECTION

VERTEX_DESCRIPTION_SECTION
V   1
15   5   10
EDGEN
1   8   33
V   2
5   5   10
EDGEN
1   2   34
V   3
5   15   10
EDGEN
2   3   35
V   4
15   15   10
EDGEN
3   4   36
V   5
15   12   10
EDGEN
4   5   25
V   6
14   12   10
EDGEN
5   6   26
V  7
 14  9  10
EDGEN
 6  7  23

V  8
 15  9  10
EDGEN
 7  8  22

V  9
 15  5  5
EDGEN
 9  12  33

V 10
 5  5  5
EDGEN
 9  10  34

V 11
 5  15  5
EDGEN
 10  11  35

V 12
 15  15  5
EDGEN
 11  12  36

V 13
 12  9  10
EDGEN
 13  16  27

V 14
 9  9  10
EDGEN
 13  14  28

V 15
 9  12  10
EDGEN
 14  15  29

V 16
 12  12  10
EDGEN
 15  16  30

V 17
 9  9  9
EDGEN
17 20 28
V 18
9 12 9
EDGEN
17 18 29
V 19
12 12 9
EDGEN
18 19 30
V 20
12 9 9
EDGEN
19 20 27
V 21
15 9 8
EDGEN
21 22 31
V 22
14 9 8
EDGEN
21 23 32
V 23
15 12 8
EDGEN
24 25 31
V 24
14 12 8
EDGEN
24 26 32
ENDVERTEX_DESCRIPTIONSECTION

EDGE_DESCRIPTIONSECTION
E 1
VIDS
1 2
PATHS
2 34
E 2
VIDS
2 3
PATHS
3 35
PATHS
14 15

E 30
VIDS
16 19
PATHS
15 16

E 31
VIDS
21 23
PATHS
21 22

E 32
VIDS
22 24
PATHS
21 23

E 33
VIDS
1 9
PATHS
8 1

E 34
VIDS
2 10
PATHS
1 2

E 35
VIDS
3 11
PATHS
2 3

E 36
VIDS
12 4
PATHS
11 12

END_EDGE_DESCRIPTION_SECTION

FACE_DESCRIPTION_SECTION

FACE 1 8 8 0 0
EDGE_NUMBERS 1 8 7 6 5 4 3 2
ADJACENT_FACES
<table>
<thead>
<tr>
<th>3 4 4 6 10 11 14 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADJACENCY_RELATIONSHIP</td>
</tr>
<tr>
<td>3 1 1 1 1</td>
</tr>
<tr>
<td>4 1 1 1 8</td>
</tr>
<tr>
<td>4 1 1 1 4</td>
</tr>
<tr>
<td>6 1 1 1 3</td>
</tr>
<tr>
<td>10 1 1 1 7</td>
</tr>
<tr>
<td>11 1 1 1 5</td>
</tr>
<tr>
<td>14 1 1 1 6</td>
</tr>
<tr>
<td>16 1 1 1 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FACE 2 4 4 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDGE_NUMBERS 16 13 14 15</td>
</tr>
<tr>
<td>ADJACENCY_RELATIONSHIP</td>
</tr>
<tr>
<td>8 9 12 15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FACE 3 4 4 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDGE_NUMBERS 1 33 9 34</td>
</tr>
<tr>
<td>ADJACENCY_RELATIONSHIP</td>
</tr>
<tr>
<td>1 4 5 16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FACE 4 8 8 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDGE_NUMBERS 8 22 31 25 4 36 12 33</td>
</tr>
<tr>
<td>ADJACENCY_RELATIONSHIP</td>
</tr>
<tr>
<td>1 1 3 5 6 10 11 13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FACE 5 4 4 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDGE_NUMBERS 9 12 11 10</td>
</tr>
<tr>
<td>ADJACENCY_RELATIONSHIPS</td>
</tr>
<tr>
<td>3 4 6 16</td>
</tr>
</tbody>
</table>
ADJACENCY_RELATIONSHIP
  3 2 2 2 9
  4 2 2 2 12
  6 2 2 2 11
  16 2 2 2 10

