

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

FINITE ELEMENT MESH GENERATION IN NON-MANIFOLD GEOMETRIES

by

Ergun Nart

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy

in

Mechanical Engineering

Lehigh University

June 2002

UMI Number: 3048967

UMI[®]

UMI Microform 3048967

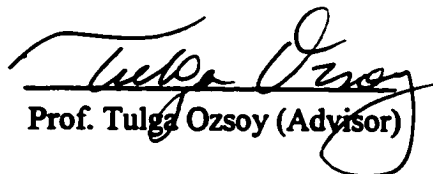
Copyright 2002 by ProQuest Information and Learning Company.

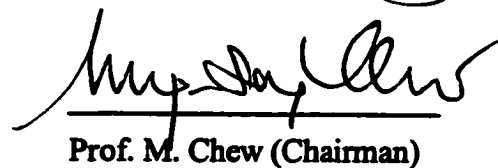
**All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.**

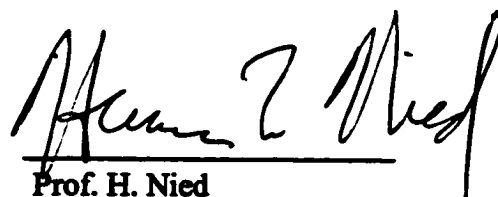
**ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346**

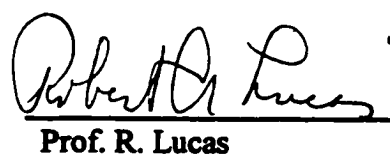
Approved and Recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

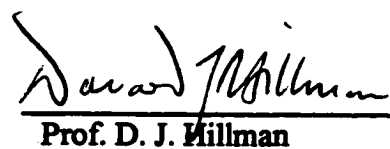
04/22/2002
(Date)


Prof. Tulga Ozsoy (Advisor)


Prof. M. Chew (Chairman)


Prof. H. Nied


Prof. R. Lucas


Prof. D. J. Millman

Acknowledgments

The author wishes to express his grateful appreciation and thanks to his advisor, Prof. M. Tulga Ozsoy, for his continuous support and valuable guidance throughout the Ph.D. study. The guidance and cooperation of the Ph.D. committee members are also appreciated.

The continuous encouragement and understanding by my wife, Reyhan, on the other hand, has been a great support during the course of this study. Thanks are also due to my parents for their every kind of contributions from a far away distance.

Table of Contents

ACKNOWLEDGMENTS	III
TABLE OF CONTENTS	IV
TABLE OF FIGURES.....	VI
ABSTRACT.....	1
INTRODUCTION.....	2
1.1 OBJECTIVE OF THE PRESENT STUDY.....	2
CHAPTER 2	4
MANIFOLD REPRESENTATION SCHEMES.....	4
2.1 WIREFRAME MODELING	4
2.2 SURFACE MODELING	5
2.3 SOLID MODELING	5
2.3.1 SOLID MODELING TECHNIQUES	7
2.3.1.1 DECOMPOSITION MODELS.....	8
2.3.1.2 CONSTRUCTIVE MODELS	8
2.3.1.2.1 HALF SPACE MODELS	8
2.3.1.2.2 CONSTRUCTIVE SOLID GEOMETRY	11
2.3.1.3 BOUNDARY MODELS	16
2.3.1.4 HYBRID MODELS	24
CHAPTER 3	25
NON-MANIFOLD REPRESENTATION SCHEME.....	25
3.1 NON-MANIFOLD SOLID MODELING	25
3.1.1 NON-MANIFOLD TOPOLOGY	27
3.1.2 DESCRIPTION OF RADIAL EDGE DATA STRUCTURE.....	31
CHAPTER 4	41
LINKED LIST STRUCTURES.....	41
4.1 LINKED STRUCTURE	41
4.1.1 SINGLE-LINKED LISTS	41
4.1.2 DOUBLE-LINKED LIST	47
4.1.3 CIRCULAR-LINK LIST.....	48
CHAPTER 5	50
EULER OPERATORS IN TOPOLOGIES.....	50
5.1. EULER OPERATORS	50
5.1.1. EULER OPERATORS IN MANIFOLD TOPOLOGY	50
6.1 EULER OPERATORS IN NON-MANIFOLD TOPOLOGY	55

CHAPTER 6.....	62
DEVELOPMENT OF NON-MANIFOLD BOOLEAN OPERATIONS.....	62
6.1. BOOLEAN OPERATIONS IN NON-MANIFOLD MODELS	62
6.1.1 INTERSECTION OPERATION	64
6.1.1.1 MERGING OPERATION.....	75
6.1.2 EXTRACTING OPERATION	75
6.1.3 SELECTION OPERATION	77
CHAPTER 7.....	80
APPLICATIONS IN MANUFACTURING	80
7.1 APPLICATIONS OF NON-MANIFOLD TOPOLOGY	80
CHAPTER 8.....	85
DEVELOPMENT OF MESH GENERATION PROCEDURE FOR NON-MANIFOLD TOPOLOGIES.....	85
8.1. FINITE ELEMENT MODELING.....	85
8.2. MESH GENERATION	87
8.2.1 DELAUNAY TRIANGULATION	88
8.2.2 INCREMENTAL DELAUNAY TRIANGULATION IN 2-D	90
8.2.3 PROPERTIES OF A TRIANGLE	94
8.2.4 INCREMENTAL DELAUNAY TRIANGULATION METHOD IN 3-D	105
8.2.5 PROPERTIES OF TETRAHEDRONS	106
8.2.6 NODE INSERTION INTO THE GEOMETRY	115
8.2.7 TWO DIMENSIONAL INTERIOR NODE INSERTION.....	116
8.2.8 MESH GENERATION ALGORITHM IN SOLIDS	121
8.2.9 THREE DIMENSIONAL INTERIOR NODE GENERATION.....	123
CHAPTER 9	124
IMPLEMENTATION	124
9.1 THE IMPLEMENTED TOPOLOGY.....	124
9.2 CODE DEVELOPMENT	126
9.2 CASE STUDY ONE	129
9.3 CASE STUDY TWO	150
9.4 CASE STUDY THREE.....	159
CHAPTER 10.....	162
CONCLUSIONS AND FUTURE WORK.....	162
10.1 CONCLUSIONS.....	162
10.2 FUTURE WORK	164
REFERENCES.....	165
VITA.....	170

Table of Figures

Figure 1: Non-manifold Geometry.	3
Figure 2: A wireframe model.....	4
Figure 3: Half-space Boolean operation.	10
Figure 4: Instances and parameterized solids.	12
Figure 5: A CSG tree.	13
Figure 6: Boundary Representation.	17
Figure 7: Edge-based boundary representation.....	18
Figure 8: The orientation of faces.	19
Figure 9: The winged-edge data structure.	20
Figure 10: "Full" winged-edge data structure.....	21
Figure 11: "Full" winged-edge data structure.....	21
Figure 12: "Full" winged-edge data structure.....	22
Figure 14: A face defined by winged-edge data structure.	23
Figure 15: Manifold and non-manifold surfaces.	25
Figure 16: Non-manifold results from the Boolean operation of manifold objects.....	26
Figure 17: Non-manifold topology.	30
Figure 18: Manifold and non-manifold topological structure relationships.	31
Figure 19: Topological relationship in loop-use [Weiler].	38
Figure 20: Topological relationship in edge-use [Weiler].	38
Figure 21: Topological relationship in vertex-use [Weiler].	40
Figure 22: Single link list.....	42
Figure 23: Adding the first list.....	44
Figure 24: Adding to the beginning of a linked list.	45
Figure 25: Adding to the middle of a linked list.....	46
Figure 26: Double-linked list.	47
Figure 27: Circular-linked list.....	49
Figure 28: Minimal set of Euler operators.	56
Figure 29: A non-manifold model.	58
Figure 30: Objects for Boolean operations.	64
Figure 31: An edge connecting two vertices.....	65
Figure 32: Intersection of a vertex and an edge.	66
Figure 33: Vertex-Edge intersection on the models.	66
Figure 34: Edge-edge intersection.	67
Figure 35: Edge-edge intersection.	67
Figure 36: Edge-edge intersection.	68
Figure 37: Determining the distance between curve edges.	68
Figure 38: Intersection of edges in the model.....	69
Figure 39: Determining the distance between a face and an edge.	70
Figure 40: Single vertex loop case.....	70
Figure 41: One vertex intersect with a face.	71
Figure 42: Two vertices included in a face.....	71
Figure 43: Two intersection vertices belonging the same loop	71

Figure 44: Two vertices belong to two loops.....	72
Figure 45: Intersection of a face and an edge.	72
Figure 46: The edge-face intersection of two models.....	73
Figure 47: Coincident faces.	73
Figure 48: Intersection of two faces.....	74
Figure 49: Face-Face intersections.	74
Figure 50: Merged object.....	75
Figure 51: Repartitioned solids.....	76
Figure 52: Extracted faces, edges, and vertices.	77
Figure 53: Repartitioned and mapped merged object.	78
Figure 54: The resultant solid.	79
Figure 55: The final intersecting non-manifold object.	79
Figure 56: Profile and outward offset.	83
Figure 57: Profile and inward offset.	83
Figure 58: Steps in creating parallel tool paths.....	84
Figure 59: Mesh hierarchy and associated entities.	87
Figure 60: Delaunay triangulation and corresponding Voronoi diagram.	89
Figure 61: Incremental Delaunay Triangulation Method.	91
Figure 62: Experimental results in 2-D.....	92
Figure 63: Experimental results in 2-D.....	93
Figure 64: Possibilities for different triangle orientation.....	94
Figure 65: Translated coordinate system.	95
Figure 66: Orientation of a triangle.	97
Figure 67: Geometry of a triangle.....	98
Figure 68: Geometry of a triangle.....	99
Figure 69: Circle through three points.....	100
Figure 70: Geometry of innercircle.....	103
Figure 71: The initial box mesh.	105
Figure 72: After inserting the first node.	106
Figure 73: The geometry of a tetrahedron.	107
Figure 74: The geometry of a tetrahedron.	108
Figure 75: Translated origin to the one of the corner of a tetrahedron.....	108
Figure 76: The innersphere of a tetrahedron.....	110
Figure 77: A side view of a tetrahedron.....	112
Figure 78: The geometry of tetrahedron.	113
Figure 79: Non-manifold solid part to be meshed.	117
Figure 80: Boundary triangulation.....	118
Figure 81: Inserting the first prospective node.	118
Figure 82: Inserting nodes.	119
Figure 83: The final node inserting.....	120
Figure 84: The final mesh after smoothing.....	120
Figure 85: The quality of the resulting mesh.	121
Figure 86: Flow diagram for NMT-mesh.	127
Figure 87: The bounding box.....	130
Figure 88: The nodes on the vertices and edges.	130

Figure 89: The primary surface triangulation.	131
Figure 90: Updating the 3D Delaunay triangulation.....	131
Figure 91: The mesh without nodes on faces and interior of the solid.	132
Figure 92: The face #1's triangulation in 2D parametric space.....	133
Figure 93: The face #1's smoothing and quality of the mesh (v1:ratio, v2:number of elements).	133
Figure 94: The face #2's triangulation in 2D parametric space.....	134
Figure 95: The face #2's smoothing and quality of the mesh (v1:ratio, v2:number of elements).	134
Figure 96: The face #3's triangulation in 2D parametric space.....	135
Figure 97: The face #3's smoothing and quality of the mesh (v1:ratio, v2:number of elements).	135
Figure 98: The face #4's triangulation in 2D parametric space.....	136
Figure 99: The face #4's smoothing and quality of the mesh (v1:ratio, v2:number of elements).	136
Figure 100: The face #5's triangulation in 2D parametric space.....	137
Figure 101: The face #5's smoothing and quality of the mesh (v1:ratio, v2:number of elements).	137
Figure 102: The face #6's triangulation in 2D parametric space.....	138
Figure 103: The face #6's smoothing and quality of the mesh (v1:ratio, v2:number of elements).	138
Figure 104: The face #7's triangulation in 2D parametric space.....	139
Figure 105: The face #7's smoothing and quality of the mesh (v1:ratio, v2:number of elements).	139
Figure 106: The face #8's triangulation in 2D parametric space.....	140
Figure 107: The face #8's smoothing and quality of the mesh (v1:ratio, v2:number of elements).	140
Figure 108: The face #9's triangulation in 2D parametric space.....	141
Figure 109: The face #9's smoothing and quality of the mesh (v1:ratio, v2:number of elements).	141
Figure 110: The face #10's triangulation in 2D parametric space.....	142
Figure 111: The face #10's smoothing and quality of the mesh (v1:ratio, v2:number of elements).	142
Figure 112: The face #11's triangulation in 2D parametric space.....	143
Figure 113: The face #11's smoothing and quality of the mesh (v1:ratio, v2:number of elements).	143
Figure 114: The face #12's triangulation in 2D parametric space.....	144
Figure 115: The face #12's smoothing and quality of the mesh (v1:ratio, v2:number of elements).	144
Figure 116: The face #13's triangulation in 2D parametric space.....	145
Figure 117: The face #13's smoothing and quality of the mesh (v1:ratio, v2:number of elements).	145
Figure 118: The face #14's triangulation in 2D parametric space.....	146
Figure 119: The face #14-15's smoothing and quality of the mesh (v1:ratio, v2:number of elements).	146

Figure 120: The face #16's triangulation in 2D parametric space.....	147
Figure 121: The face #16's smoothing and quality of the mesh (v1:ratio, v2:number of elements).....	147
Figure 122: The node in the 3D model.	148
Figure 123: One surface mesh in 3D space.	148
Figure 124: The final shrunk mesh for manifold case.	149
Figure 125: The stretch histogram of the final mesh. (v1: stretch, v2: number of elements).	150
Figure 126: The solid view of a non-manifold model.	150
Figure 127: Before union Boolean operation.....	151
Figure 128: After intersection operation.....	151
Figure 129: The merged edge and the solid.....	152
Figure 130: Before intersection operation.	152
Figure 131: After intersection operation.	153
Figure 132: The boundary discretization for the non-manifold model.	153
Figure 133: The boundary triangulation of the non-manifold model.	154
Figure 134: The front face triangulation in absolute coordinates.	154
Figure 135: The front face triangulation after internal nodes are inserted.	155
Figure 136: Boundary triangulation of the model.....	155
Figure 137: Zoomed view of the interface.....	156
Figure 138: Shrunk tetrahedrons in the solid part of the model.	156
Figure 139: Shrunk tetrahedrons in the non-manifold model.....	157
Figure 140: Shrunk tetrahedrons in the non-manifold model.....	158
Figure 141: A face cutting a solid.....	159
Figure 142: Intersection curves created by Boolean operation process.....	159
Figure 143: Boundary discretizing in subtraction Boolean operation.	160
Figure 144: Boundary triangulation in subtraction Boolean operation.	161
Figure 145: Shrunk tetrahedrons in the non-manifold model.....	161

Abstract

In this study, a mesh generation process for non-manifold geometries has been developed. The procedure has been implemented in three phases. The first phase is the representation of the non-manifold topology. For this purpose, a special version of a radial edge structure has been constructed by gathering the connectivity information among the topological elements.

During the second phase, the special Boolean operations for Non-manifold Geometries (NMG) have been constructed. This has been accomplished by employing the curve-surface and surface-surface intersection processes, followed by classification of the resulting points and curves with respect to the specified geometric operations.

Having the necessary geometric information prepared, the last phase of the process is to employ a mesh generation task on non-manifold geometry. This is done by first generating nodes on the boundaries, and then by generating nodes inside boundaries.

Delaunay Triangulation with incremental node insertion method has been primarily used since it allows constrained nodes to be used in every stage of the mesh generation process. The method also tends to create triangles and tetrahedrons with less distortion.

A newly developed Boolean operation of subtraction type has also been implemented for generating non-manifold geometries. This is a unique tool for fracture mechanics for various type of analysis, such as stress distribution analysis at crack tips and crack propagation estimations.

Chapter 1

Introduction

1.1 OBJECTIVE OF THE PRESENT STUDY

In order to solve a wide variety of problems in mathematical physics on very general domains, finite element analysis can be used as a powerful numerical analysis procedure. The disadvantage of the method are the amounts of computational effort required to solve the systems of equations that are generated by complex non-linear systems and the amount of time required to create a valid finite element model for the analysis. Not surprisingly, the model generation portion of the process dominates the overall cost of the finite element model. However, this cost can be reduced by means of tight integration of the geometric and finite element modeling and the development of automated finite element modeling tools.

Geometric modeling offers several methods, which can be deployed in order to reduce the cost of the finite element analysis (FEA). Simplification of the geometry is one of them. The geometry is simplified by means of the following two procedures.

1. Dimensional reduction
2. Insignificant feature removal

After the original geometry has been simplified, it is usually represented as a wireframe, a surface model, or a solid model.

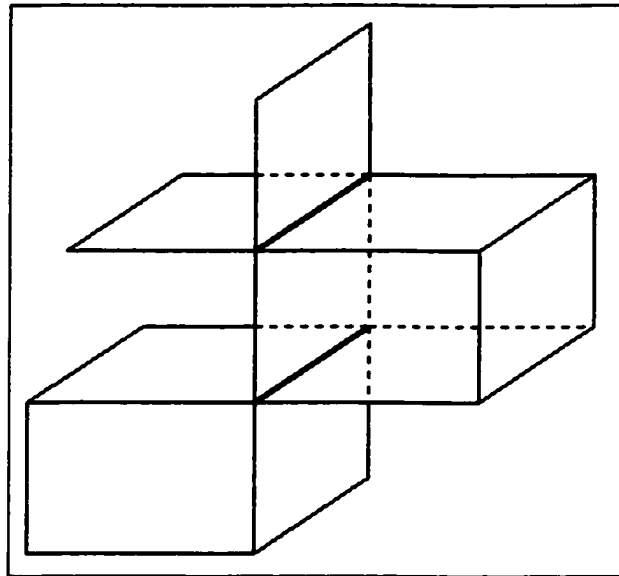


Figure 1: Non-manifold Geometry.

Since the resultant model requires less finite element nodes, computational time is reduced dramatically in the analysis. But, the resultant model in Figure1 cannot be represented by the conventional solid modeling topology, because the geometry is simplified to a non-manifold geometry, which is a combination of 1-D, 2-D, and 3-D geometries.

In today's world, non-manifold topology is in the process of full integration with finite element modeling (FEM). Although some commercial software vendors have already implemented Radial Edge Structure representation, none of them supported the full capabilities of Boolean operations to deal with the construction of non-manifold geometries.

Chapter 2

Manifold Representation Schemes

2.1 WIREFRAME MODELING

Wireframe modeling, one of the earliest geometric modeling techniques, represents objects by edge curves and points on the surface of the object. Actually wireframe-modeling systems is an early effort in automated drawing. Although this would be a natural representation, assuming that the object is to generate line drawing of the design, projected in certain directions, unfortunately, they may be ambiguities in interpreting the representation. A simple, well-known example of this phenomenon is shown in Figure 2.

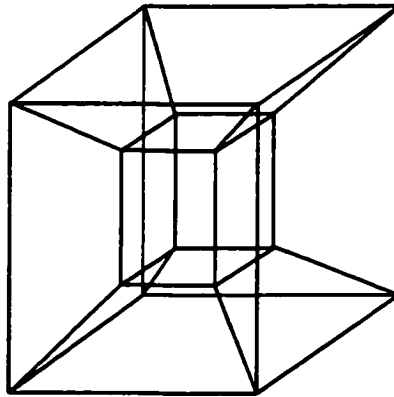


Figure 2: A wireframe model.

The figure shows a block with a hole through its center. It is not possible to understand the direction of the hole, since it could lie in any of the three principal directions. For this reason, wireframes are not the preferred object representations.

On the other hand, wireframe objects have small storage requirements and can be accessed and displayed quickly.

The resulting line drawings have an acceptable visualization aid in many situations, and can provide a quick feedback between the modeling system and the designer. For this reason, many modeling systems retain the capability to generate wireframe drawings.

2.2 SURFACE MODELING

Surface modeling techniques go one step further than wireframe representations by also providing mathematical description of the shape of the surfaces.

The surfaces may have various types such as the parametric surface patches and, in particular, various types of spline surface. Actually, surface design with splines originated in the automobile industry (by Pierre Bezier, France), for car body design. The representation also has found itself an application area in ship hulls design; and in the aircraft industry, for design of wings, fuselages, and so on, because surface modeling technique gives a way to create graphics display and numeric control machining of carefully constructed models.

The difficulty of evaluating and representing the intersection of parametric surface patches has triggered the development of a new representation scheme, solid modelers.

2.3 SOLID MODELING

Solid modeling explicitly or implicitly contains information about the closure and connectivity of the volumes of solid shapes to represent the physical objects. Solid

modeling has a number of advantages over previous surface modeling techniques, because it guaranties that any model, which is created will form closed and bounded objects. Hence it will be more closely related to physically realizable shapes that can be guaranteed for surface models. Solid models, unlike surface models, allow a modeler system to distinguish the inside and outside of a volume, to determine the center of gravity, etc. Typical solid modeling systems also offer tools for the creation and manipulation of complete solid shapes, while maintaining the integrity of the representation.

The majority of current Solid Modeling systems in the market find its roots from a small number of developments by research groups. Braid [35] and others, of the university of Cambridge, England, first introduced their BUILD system in Budapest, 1973; Okino and his colleagues, of Japan's Hokkaido University, described TIPS. Both BUILD and TIPS uses the same principal of creating complex shapes by combining geometric "primitives" by using different way of data representation within the computer; the Japanese system used what is now called " Constructive Solid Modeling", or CSG, while the English system employed the technique now known as " Boundary Representation", or B-Rep.

Boumgart[32], of Stanford University, has made a novel addition to the B-Rep approach in 1974. He simply described GEOMED, a system based on the "Winged Edge Representation". This technique utilizes substantial performance advantages over the others. Spur, and his colleagues from technical university of Berlin have developed COMPAC system in 1975. On the other hand, Eastman and his colleagues at Canegie-Mellon university of Pittsburg, Pennsylvania, developed their architecturally oriented

GLIDE system in 1977. Voelaker and Requicha[36] of the university of Rochester, New York, published their work on PADL-1, a system that incorporated the advantages CSG and B-Rep by maintaining a dual representation.

2.3.1 SOLID MODELING TECHNIQUES

In solid modeling, representations simply reduce the infinite point set in a finite amount of computer storage in a more generic fashion. According to Mantyla [31], representations achieving this may be divided in three large classes as follows;

1. Decomposition Models, that represents a complex object as a collection of simple objects, and combine with a single “gluing” operation.
2. Constructive Models, that represents a complex object as a combination of primitive objects. Each of the primitive objects is represented as an instance of a primitive solid type. Constructive modeling has more general construction operations than mere gluing.
3. Boundary Models, that represents a complex object in terms of its boundary. The boundary of a three-dimensional “solid” object is represented as a collection of faces. Faces are represented with one-dimensional curves, edges. Hence, boundary models are based on the surface oriented formulation of solid objects although they seem similar to the other representation methods.

2.3.1.1 DECOMPOSITION MODELS

Since the space defined by a solid cannot be represented in terms of points belonging to a solid, tiny cubes that are contained (completely or partially) in the solid can easily be used to approximate the body in which the cubes have to be non-overlapping and of uniform size and orientation. They form a regular subdivision of the space.

The solid representation employs this kind of approximation is called “the exhaustive enumeration”. Tomography, which supplies three-dimensional digital information, is a good source of data to start with. Unfortunately, these direct solid description techniques cannot be used in solid modeling.

On the other hand, by using other representation such as the constructive or boundary models, exhaustive enumeration can be created. Boolean set operations, or other suitable algorithms for exhaustive enumeration can be employed to create other objects as well.

2.3.1.2 CONSTRUCTIVE MODELS

2.3.1.2.1 HALF SPACE MODELS

The basic idea behind the Half Space Models is to start from some sufficiently simple objects that can be represented directly, and model other objects in terms of very general combinations of the simple objects. Half-space models use this approach in direct fashion.

For example, a class of objects can be represented in terms of an analytic function $f(x,y,z)$ defined everywhere in 3-D space. Because $f(x,y,z)=0$ divides the defined space into

subsets, objects defined by $f(x) \geq 0$ and $f(x) \leq 0$ are referred to as half spaces. For instance, functions;

$$ax + by + cz + d \geq 0$$

$$x^2 + y^2 - r^2 \leq 0$$

yield useful objects, namely the plane ($ax + by + cz = 0$) that defines all points on the positive side, and the cylindrical half-space that consists of all points or inside on infinite cylinder whose axis z-axis and r =radius.

Other half-spaces include the remaining natural quadratic surfaces such as spheres and cones, certain higher-order surfaces such as Torus. It can be seen that this collection includes both unbounded and bounded half-spaces. On the other hand, not all surfaces are half-spaces. For example, the various surface patches widely used in surface models are not half-spaces, because they do not divide the space into distinct subsets. This means that the modeling space of a half-space modeler cannot easily be extended to cover also objects bounded by surface patches.

Boolean Set Operations:

Half-spaces establish the basic modeling primitives used half-space model. The natural modeling procedures for half-space models are the Boolean set operations union ($\cup$), and intersection ($\cap$), and set difference ($\setminus$).

Boolean set operations are used to construct the half-space model by combining instances of half-space. For instance, to describe a finite cylinder of length h , one

cylindrical half-space and two planar half-spaces are needed, then they are combined together with the set operation “ $\cap$ ”.

$$H1: x^2 + y^2 - r^2 \leq 0$$

$$H2: z \geq 0$$

$$H3: z - h \leq 0$$

$$C = H1 \cap H2 \cap H3$$

This construction is illustrated in Figure 3.

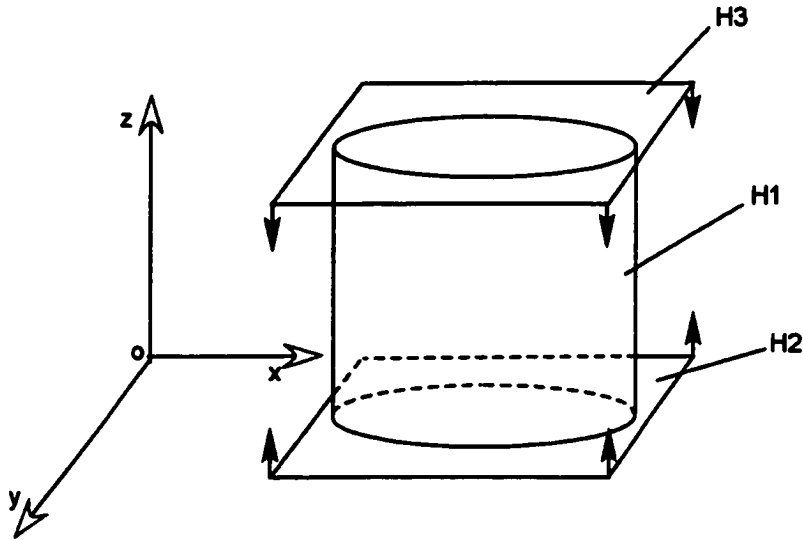


Figure 3: Half-space Boolean operation.

The availability of half-spaces, and the generality of the operations available for combining them determine the ability of creating half-space models. Half-spaces are usually objects with infinite point sets.

Even a combination of them may be infinite, and hence not all combinations are valid solids (If finiteness is a required characteristic of a “solid” object).

2.3.1.2.2 CONSTRUCTIVE SOLID GEOMETRY

Although half-space models are an easily understandable approach to solid modeling, for users, however, it is easier to operate with bounded primitives instead of the unbounded half-spaces.

It can be seen that combinations of half-spaces may be unbounded objects with point sets that are not considered valid solids also. To avoid the generation of unbounded sets, the so-called Constructive Solid Geometry (CSG) approach to solid modeling uses only bounded point sets as its primitives.

Instances and Parameterized Solids:

A new shape can be defined by using a simple linear transformation of an existing shape. For a unit cube, there are several ways of transforming it into a new shape by using scaling operators. Scaling of all 3-D components creates new different-size cubes if equal scaling is used.

On the other hand, differential scaling creates a variety of rectangular solids. Each new shape is a particular instance of the initial cube. Figure 4 shows examples of specific instances of an original shape. It can be seen that such transformation affects the geometry but not the topology of a shape.

The shapes in this figure happen to be regular convex shapes. The original, or initial shape, is often called the primitive shape, and the result of its transformation produces an instance

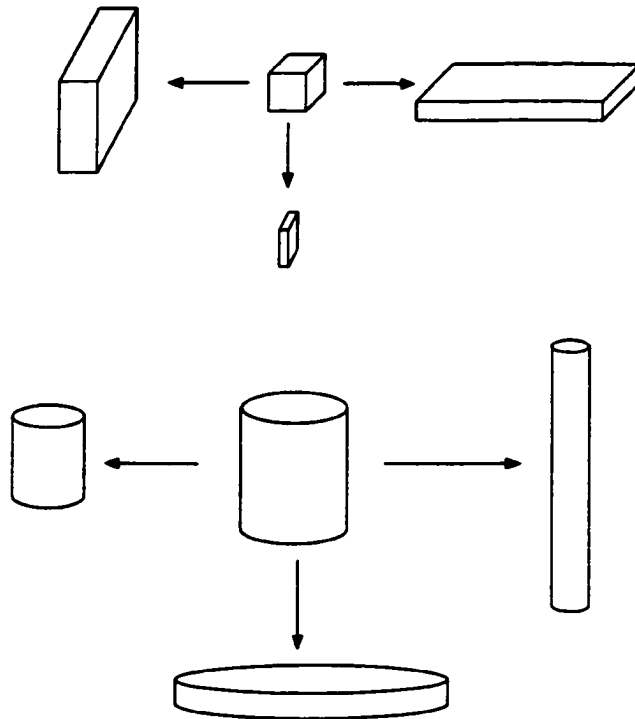


Figure 4: Instances and parameterized solids.

Representation of CSG Models:

CSG employs the “building block” approach to solid modeling. The user of CSG modeler can use only parameterized instances of solid primitives, and Boolean set operations on them.

Each primitive is actually formed as a combination of half-spaces. The user has no direct access to individual half-spaces to manipulate, however. Hence the

mathematical modeling space of CSG models is exactly the same as that of pure half-space models. Boolean set operations with a rigid motion are the main differences arise (Figure 5).

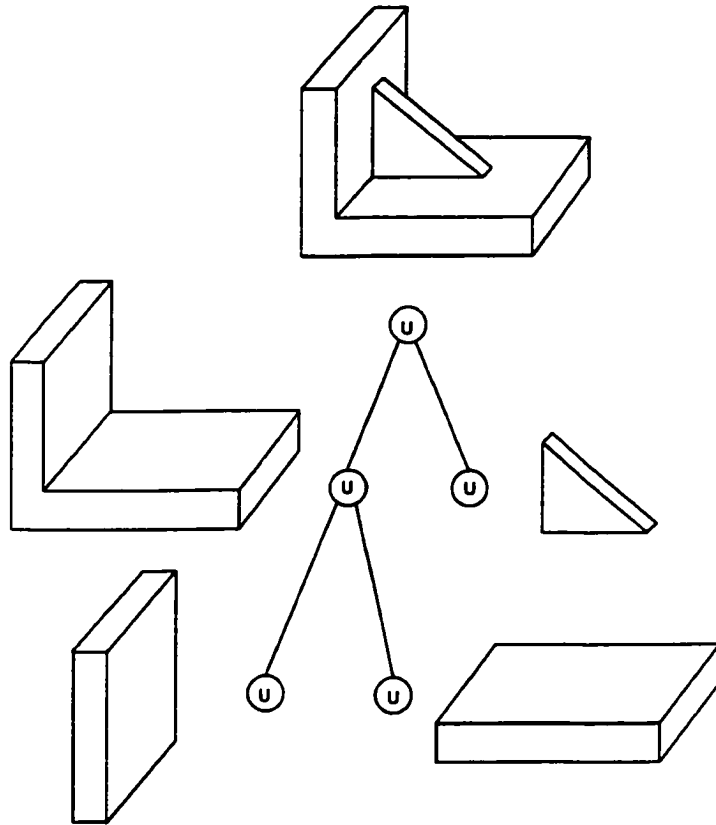


Figure 5: A CSG tree.

The CSG tree is an implicit description of the geometry of the solid modeled that must be evaluated in some fashion in order to create graphical output or perform calculations. After constructing a CSG object, its geometry can be interrogated in various ways. The important query is to test whether a point (x,y,z) is inside a solid, is on its surface, or is outside of it. This query is usually addressed as a point/solid

classification. Other such queries, such as classification of how a surface intersects a solid, and a test of whether two solids intersect in a non-empty volume has to be determined in order to evaluate the geometry. On the other hand, it can be seen that, after constructing a CSG object, objects are already positioned appropriately with respect to each other. To make this positioning meaningful, a relationship between the local coordinate frames of the objects is established. A simple method is to identify the local frames with a single, universal coordinate frame. The universal frame is often referred to as the world coordinate system.

Point/Solid Classification:

Point/solid classification can be fulfilled with an algorithm that has a simple conceptual structure. If a point is inside or outside of a solid, it is trivial to evaluate its status since the boundary of every instant object are known. On the other hand, it can be realized that a difficulty may arise when the point lies on the surface of a primitive, and this problem gives a way for the introduction of neighborhoods concept. For two-way classification, a neighborhood model can easily be introduced because the combination rules are simple.

Curve/Solid Classification:

Another process to overcome is the classification of a space curve against a solid. The algorithm for classifying a line or curve against the solid is put in order exactly like the point/solid classification

1. Send the line or curve description to the leaves. Partition the curve into segments labeled inside, outside, or on the surface of the primitive.

2. Propagate the segments back upward, and merge them appropriately.

Wireframe generation task is a good example on the use of the edge-solid classification. Based on the edge/solid classification, the wireframe generation algorithm can be combined from the following steps:

1. Generate the set of all candidate edges by intersecting curves of all half-spaces appearing in CSG tree.
2. Classify each edge against the CSG tree, and add the component edge on the surface to the result.

Surface/Solid Classification:

A surface gets intersected with a solid in a number of areas. Each area consists of curve segments, and each segment is on the intersection of the surface with one of the primitives of the solid. The general strategy is simple as following:

1. Intersect the surface with each of the primitives.
2. Classify the resulting curves, and determine the boundary edges of those surface areas with labeling them as inside or outside, or on the solid surface.
3. Combine the segments constructing a boundary representation of the respective surface areas.

Surface/Solid classification can be effectively used to convert from a CSG to a boundary representation. Such a conversion algorithm simple uses on the “generate and classify” procedure. In this procedure, all pairs of intersecting primitives in a CSG object are considered. After obtaining for each a set of space curves in which they intersect, each curve against the solid is classified. Those segments that are on the surface of the object are easily determined. Each segment will be an edge of the boundary representation. These segments now define, on the surface of the primitives, areas that will be the faces of the boundary representation. By considering the neighborhoods, the topological information needed to determine the adjacencies and incidence of the various faces is derived.

2.3.1.3 BOUNDARY MODELS

A solid can also be represented unambiguously by describing its surfaces, and topologically orienting them such a way that each surface points, on which side of the solid interior lies, can be determined.

This description consists of two parts, a topological description of the connectivity and orientation of vertices, edges, faces, and a geometric description for representing these surface elements in space. The basic rules of boundary models are formed by the three object types, face, edge, end vertex, and the geometric information attached to them.

In addition to geometric information such as face and curve equations and vertex coordinates, a boundary model must also represent how the faces, edge, and vertices are related to each other. All information of the geometry of the entities is bundled under the

term *geometry* of a boundary model, and the information of their interconnections is named *topology*. All boundary models represent faces in terms of explicit nodes of a boundary data structure. Many alternatives for representing the geometry and topology of a boundary model are possible, such as Polygon-Based Boundary models, Vertex-based Boundary Models, and Edge-Based Boundary Models.

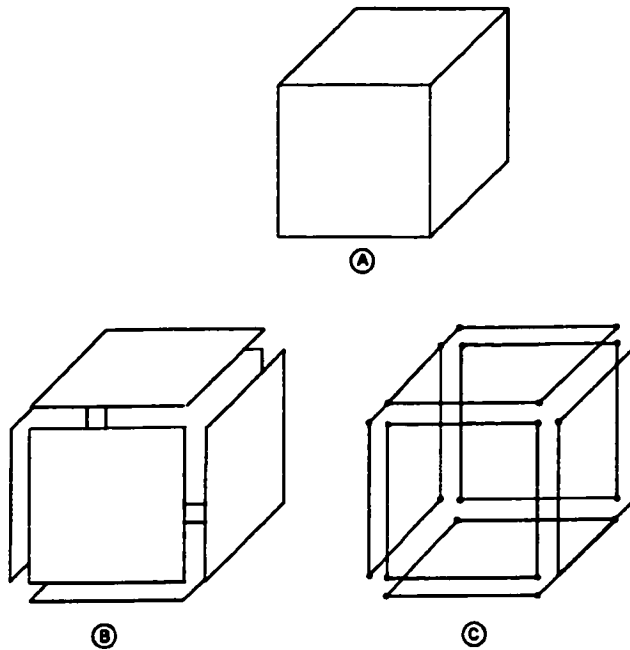


Figure 6: Boundary Representation.

Edge-Based Boundary Models:

An edge-based boundary model represents a face boundary in terms of edges, and loops. Vertices of the face are represented only through edges. Data structure indicates an orientation for each edge, and each face.

For example, e_1 has two vertices v_1, v_2 in clockwise orientation, also face f_1 consists of these edges in clockwise (Figure 7).

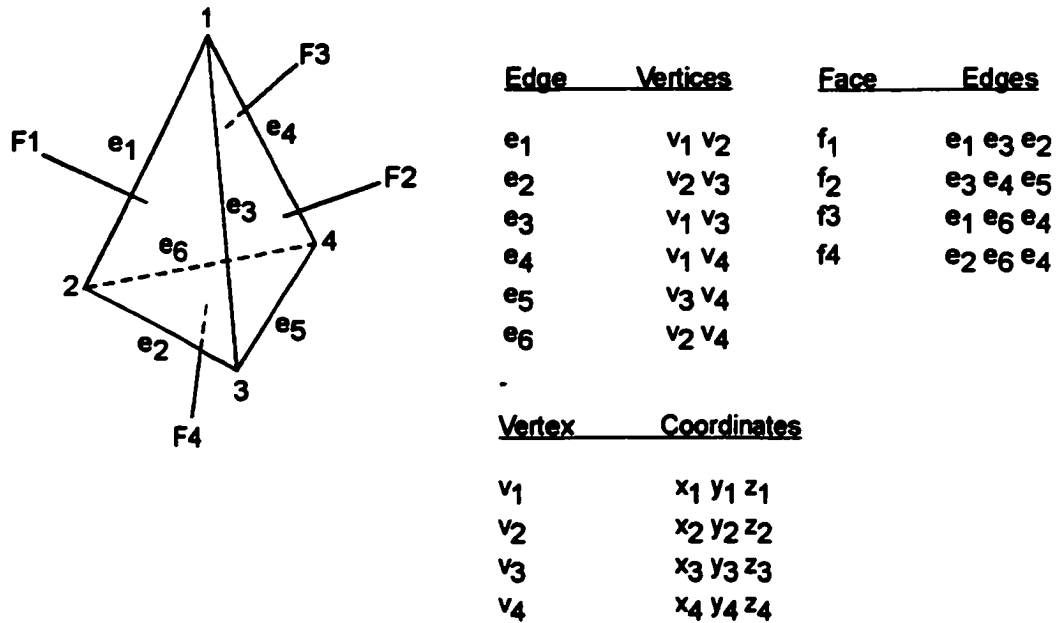


Figure 7: Edge-based boundary representation.

Winged-Edge Representation:

The winged-edge structure offers a way to represent the edge adjacency information as a single unified structure. It was originally developed by Baumgard[32] at Stanford, CA in the early 70's.

It was used to model environments of planar polyhedral solids in computer vision research for robotics. To represent the boundary graph of the topological adjacencies of faces, edges, and vertices embedded in the surface of planar-faced polyhedral solid

models, the winged-edge representation is also employed successfully. Groups of researchers at Carnegie-Mellon and Cambridge universities independently enhanced the winged-edge representation. So that disconnected graphs are allowed by making additions to the supporting structures. These improvements are implemented in these groups' respective solid modeling systems, GLIDE and BUILD2.

The topological information stored in the winged edge structure for each edge forms the adjacencies of the given edge with other edges, vertices, and faces. Each edge e presents in exactly two faces. Exactly two other edges e^+ and e^- are located after e in these faces. Moreover, because of the consistent orientation of faces, e occurs exactly once in its positive orientation, and exactly once in the opposite orientation (Figure 8).

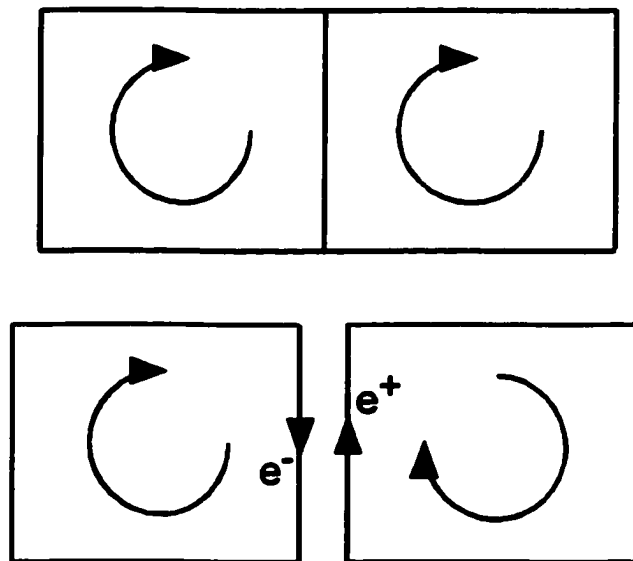


Figure 8: The orientation of faces.

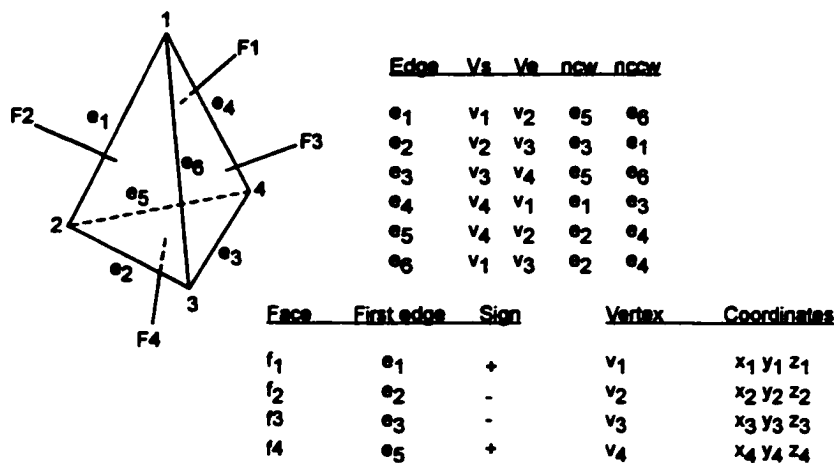


Figure 9: The winged-edge data structure.

The winged-edge data structure uses these structural properties by associating the identifiers of two “next” edges with an edge label. By convention, this data is denoted by ncw and nccw for “next clockwise” and “next counterclockwise”; in particular, ncw points the next edge in the face where the edge occurs in its positive orientation, and nccw the next edge in the other face.

In this representation, faces always include the identifier of an arbitrary edge and a bit that indicates its orientation. Figure 10 shows an example of winged-edge data structure in which sign + and sign – denote the orientations of the edge.

Starting from the edge directly associated with a loop, all other edges may be retrieved by following the ncw and nccw pointers. For instance, in the case of Figure 9, the boundary of face 4 is extracted by starting from e5. The next edge is given by $ncw(e_5)=e_2$, similarly we get $ncw(e_2)=e_3$ as the next edge is $ncw(e_3)=e_5$ again, and we know that all edges have been extracted.

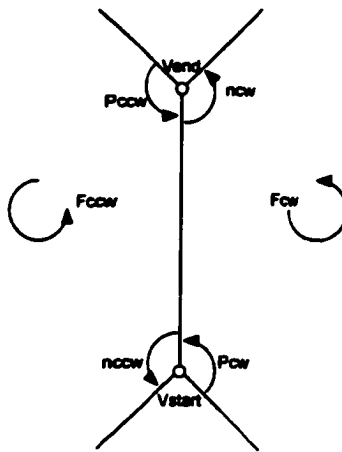


Figure 10: "Full" winged-edge data structure.

In the most general variation, edge nodes of the winged-edge data structure also carry out the identifiers *fcw* and *fccw* of its neighbor faces, and analogously to *ncw* and *nccw*, the identifiers *pcw*, *pccw* of the previous edge in these faces. Thus, the orientation indicator of previous winged-edge structure now becomes redundant. The resulting "full" winged-edge data structure is shown in Figure 11-12.

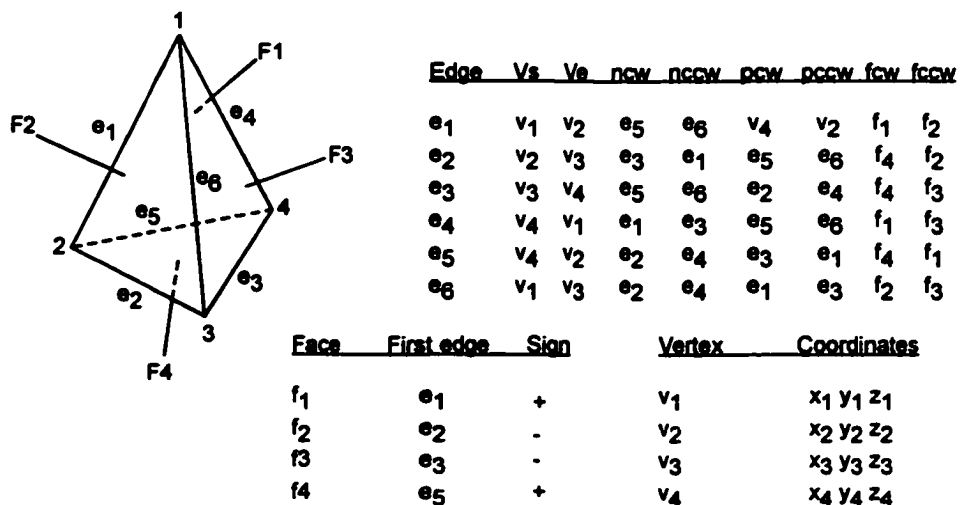


Figure 11: "Full" winged-edge data structure.

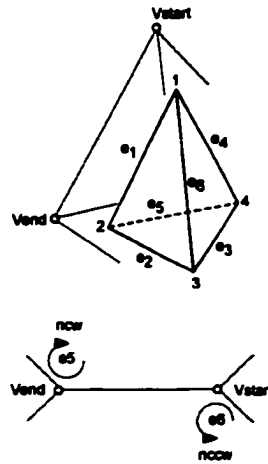


Figure 12: "Full" winged-edge data structure.

Face With Several Boundaries:

So far, it is assumed that each face is simple-connected, which means a face has just one boundary curve. However sometimes, faces that have several boundary curves are dealt with in Boundary representations. This problem can be solved in two ways. One approach is to model faces with non-simple boundaries by connecting the boundary segments with auxiliary edges.

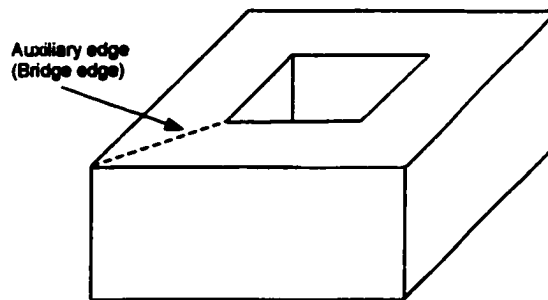


Figure 13: Auxiliary edge (bridge edge).

In the case of Figure 13, it is done adding the dashed edge. Unfortunately, these special edges should be avoided in line drawings. However, in the winged-edge data

structure, this is not necessary because auxiliary edges are considered to occur twice in the same face, and can be detected by comparing the fcw and fccw data items.

In the other approach, a separate loop node is added, and associated each face with a list of loops. References to faces would be replaced by references to the appropriate loops: for example, face identifiers fcw and fccw in edge nodes of the full winged-edge data structure would be replaced with the loop identifies lcw and lccw. All data structures can be readily generalized in the same way (Figure 14).

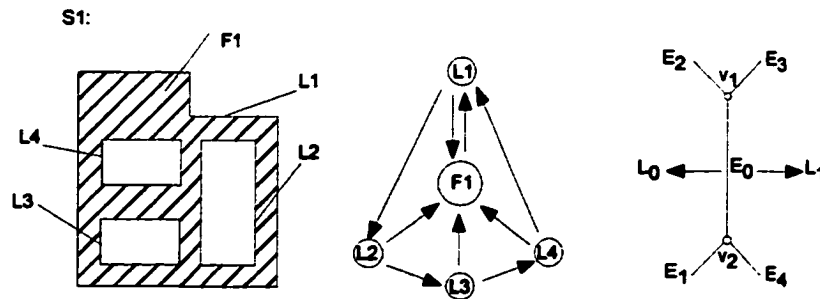


Figure 14: A face defined by winged-edge data structure.

Modified Winged Edge Representation:

The modified winged-edge or modified W-E structure is a slight but important variation on the W-E structure. Like the W-E structure, it represents the edge adjacency information as a single unified structure. In fact, it is identical to the W-E structure to the W-E structure except that it contains additional data. The difference is that each of the ncw and nccw pointers are accompanied by edge ncw-half and nccw-half fields which indicate exactly which side of the unified edge pointed at is intended. This reduces algorithm complexity which is particularly troublesome in curved surface domains.

2.3.1.4 HYBRID MODELS

None of these three major approaches to solid modeling is superior to the others in all respects. This initiates the use of multiple simultaneous representations in a serious modeling system. For example, constructive models are the most advanced of the major approaches to solid modeling. Because the directly available methods for the generation of graphical output or the creation of data for numerical algorithms are slow, a conversion to other representation has to be performed to do these tasks.

Boundary models are directly useful for graphical applications. Especially polyhedral models are useful for numerical application as well, if the limited modeling space captured by polyhedra can be tolerated. On the other hand, without the help of other representations and conversions, their creation is difficult, however. Decomposition models are superior to the others as sources of data for numerical algorithms, even when they are approximate. Their creation is complicated without conversion from the other representations. Clearly, a combination of the three major approaches to solid modeling would be superior to any of them alone. This has led designers of solid modeling systems to investigate possibilities of combining the basic approaches. A hybrid solid modeler is capable of supporting several co existing solid representations, and tries to pick the most suitable of them for each task. These representations are guaranteed to be consistent up to the level made possible by differences in the inherent modeling spaces. Consistency enforcement is done in conversion algorithms that can go from one representation to another.

Chapter 3

Non-manifold Representation Scheme

3.1 NON-MANIFOLD SOLID MODELING

Non-manifold geometric modeling, developed by Weiler[12,13,14], is a new modeling form. Essentially it removes all constraints traditionally associated with manifold solid modeling forms by including all of the capabilities of the previous three modeling forms in a unique representation. It simply extends the representational domain beyond that of the previous modeling forms.

Some volume based solid modeling systems allow some non-manifold conditions, but do not allow the full range of non-manifold conditions including boundary objects such as surfaces and wireframe edges. The full range of non-manifold systems allowed all such conditions is a new concept.

It is known that in a manifold solid representation, every point on a surface is two-dimensional. This means that every point has a neighborhood that is homeomorphic to two-dimensional space. It is topologically “flat” when the surface is examined closely in a small enough area around any given point (Figure 15).

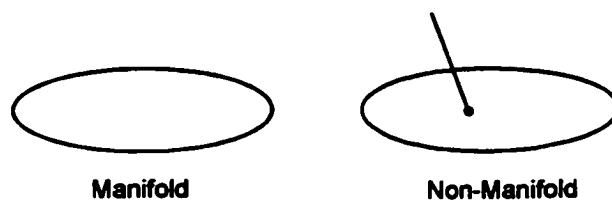


Figure 15: Manifold and non-manifold surfaces.

Up to now, manifold representations have been used in boundary-based solid modeling systems which, store topological adjacencies. Non-manifold is a geometric modeling in which topological situations are not two-manifold. This means that in non-manifold situations the surface area around a given point on a surface might not be “flat” in the sense that the neighborhood of the point need not be a simple two-dimension disk.

By using this property, topological conditions such as sphere touching upon another surface at a single point, more that two faces meeting along a common edge, and wire edges emanating from a point on a surface are possible to be represented easily. A non-manifold representation can form a general wire mesh with surfaces and enclose volumes embedded in space.

On the other hand, the Boolean operations are not closed under the manifold representations. The regularized set operations are designed to permit only volume filling results from the Boolean operations on manifold inputs. However, with some manifold inputs, the results of Boolean operations, regularized or not, yield non-manifolds and therefore they cannot be represented under manifold representations.

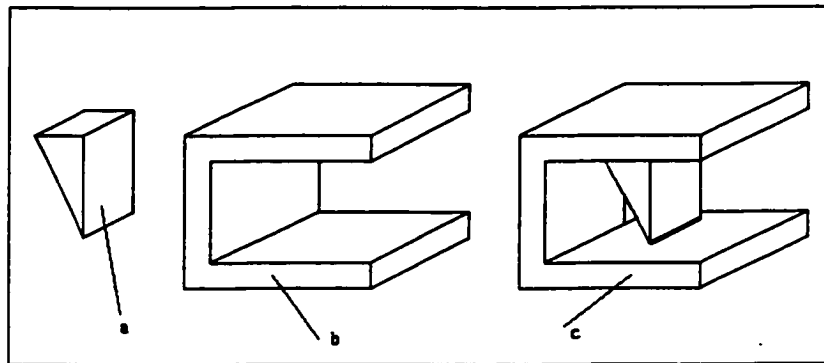


Figure 16: Non-manifold results from the Boolean operation of manifold objects.

For example, a small part reaching out from the main volume of an object and then touching back on the surface of the same object along a line cannot be directly represented with manifold, and creation of such an object even with regularized set operations cannot result a valid manifold objects (Figure 16). Non-manifold representations don't deal with these singularities by representing non-manifold situations directly instead of restricting the domain of the output.

In general, non-manifold representations can represent a larger variety of objects, and can support a wider variety of applications than manifold representations, but they use a larger size data structure. Boolean operation implementations must also detect and deal with non-manifold results, which are uniformly and cleanly represented and manipulated. Boolean operations with representation of non-manifold results are only choice if the accurate results are desired.

In non-manifold representations, interior volume structures in an object and the relationships between them, such as in composite objects and finite element meshes can be preserved. Perhaps most importantly, generalized non-manifold representations can represent wireframe, surface, and solid modeling representations simultaneously in a single uniform format.

3.1.1 NON-MANIFOLD TOPOLOGY

Since topological element adjacency relationships concern the relationships between individual topological elements, adjacency relationships between elements have to be described more clearly.

In his work, Weiler[12] describes at least seven distinct element types, including six basic topological element types involved in a non-manifold topology representation. It can be seen that lower dimensional elements are used as boundaries of higher dimensional elements in a hierarchical fashion (Figures 17, and 18). The topological elements are described in Weiler[12] as following:

A “*Model*” is a single three-dimensional topological modeling space, containing one or more different (though perhaps adjacent) region in space. A model is not a topological element as others, but acts as a common base for all topological elements contained in a geometric model. It allows the naming and manipulation of multiple models by a geometric modeling system.

A “*Region*” is a volume of space. It must be at least one in a model. Only one region in a model may be infinite; all others must be finite, and if there are more than one region in a model, all regions carry out one boundary. For instance, a single solid would consist of two regions in a model, one for the inside of the object, and one for the outside, which has an infinite extent.

A “*Shell*” is an oriented boundary surface of a region. It is also called “facet”. A region may have no shell only where all space exists as a single region. It may happen in the initial state where no modeling has been done, or after all components of a model have been deleted. A shell may include a connected set of faces. These faces form a closed volume or may be on open set of adjacent faces. They may contain a wireframe, or a combination of these, or even a single point.

A “*Face*” is a bounded part of a shell. It is orientable, but non-oriented. Two region shells may use different sides of the same face. Thus only the use of a face by a

shell is oriented. A face consists of surfaces it covers. But it does not include its boundaries.

A “*Loop*” forms a connected boundary of a single face. A face may consist of one or more loops. Loops normally are ordered in alternating sequence of edges and vertices in a complete circuit, but they may also consist of only a single vertex. Loops are also orientable but not oriented. They bound a face in which two different shells may use them. Therefore, the use of a loop is oriented only.

An “*Edge*” is a part of a loop boundary between two vertices. Topologically, an edge is a boundary curve segment. It may be part of a loop boundary for one or more faces. Every edge has at most one vertex at each end. For a circular edge, there is the same vertex at each end. Although an edge can be oriented, it is not oriented. The use of an edge is oriented only.

A “*Vertex*” is simply a point in space. No two vertices may exist at the same geometric location. Single vertices may also be part of boundaries of faces, and shells.

Although not directly part of the adjacency relationship as described, at least four additional types of topological elements adjacency “uses” associated with the face, loop, edge, and vertex elements may also be included, and in some representations they may be directly represented.

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

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

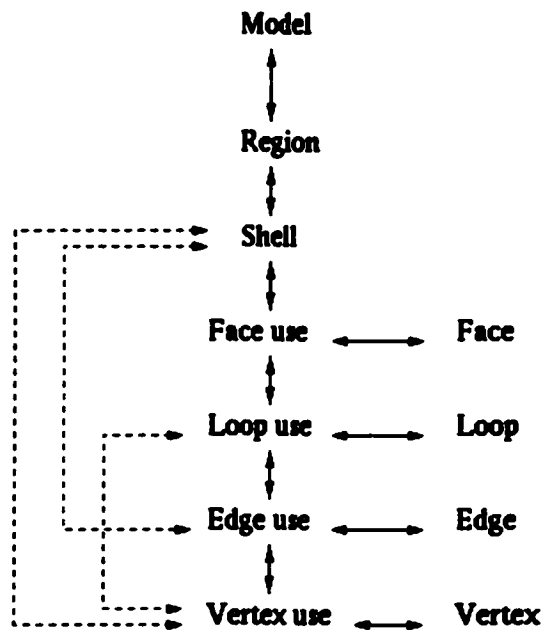


Figure 17: Non-manifold topology.

An “*Edge-use*” is an oriented bounding curve segment on a loop-use of a face-use and represents the use of an edge by that loop-use, or if a wireframe edge, by end-point vertices. Oriented is specified with respect to edge geometry.

There may be many uses of a single edge in a model, but there will always be an even number of edge-use (since each use by each face side). A wireframe edge produced two edge-uses, one for each end of the edge.

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

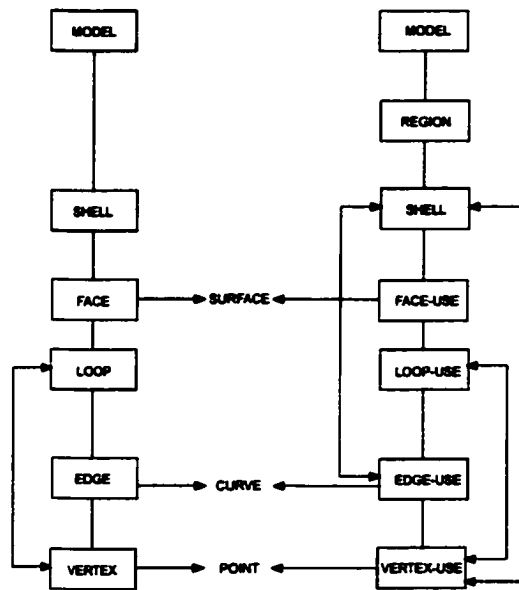


Figure 18: Manifold and non-manifold topological structure relationships.

3.1.2 DESCRIPTION OF RADIAL EDGE DATA STRUCTURE

According to Weiler, several decisions were made during design of the radial edge non-manifold data structure:

1. The top-down hierarchical relationships of the topological elements, from higher to lower dimensional elements, and the bottom-up hierarchical relationships of the topological elements, from lower to higher dimensional elements are represented in the data structure as shown in Figure 17. The “up pointer” and “down pointer” used in later data structure descriptions are simply for these relative positions in this hierarchy.

2. The $V\{E\}$ adjacency relationship containing the unordered list of edges connected to a vertex is represented to capture the adjacencies of different volumes touching at a single non-manifold point. Those adjacency relationships are used to capture the edge adjacencies in a wireframe. Because the vertex is the only structure common between such adjacent structures in these situations, the vertex and vertex-use structures are used to store such adjacency information.
3. The $E\{L\}$ adjacency relationship contains the ordered list of loops surrounding an edge. Because the same volume may be adjacent to an edge from several directions at once, the radial face ordering around an edge is necessary to allow the adjacencies of the volumes at the edge. This key feature of the Radial Edge Data Structure becomes a source to the name “Radial Edge”
4. The topology represents the adjacency uses of the face, loop, edge, and vertex elements directly. Thus, representing each face with two face-use (face side) structures and each with an edge-use structure for each use by each face-use form some of the other key ideas of the radial edge structure.
5. Wireframe edges are represented by two edge-uses, one for each end of the edge. The vertex structures maintain the connectivity between the other edges.

3.1.3 DATA STRUCTURES

Model: The model simply forms the top of the hierarchical for the non-manifold structures.

```
Model:
struct model {
    long          index;
    long          maxindex;
    struct topology_list header_reg;
    struct attribute att_m;
};
```

The list “header-reg” keeps track of all regions in the model space. The model attributes structure is used to store information for entire model. The long integer “index” contains the number of the model. Max index is the integer for the number of structure, which have been allocated in the model. The double-linked list’s structure used is as following:

```
Struct topology_list {
    long          index;
    struct topology_list *next;
    struct topology_list *prev;
};
```

Region: Regions are used to store all shells within the model space. The topology-list is included in the model’s list of regions through the liked list node at the head of the region structure.

```
Regions:
struct regions {
    long          index;
    struct model  *mp;
    struct topology_list tp;
    struct topology_list rshell;
    struct att_reg *rap;
};
```

The region structure consists of a linked list “*tp*”, a pointer to the parent model “*mp*” and a list of shells, which make up the region “*rshell*”. The region attributes structure is used to store a bounding box for the entire region.

Shell: The shell represents a set of collected items. The member “*tp*” is a linked list node to permit all shells to be collected into a list.

```
Shells:
struct shell {
struct topology_list    tp;
struct region           *sr;
struct att_shell        *sa;
struct topology_list    fu;
struct topology_list    lu;
struct topology_list    eu;
struct vertexuse        *vu;
long                    index;
};
```

The member “*sr*” is a pointer to the parent region for the shell. The attribute for the shell is referred to through the pointer “*sa*”. The lists *fu*, *lu* and *eu* keep track of the face-uses, loop-uses, and edge-uses respectively. The “*vu*” points to a single vertex-use if the shell contains a single vertex.

Face: The face represents a planar or curvilinear surface. The pointer “*fu*” is a pointer to one of the two face-uses of the face, the other being reached through that face-use’s “*fumate*” pointer. The pointer “*fa*” references the attributes including geometry.

```
Face:
struct face {
struct faceuse        *fu;
struct att_face        *fa;
long                    index;
};
```

Loop: A loop defines a boundary. A loop may contain either a single vertex, or a series of one or more edges. The most of this information is located in the use structure. The “*lu*” links the loop structure to one of the loop-uses of the loop. After that, the other use of the loop can be found through the “*loopmate*” structure pointer.

```
Loop:
    struct loop {
        struct loopuse          *lu;
        struct att_loop         *la;
        long                    index;
    };

```

Edge: The edge represents a curve, which contains two vertices. The member “*eu*” points to one of the uses of the edge. The member “*att-edge*” points to the edge attributes including geometry. It can be seen that the edge structure does not refer the end points of the edge. The end points are linked through the edge-use structure.

```
Edge:
    struct edge{
        struct edgeuse          *eu;
        struct att_edge         *ea;
        long                    index;
    };

```

Vertex: The vertex is the basic element in the topology. A vertex represents a single point in the model. It also links the topological model with the geometrical data.

The member “*vu*” forms the head of a double-linked list of all the uses of this vertex. The “*att_vertex*” is a pointer to the attributes. The member “*vertex_geo*” links the

vertex to its geometry. The index element keeps the structure index for the particular vertex instance.

Vertex:

```
struct vertex {
    struct topology_list    vu;
    struct vertex_geo       va;
    struct att_vertex       *vt;
    long                    index;
};
```

```
struct vertex_geo {
    long    index;
    double  coord;
};
```

Face-use: A face has two face-uses associated with it. The face-uses form the sides of a face. Therefore, they are oriented.