FACE  6 4 4 0 0
EDGE_NUMBERS  11 35 3 36
ADJACENT_FACES
  1 4 5 16
ADJACENCY_RELATIONSHIP
  1 2 2 2 3
  4 3 3 3 36
  5 1 1 1 11
  16 3 3 3 35

FACE  7 4 4 0 0
EDGE_NUMBERS  17 20 19 18
ADJACENT_FACES
  8 9 12 15
ADJACENCY_RELATIONSHIP
  8 2 2 2 17
  9 2 2 2 20
  12 2 2 2 18
  15 2 2 2 19

FACE  8 4 4 0 0
EDGE_NUMBERS  17 28 14 29
ADJACENT_FACES
  2 7 9 12
ADJACENCY_RELATIONSHIP
  2 2 2 2 14
  7 1 1 1 17
  9 3 3 3 28
  12 3 3 3 29

FACE  9 4 4 0 0
EDGE_NUMBERS  20 27 13 28
ADJACENT_FACES
  2 7 8 15
ADJACENCY_RELATIONSHIP
  2 2 2 2 13
  7 1 1 1 20
  8 3 3 3 28
  15 3 3 3 27

FACE  10 4 4 0 0
EDGE_NUMBERS  23 7 22 21

138
ADJACENT_FACES
1 4 13 14
ADJACENCY_RELATIONSHIP
1 2 2 2 7
4 1 1 1 22
13 1 1 1 21
14 3 3 3 23

FACE 11 4 4 0 0
EDGE_NUMBERS 24 25 5 26
ADJACENT_FACES
1 4 13 14
ADJACENCY_RELATIONSHIP
1 2 2 2 5
4 1 1 1 25
13 1 1 1 24
14 3 3 3 26

FACE 12 4 4 0 0
EDGE_NUMBERS 29 15 30 18
ADJACENT_FACES
2 7 8 15
ADJACENCY_RELATIONSHIP
2 2 2 2 15
7 1 1 1 18
8 3 3 3 29
15 3 3 3 30

FACE 13 4 4 0 0
EDGE_NUMBERS 24 31 21 32
ADJACENT_FACES
4 10 11 14
ADJACENCY_RELATIONSHIP
4 4 4 4 31
10 2 2 2 21
11 2 2 2 24
14 2 2 2 32

FACE 14 4 4 0 0
EDGE_NUMBERS 26 6 23 32
ADJACENT_FACES
1 10 11 13
ADJACENCY_RELATIONSHIP
1 2 2 2 6
10 3 3 3 23
11 3 3 3 26
13 1 1 1 32
FACE 15 4 4 0 0
EDGE_NUMBERS 27 16 30 19
ADJACENT_FACES
  2 7 9 12
ADJACENCY_RELATIONSHIP
  2 2 2 2 16
  7 1 1 1 19
  9 3 3 3 27
  12 3 3 3 30

FACE 16 4 4 0 0
EDGE_NUMBERS 34 2 35 10
ADJACENT_FACES
  1 3 5 6
ADJACENCY_RELATIONSHIP
  1 2 2 2
  3 3 3 3 34
  5 1 1 1 10
  6 3 3 3 35

END_FACE_DESCRIPTION_SECTION
Vita

Rajendra V Auradkar was born on June 10, 1964 at Bangalore, India. He completed his Bachelors in Industrial and Production Engineering from Bangalore University in May 1987. He worked for Pythagoras Communications Limited, India from September 1987 to August 1988 as a Computer Analyst. In August 1988 he joined the B.M.S. College of Engineering as an Instructor.

In pursuance of higher studies he resigned from his job and began his graduate work in Industrial Engineering at Virginia Tech in Fall 1989. On completion of his graduate studies in November 1992, he will join Unisys Corporation, Minneapolis as a Computer Software Consultant.