Face-use:

```
struct faceuse {
    struct topology_list    tp;
    struct shell            *sp;
    struct faceuse          *mate;
    struct face             *fp;
    struct topology_list    lu;
    struct att_faceuse      fua;
    int                     side;
    long                    index;
};
```

The member “tp” in the face-use structure is a linked list node. The shell keeps a list of all the face-uses. The pointer “sp” points to the parent shell of the face-use. The other use of the same face is linked the face-use pointer “mate”. The “side” member indicates which side of the face is represented by the face-use. The face-use is linked to a particular face through the “fp” pointer. The member “lu” keeps track of all loops in the face. “fua” is for the storage of attributes.

Loop-use: The loops-use structure is used to handle most of the loop details. The first member of the structure is a linked list node, so that faces and shells can be related to collections of loop-uses. The union “*up*” handles for all possible parents of the loop-use. The “*mate*” member points to the other use of the same loop as this loop-use. The “*zor*” member defines whether this loop encloses space, or excludes space within a face.

The loop-use is linked to a particular loop through the pointer “*lu*”. The structure member “*dh*” is a list head. It is used to access the component, elements which form the loop.

```

Loop-use:
    struct loopuse {
        struct topology_list    tp;
        union {
            struct faceuse      *fu;
            struct shell        *sp;
        } up;
        struct loopuse          *mate;
        int                     zor;
        struct loop              *lp;
        struct topology_list     dh;
        long                     index;
    };

```

When the loop is consists of one or more edges, this list will include a series of edge-uses. If the loop represents a single vertex, the only item in the list will be the vertex-use associated with the loop-use. When the loop-use is formed on edge-uses, the loop-use will be arranged to form a close loop. The mates to these edges will produce a circuit for the loop-use mate (Figure 19).

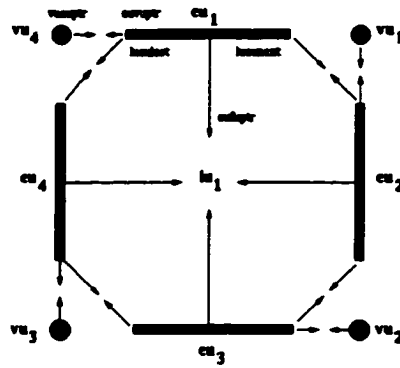


Figure 19: Topological relationship in loop-use [Weiler].

Edge-use: Each edge-use represents one vertex via the vertex-use structure in which the pointer “ vu ” handles it. On the other hand, the other edge-use is linked through the first edge-use’s mate edge-use (Figure 20).

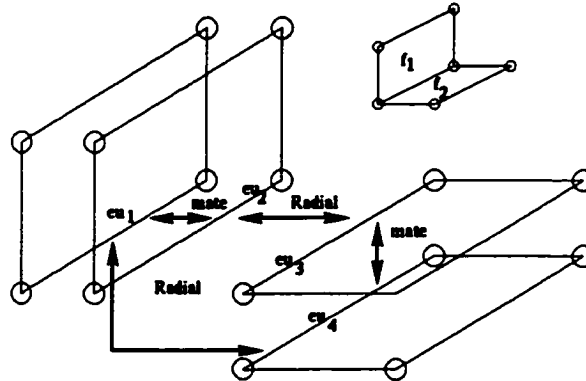


Figure 20: Topological relationship in edge-use [Weiler].

The first member in the structure is a linked list node. Therefore, loops and shells may handle collections of edge-uses. The union “ up ” keeps track of all possible parent of the edge-use. The “ $mate$ ” and “ $radial$ ” pointers form a linked list of the “radial uses” of an edge. The edge-use is associated with a particular face through the “ ep ” pointer.

Edge-use:

```

struct edgeuse {
  struct topology_list    tp;
  union {
    struct loopuse        *lu;
    struct shell           *sp;
  } up;
  struct att_edgeuse      eua;
  struct edgeuse          *mate;
  struct edgeuse          *radial;
  struct edge             *ep;
  struct vertexuse        *vu;
  long                    index;
};

```

Vertex-use: The “*tp*” element of the vertex-use structure is a list of all uses of the vertex. A vertex-use may be needed by any of the higher level object: shell, loop-use, or edge-use. The union “*up*” contains a pointer to each of these three structure types. The element “*vp*” is a pointer to the vertex being used by this vertex-use. The vertex-use may have an associated vertex-use attribute structure. This structure is referenced through the “*vua*” pointer (Figure 21).

Vertex-use:

```

struct vertexuse {
  struct topology_list    tp;
  union {
    struct loopuse        *lu;
    struct shell           *sp;
    struct edgeuse        *eu;
  } up;
  struct att_vertexuse    vua;
  struct vertex           *vp;
  long                    index;
};

```

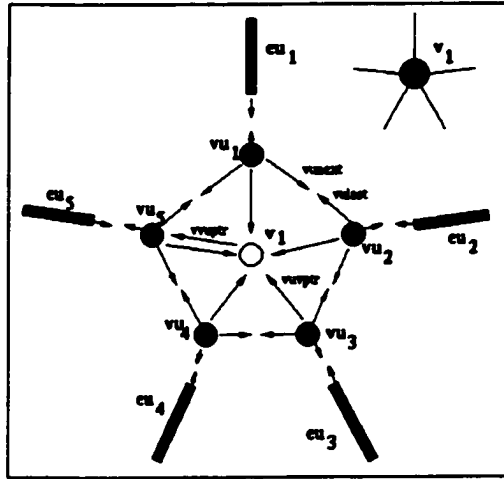


Figure 21: Topological relationship in vertex-use [Weiler].

Chapter 4

Linked List Structures

4.1 LINKED STRUCTURE

“Linked list” is a general classification for several methods of storing information in which the data stored is connected or linked together. This linking takes place with the use of pointers.

There are three types of linked lists that are commonly used:

1. Single-Linked List
2. Double-Linked List
3. Circular-Linked List

The specific type of linked list is determined by how the data groups are connected. A single-linked list has a single connection between each group of information. A double-linked list has two connections between each data group. The circular list is another form of linked list that provides additional flexibility in traversing elements. A circular list may be single-linked or double-linked, but its distinguishing feature is that it has no tail.

4.1.1 SINGLE-LINKED LISTS

Single-linked lists, and linked lists in general, are used for several reasons. The main reason is speed. When working with sorted disk files, it can be time consuming to add in a new element.

If a new element is added to the beginning of a disk file, each of the following elements must be shifted. With a linked list, it is worked in memory. Because memory is much more faster than disk access, you can manipulate the data in the list faster. In addition, sorted disk files are generally stored in sorted order with a linked list. Pointers are used to keep the element sorted. If a new element is to be added, it can be placed anywhere. Only the links need to be adjusted. This again increases the speed.

Link lists involve using a structure that contains a pointer member. The pointer member, however, is special. In the structure:

```
Struct element {  
    int      data;  
    struct element *next;  
};
```

The pointer contains the address of another structure. In fact, it is a pointer to a structure of its own type. The pointer in the structure, called “next” in this case, points to another element structure. This means that each structure or link can point to another structure at the same time. Figure 22 illustrates a single link using the preceding element structure and a linked list of such structures.

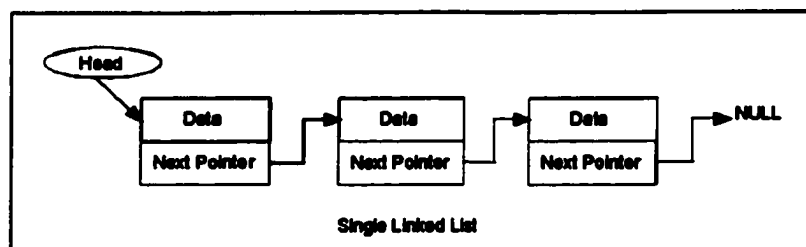


Figure 22: Single link list.

Notice that in Figure 22, each element points to the next. The last element does not point to anything. To show that the last element does not point to an additional link, the pointer is assigned the value of NULL.

To prevent the loss of links, one must set up an additional pointer. The pointer is commonly referred to as a head pointer. The head pointer always points to the first element in the link. If one knows where the first pointer is, he can access its pointer to the second element.

The second element's pointer can then be accessed to get the third. This can continue until you reach a NULL pointer, which would signify the end of the list. It is possible that the head pointer could be NULL, which would mean the list is empty.

Using a linked list:

Linked list are used similar to disk files. Elements or links can be added, deleted, or modified. Modifying on element presents no real challenge; however, adding and deleting on element may.

As stated earlier, elements in a list are connected with pointer. When a new element is added, the pointer must be adjusted. Where the new element is added affects how pointers are modified.

Elements can be added to the beginning, middle or end of a linked list. In addition, if the element is the first to be added to the list, the process is a little different.

Adding the first link:

If the first element is added to a linked list, the head pointer is set to NULL. The head pointer should be changed to point to the new element. In addition, because the

element being added is the only element, the “next” pointer should be set to NULL.

Figure 23 illustrates the final result.

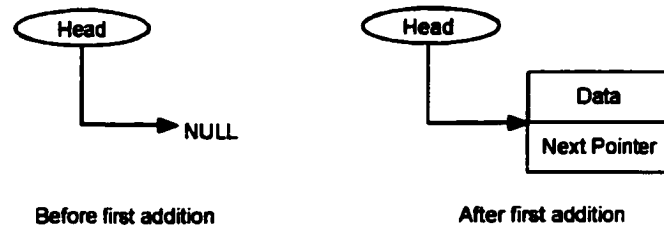


Figure 23: Adding the first list.

The following code fragment includes the element structure defined previously along with two type definitions.

```
Struct element {
    int          data;
    struct element *next;
};

typedef struct element ELEMENT;
typedef ELEMENT *Link;
```

The element structure will be used by means of the two type definitions. When an instance of element needs to be declared, ELEMENT will be used. ELEMENT is a defined data type of the element structure. The second defined data type, Link, is a pointer to an element structure. The following code fragment illustrates adding an initial element to a linked list.

```

Link first = NULL;
Link new = NULL;

new = (Link) malloc ( sizeof(ELEMENT));
scanf("%d", & (next->data));

if( first = NULL)
{
    new->next = NULL;
    first = new;
}

```

Adding to the beginning:

Adding an element to the beginning of a linked list is similar to adding an element to a new list. When an element is added to the beginning, two steps are involved. First, the “Next” pointer of the new element must be set to the original first element of the list.

This can be done setting the new element’s “Next” pointer equal to the head pointer. Once this is done, the head pointer must be reset to point to the new element that now begins the list (Figure 24).

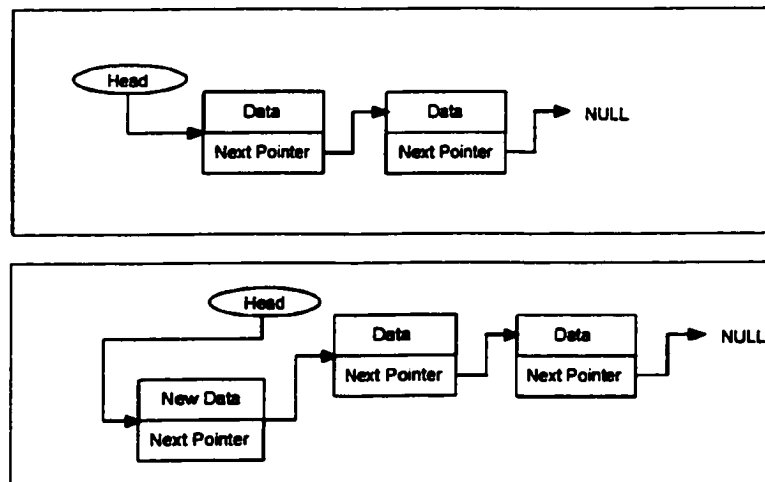


Figure 24: Adding to the beginning of a linked list.

Using the element structure, the following code fragment illustrates the process of adding an element to the beginning of a linked list.

```
{  
    new->next = first;  
    first = new;  
};
```

Adding to the middle:

Adding an element to the middle of a list is a little more complicated, yet this process is still relatively easy. Once the location of the new element is determined, you will adjust the pointers on several elements. Figure 25 illustrates the process of adding an element to the middle of a linked list.

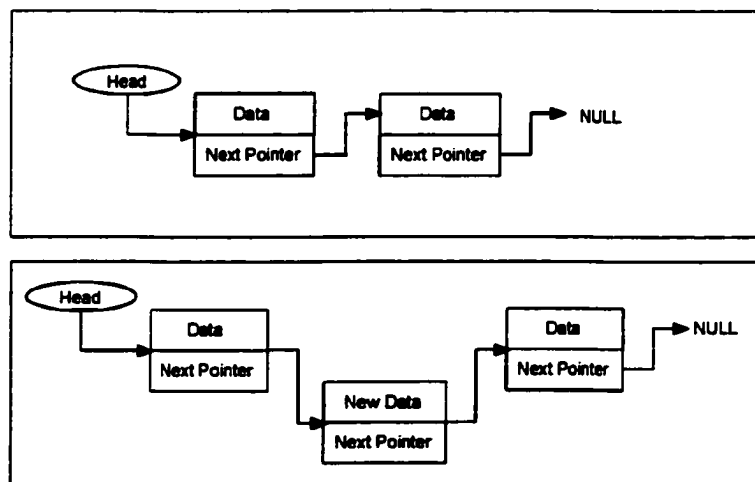


Figure 25: Adding to the middle of a linked list.

It can be seen from Figure 25 that when a new element is added to the middle, two pointers have to be adjusted. The “next” pointer of the previous element has to be

adjusted to point to the new element. In addition, the “Next” of the new element needs to be set to the original value of the “Next” pointer in the previous element. Once these pointers are readjusted, the element is part of the list. The following code fragment illustrates this addition:

```
Insert_link ( Link prev_link , Link new_link)
{
    new_link->next = prev_link->next;
    prev_link->next = new_link;
};
```

4.1.2 DOUBLE-LINKED LIST

Single-linked lists enable the user to move between the elements starting at the top and working toward the bottom (or end). Sometimes it is advantageous to be able to work back toward the top. one can traverse a list from both ends by adding a second set of links (pointers) between the elements. The double set of pointers causes the list to be double-linked. Figure 26 illustrates a double-linked list.

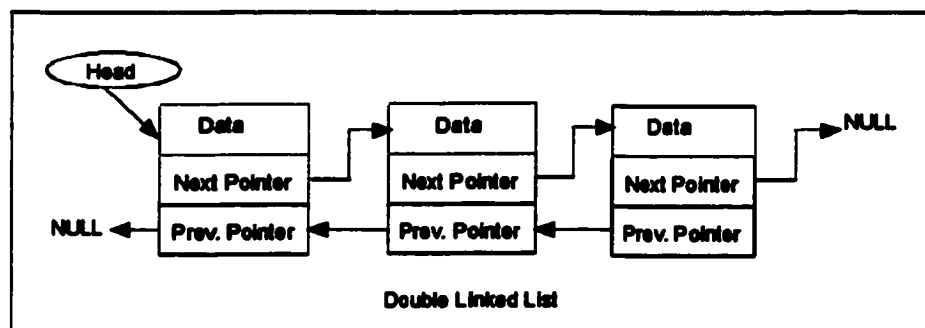


Figure 26: Double-linked list.

In figure 26, it can be noticed that all components of a single-linked list are present. In addition, a head pointer and a tail pointer are both present. As with a single-linked list, the head pointer will always point at the top or first element of the list. The tail pointer will always point to the last. If there are not elements in the list, the head and tail pointers will both be NULL. If there is only one element, then the two pointers will be equal to the first and only element.

A structure for an element in a double-linked list is different from that of a single-linked list in that it contains an additional pointer. The format of a structure for a double-linked list would be similar to the following:

```
Struct      element {  
  
    < data >  
    struct element      *next;  
    struct element *previous;  
  
};
```

4.1.3 CIRCULAR-LINK LIST

The circular list is another form of linked list that provides additional flexibility in traversing elements. A circular list may be single-linked or double-linked, but its distinguishing feature is that it has no tail.

In a circular list, the next pointer of the last element points back to its first element rather than to NULL. In the case of the first element is set to point the last element as well. Whether dealing with a single or double-linked list, it is not needed to worry about reaching an element from which can be traversed no further as moved from

element to element. Instead, the traversal simply continues back to the first element. Traversing a list this manner produces a circular pattern hence its name (Figure 27).

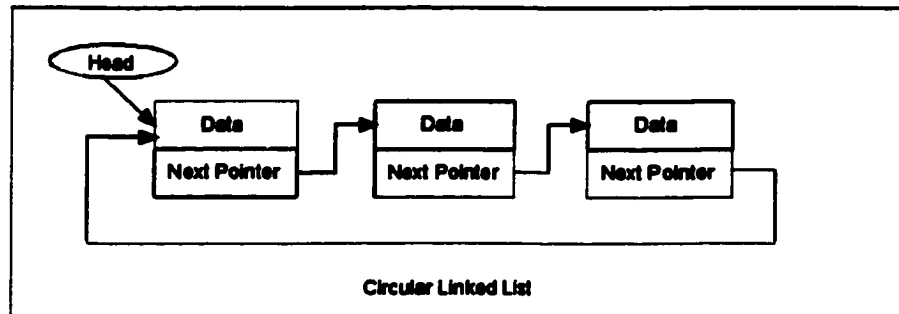


Figure 27: Circular-linked list.

Chapter 5

Euler Operators in Topologies

5.1. EULER OPERATORS

The Euler operators are set of operations. They simply manipulate manifold or non-manifold topology representations in a low level incrementally, in which the topology is constructed edge by edge.

Euler operators are used as an intermediate procedure in some modeling systems. The advantage of using the operators comes with reducing the operations implemented and the data structures used to represent the surface topology. Thus, in principle, the representation of the modeling system could be changed with small impact on the modeling system's implementation. On the other hand, Euler operators result in topological consistency throughout the modeling process.

5.1.1. EULER OPERATORS IN MANIFOLD TOPOLOGY

The basic Euler operators such as MSFLV, ME, GLUE, and KE can sufficiently be used to create any topology. On the other hand, other operators are included for convenience and flexibility to the surface construction process.

Baumgard [32] first introduced the name of Euler operators. The "M" stands for "make" or "create" and "K" stands for "kill" or "delete". S, F, F, E, and V, which stand for shell, face, loop, edge, and vertex, respectively, follow first letters signifying the types of topological elements creates or deleted.

The Euler operators create new components in a topology. These incremental changes always balance the Euler-Poincare equation. Thus if Euler operations are treated as non-interruptible operators, the data structures are always constrained to represent a valid manifold topology at every stage.

Recall the Euler-Poincare formula;

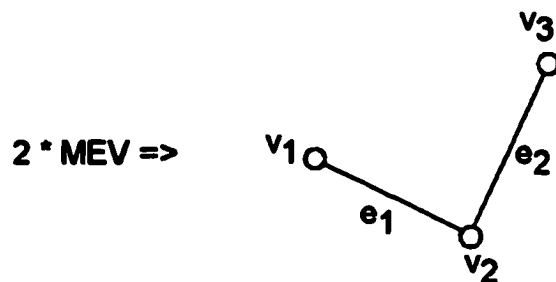
$$V - e + f - h = 2(s - p)$$

where v, e, f, s, h, and p denote the numbers of vertices, edges, faces, solid, holes, and passage in a model. We can clarify the use of Euler operators by means of a single example, the construction of a triangle prism with one triangle prism hole. The construction is illustrated in the following figures in terms of plane model and 3-D space models.

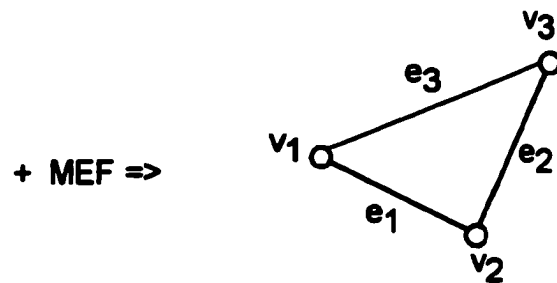
The construction starts by creating the basic model by means of a MVFS (make vertex, face, and solid)

MVFS => 

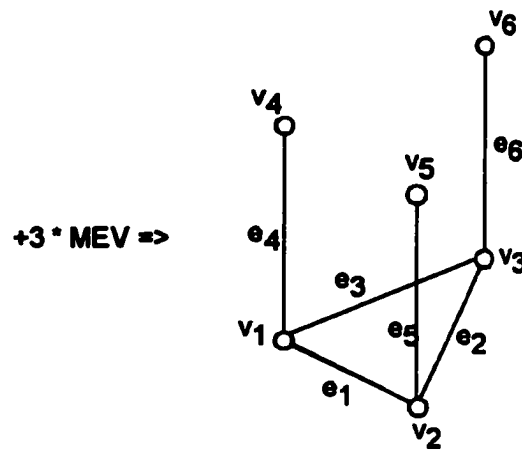
By applying MEV two times, two edges and vertices are added so as to create;



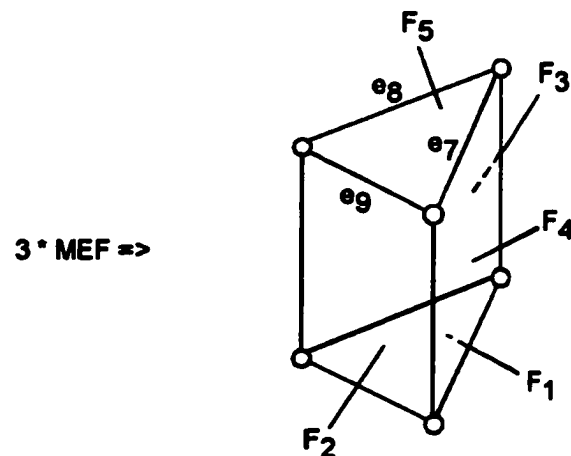
By joining the end vertices of the edge with a MEF, the face below is created.



Three MEV's lead us to the model below that has side edges of the emerging prism.

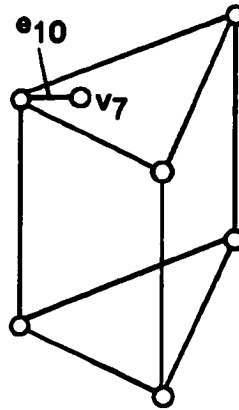


Their tips are joined with 3 MEF's to form the side faces.



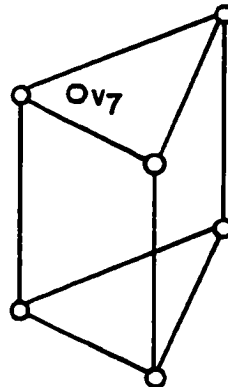
We now have formed the model of a simple prism. To create the hole, we first need to form an interior face on the top of the box. We do that by adding an edge on the top face with a MEV,

+ MEV =>



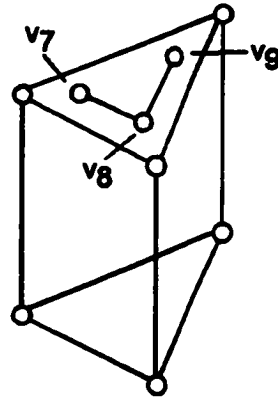
and removing the edge with KEMS (Kill Edge, Make Shell) yields;

+ KEMS =>



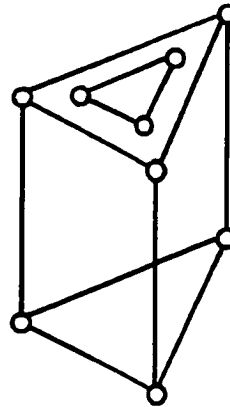
This gives us an empty loop with one vertex only inside the top face. After two additional edges drawn from that vertex with MEV's;

+ 2 * MEV =>



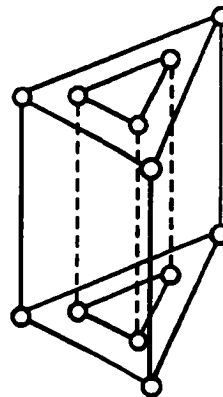
The interior face can be formed with MEF;

+ MEV =>



Now, the side faces of the hole can be formed with MEV's and MEF similarly.

+ 3 * MEV + 3 * MEF =>



We now have formed all edges and vertices of the desired result. To finish the description ,the bottom face of the emerging hole is “subtracted” from the bottom face of the prism by means of a KFMSH (Kill face , make shell, hole).

6.1 EULER OPERATORS IN NON-MANIFOLD TOPOLOGY

The non-manifold topology operators can be classified by two characteristics. The first, some of the operators described by Weiler[12] handle either manifold or non-manifold situations. Different kinds of specifications are required to unambiguously construct a model. Manifold version of the operators also provides compatibility with existing higher level geometric modeling functions, which originally designed for manifold situations using the Euler operators. However, some additional information is required because of the non-manifold environment. On the other hand, when no difference in specification is required, general operators handle both manifold and non-manifold situations.

Second, the operators are functionally “constructive” or “ deconstructive” in terms of the *number* elements existing in the model after their use. According to Weiler, several other operators, which can be designed to promote efficiency or convenience for given applications, one interesting issue is to determine a minimal set of operators, which can define any model in the representation. There can be many such minimal sets of operators, since many different operators can be designed, which have overlapping functionality.

These operators incrementally construct a model toward a given specification in different sized steps. Eventually they yield the same result. The set of operators shown in Figure 28 is one of such minimal set.

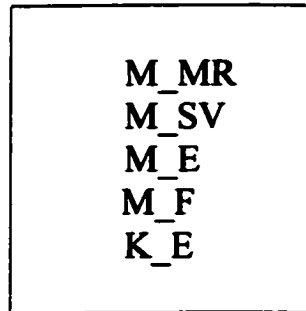


Figure 28: Minimal set of Euler operators.

The minimal set is ideal for simplicity of the functionality of the operators while any possible model may be constructed with them. However, manipulation of the existing model without additional operators is not convenient.

Weiler suggests the following algorithm to be used for any model with above set of operators from some existing specification of the final boundary model.

1. Create the model and region by using M_MR.
2. Create a new shell in the region for every vertex in the model by using M_SV.
3. Connect appropriate vertices for every edge to be in the model by using M_E.
4. Connect loops together, which share the same face including “hole” loops in the faces and vertices, which will be a single loop vertex in the model. Thus all edges and vertices sharing a common face are connected. It is required to

perform $n-1$ M_E operation for a face with n -loops in the model. These additional edges are called “loop edges”.

5. Create faces in the model excepting face with a single vertex by using M_F, being sure to appropriately include loop edges into the boundary descriptions.
6. Create a self loop edge for every vertex by using M_E, which will be the sole boundary of a face in the model, making sure they will be created in appropriate regions. Call these additional edges “face edges”
7. Do two M_F's for each self loop edge in the face edge list, creating the new regions.
8. Eliminate each loop edge and face edge by using K_E.

For efficiency and convenience, a practical modeling system employs more than this minimal set. By adding more operators to the minimal set, inconvenient restrictions are removed on the order of operations necessary to construct and modify objects. According to Weiler[12], the first M_EV is added to the minimal set to complete basic wireframe construction capabilities. Then M_RSFL, MM_EV, and MM_E are added for convenience in manifold modeling situations, followed by others for more specialized situations.

Example of use of the non-manifold operators:

In order to show the simple application of non-manifold operators, the following example is given. The geometry consists of one cube, one laminar face, one wire edge, and two vertices placed in the model as follows (Figure 29):

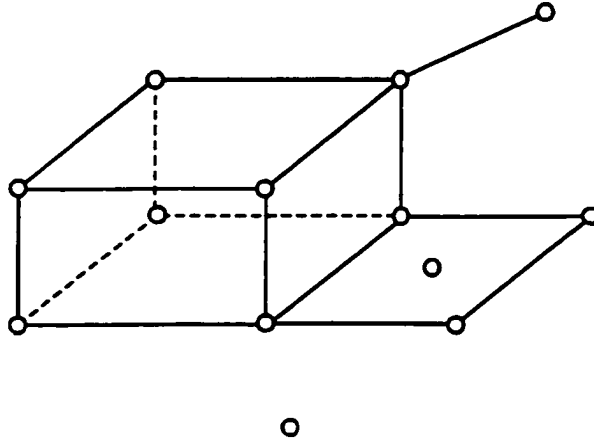
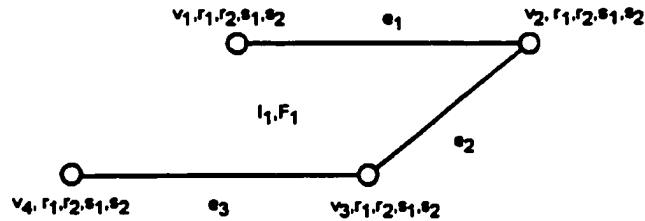


Figure 29: A non-manifold model.

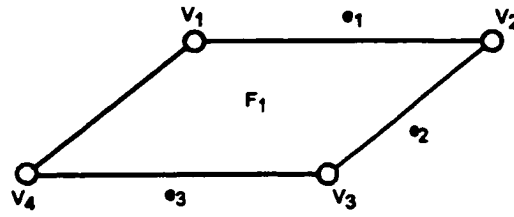
In order to construct the above model, we first create the initial model by using **M_MR** (Make Region) operator. Then a new shell in the region is created for every vertex in the model by using **M_SR** (Make Shell, Vertex).



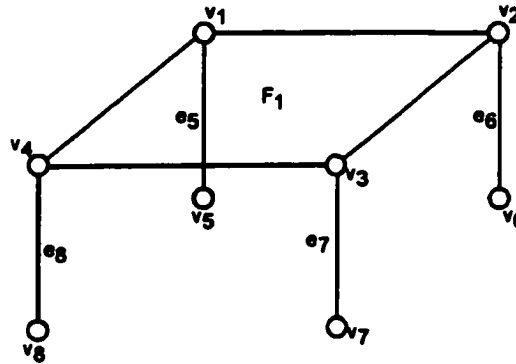
After creating the first vertex in the model, we apply **M_RSFL** operator to generate the first loop, and face. We execute the **MM_EV** operator three times to create 3 more vertices and their corresponding edges:



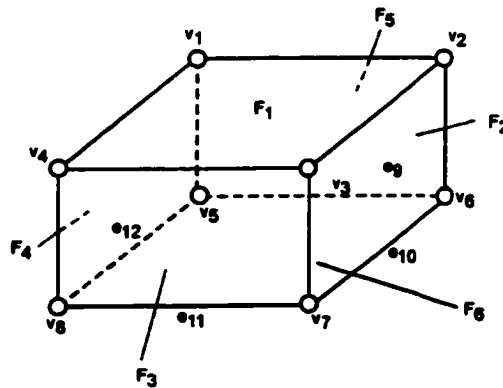
Connecting the appropriate vertices by using **MM_E** (Make Edge) yields;



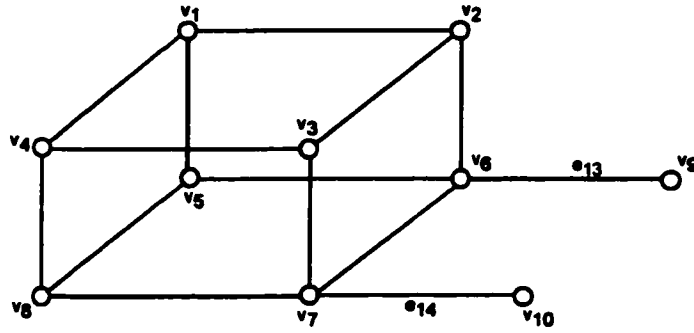
Then, we continue introducing the new vertices and corresponding edges by using MM_EV four times;



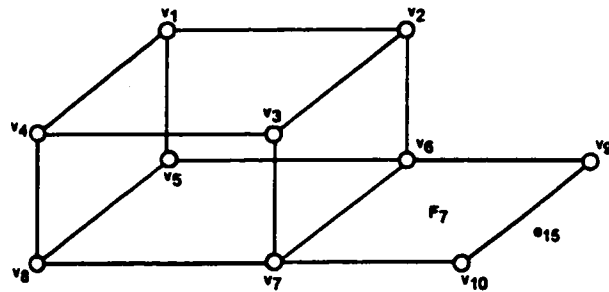
For the next step of creating the model, MM_E operator is used four times to form the remaining faces;



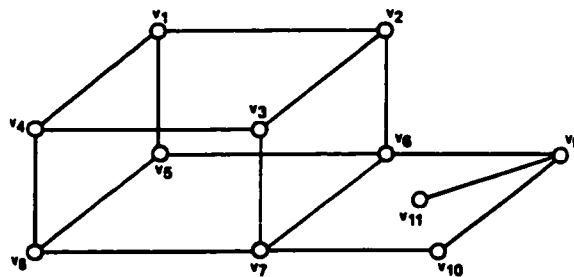
To create the laminar face; we apply M_EV two times to get:



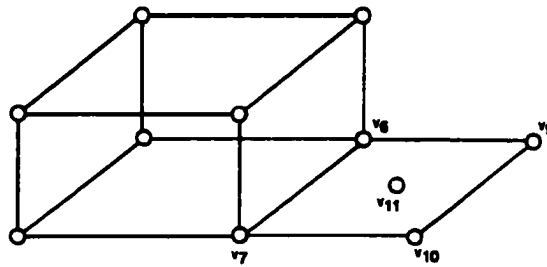
After that, we put an edge, and a face by using M_E , and M_F operators:



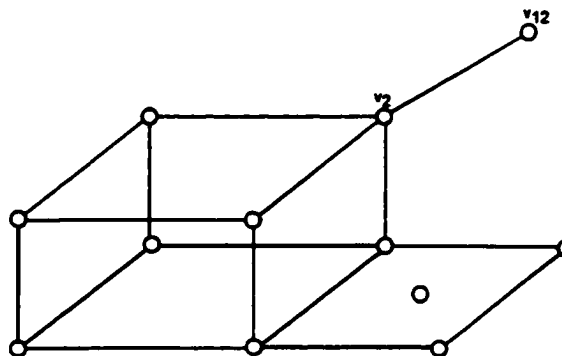
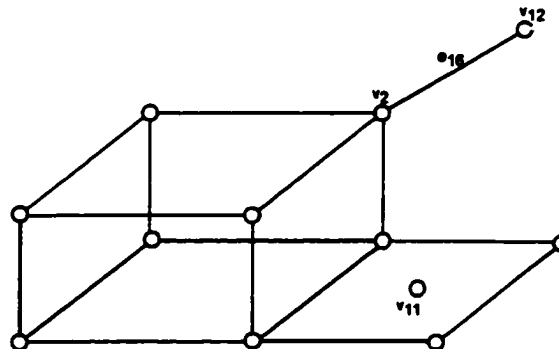
Now, we can introduce the single vertex loop. To do that, we have to create a temporary edge first:



Then, we remove the temporary edge by using M_K (Non-manifold Kill Edge), we end up with the single vertex loop inside the face7:



We have completed the majority of the model. We are now to perform two more Euler operations. One is for creating the wire-edge, and the other is for the single vertex shell. The wire-edge is created by applying M_EV operator. On the other hand, the single vertex shell is generated by M_SV (Make Shell, Vertex) operation as shown in following figures respectively:



v13

Chapter 6

Development of Non-manifold Boolean Operations

6.1. BOOLEAN OPERATIONS IN NON-MANIFOLD MODELS

Boolean operations are algorithms for determining the union, intersection, or set difference of two solids. They are used in Boundary Representations and dual (or hybrid) representation models. They can also be used to convert solids represented by CSG tree to an equivalent B-Rep. In modeling with Boolean operations, a new solid model is gradually determined by two intersecting solid models. Boolean operations are very useful, and they are used in most modeling systems. However, they have serious problems such as;

1. It is time consuming to reshape a boundary model defined by Boolean operations.
2. The realization of cancel operations is difficult.

In CSG modeling, models are easily reshaped when modifying a CSG tree, however it is subsequently time consuming to obtain boundary data. In conventional modeling, the original primitive objects are not left in the resultant shape. When two primitive objects are unified, for example, topological elements inside the unified object are deleted. This is the main reason why reshaping is time consuming and cancel operation is difficult.

Thus, the Boolean operations can be manipulated by means of non-manifold geometric modeling techniques. It is possible to maintain the shape of the original primitive objects in the topological structure. The following algorithm, which is used in conventional solid modeler, consists of 4 steps:

1. *Intersection*: cross intersection between entities of each primitive.
2. *Classification*: determine in/out/on classification of original geometry and intersection geometry relative to each primitive.
3. *Selection*: use classification results to solve the Boolean operation: keep geometry satisfying the operation and throw away the rest.
4. *Topological Construction*: establish necessary adjacency relationship between the kept pieces of the geometry.

This algorithm is limited to solids and do not include wireframe and surface information. Thus, the algorithms cannot represent non-manifold results. Manifold nature of their topological framework simply does not allow them.

This problem is solved in non-manifold modeling by dividing a Boolean operation into four operations:

1. Intersection operation
2. A merging operation: which is independent of the operation sequence
3. An extracting operation: which is dependent on the operation sequence.
4. Selection operation.

6.1.1 INTERSECTION OPERATION

There are six types of intersection pairs of topological elements for actual intersection computation. The intersection process has to be from lower dimensional elements to higher dimensional elements. So the primitive intersection are performed in the following order:

- a. Vertex-Vertex
- b. Vertex-Edge
- c. Vertex-Face
- d. Edge-Edge
- e. Edge-Face
- f. Face-Face

intersections.

In order to visualize the intersection process on a model, we will consider two objects for the process as shown in figure 30. For every Boolean operation, the result on those objects will be given with the text.

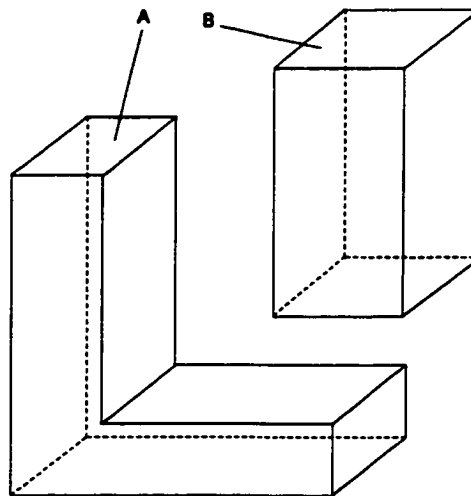


Figure 30: Objects for Boolean operations.

A. Vertex-Vertex Intersections

If a distance between two vertices is less than the specified tolerance value, vertex intersections between two objects are checked for each pair of vertices. Then those two vertices are noted as coincident (Figure 31).

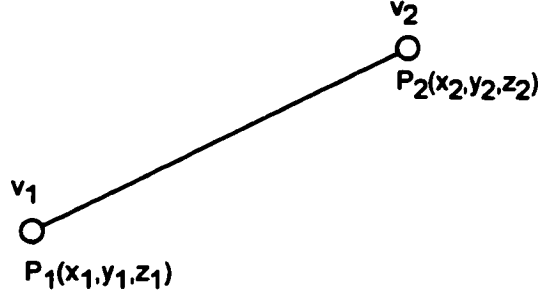


Figure 31: An edge connecting two vertices.

where

$$d_{\min} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$
$$d_{\min} > t$$
$$t = \text{tolerance}$$

B. Vertex-Edge Intersections

In the vertex-edge intersection, if the distance between a vertex and edge is smaller than the specified tolerance value, then the edge is split creating a new vertex at the projected position. This new vertex and the intersecting vertex are labeled as coincident as shown in Figure 32 and 33.

$$d_{\min} = |P - q|$$

$$(P - q) \cdot P' = 0$$

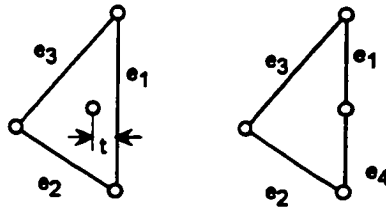
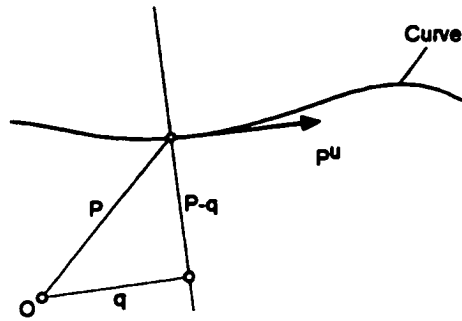


Figure 32: Intersection of a vertex and an edge.

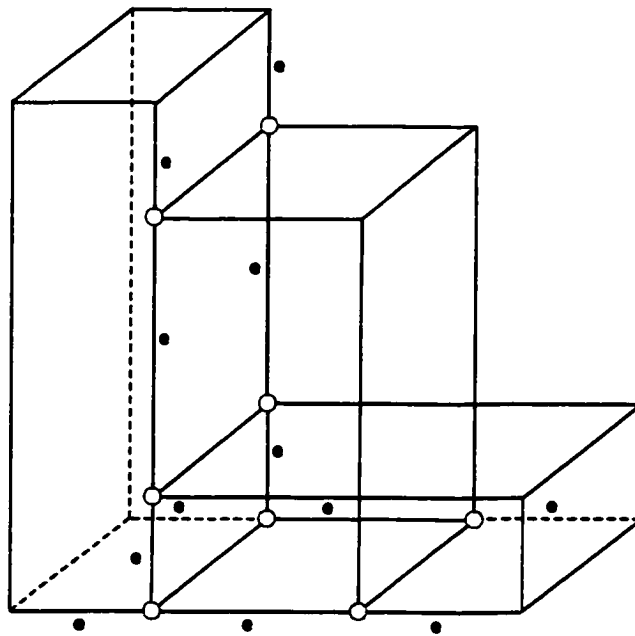


Figure 33: Vertex-Edge intersection on the models.

C. Edge-Edge Intersection

For edge-edge intersection, boundary incidence between the edge pairs is checked. Before checking the geometric intersection, the cases shown below are handled by vertex-edge intersection.

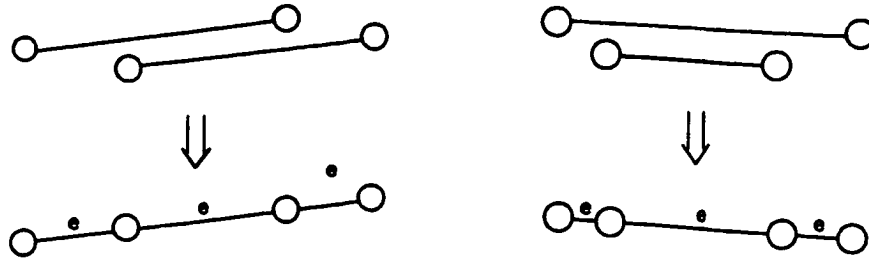


Figure 34: Edge-edge intersection.

If two edges have two vertices, which have been found coincident with each other, then the two edges are labeled as coincident.

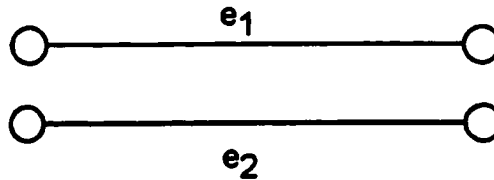


Figure 35: Edge-edge intersection.

If they each have one vertex coincident, then no geometric test is performed. Edges are then assumed to be open line segments. Other than these situations, the shortest distance is found between two edges.

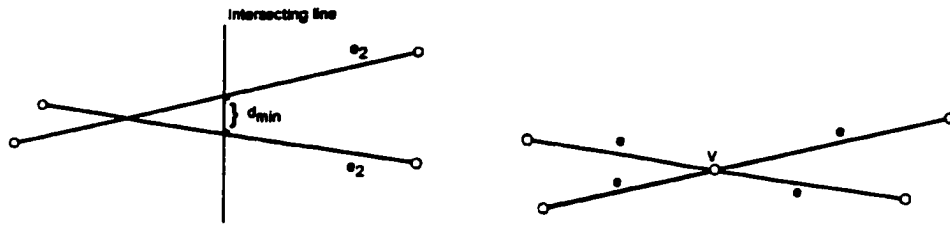


Figure 36: Edge-edge intersection.

If the distance between them is less than the tolerance value, the both edges are split at the crossing points. Then two newly created vertices are labeled as coincident. If the edges are not lines, the following intersection calculation, shown in the figure 37, is applied.

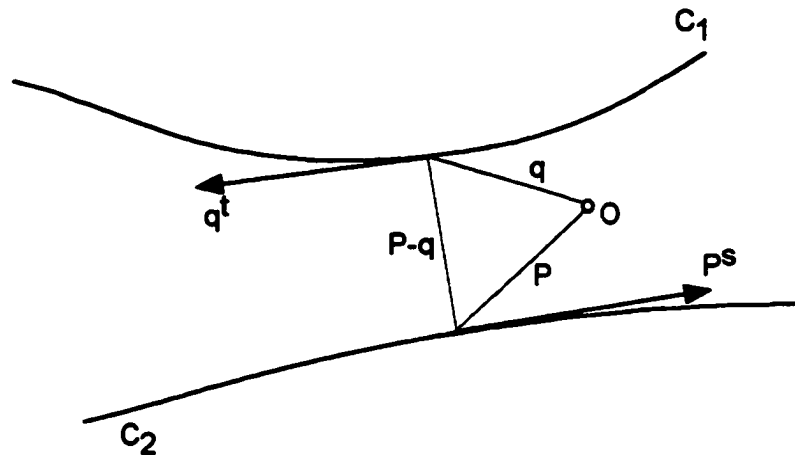


Figure 37: Determining the distance between curve edges.

$$(P - q) \cdot P^s = 0$$

$$(P - q) \cdot q' = 0$$

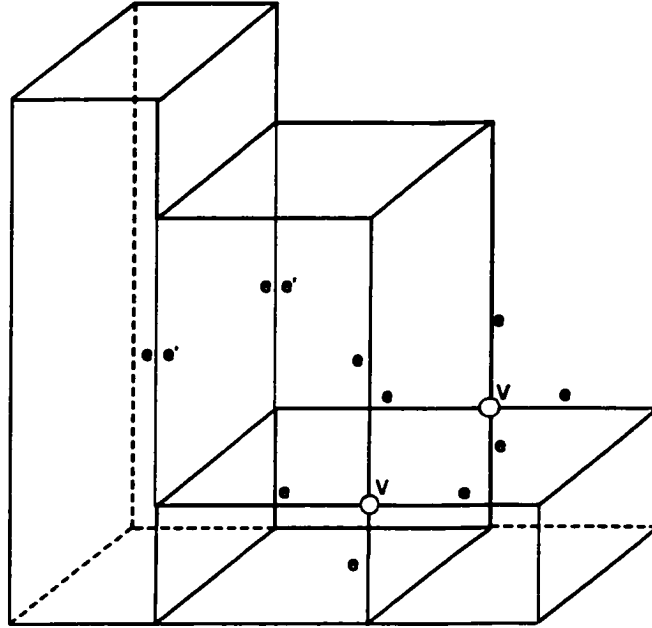


Figure 38: Intersection of edges in the model.

D. Vertex-Face Intersection

Since all the vertex-vertex and vertex-edge intersections have been performed, only the distance after the vertex is projected to the face is calculated. If the distance is less than the tolerance value and the projected position is within the face, then a vertex-loop is generated on the face. The new vertex and the intersecting vertex are labeled as coincident (Figure 39 and 40).

$$(P - q) \times (P^u - P^v) = 0$$

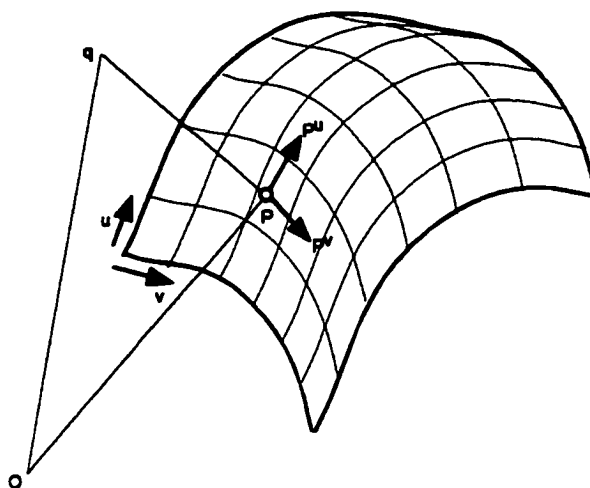


Figure 39: Determining the distance between a face and an edge.

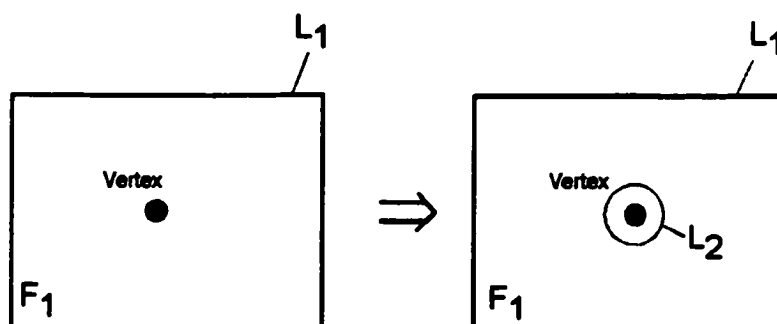


Figure 40: Single vertex loop case.

E. Edge-Face Intersection

In the edge-face intersection, the boundary incidence between the edge and the face is checked. Minimum zero and maximum two vertices of the edge may be found to incident with the vertices on the boundary of the face. If there is only one such vertex as shown in figure 41,

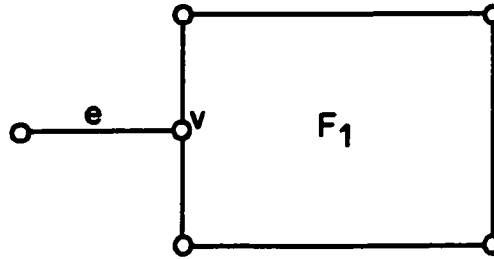


Figure 41: One vertex intersect with a face.

No geometric test is performed because primitives are open and linear. When two such vertices are found, it means the edge is within the face,

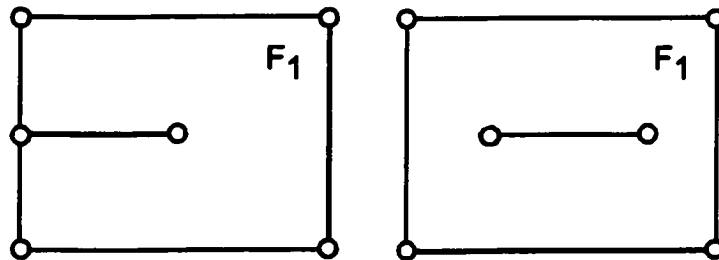


Figure 42: Two vertices included in a face.

If the edge lies within the face, a new edge is created on the face between those incident vertices. This newly created edge and the intersecting edge are labeled as coincident. If the two vertices belong to the same loop of the face as shown in figure 43;

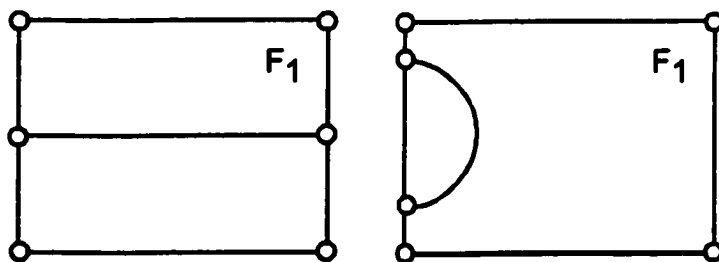


Figure 43: Two intersection vertices belonging the same loop

A new face is created along with the new edge. In this case, that loop is split into two. When the two vertices belong to two different loops (Figure 44);

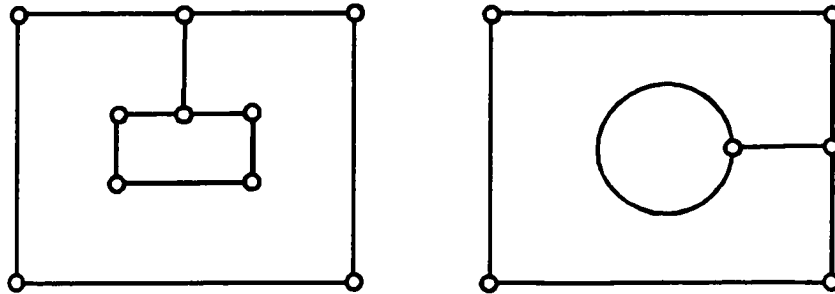


Figure 44: Two vertices belong to two loops.

However, no new face is created but those loops are merged to be one with the correct sequence of edges. One or both of these loops may as well vertex-loops. When no vertices are found to be coincident, the intersection point between the edge line and the face plane is found (Figure 45):

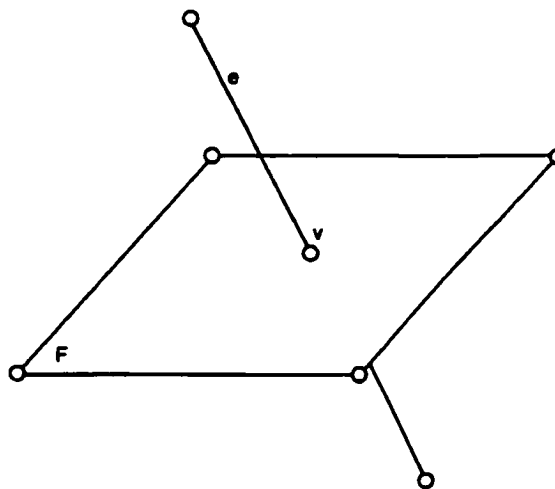


Figure 45: Intersection of a face and an edge.

If this point lies within both the edge and the face, the edge is split at that point by creating a new vertex. At the same time, a new vertex loop is created on the face. These newly created vertices are labeled as coincident (Figure 46).

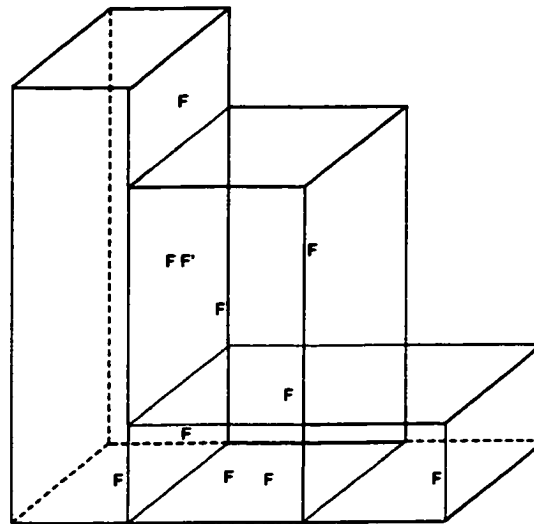


Figure 46: The edge-face intersection of two models.

F. Face-Face Intersections

In the last stage of intersection process, face-face intersections are checked. If all elements that make up the boundary of one face are identical, then the two faces are labeled as coincident (Figure 47).

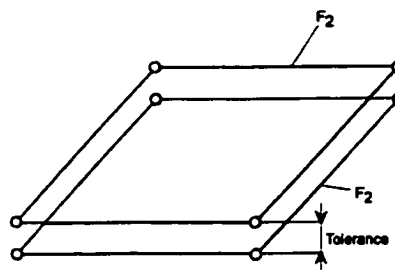


Figure 47: Coincident faces.

If they are not identical, the two faces are tested for co- linearity. If faces are co- linear, the intersection is not performed. However, if they are not co-planar, the intersection line between the two faces is performed and the coincident vertex pairs are sorted along the intersection line. The next step, two edges between the consecutive pairs of vertices are created. If the edge between the corresponding vertices lies within both of the faces, one new edge is created for each face using the appropriate vertices. These two new edges are labeled as coincident (Figure 48, and 49).

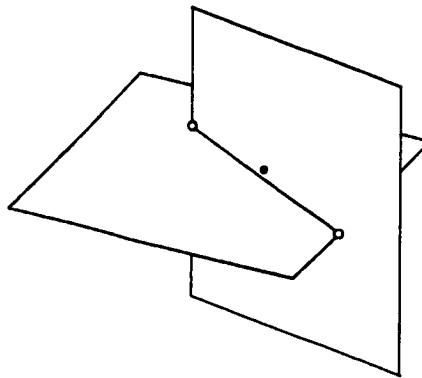


Figure 48: Intersection of two faces.

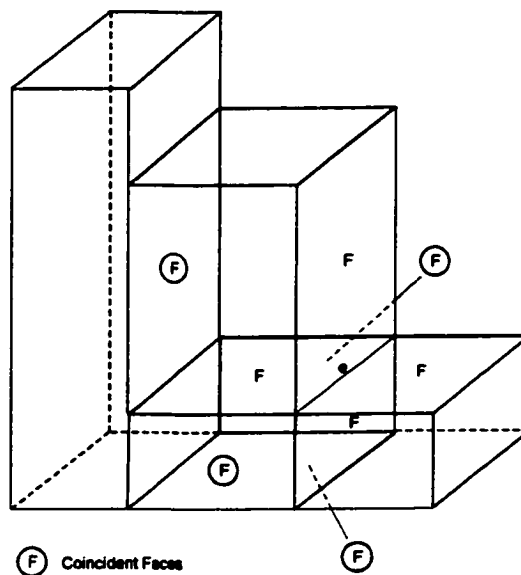


Figure 49: Face-Face intersections.

6.1.1.1 MERGING OPERATION

In the merging operation, the coincident topological elements of each object are merged, in the order of vertices, edges, and faces, along with the attributes of coincident topological elements. The merged object becomes the base for further Boolean operations and canceling operation (Figure 50).

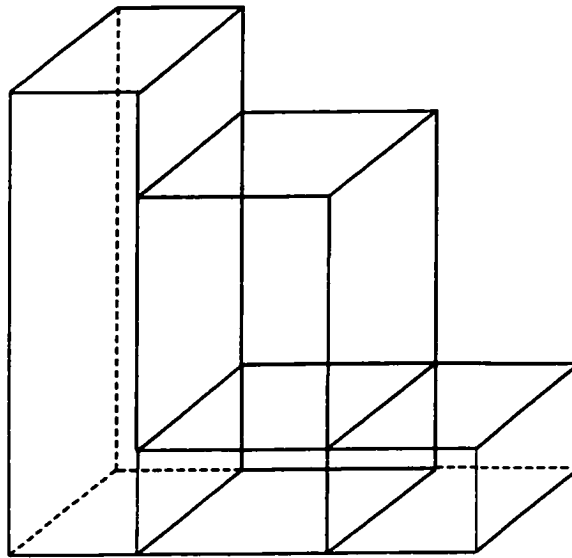


Figure 50: Merged object.

6.1.2 EXTRACTING OPERATION

Extracting operation is done for every topological element. This is performed in two stages:

- A. Extracting sub solids by repartitioning the merged object and mapping them to the original solids.
- B. Extracting faces, edges, and vertices according to the Boolean operation.

A. Extracting Solids

In the extracting operation, a categorization process is made through reducing the space into volumetric regions. In the merged object, each face can be used by two groups. Therefore, all possible groups are formed by traversing both sides of each face. Once a face side is assigned into one group, then it is labeled to avoid further inclusion to other groups. These groups are, then, formed shells. After this process, the merged object is completely decomposed into volumes. Resultant volumes are mapped into the solids of the original objects (Figure 51).

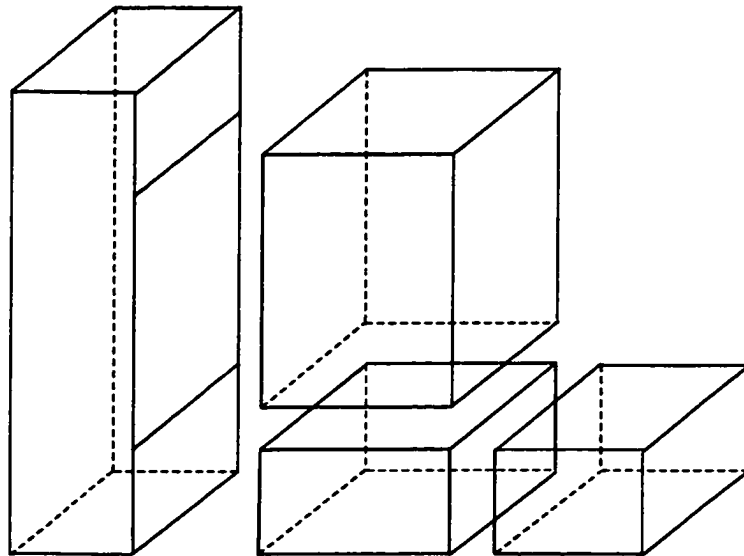


Figure 51: Repartitioned solids.

B. Extracting faces, edges, and vertices

All topological elements are given attributes, which include the information about their origin. By using that information, all vertices, edges, faces involved in the Boolean operation can easily be extracted.

For $A \cap B$ Boolean operation, all topological elements both member of object A and object B are extracted. So that, all dangling faces, edges, and single vertex shells are easily determined and extracted.

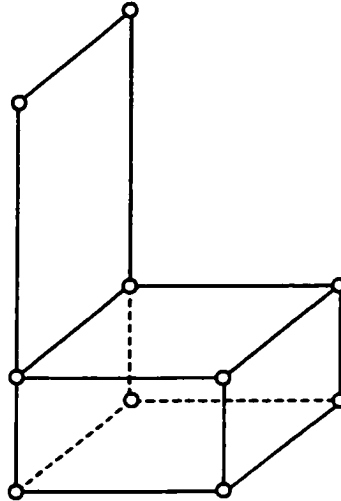


Figure 52: Extracted faces, edges, and vertices.

The execution time for this process is very short, because the system does not execute any geometric calculation or topological modification, but only traverses topological elements in the merged object. However, if there is no new volume is created in repartitioning of volumes in the solid extraction process, the exhaustive repartitioning is performed (Figure 52).

6.1.3 SELECTION OPERATION

After having obtained a merged object, any Boolean operation is trivially achieved by selecting the appropriate point set elements in the object (Figure 53). The model R in Figure 53 consists of one merged object, which has four sub-solids. The original Block 1 has three sub solids and the block number 2 has two sub solids.

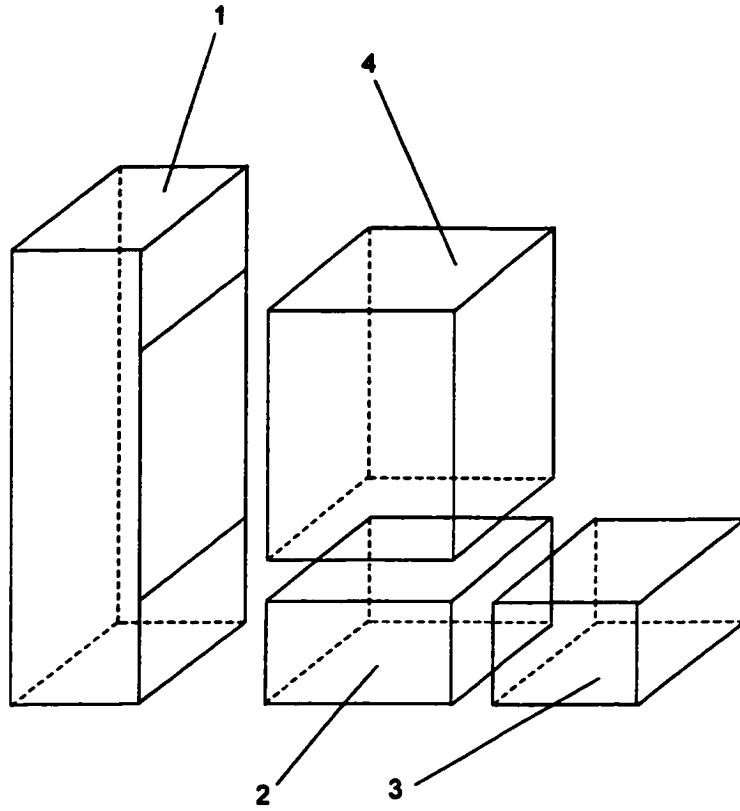


Figure 53: Repartitioned and mapped merged object.

If the Intersection Boolean operation is applied, first the original objects can be written in terms of sub solid sets as follows;

$$A^{IN} = \{R^1, R^2, R^3\}$$

$$B^{IN} = \{R^2, R^4\}$$

Then, the resultant object can be obtained from;

$$A \cap B = A^{IN} \cap B^{IN} = \{R^2\}$$

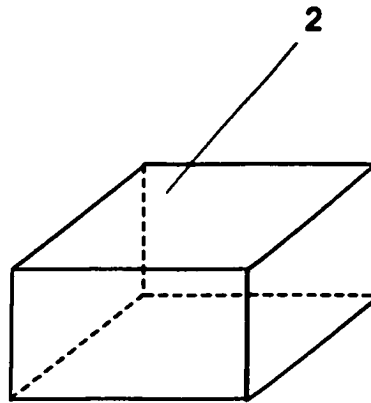


Figure 54: The resultant solid.

But the solid in Figure 54 is true not result of the intersecting operation. The final shape of the model shown in figure 55 is obtained by merging the extracted solids and extracted shell elements.

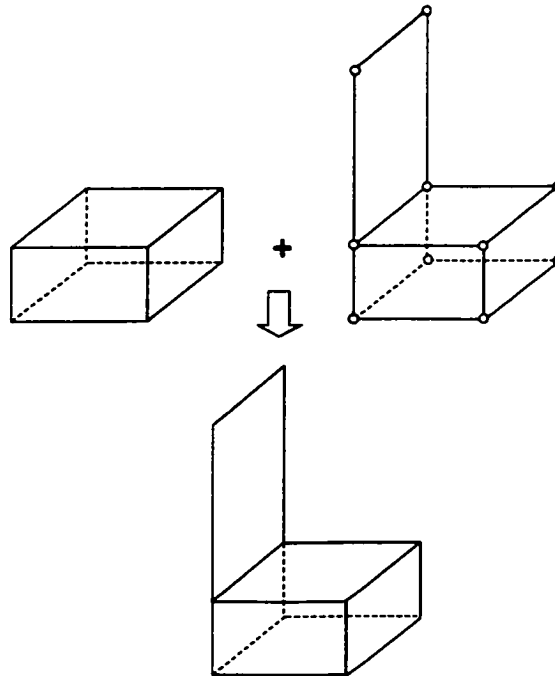


Figure 55: The final intersecting non-manifold object.

Chapter 7

Applications in Manufacturing

7.1 APPLICATIONS OF NON-MANIFOLD TOPOLOGY

The non-manifold representation is an increasingly popular technology, and it can represent the partitioning of the entire three-dimensional space in which the solid is embedded. It represents structures on the solid's surface, structures inside and outside the solid easily. Non-manifold topology, NMT, can also employ three capabilities to a geometric modeler that cannot be obtained using manifold boundary representations. These three capabilities are:

- A non-manifold topological data structure can partition the entire, infinite space into 3,2,1, and 0-dimensional point sets (volumes, faces, edges, and, vertices).
- A non-manifold topological data structure can preserve all the elements used to construct a model. This is in contrast to manifold modelers, which must discard elements that do not lie on the boundary of a solid.
- A universal *merge* operation can be defined for combining non-manifold models. No topological elements are discarded in a merge operation, thus it preserves topological information.

Only, the last two capabilities can be understood by looking the nature of *merging* operation in NMT. On the other hand, Boolean operations are usually dependent on the operation sequence, which means they don't satisfy the associative or commutative law.

Thus, Masuda [5] proposed a method to solve this problem. In this method, Boolean operation is divided into two operations: a merge operation, which is independent of the operation sequence, and an extracting operation, which is dependent on the operation sequence. An addition to that, in conventional boundary modeling, it is known that the original primitive objects are not left in the resultant shape when two primitive objects are unified, and topological elements inside the unified object are deleted. Therefore, the shapes of the original primitive objects are partly lost from the topological structure.

In some cases, geometric models could not be manipulated consistently, because the combined models are non-manifold, even though the original ones were not. However, it is possible to maintain the shapes of the original primitive objects in the topological structure by using the non-manifold geometric modeling techniques. In these techniques, the system generates intersection edges and vertices on each object. If a topological element is split, its attributes are inherited to a split topological element. The system also merges the coincident topological elements of each object in the order of vertices, edges, and faces. The attributes of the coincident topological elements are merged too. If a merged-edge is shared by more than two faces, faces around the edge are sorted to maintained the radial-edge orders. When two models are merged, volumes may be split. The split volumes are reconstructed by detecting their boundary faces connected by the radial-edge pointers. The attributes of reconstructed volumes are also maintained.

It can be seen that non-manifold representations preserves the all topological elements used to constructed a model, topological information, and other information (attributes) at the same time. This ability is the answer to most of the existing manufacturing simulation tools that have shortcomings in addressing the integration of

process and other non-geometric information with geometry. This integration can be done using the attributes easily. However, it should be known that attributes can store user-defined data, but they are primarily labels. Attributes have no significance to the merge operation, but they are carried along as it splits and merge topological elements. In more precise terms, an attribute assigned to a topological element is a *persistent* label on every point in that point set.

In Charlesworth[11], automatic Generation of Machining Tool Path for 2 1/2 Dimensional Pockets is explained by using NMT. The main idea behind many of the automatic pocketing procedures reported in the literature (Persons[37], Guyder[38], Suh[39], Held[40], or Southwood[41]) is to reduce to pocket to a two-dimensional profile and define tool paths in terms of offsets from the profile.

The tool paths can be obtained from successive offsets, or they may be composed of parallel motions bounded by a single offset. The typical automatic pocketing procedure obtains machining instructions from the pocket profile in four major steps:

1. Generate offsets and the other curves for the cutter.
2. Trim the curves to obtain feasible paths.
3. Order the paths into a sequence that satisfies good machining practices.
4. Generate instructions to move the cutter along the path.

In this research, the authors work on the step 2 as an application of non-manifold topology. Their procedure starts with the generation of offset curves. Then, they model the offset of these two-dimensional curves as areas to construct the offset of the entire profile from the collection of these areas. So, each generated offset curve forms an area according to the movement of the cutter tool. They are called “offset regions”.

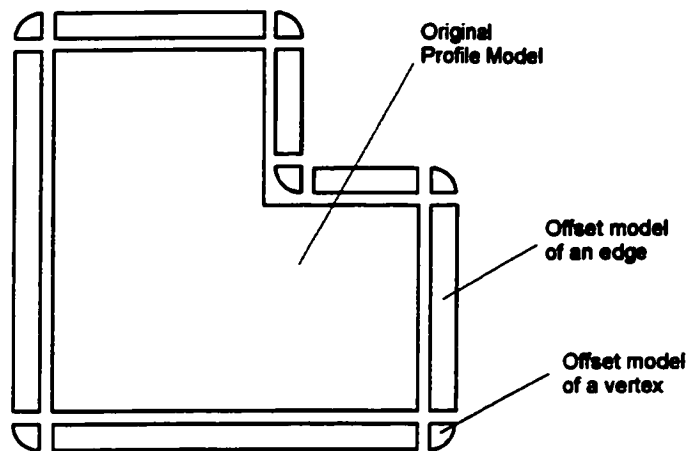


Figure 56: Profile and outward offset.

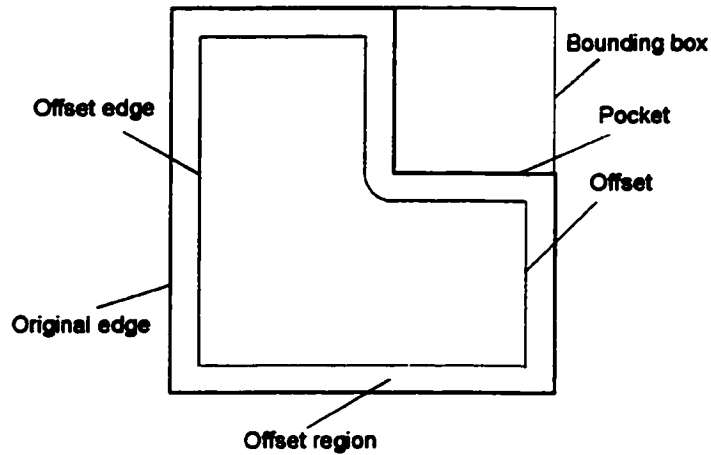


Figure 57: Profile and inward offset.

Offset edges are the edges that form the boundary of region that was not machined yet. In the meantime these offset edges and offset regions attributes are attached to offset edges and offset regions, respectively. Then, a series of possible parallel motion tool paths is generated. These tool paths and the profile and offset curves

are merged by selecting only those edges that have an “offset edge” attribute and no “offset region” attribute (Figure 56-58).

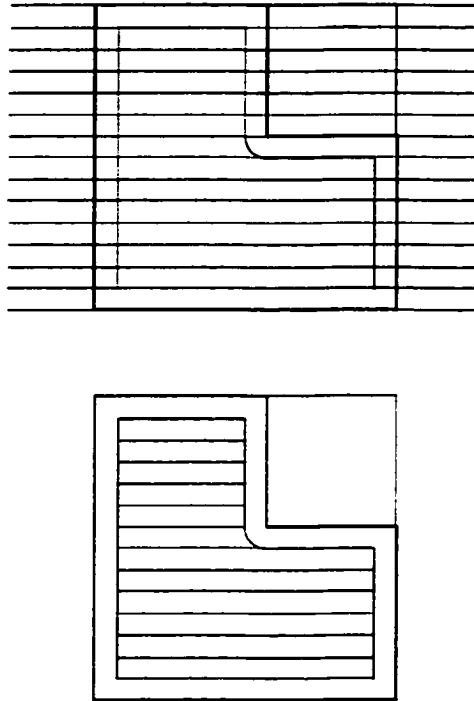


Figure 58: Steps in creating parallel tool paths.

Chapter 8

Development of Mesh Generation Procedure for Non-manifold Topologies

8.1. FINITE ELEMENT MODELING

The finite element modeling process starts with the geometric representation of the object, and ends with the production of analysis results. The basic steps in this process are:

1. Definition of analysis attributes.
2. Simplification of the geometry.
3. Finite elements generation within the domain.
4. Solution of the algebraic equations associated constructed the finite element model.
5. Association of the results with the original model.

It can be easily seen that all steps except the step four are geometry-based operations. They depend on direct interactions with the geometric representation. On the other hand, the domain to be discretized into a finite element model won't be the same as that of the original object. An addition to that, the simplification of the geometry may occur by ignoring specific geometric features, such as small holes, and fillets, or by using beams and shells element to reduce the computation time in finite element analysis.

Once the geometric model has been modified into the idealized model that is to be analyzed, the mesh generation process begins by employing a set of algorithmic procedures to reduce the idealized model to a finite element model.

Then the analysis attribute information is associated with the finite element model. The analysis constructs and solves the various sets of algebraic equations that govern the model, by using all provided data as an input.

The most fundamental data in the generation of a finite element model is the geometry. A boundary based model data structure provides a general framework for this data structure. Since finite element models commonly consist of combinations of three-dimensional (solid elements), two-dimensional (shell elements), one-dimensional (beam elements) entities, it is necessary to establish a model representation that can handle all three without the need for special cases.

The commonly used boundary representation for solid modeling systems can only be represent two-manifold geometries which means that even a mesh of solid elements alone would require special consideration.

However, the radial edge data structure [12] can house combined solid, surface, and wire-frame geometries in a consistent manner. Therefore, it is an ideal representation scheme for the finite element model data structure. In particular, vertices point to points in space, edge to one or more curves, faces to one of more surface patches and regions may be point to volume entities. The mesh data structure, which stores the finite elements, should also use a hierarchic boundary-based structure. This hierarchy is shown in Figure 59. Each finite entity points to the lowest order modeling topology entity, which it is inherently a part.

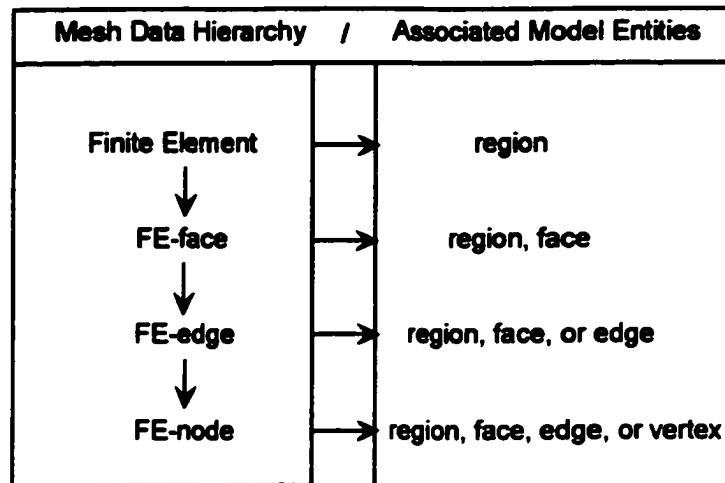


Figure 59: Mesh hierarchy and associated entities.

8.2. MESH GENERATION

The generation of a finite element mesh can simply be described as the discretization of a given geometry into a set of topologically simple pieces. This view, to be referred to as top-down finite element model generation, is consistent with the use of a geometric modeling system to define the geometry and finite element mesh generation to discretize it.

Up to now, there has been a number of new algorithmic procedure developed for the generation of finite element meshes designed to discretize general domains automatically without the need for user intervention. Since an automatic mesh generator must be capable of discretising a general domain into a valid finite element mesh, without user intervention, these techniques can only be operated using a computerized geometric

representation of the object to be meshed. The automatic mesh generators that have been developed so far use one, or a combination of the following algorithmic approaches:

1. Point placement and Delaunay triangulation.
2. Spatial decomposition and sub-domain meshing.
3. Removal of individual pieces until the domain is reduced to the pieces desired.
4. Recursive subdivision.

8.2.1 DELAUNAY TRIANGULATION

Delaunay triangulation is the geometric dual of the Voronoi diagram (which is also known as the Dirichlet tessellation). The definition of Delaunay triangulation is as follows:

Given a set P of M points in n -dimensional space, there exists a region V_i such that:

$$V_i = \{x : |x - P_i| \leq |x - P_j| \}$$

$$i \neq j$$

The collection $V = \sum_{i=1}^M V_i$ is designed as the Voronoi diagram. Given a Voronoi diagram V , the lines connecting the points P_i to P_j where V_i and V_j are neighboring tiles corresponding to the points P_i to P_j form the Delaunay triangulation.

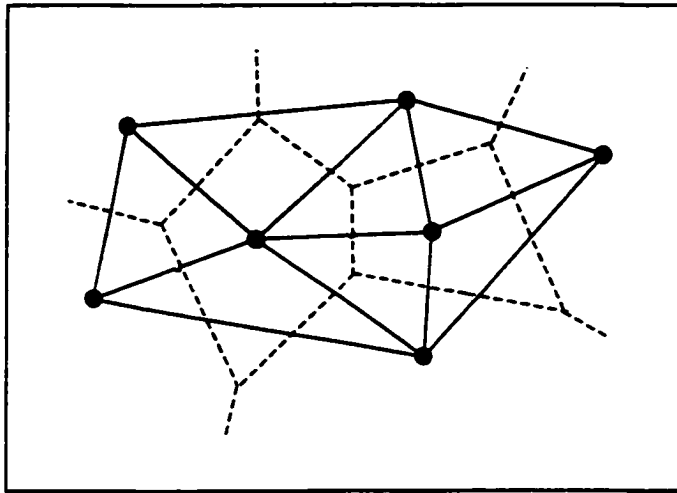


Figure 60: Delaunay triangulation and corresponding Voronoi diagram.

The Voronoi diagram for seven points in the plane is shown by the dotted lines in Figure 60. Each Voronoi polygon is constructed by drawing perpendicular bisectors through the lines that connect each point to its nearest neighbors. Every polygon is associated uniquely with a single point and all sites within a polygon are closer to this point than any other point in the data set. Because of this last property, Voronoi diagrams are often used in geometric algorithms, which require nearest neighbor searching.

Once the Voronoi diagram for a set of points is known, the corresponding Delaunay triangulation is readily computed by connecting all pairs of points, which share a polygon boundary. Generally speaking, each vertex of the Voronoi diagram is located at the point of contact of three adjacent polygons, and hence, defines the circumcenter for a Delaunay triangle. This last condition implies that the circumcircle for each triangle is “empty”, since it may not contain a vertex.

One major advantage of the Delaunay triangulation, as opposed to a triangulation constructed heuristically, is that it automatically voids the creation of long thin triangles, with small included angles, wherever possible. Indeed, Lawson [33] has proved that the Delaunay triangulation is, by definition, locally equiangular.

This means that for every convex quadrilateral formed by two adjacent triangles, the minimum of the six angles in the two triangles is greater than it would have been if the alternative diagonal had been drawn and the other pair of triangle chosen. Because of this property, Delaunay triangulation is natural choice for mesh generation in finite element analysis and contour plotting applications.

8.2.2 INCREMENTAL DELAUNAY TRIANGULATION IN 2-D

In literature, the first approach to implement the Delaunay triangulation is suggested by Watson [21]. His algorithm simply inserts every point of a set into the 2-D space one by one. This simplifies the triangulation procedure dramatically for the data sets that have a large number of points. There are several algorithms that use the same basic idea. In one of them, one starts with the boundary box's triangles, and modify them by adding new points one by one.

Boundary box as suggested in reference [34] is used because it eliminates having triangle with unacceptable distortion, so called silver triangle, on the convex hull, which are caused by floating point errors. Moreover, this starting technique simplifies the algorithm, because all inserted nodes always exist within the boundary box.

The algorithm can be outlined as follows (Figure 61 and 63):

1. Create a square box in which all nodes on the geometric model exist within the boundary box, construct two triangles.
2. Insert point P_1
3. Find one triangle, which contains P_1 in its boundary.
4. Find the neighboring triangles.
5. Eliminate the neighboring triangles that their circumcircle do not contain P_1 .
6. Remove the common edges and form the bounding polygon R .
7. Decompose the region R into triangles at the apex of P_1 with the base edges on the boundary of R .

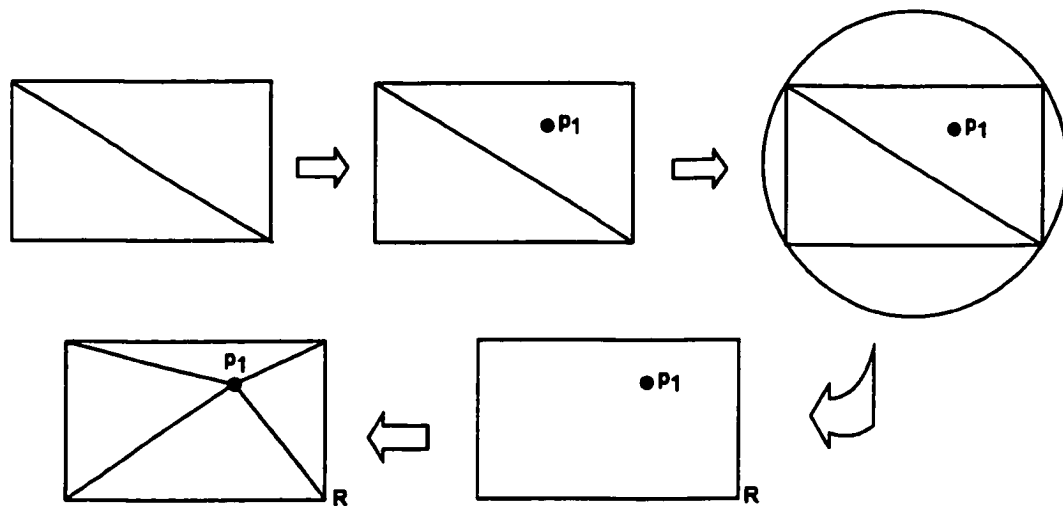


Figure 61: Incremental Delaunay Triangulation Method.

Figure 62 and 63 demonstrates in general the use of this algorithm in the implementation.

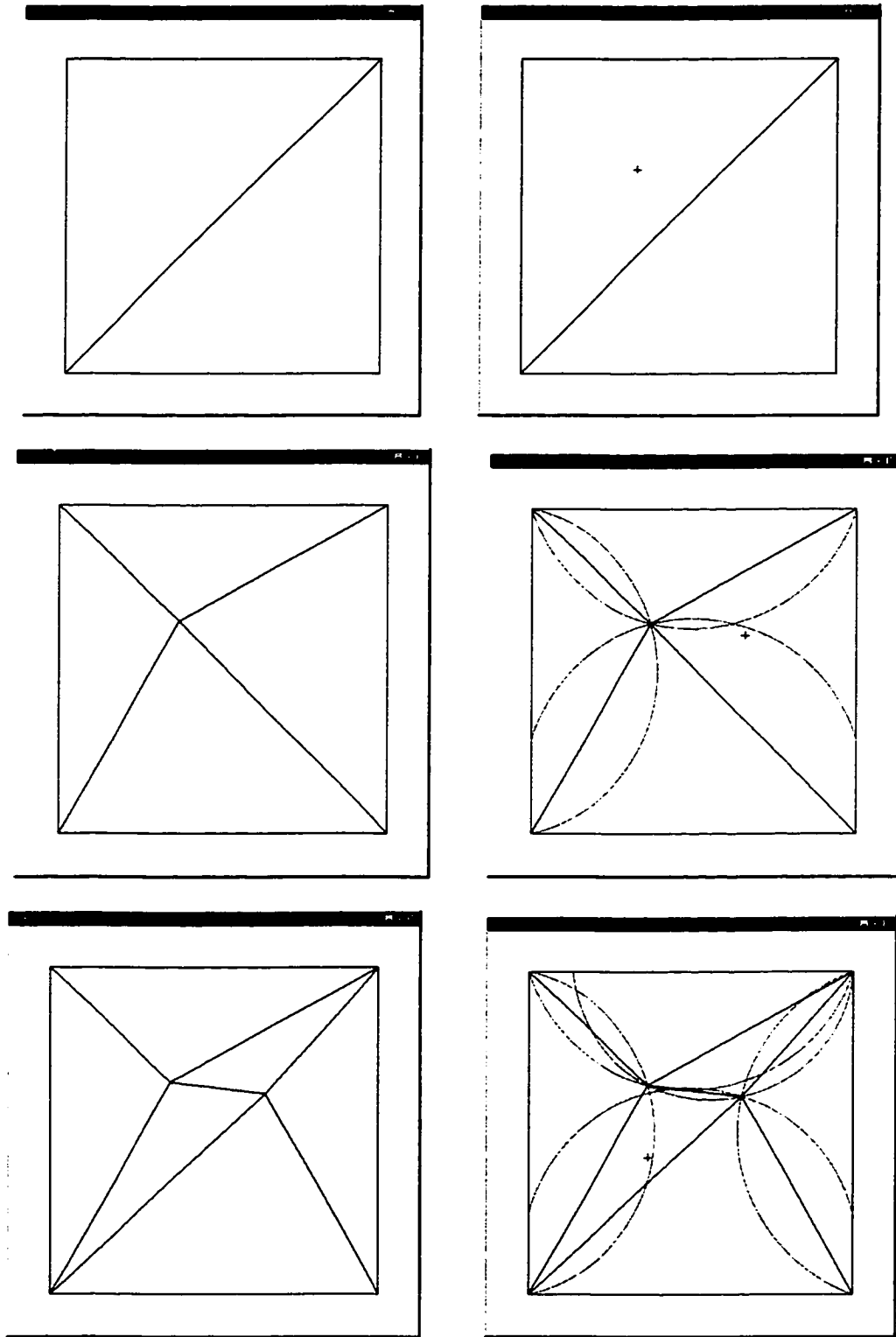


Figure 62. Experimental results in 2-D

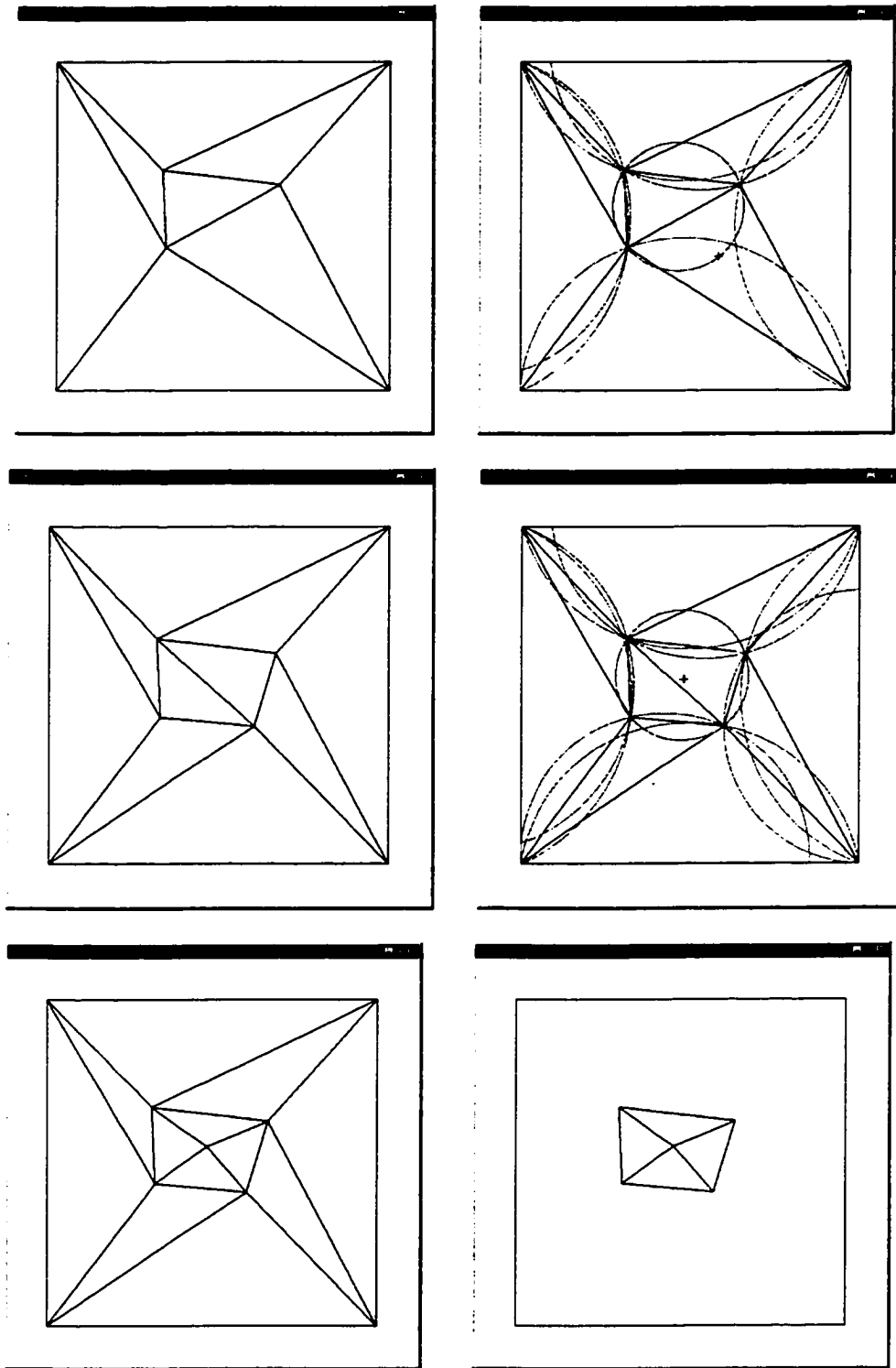


Figure 63: Experimental results in 2-D.

8.2.3 PROPERTIES OF A TRIANGLE

There are so many references describing the properties of a triangle, which are very useful in Delaunay triangulation in 2-D. Suppose that an ordered triple (P,Q,R) of three points in x-y plane is given. The Counter-clockwise or clockwise direction of the triangle, when visiting these points in the given order, is needed to be determined. Figure 64 shows the possibilities, which is referred to as positive and negative orientation, respectively.

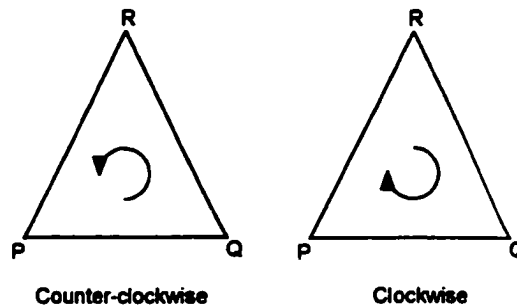


Figure 64: Possibilities for different triangle orientation.

There is a third possibility, namely that the points P,Q, and R lie on a straight line. The orientation is considered to be zero in this case. By using only the coordinates X_P , Y_P , X_Q , Y_Q , X_R , and Y_R , the orientation of the triangle can be found.

Since a translation of three points does not alter their orientation, the three x-coordinates and y-coordinates are subtracted by X_P , and Y_P respectively. Then, three points becomes $O(0,0)$, $A(X_A, Y_A)$, $B(X_B, Y_B)$, shown in Figure 65, where

$$\begin{aligned} X_A &= X_Q - X_P & X_B &= X_R - X_P \\ Y_A &= Y_Q - Y_P & Y_B &= Y_R - Y_P \end{aligned}$$

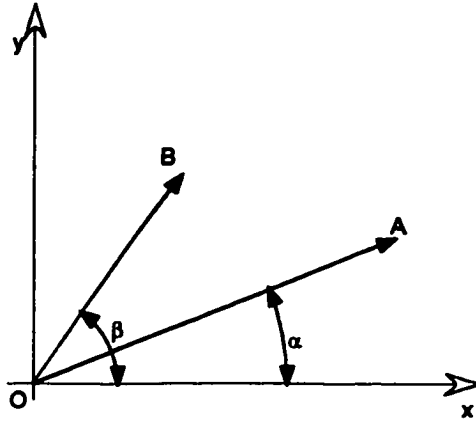


Figure 65: Translated coordinate system.

After the angle α , between vectors A and the positive x-axis, and β , the angle between B and the positive x-axis, are defined. It can be easily seen that the orientation of OAB depends on the angle $\beta - \alpha$. If this angle lies between 0 and π , the orientation is clearly positive, but it is negative if this angle lies between π and 2π (or between $-\pi$ and 0). On the other hand, the orientation in question depends on the value of $\sin(\beta - \alpha)$ rather than on the angle $\beta - \alpha$ itself. More specifically, the orientation has the same sign as;

$$\sin(\beta - \alpha) = \sin\beta \cdot \cos\alpha - \cos\beta \cdot \sin\alpha = \begin{vmatrix} \cos\alpha & \sin\alpha \\ \cos\beta & \sin\beta \end{vmatrix} = D_o$$

Since we have

$$\cos\alpha = \frac{X_A}{|A|}, \cos\beta = \frac{Y_A}{|A|}, \cos\beta = \frac{X_B}{|B|}, \sin\beta = \frac{Y_B}{|B|}$$

If we substitute these into the previous equation, we get

$$\frac{\begin{vmatrix} X_A & Y_A \\ |A| & |A| \end{vmatrix}}{\begin{vmatrix} X_B & Y_B \\ |B| & |B| \end{vmatrix}} = D_o$$

or

$$\begin{vmatrix} X_A & Y_A \\ X_B & Y_B \end{vmatrix} = D_o \cdot |A| \cdot |B| = D$$

Also D can be defined in terms of point P, Q, and R;

$$\begin{vmatrix} X_Q - X_P & Y_Q - Y_P \\ X_R - X_P & Y_R - Y_P \end{vmatrix} = D$$

or

$$\begin{vmatrix} X_P & Y_P & 1 \\ X_Q & Y_Q & 1 \\ X_R & Y_R & 1 \end{vmatrix} = D$$

This yields the equation of the Area of the triangle. So, the orientation of (P, Q, R) is determined by computing the value of D.

IF $D > 0$, the points P, Q, and R, in that order, are counter-clockwise.

IF $D = 0$, the points P, Q, and R show a straight line.

IF $D < 0$, then points P, Q, and R, in that order, are clockwise.

A. Testing whether a point lies inside a triangle

After determining the orientation of a triangle, it is easy to test whether a point lies inside a triangle. It can be written that;

$$\text{Orientation}(A, B, P) = \text{Orientation}(B, C, P) = \text{Orientation}(C, A, P)$$

If and only if P lies inside triangle ABC as Figure 66 shows;

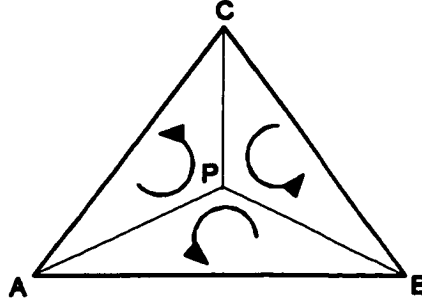


Figure 66: Orientation of a triangle.

Alternatively, the following equation can be used;

$$|\text{Area}(A, B, C)| = |\text{Area}(B, C, P)| + |\text{Area}(C, A, P)| + |\text{Area}(A, B, P)|$$

If there is equality between RHS and LHS of the above equation, the point P is inside the triangle. If not, the point P is outside.

B. Quality of a triangle

The goal of shape control in mesh generation algorithms is to produce triangles, which are as well shaped as possible. This accomplished indirectly by heuristically developed strategies for node placement in the selected triangles. For most finite element

analyses, the ideal element shape is the equilateral triangle. A useful measure of how closely an arbitrary triangle approaches this ideal is the ratio of its inradius (r) to circumradius (R). This ratio can be easily driven as following;

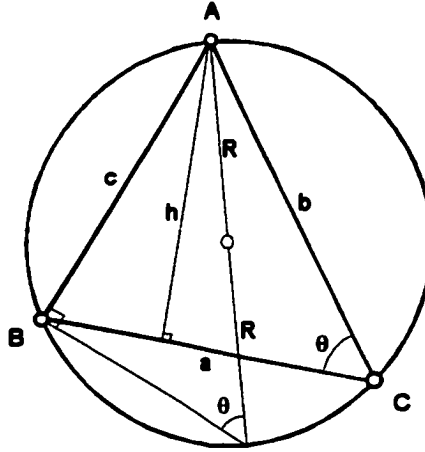


Figure 67: Geometry of a triangle.

It can be written that

$$\sin \theta = \frac{h}{b} = \frac{c}{2.R}$$

or

$$R = \frac{b.c}{2h}$$

substituting $h = \frac{2.S}{a}$ into the R, we get

$$R = \frac{a.b.c}{4.S}$$

where S is the area of the triangle. For the innerradius, it can be written that

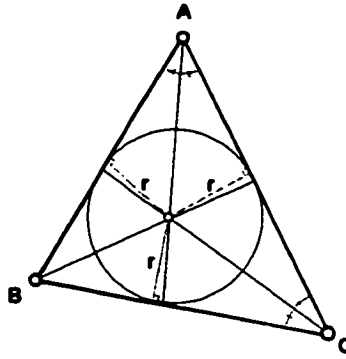


Figure 68: Geometry of a triangle.

$$\frac{r}{2} \cdot (a + b + c) = S$$

or

$$r = \frac{2.S}{(a + b + c)}$$

then the ratio between r and R would be

$$\frac{r}{R} = \frac{\frac{2.S}{(a + b + c)}}{\frac{a.b.c}{4.S}} = \frac{8.S^2}{a.b.c.(a + b + c)}$$

On the other hand, in the case of an equilateral triangle in which $a=b=c$, the ratio would be

$$\frac{r}{R} = \frac{4.\left(\frac{3.a}{2} - a\right)^3}{a^3} = \frac{1}{2}$$

This ratio r/R achieves its maximum value of one-half when the triangle is equilateral. It is convenient to multiply the ratio by two to obtain the normalized shape ratio:

$$\delta = \frac{2.r}{R}$$

C. Finding the center coordinates of innercircle and circumcircle

Since only nodes on the corner of triangles are known, we have to start from finding the equation of a circle defined by three points.

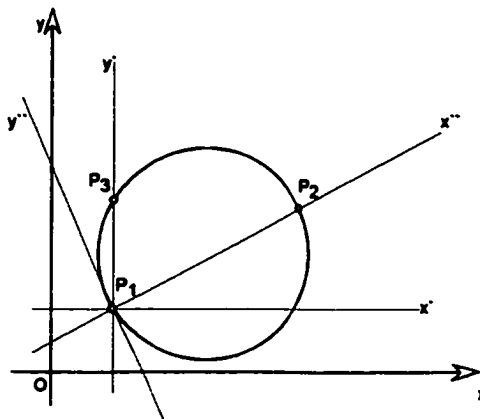


Figure 69: Circle through three points.

A direct solution technique is to write three simultaneous non-linear equations in h , k , the x and y coordinates of the center of the circle, and R , the radius.

$$(x_1 - h)^2 + (y_1 - k)^2 = R^2 \dots\dots\dots 1a$$

$$(x_2 - h)^2 + (y_2 - k)^2 = R^2 \dots\dots\dots 1b$$

$$(x_3 - h)^2 + (y_3 - k)^2 = R^2 \dots\dots\dots 1c$$

The solution algorithm which first subtracts eqn.(1b) from eqn.(1c) reduces the solution to a linear problem. Specifically, the solution algorithm is

$$[(1a) - (1b)](x_3 - x_2) - [(1c) - (1b)](x_1 - x_2)$$

The resulting equation in the unknown k is

$$k = \frac{[(x_1^2 - x_2^2) + (y_1^2 - y_2^2)](x_3 - x_2)}{2[(y_1 - y_2)(x_3 - x_2) - (y_3 - y_2)(x_1 - x_2)]} - \frac{[(x_3^2 - x_2^2) + (y_3^2 - y_2^2)](x_1 - x_2)}{2[(y_1 - y_2)(x_3 - x_2) - (y_3 - y_2)(x_1 - x_2)]}$$

Similarly, h is calculated from $[(1a) - (1b)]$, and eqn. k as

$$h = \frac{(x_1^2 - x_2^2) + (y_1^2 - y_2^2) - 2k(y_1 - y_2)}{2(x_1 - x_2)}$$

Examination of h,k reveals some difficulties. In particular, if the dominator of either equation is zero, then alternate methods must be used. Further, the condition to check for an infinite radius or for collinear points is not immediately obvious. To overcome these difficulties, the following solution procedure is adopted:

1. Translate such that one of the points is at the origin of the coordinate system.
2. Rotate about the origin such that one of the either points lies on the positive x-axis.

3. Check for collinear points.
4. Solve in the translated and rotated coordinate system for the center and radius of the circle.
5. Rotate about the origin back to the intermediate coordinate system.
6. Translate back to the original coordinate system to get the center of the circle in the original coordinate system.

Figure 66 illustrates this technique where P_1 has been translated to the origin and P_2 rotated to the positive x-axis. The governing equations in the (x'', y'') coordinate system are

$$\begin{aligned}h''^2 + k''^2 &= R^2 \\(x_2'' - h'')^2 + k''^2 &= R^2 \\(x_3'' - h'')^2 + (y_3'' - k'')^2 &= R^2\end{aligned}$$

Subtracting the first equation from the second one yields

$$x_2'' - 2x_2'' \cdot h'' = 0$$

or

$$h'' = \frac{x_2''}{2}$$

Subtracting the first equation from the last one and substituting the equation the last equation yields;

$$k'' = \frac{x_3''}{2y_3''}(x_3'' - x_2'') + \frac{y_3''}{2}$$

and from the first equation;

$$R = \sqrt{(h''^2 + k''^2)}$$

Examination of the equations reveals that if $y_3'' = 0$ the k is infinite. However, y_3'' can be only zero if the three points are collinear. Thus, this is a simple obvious check for the collinear case.

After determining the radius and the coordinates of the center point of the circumcircle, the center point of the innercircle is needed by the node insertion algorithms. To calculate the value, the basic principles of geometry are investigated. It is well known that

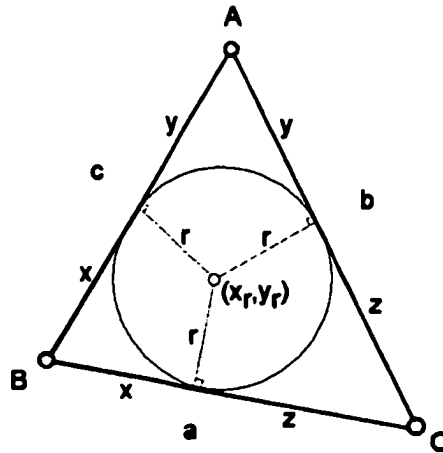


Figure 70: Geometry of innercircle.

$$2x + 2y + 2z = a + b + c = 2u$$

and

$$x + y + z = u$$

It can be seen that

$$c = x + y$$

$$b = y + z$$

$$a = x + z$$

x, y, and z can be written as

$$x = u - b$$

$$y = u - a$$

$$z = u - c$$

After finding x, y, and z values, the corresponding coordinates are determined by using parametric formulation as following:

$$x_1 = x_A + t_1(x_C - x_B)$$

$$y_1 = y_B + t_2(y_C - y_B)$$

$$x_2 = x_B + t_2(x_A - x_B)$$

$$y_2 = y_B + t_2(y_A - y_B)$$

$$x_3 = x_C + t_3(x_A - x_C)$$

$$y_3 = y_C + t_3(y_A - y_C)$$

$$t_1 = \frac{x}{a}$$

$$\text{where } t_2 = \frac{y}{c}$$

$$t_3 = \frac{z}{b}$$

P(x1,y1), P(x2,y2), and P(x3,y3) are the three points that their corresponding innercircle center coordinates can be found by following the procedure previously explained.

8.2.4 INCREMENTAL DELAUNAY TRIANGULATION METHOD IN 3-D

Despite the difference in dimensions, in 3-D case, the similar steps is followed to construct the Delaunay triangulation. The outline of the steps is as following;

1. Create a box in which all nodes on the geometric model exist within the boundary box, then construct five tetrahedrons out of this boundary box.
2. Insert point P_i .
3. Find the tetrahedron, which contains P_i in its boundary.
4. Find the neighboring tetrahedrons
5. Eliminate the neighboring tetrahedrons that their circumsphere do not contain P_i .
6. Remove the common facets and form the bounding region.
7. Decompose the region tetrahedrons with the apex at P_i with the base facets on the boundary.

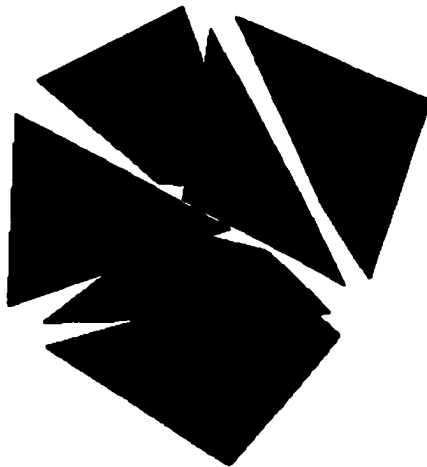


Figure 71: The initial box mesh.

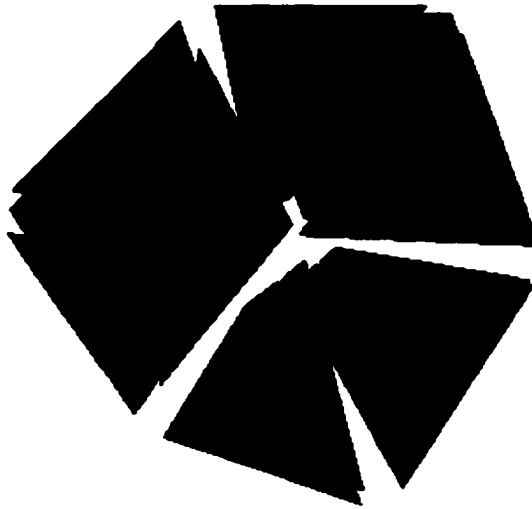


Figure 72: After inserting the first node.

The results of the above algorithm can be seen in Figure 71,72. In order to generate mesh by using the tetrahedron, it is crucial to know the properties of a tetrahedron. To find them, the simple rules of geometry are utilized.

8.2.5 PROPERTIES OF TETRAHEDRONS

The tetrahedron not surprisingly exhibits properties similar to those of the triangle. It is possible to use the similar methods to test whether a point lies inside tetrahedron by using the orientation of the facets of the tetrahedron. Another simple method can be used to do that.

This method utilizes the volumetric equality between internal tetrahedrons that are formed by the test point and the faces of a tetrahedron, and the bounding main tetrahedron (Figure 73). This checking can be made using the following formula:

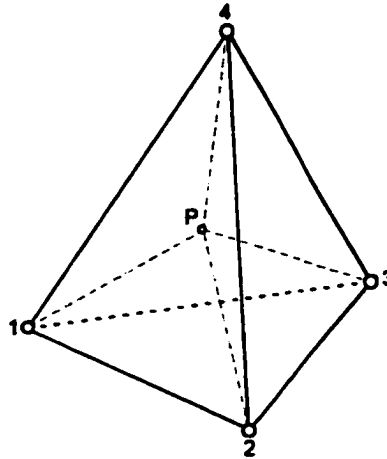


Figure 73: The geometry of a tetrahedron.

$$V_{1234} = \int dx.dy.dz = \frac{1}{6} \begin{vmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{vmatrix}$$

If and only if P lies inside the tetrahedron, when

$$|V_{1234}| = |V_{123P}| + |V_{234P}| + |V_{143P}| + |V_{124P}|$$

satisfies the equality.

A. Finding the center coordinates of innersphere and ciumsphere

Since only nodes on the corner of the tetrahedron are known, the equation of a sphere is determined by using these four points (Figure 74).

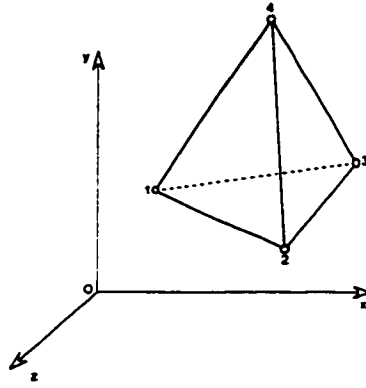


Figure 74: The geometry of a tetrahedron.

A direct solution technique is to write four simultaneous non-linear equations in h, k, f , the x and y coordinates of the center of the sphere, and R , the radius.

$$(x_1 - h)^2 + (y_1 - k)^2 + (z_1 - f)^2 = R^2$$

$$(x_2 - h)^2 + (y_2 - k)^2 + (z_2 - f)^2 = R^2$$

$$(x_3 - h)^2 + (y_3 - k)^2 + (z_3 - f)^2 = R^2$$

$$(x_4 - h)^2 + (y_4 - k)^2 + (z_4 - f)^2 = R^2$$

The origin of the coordinate system is translated to the point P_1 (Figure 75), then, the set of equation in an open form becomes as follows:

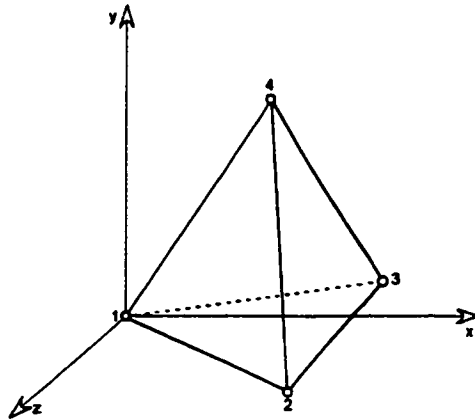


Figure 75: Translated origin to the one of the corner of a tetrahedron

$$h'^2 + k'^2 + f'^2 = R^2$$

$$h'^2 + k'^2 + f'^2 - 2(x'_2 h' + y'_2 k' + z'_2 f) + x'^2_2 + y'^2_2 + z'^2_2 = R^2$$

$$h'^2 + k'^2 + f'^2 - 2(x'_3 h' + y'_3 k' + z'_3 f) + x'^2_3 + y'^2_3 + z'^2_3 = R^2$$

$$h'^2 + k'^2 + f'^2 - 2(x'_4 h' + y'_4 k' + z'_4 f) + x'^2_4 + y'^2_4 + z'^2_4 = R^2$$

If the first equation is subtracted from the others, they yield;

$$2(x'_2 h' + y'_2 k' + z'_2 f) - x'^2_2 - y'^2_2 - z'^2_2 = 0$$

$$2(x'_3 h' + y'_3 k' + z'_3 f) - x'^2_3 - y'^2_3 - z'^2_3 = 0$$

$$2(x'_4 h' + y'_4 k' + z'_4 f) - x'^2_4 - y'^2_4 - z'^2_4 = 0$$

Also these resultant equations are reduced into

$$x'_2 h' + y'_2 k' + z'_2 f = \frac{x'^2_2 + y'^2_2 + z'^2_2}{2} = Q_1$$

$$x'_3 h' + y'_3 k' + z'_3 f = \frac{x'^2_3 + y'^2_3 + z'^2_3}{2} = Q_2$$

$$x'_4 h' + y'_4 k' + z'_4 f = \frac{x'^2_4 + y'^2_4 + z'^2_4}{2} = Q_3$$

Since three unknowns and three equations indicate that the set of equations can easily be solved for h, k, and f. The results will be as following;

$$h' = \frac{\begin{vmatrix} Q_1 & y'_2 & z'_2 \\ Q_2 & y'_3 & z'_3 \\ Q_3 & y'_4 & z'_4 \end{vmatrix}}{\text{Det}}, \quad k' = \frac{\begin{vmatrix} x'_2 & Q_1 & z'_2 \\ x'_3 & Q_2 & z'_3 \\ x'_4 & Q_3 & z'_4 \end{vmatrix}}{\text{Det}}, \quad \text{and} \quad f' = \frac{\begin{vmatrix} x'_2 & y'_2 & Q_1 \\ x'_3 & y'_3 & Q_2 \\ x'_4 & y'_4 & Q_3 \end{vmatrix}}{\text{Det}}$$

where $Det = \begin{vmatrix} x'_2 & y'_2 & z'_2 \\ x'_3 & y'_3 & z'_3 \\ x'_4 & y'_4 & z'_4 \end{vmatrix}$

After that, the origin is translated back to the original position, so the center of the circumsphere becomes

$$h = h' + x_1$$

$$k = k' + y_1$$

$$f = f' + z_1$$

and the corresponding radius is

$$R = \sqrt{h'^2 + k'^2 + f'^2}$$

For further investigation, after determining the radius and the coordinates of the center point of the circumsphere (Figure 76), the center point of the innersphere has to be found, which is used by the node insertion algorithms. From basic geometry, it is well known that the volume of a tetrahedron is defines as

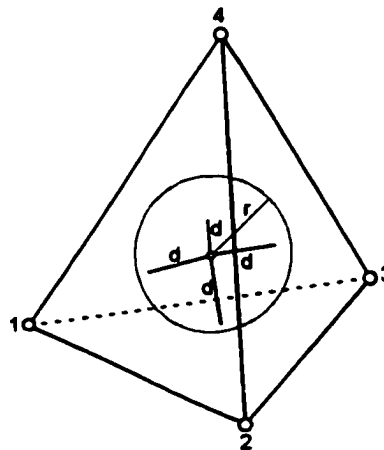


Figure 76: The innersphere of a tetrahedron.

$$V = \frac{1}{3}.d.(S_1 + S_2 + S_3 + S_4)$$

or as in other form of

$$d = r = \frac{3V}{(S_1 + S_2 + S_3 + S_4)}$$

where V is the volume and r is the innerradius of the tetrahedron. In the space analytic geometry, the distance of a point to a plane is found using the following equation;

$$d = \frac{Ax_1 + By_1 + Cz_1 + D}{\pm \sqrt{A^2 + B^2 + C^2}}$$

where the plane's equation is;

$$Ax + By + Cz + D = 0$$

In theory, to determine which sign to be used in place of the $\pm$ sign in above equation, the opposite sign of the coefficient D is used. It is found $d > 0$ if the point P and the origin of the coordinate system are in the different sides with respect to the plane. Otherwise d is less than zero. In order to obtain the coordinate of the innersphere, a set of equations is written, which results the innersphere center to the corresponding four facets of a tetrahedron by using the distance formula between a points and a plane as

$$\begin{aligned}
A_1x_1 + B_1y_1 + C_1z_1 &= \pm d\sqrt{A_1^2 + B_1^2 + C_1^2} - D_1 = F_1 \\
A_2x_1 + B_2y_1 + C_2z_1 &= \pm d\sqrt{A_2^2 + B_2^2 + C_2^2} - D_2 = F_2 \\
A_3x_1 + B_3y_1 + C_3z_1 &= \pm d\sqrt{A_3^2 + B_3^2 + C_3^2} - D_3 = F_3
\end{aligned}$$

After determining the sign of d with respect to each facets of the tetradedron, the set of equation is easily solved for the three unknowns to get the corresponding coordinates of the point P.

B. Quality of a tetrahedron

The goal of shape control in mesh generation algorithms is to produce tetrahedrons, which are as well shaped as possible. This accomplished indirectly by heuristically developed strategies for node placement in the selected tetrahedrons. For most finite element analyses, the ideal element shape is the equilateral tetrahedrons. A useful measure of how closely an arbitrary tetrahedron approaches this ideal is the ratio of its inradius (r) to circumradius (R). This ration can be easily driven as following;

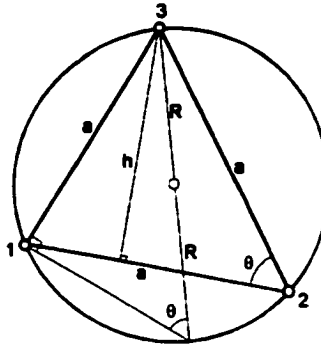


Figure 77: A side view of a tetrahedron.

It can be written that

$$\sin \theta = \frac{h}{a} = \frac{a}{2.R}$$

or

$$R = \frac{a^2}{2h}$$

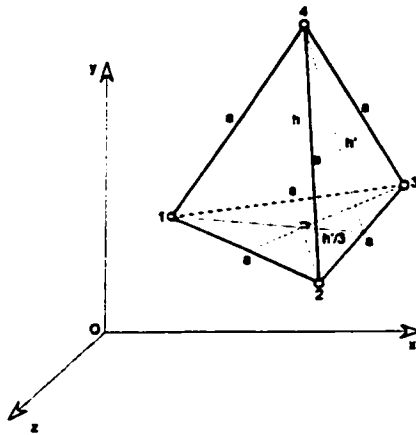


Figure 78: The geometry of tetrahedron.

From the figure 78, it can be seen that

$$h' = a \cdot \sin 60 = a \frac{\sqrt{3}}{2}$$

also

$$h'^2 = h^2 + \frac{h'^2}{9}$$

Then, solving above equation yields that

$$h = a \frac{\sqrt{6}}{3}$$

Substituting h into the circumradius equation gives,

$$R = \frac{3.a}{2\sqrt{6}}$$

One of the facet areas is calculated as $S = \frac{1}{2}h'a = a^2 \frac{\sqrt{3}}{4}$

The volume of the tetrahedron is $V = \frac{1}{3}hS = a^3 \frac{\sqrt{2}}{12}$

The innerradius would be

$$r = \frac{3V}{4S} = a \frac{\sqrt{6}}{12}$$

The quality ratio of a tetrahedron would be

$$\frac{r}{R} = \frac{1}{3} \text{ or } \delta = \frac{3r}{R}$$

where δ is between 0 and 1.

The above formula is actually computationally expensive for regular cases. There are several proposed tetrahedron quality measures available. One of them is proposed by Golias and Tsiboukis [30] as follows:

$$\delta = \frac{V}{V_{\text{optimal}}} = \frac{12V}{\sqrt{2}L_{\text{max}}^3}$$

where L_{max} is the longest edge of the tetrahedron.

However, the above formula can be used for mesh refinement purposes. For incremental node insertion method, the center coordinates of innersphere and

circumsphere of the tetrahedron has to be determined for every tetrahedron. So, the computation of the quality measure does not increase the computational time a lot.

8.2.6 NODE INSERTION INTO THE GEOMETRY

The simplest point insertion approach is to define nodes from a regular grid of points covering the domain at a specified nodal density. In order to provide for varying element sizes, a user specified sizing function can also be defined and nodes inserted until the underlying sizing function is satisfied.

Another approach is for nodes to be recursively inserted at triangle or tetrahedral centroids. Weatherill and Hassan [4] propose a tetrahedral mesh generation scheme where nodes are inserted at a tetrahedron's centroid provided. The underlying sizing function is not violated. Thus, the implicit spacing function on the boundary is continuous piecewise linear function of the cumulative chord length around the boundary.

Finally, a boundary Delaunay triangulation is computed using this expanded set of boundary nodes and spacing function in each triangular element is taken to be the linear function that interpolates the spacing value at its three nodes. The resulting implicit spacing function is then a continuous bivariate piecewise linear function which takes on the assigned spacing values at all of the boundary nodes.

For an implicit spacing function, it is convenient to modify $d(x,y)$ as each interior node is inserted. When a prospective new node is generated, the current representation of $d(x,y)$ is first evaluated at this location, and the numerical value obtained is assigned to the new node. After the triangulation is modified to absorb the new node, the assigned spacing value, together with those of neighboring nodes, defines a locally modified

spacing function on the triangles or tetrahedron formed by the new node. This process is repeated as each new node is added to the mesh, so a spacing function on each element of the current mesh is available at each stage.

An alternative approach is to define new nodes at element circumcircle/ Sphere centers as proposed by Chew[25]. When a specific order of insertion is followed, this technique is often referred to as “Guaranteed Quality” as triangles or tetrahedrons can be generated with a minimum bound on any angle in the mesh.

In this study, node insertion begins with the boundary node placement process. It is followed by the interior node placement stage. On the other hand, node insertion approach requires that information on the desired local node spacing be available at any point in the domain of the mesh in order to control the insertion process.

When an implicit spacing function is to be used, the spacing at each given boundary node may either be assigned directly by the analyst or deducted from the nodal position given. In the later case, the value of $d(x,y)$ at each given node is assigned to be the distance between that node and the nearest of the others. On the edges connecting adjacent boundary nodes, $d(x,y)$ is assumed to vary linearly and to assumed the previously computed values at the nodes at each end.

8.2.7 TWO DIMENSIONAL INTERIOR NODE INSERTION

The interior node placement strategy has similarity with the algorithm used in the literature[.]. The algorithm that has been used for the solid part in Figure 79 can be broken into five steps:

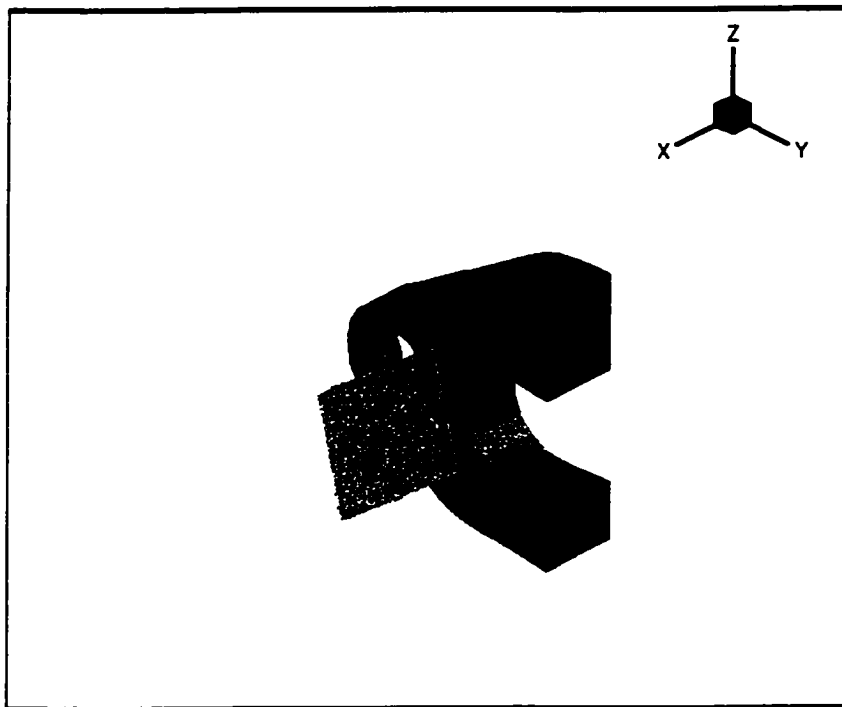


Figure 79: Non-manifold solid part to be meshed.

1. **Boundary triangulation:** Given an ordered set of boundary nodes, construct a boundary Delaunay triangulation (Figure 80).
2. **Triangle Selecting:** First eliminate the triangles that are not in the face boundaries. Then find a triangle, which contains its circumcenter, or if none can be found, take a minimum ill-shaped triangle. If none can be found go to step-6.
3. **Prospective node generation and spacing test:** Generate a prospective node inside the triangle. Compute the distance from this point to each node of the triangle. If the node is too close to any of these nodes, then, reject it.

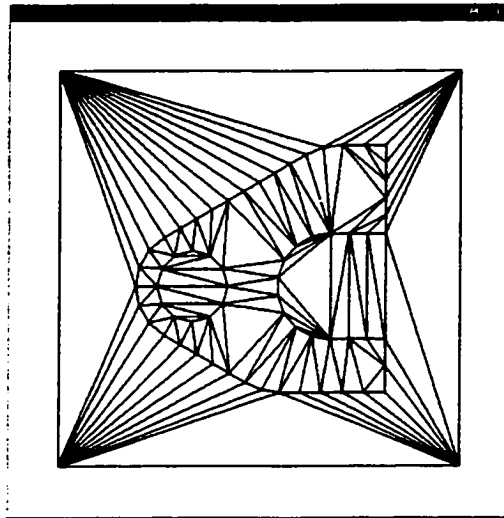


Figure 80: Boundary triangulation.

4. **Node insertion:** If the prospective node is acceptable, insert the new node and create the new Delaunay triangulation. If the target was an ill-shaped triangle, go to the step-5. If not return to step 2 (Figure 81-83).

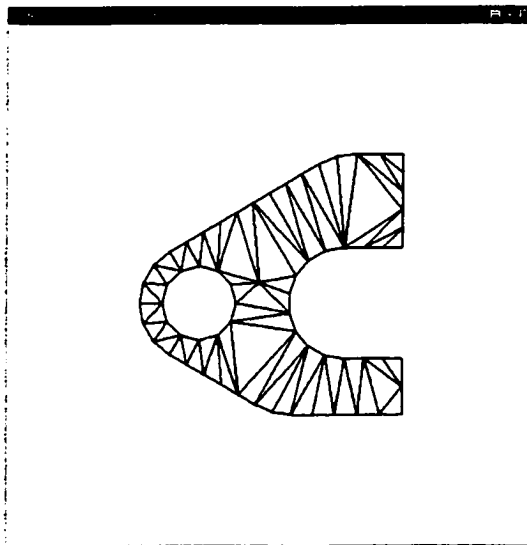


Figure 81: Inserting the first prospective node.

5. **Local mesh smoothing:** Move the interior node of the triangles around the newly inserted node to the new location using the Laplacian mesh smoothing technique. Then, go to the step-2
6. **Global mesh smoothing:** Move all the interior nodes to new locations using the Laplacian mesh smoothing technique except the boundary nodes (Shown figure 84).

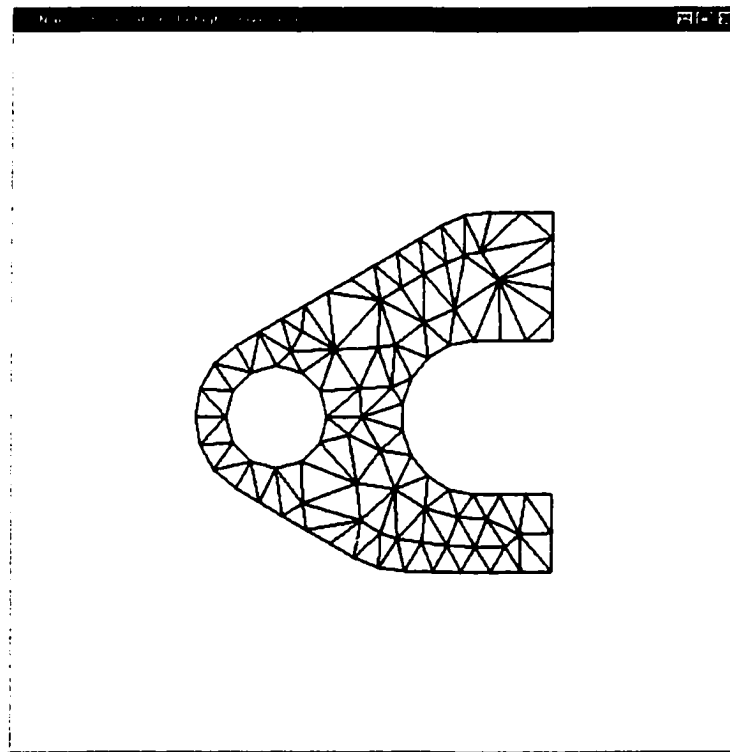


Figure 82: Inserting nodes.

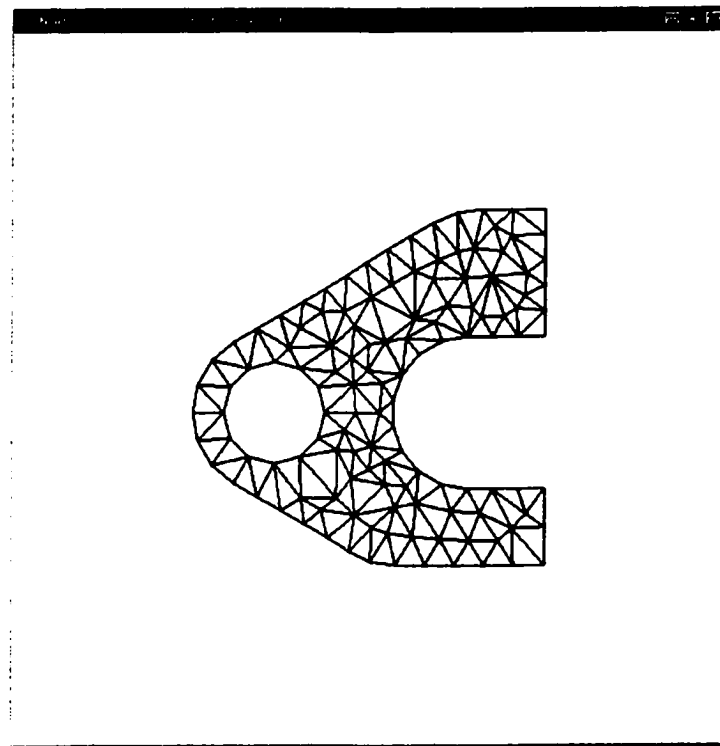


Figure 83: The final node inserting.

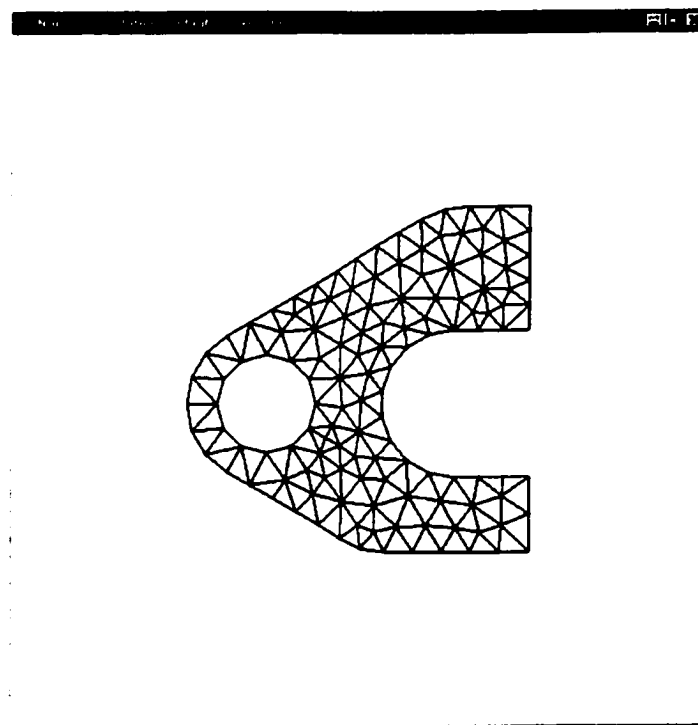


Figure 84: The final mesh after smoothing.

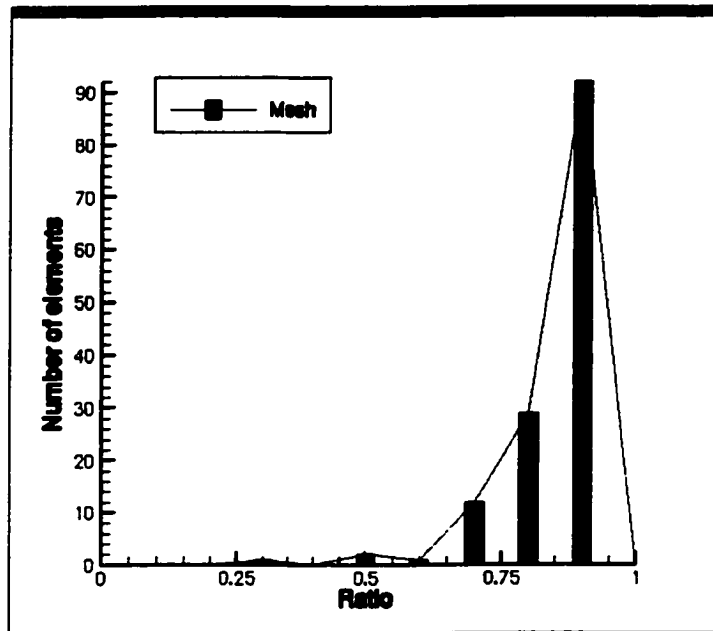


Figure 85: The quality of the resulting mesh.

The node generation process stops when no triangle can be found in which acceptable interior nodes can be placed. The number of interior nodes generated is controlled largely by spacing function.

On the other hand, shape control has to be done to select the prospective triangles. This is accomplished indirectly by heuristically developed strategies for node placement in the selected triangles for most finite element analysis. The ideal element shape is the equilateral triangle. A useful measure of how closely an arbitrary triangle approaches this ideal is the ratio of its innerradius r to circumradius R .

8.2.8 MESH GENERATION ALGORITHM IN SOLIDS

There are several algorithms used in the literature to insert nodes in Solids. The following algorithm has similar steps with reference [19], but it differs in details. The

algorithm starts with sparsely placed nodes on boundaries of the geometric model and corresponding 3-D Delaunay triangulation. Nodes are then inserted in an incremental manner. The finite element mesh is generated as following steps;

- 1. Compute a bounding box for the geometric model. Locate temporary nodes on vertices of the bounding box so that all nodes on the geometric model are included in the box. Then, form 6 tetrahedrons in the bounding box.**
- 2. Generate nodes on the vertices of the geometric model, update the Delaunay triangulation.**
- 3. Generate nodes along edges using the maximum allowable mesh size. Update the Delaunay triangulation.**
- 4. Compute the location of face nodes using 2-D Delaunay triangulation method.**
- 5. Transform the face nodes into the 3-D space. Insert the face nodes randomly and update the Delaunay triangulation.**
- 6. Remove the tetrahedrons corresponding to empty regions before internal node insertion for preventing unnecessary node insertion.**
- 7. Insert the region nodes inside the solid region, and update the triangulation.**
- 8. Remove the tetrahedrons corresponding to empty regions after internal node insertion finished.**
- 9. Smooth the tetrahedrons inside the solid by moving the internal nodes only.**

8.2.9 THREE DIMENSIONAL INTERIOR NODE GENERATION

Region nodes are inserted after the nodes of all faces in the solid model are generated and inserted. In other words, boundary tetra-faces are already identified and marked. Region nodes are then inserted so that these boundary tetra-faces do not change.

The procedure for region node insertion is as follows;

1. Select a target tetrahedron, which is larger than the specified size. If no large tetrahedron is found, take an ill-shape tetrahedron. If node can be found go to Step-4.
2. Computer the new node location, and if the new node is acceptable insert the node, and update the 3-D Delaunay triangulation go to Step-3, if not go to Step-1.
3. If any of resultant tetrahedrons is an ill-shaped tetrahedron, apply a local smoothing by moving only the newly inserted node among the all neighboring tetrahedrons' nodes. Then go to Step-1. If not go to Step-1.
4. Smooth the mesh by using the Laplacian mesh smoothing technique.

The node location P_{node} is computed by interpolation between the circumcenter C_c and the innercenter C_i of the tetrahedron in the following way:

$$P_{node} = uC_c + (1 - u)C_i$$

where

$$u = \begin{cases} 1.0 \rightarrow \delta > 0.8 \\ 0.5 \rightarrow \delta < 0.8 \end{cases}$$

Chapter 9

Implementation

9.1 THE IMPLEMENTED TOPOLOGY

In this study, a modified version of non-manifold topology has been implemented. For this reason, non-manifold data structures have been simplified to a minimal degree for which the corresponding mesh generation task can be done. Therefore, the modification has been carried away to the state in which the models include only the links to other models, the regions keep the links to their model and contain the structure of the corresponding shells.

```
Model:
struct model {
    int                index;
    int                maxindex;
    struct topology_list header_reg;
};
```

```
Regions:
struct regions {
    int                index;
    struct model        *mp;
    struct topology_list tp;
    struct topology_list rshell;
};
```

```
Struct topology_list {
    long                index;
    struct topology_list *next;
    struct topology_list *prev;
};
```

The shell structure contains the pointer to its region, and it includes the face-uses and shells.

Shells:

```
struct shell {  
    struct regions      *rg;  
    struct topology_list tp;  
    struct topology_list fu;  
    int                 index;  
};
```

The main focus has been made on the face-use structure. In this structure, all information about edges, loops, and finite element mesh nodes are kept. The boundary and inside nodal points are stored separately. The face-uses store only the nodes inside the face.

Face-use:

```
struct faceuse {  
    struct topology_list tp;  
    struct shell        *sh;  
    int face_number;  
    int face_edge_number;  
    int edge_id_number[];  
    int loop_number;  
    int loop_edge_number[][];  
    int loop_type[];  
    int face_point_number;  
    int face_inside_point_id[];  
};
```

The nodes on the boundary edges are stored in edge-use structure.

```
Edge-use:
    struct edgeuse {
        struct    topology_list    tp;
        int       edge_type;
        int       edge_number;
        int       point_number;
        int       point_id_number;
        int       first_point;
        int       last_point;
    };
```

Such a structure allows easy maintenance of the necessary information. If there is any need, the other topological information can be derived without any problem by traversing the radial edge structure.

The other advantage of this modified topology is the fact that topological information is incorporated into the finite element mesh information. The resultant topology has less storage to maintain, and gives a good transition between the topology and geometrical information.

9.2 CODE DEVELOPMENT

The software called “NMT-MESH” was developed during this research. The program originally was designed for PC environment running under Windows NT operating system, however, it may be run in unix workstations with a few adjustments.

NMT-MESH has a simple flow diagram shown in Figure 85. The program consists of four major modules.

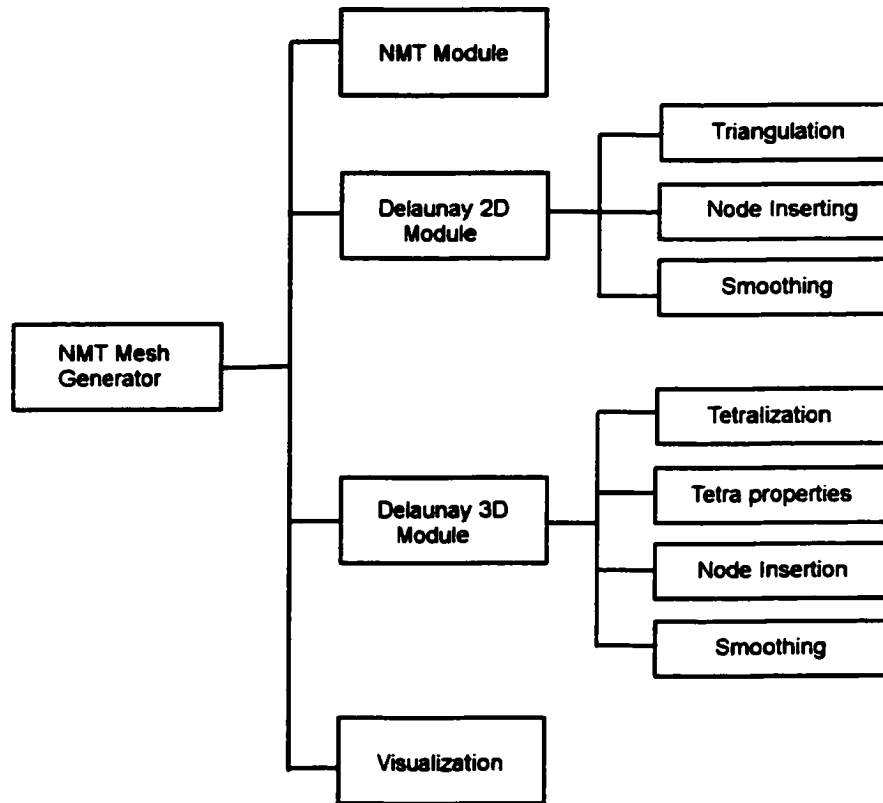


Figure 86. Flow diagram for NMT-mesh.

NMT Module:

This module consists of more than 3000 lines of code written in C language. The module utilizes several topological and geometrical subroutines from Unigraphics Application Programming Interface (API[®]) library [20] in addition to the subroutines developed to represent the non-manifold topology. These routines were extensively used in developing the non-manifold geometries used in the examples.

In the beginning of the process, the module loads the 3D model first, extracts all topological information, and forms the new non-manifold data structure. Then, it lets the user introduce 1D or 2D objects next to the model. After that, the module applies the desired Boolean operations, explained in Chapter 6, to update the model and topology. Then, the module creates nodes on the boundary edges of the non-manifold model, and stores them in the topological structure.

After finishing the node generation on the boundary, the module writes two types of output files for each face. The first type file contains the nodal identification number and the corresponding coordinates. The second type of the file stores arrays of nodes, which form the boundary loops in counterclockwise direction.

Delaunay-2D:

This module consists of three subroutines, and contains 4080 lines of code written in Fortran and C language. The triangulation subroutine first takes the files written by NMT module one by one and generates the boundary triangulation incrementally as explained in Chapter 8.

By using the algorithm detailed in Chapter 8, the node insertion subroutine inserts new nodes. In the mean time, local mesh smoothing technique is used to improve the mesh quality. This process is carried away until no more nodes can be inserted into the boundary.

After the node insertion process is done, the mesh global smoothing procedure is applied by the smoothing subroutine. Then, the Delaunay-2D module generates two

output files for every face in the model, which include the connectivity information for each triangle and coordinates of nodes with their global node identification numbers.

Delaunay-3D:

Like 2D module, this module consists of three subroutines. It contains a total 5491 lines of code written in Fortran and C languages. The tetralization subroutine employs another subroutine for the calculation of properties of selected tetrahedrons. The module reads all files written by Delaunay-2D module, and assembles them in one unique file. Then, it generates the boundary tetralization incrementally since there are only boundary nodes at the beginning. The algorithm, which does this tetralization, is explained in Chapter 8.

Having all initial boundary tetrahedrons generated, the tetrahedons in the empty regions and on the boundary surfaces with zero volume are eliminated before any node insertion inside the boundary. After that, the new node insertion algorithm explained in Chapter 7 is used to place the new nodes inside the boundary. The node insertion subroutine does that until no more nodes can be inserted.

In the meantime, local smoothing is employed to produce tetrahedrons with less distortion that are to be selected in the ongoing process. The final smoothing module applies the Lagrangian smoothing to the mesh to improve the quality of the tetrahedrons.

9.2 CASE STUDY ONE

For this first case, a manifold model is meshed. As explained in section Chapter 8 and shown in Figure 87, the program first forms the bounding box of the geometry being meshed.

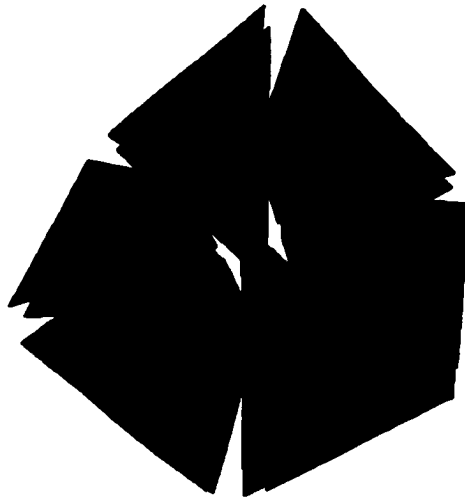


Figure 87: The bounding box.

Then, the nodes are generated at the vertices of the model and on the edges as shown in figure 88.



Figure 88: The nodes on the vertices and edges.

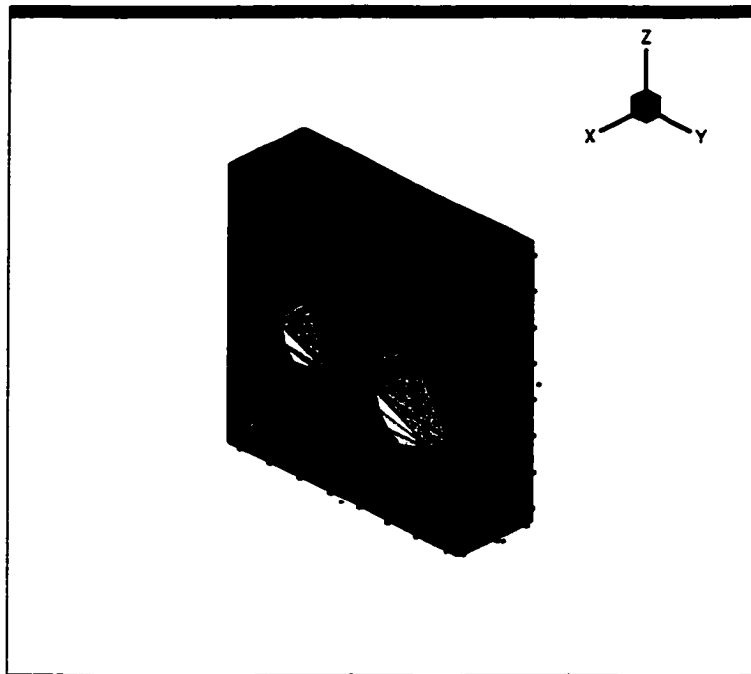


Figure 89: The primary surface triangulation.

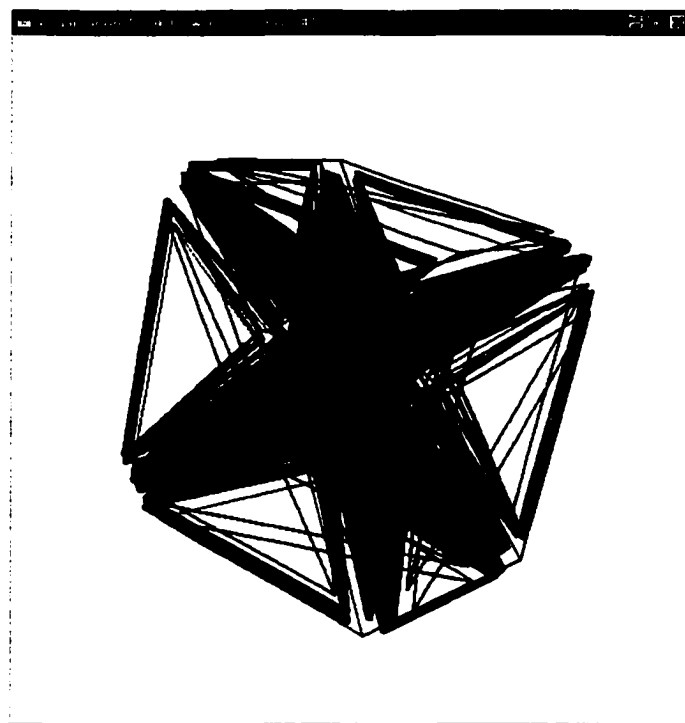


Figure 90: Updating the 3D Delaunay triangulation.

Then, the Delaunay triangulation is updated as shown in figure 90. After eliminating the bounding tetrahedrons, the mesh remains as in figure 91. Notice that, we have still some tetrahedrons that don't belong to the solid part.

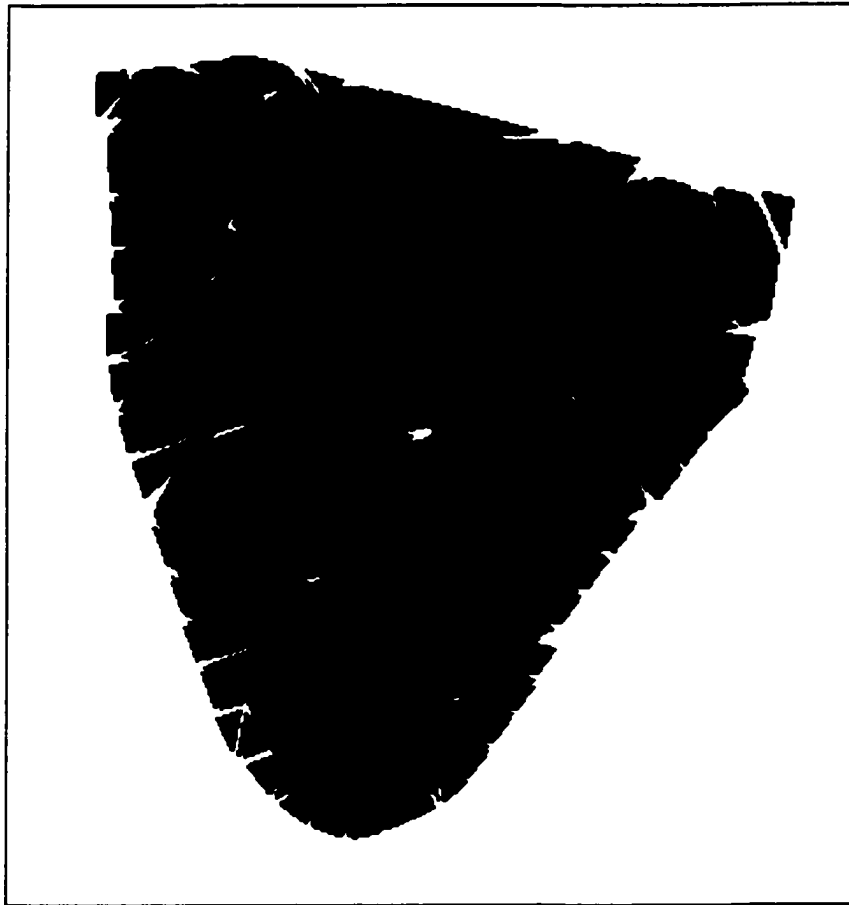


Figure 91: The mesh without nodes on faces and interior of the solid.

In parametric space, we insert the surface nodes for each face, then use the Laplacian smoothing, which moves the every internal node to the center of its surrounding triangles, to improve the mesh quality as shown in Figure 92-145.

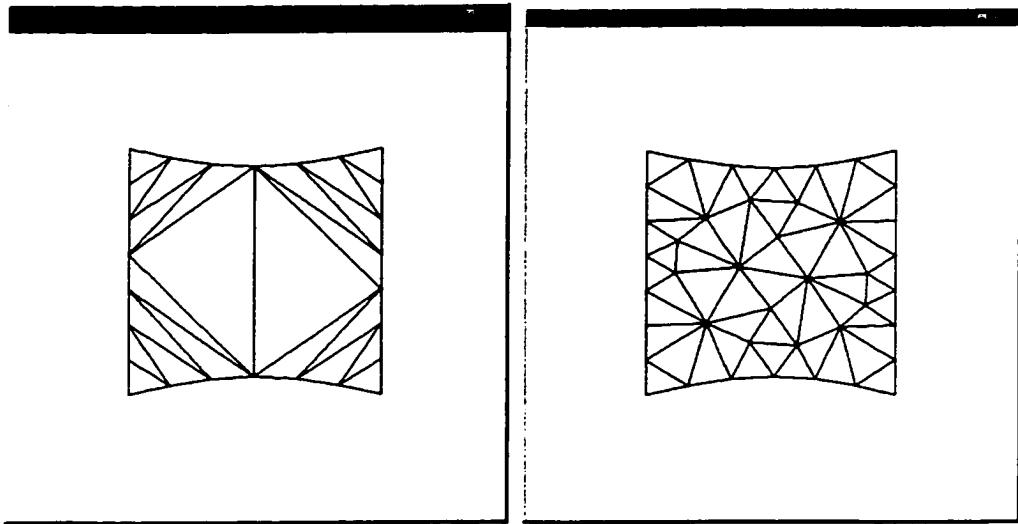


Figure 92: The face #1's triangulation in 2D parametric space.

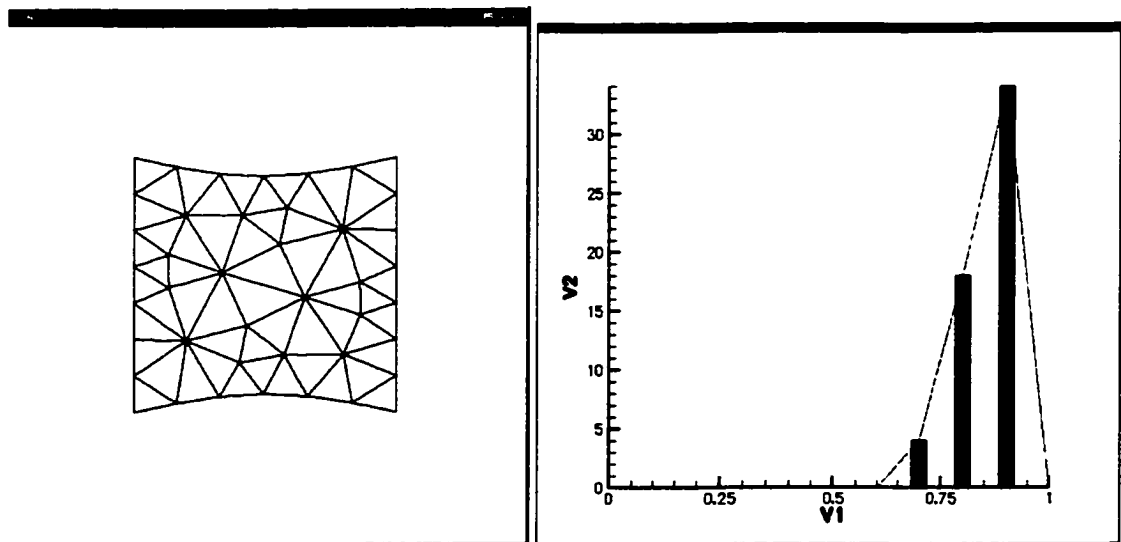


Figure 93: The face #1's smoothing and quality of the mesh (v1:ratio, v2:number of elements).

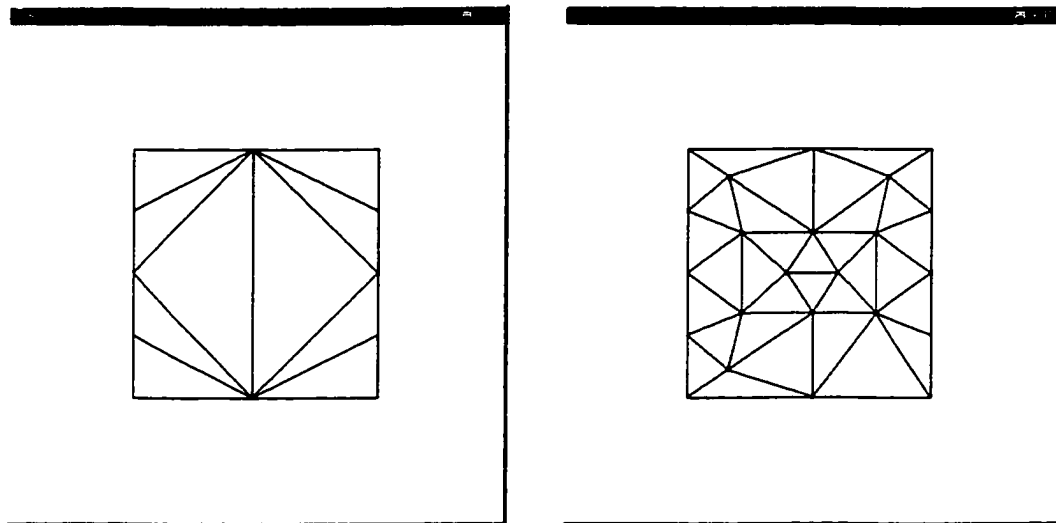


Figure 94: *The face #2's triangulation in 2D parametric space.*

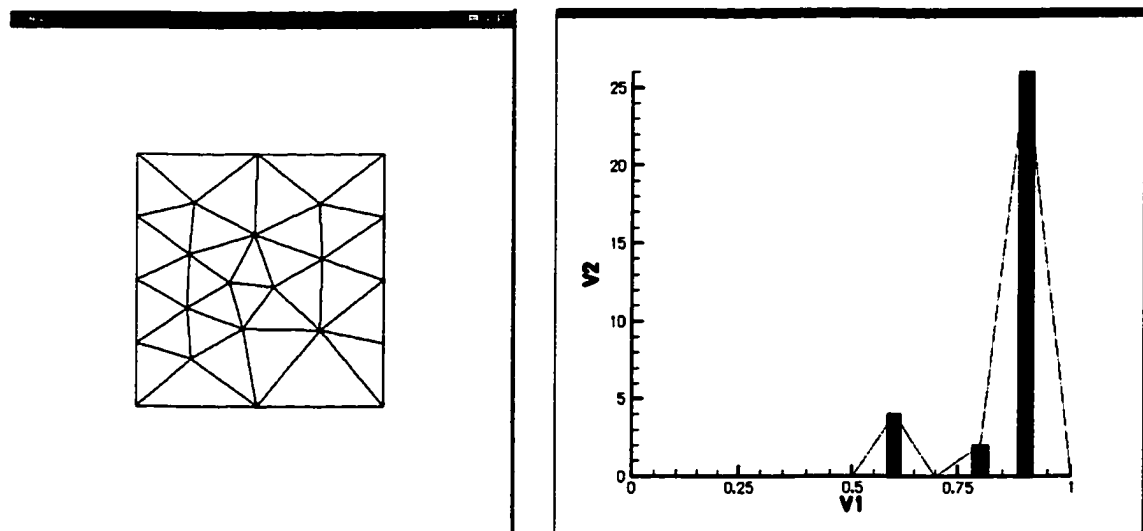


Figure 95: *The face #2's smoothing and quality of the mesh (v1:ratio, v2:number of elements).*

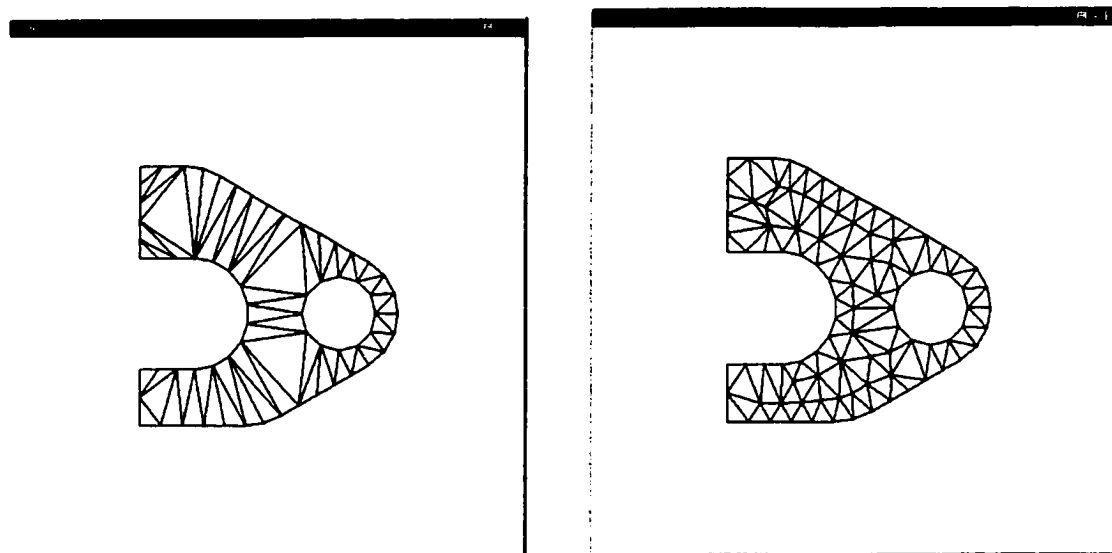


Figure 96: The face #3's triangulation in 2D parametric space.

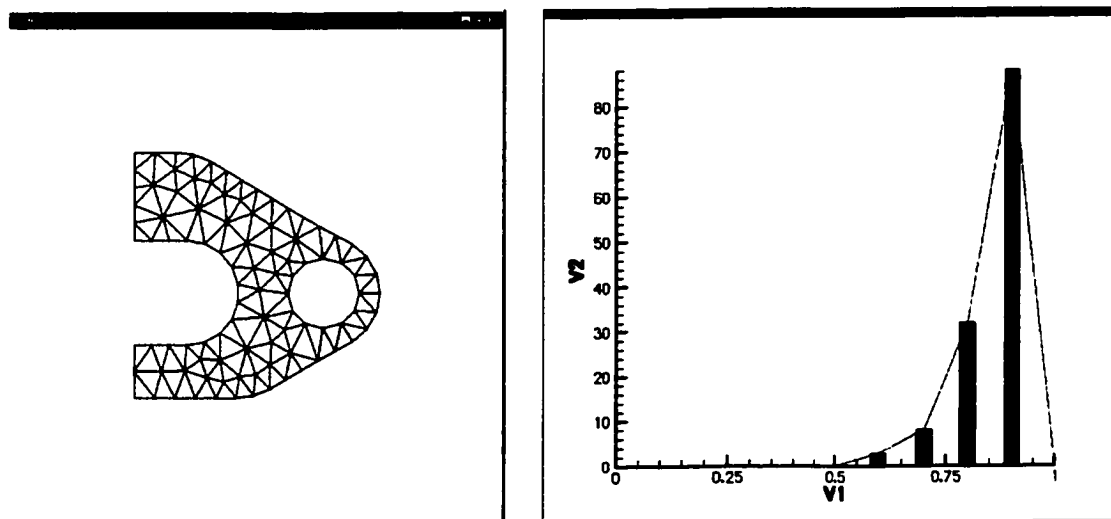


Figure 97: The face #3's smoothing and quality of the mesh (v1:ratio, v2:number of elements).

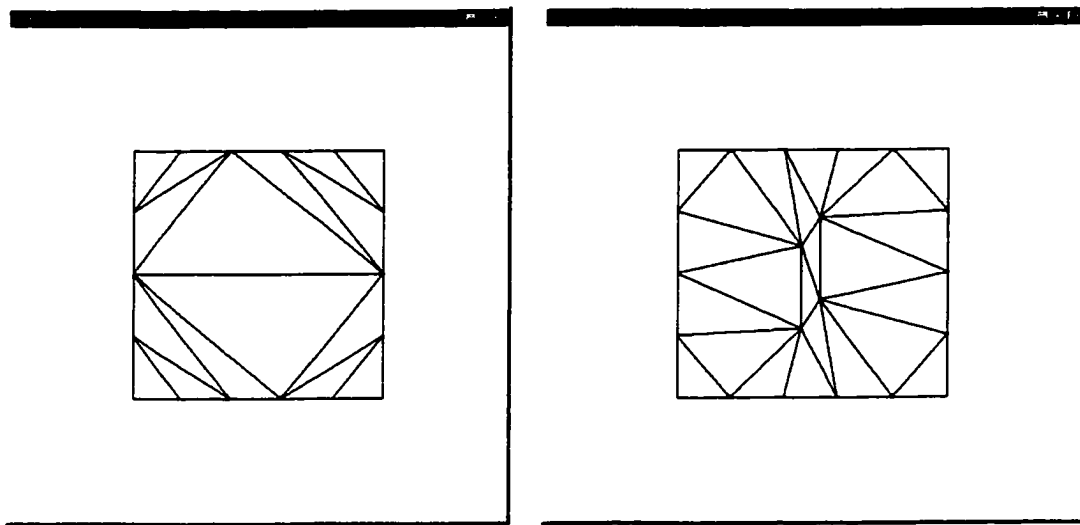


Figure 98: *The face #4's triangulation in 2D parametric space.*

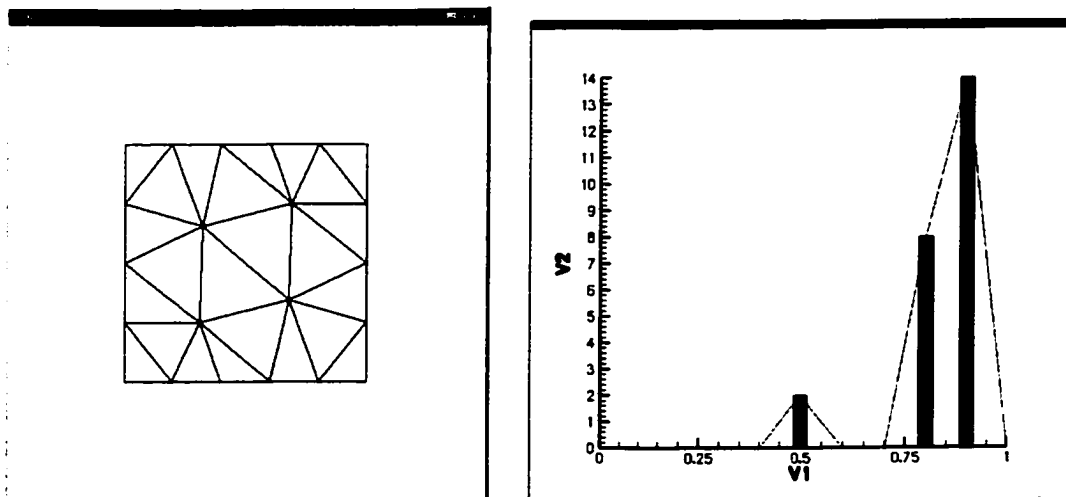


Figure 99: *The face #4's smoothing and quality of the mesh (v1:ratio, v2:number of elements).*

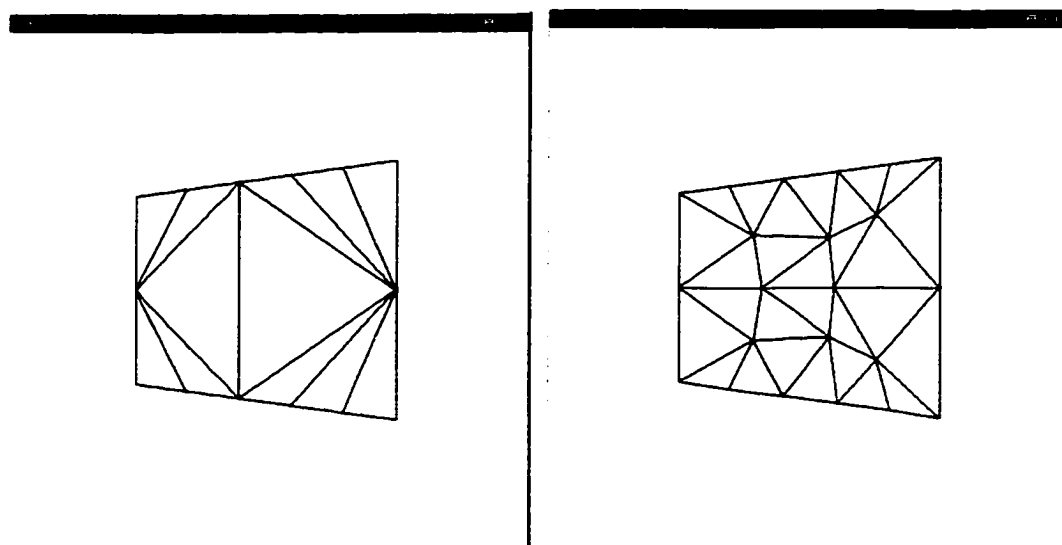


Figure 100: The face #5's triangulation in 2D parametric space.

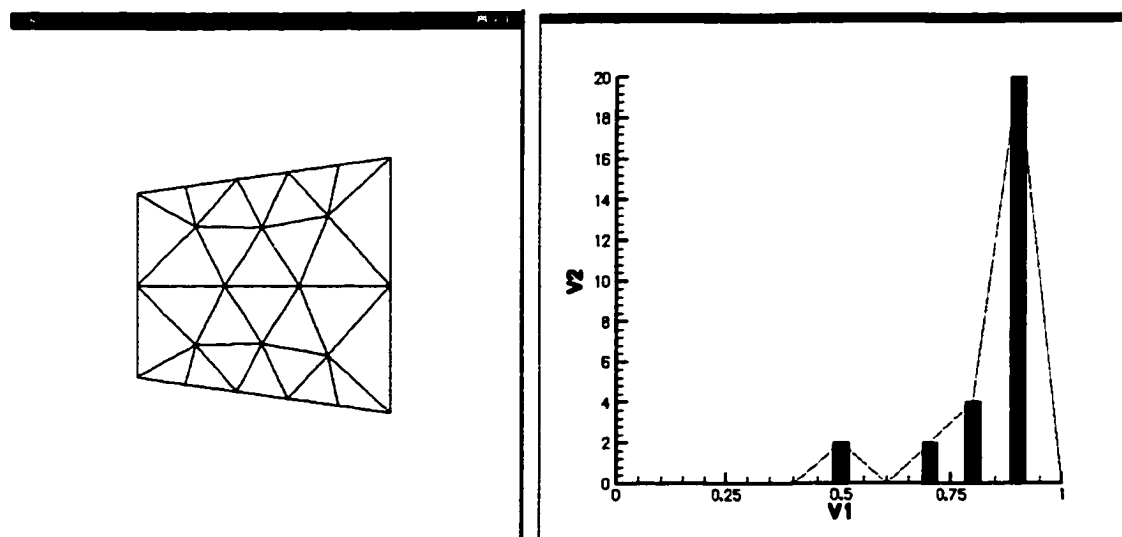


Figure 101: The face #5's smoothing and quality of the mesh (v1:ratio, v2:number of elements).

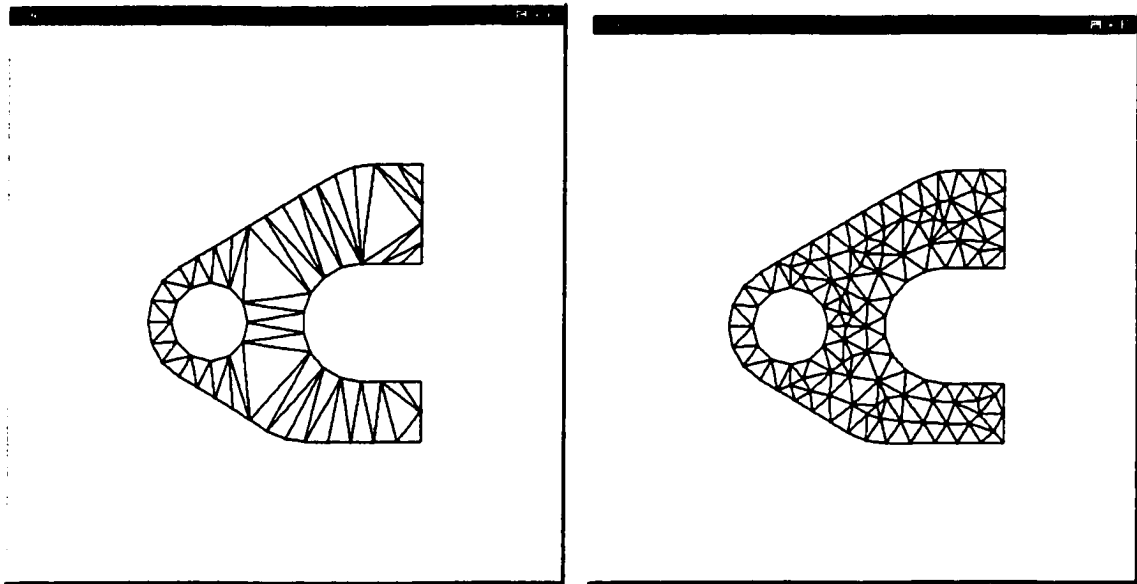


Figure 102: The face #6's triangulation in 2D parametric space.

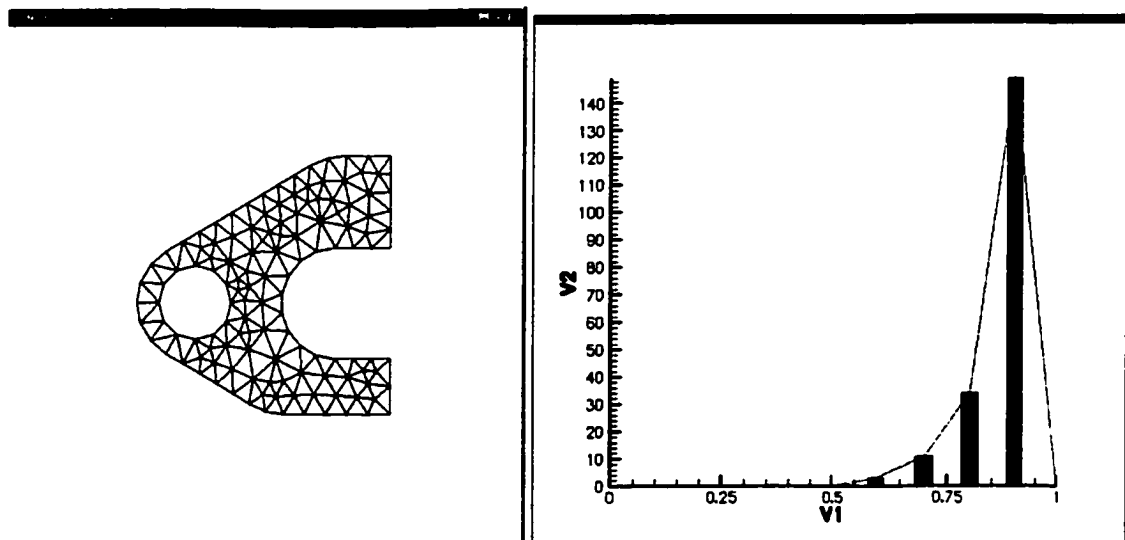


Figure 103: The face #6's smoothing and quality of the mesh (v1:ratio, v2:number of elements).

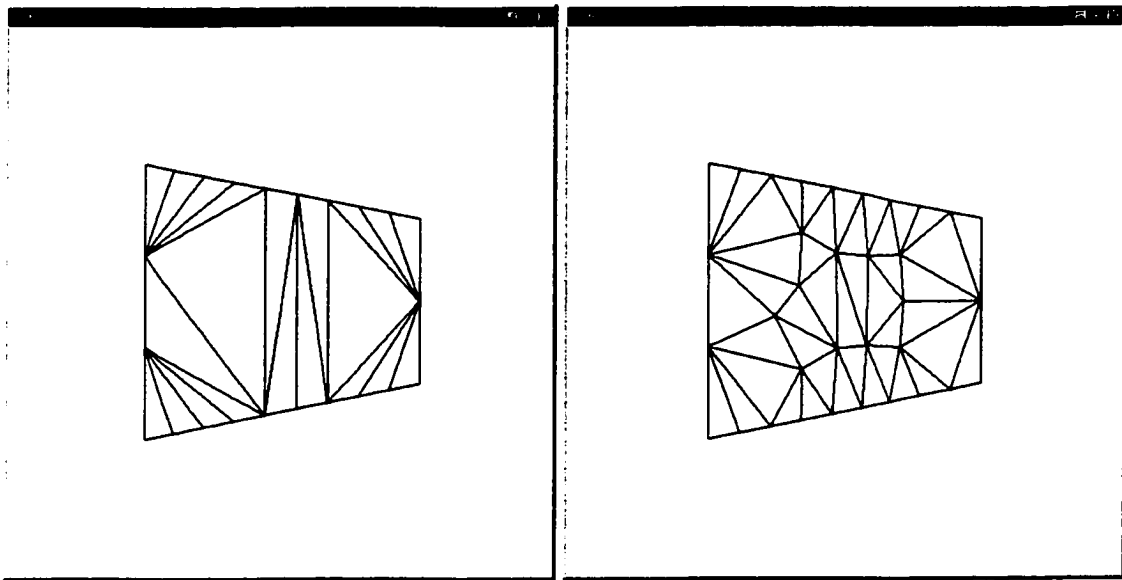


Figure 104: *The face #7's triangulation in 2D parametric space.*

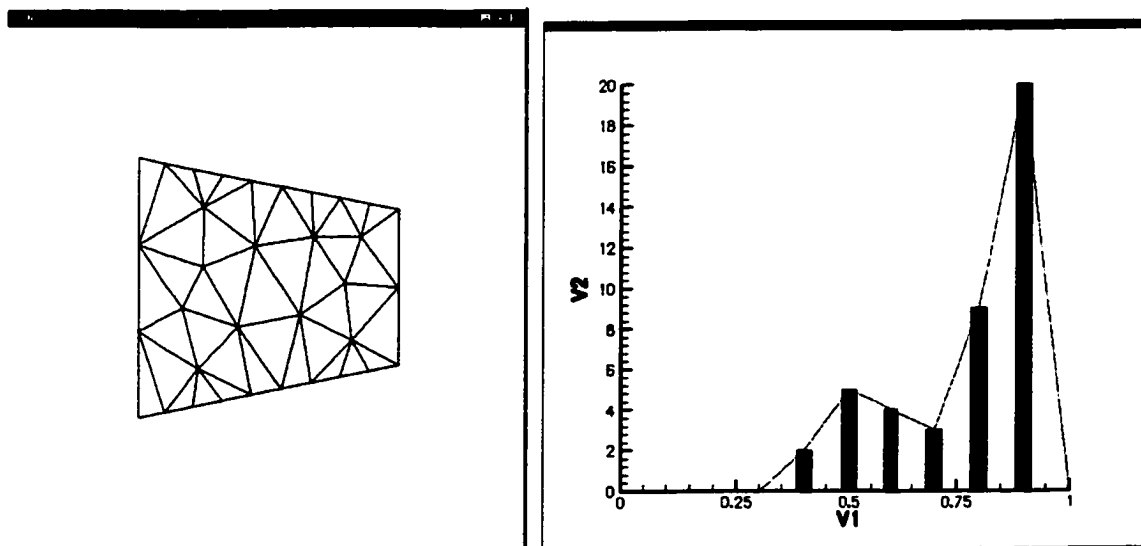


Figure 105: *The face #7's smoothing and quality of the mesh (v1:ratio, v2:number of elements).*

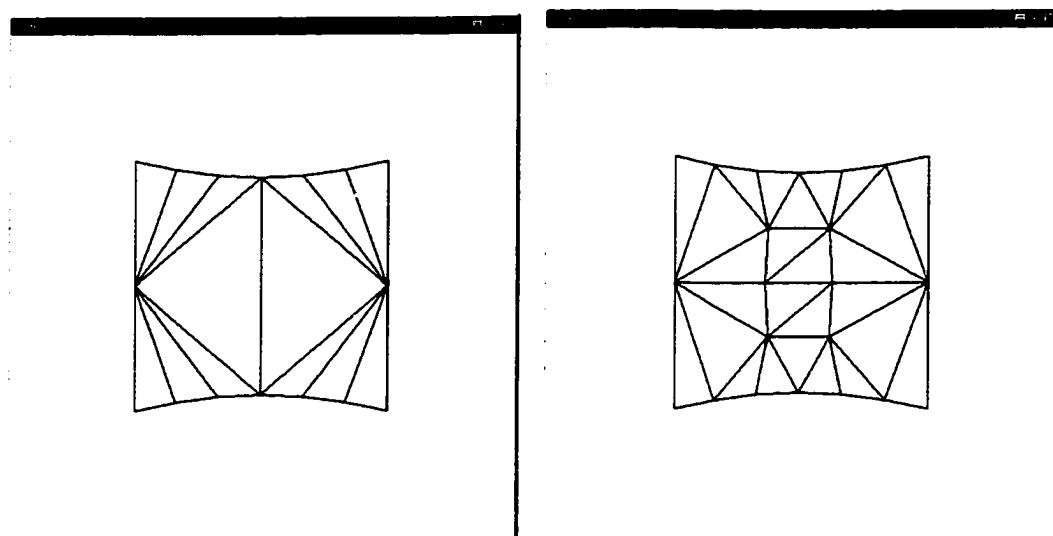


Figure 106: The face #8's triangulation in 2D parametric space.

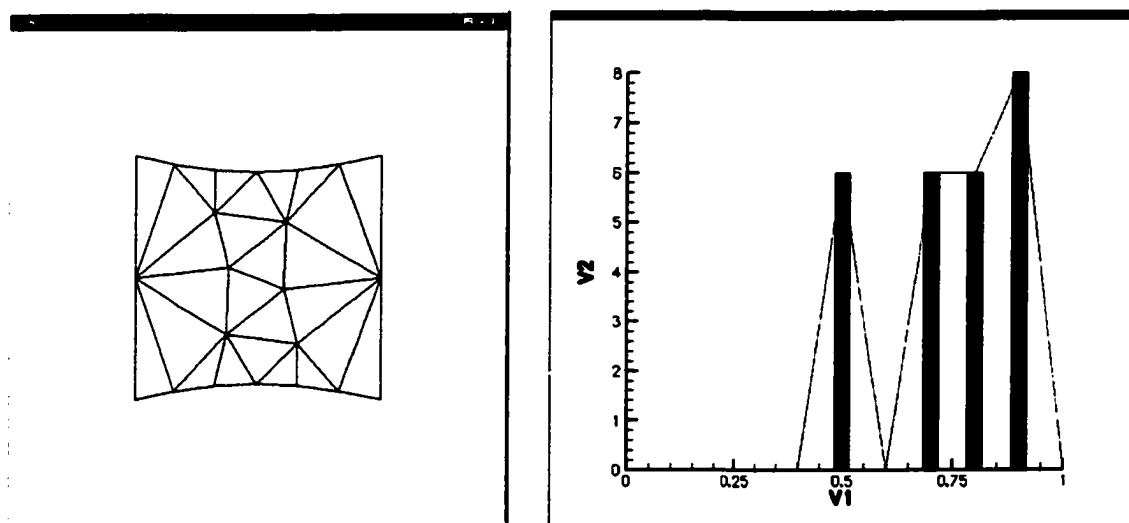


Figure 107: The face #8's smoothing and quality of the mesh (v1:ratio, v2:number of elements).

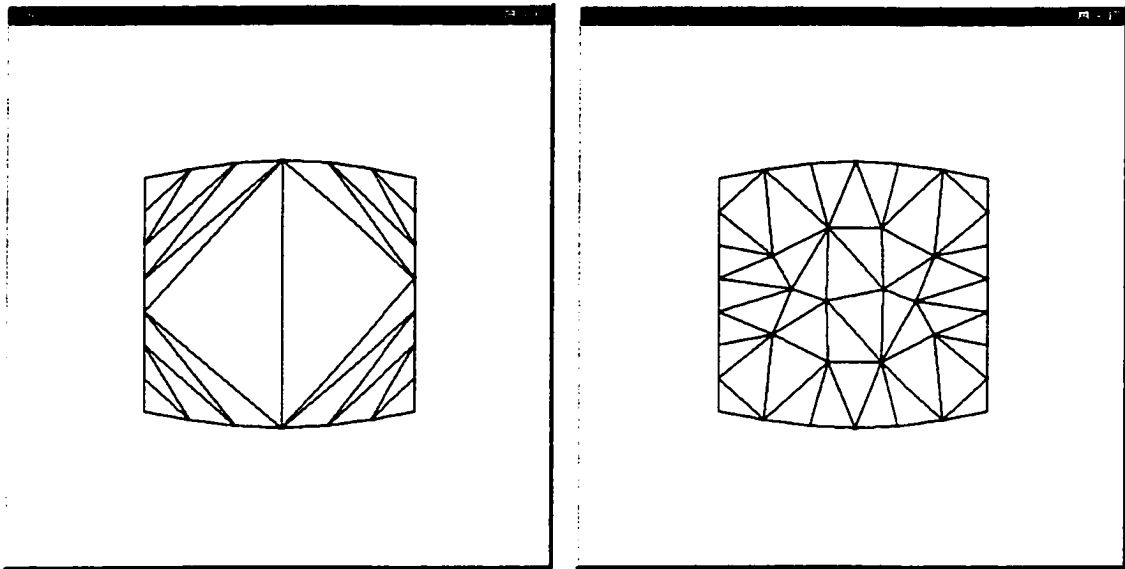


Figure 108: The face #9's triangulation in 2D parametric space.

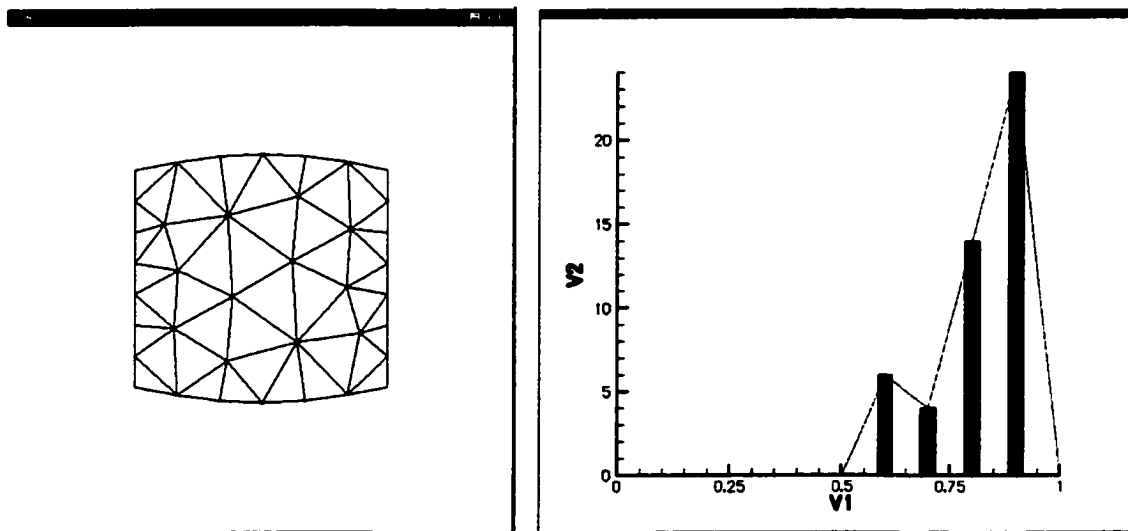


Figure 109: The face #9's smoothing and quality of the mesh (v1:ratio, v2:number of elements).

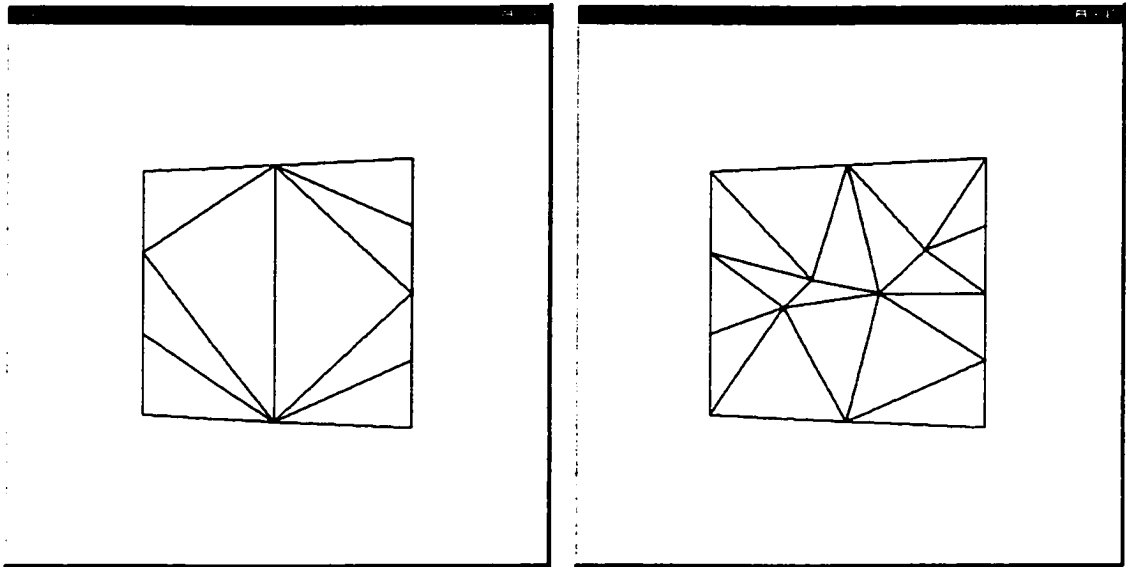


Figure 110: The face #10's triangulation in 2D parametric space.

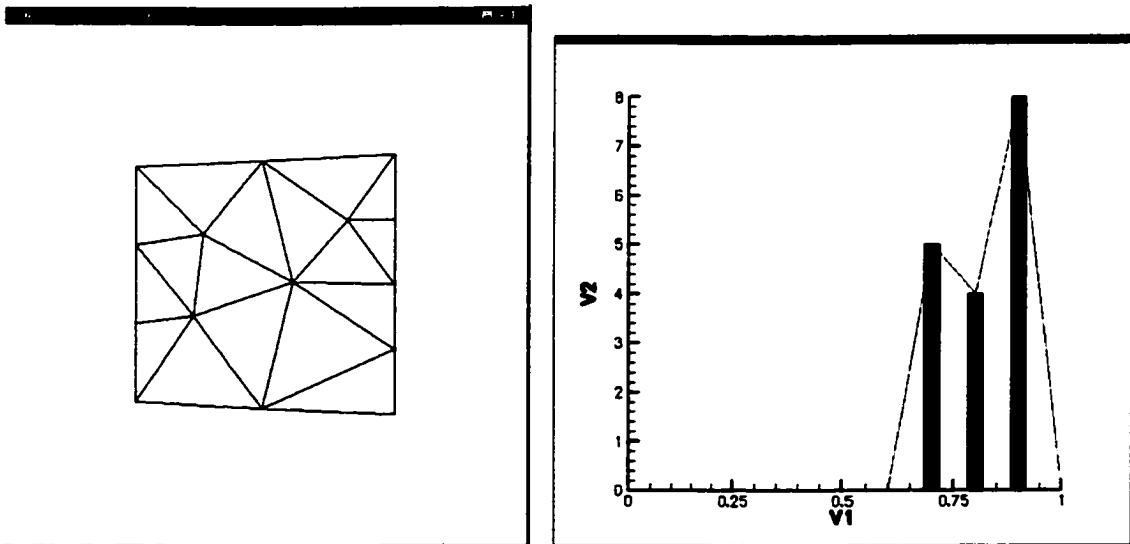


Figure 111: The face #10's smoothing and quality of the mesh (v1:ratio, v2:number of elements).

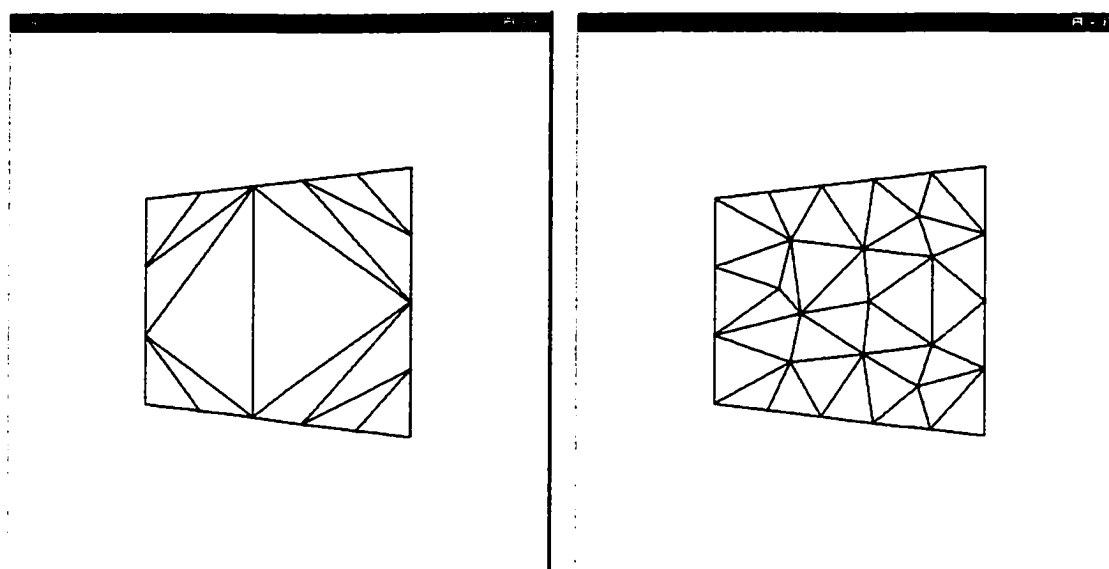


Figure 112: The face #11's triangulation in 2D parametric space.

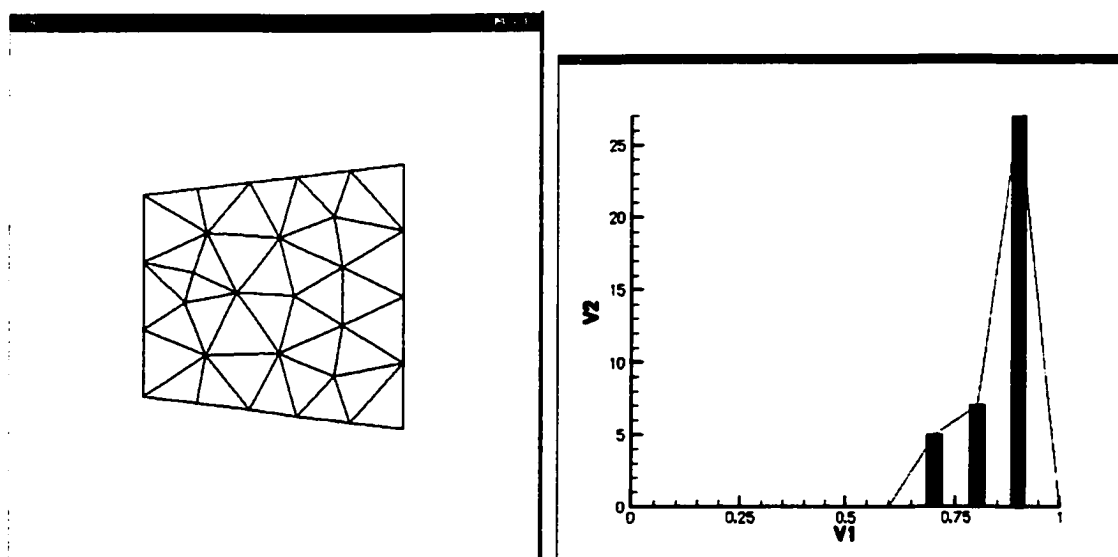


Figure 113: The face #11's smoothing and quality of the mesh (v1:ratio, v2:number of elements).

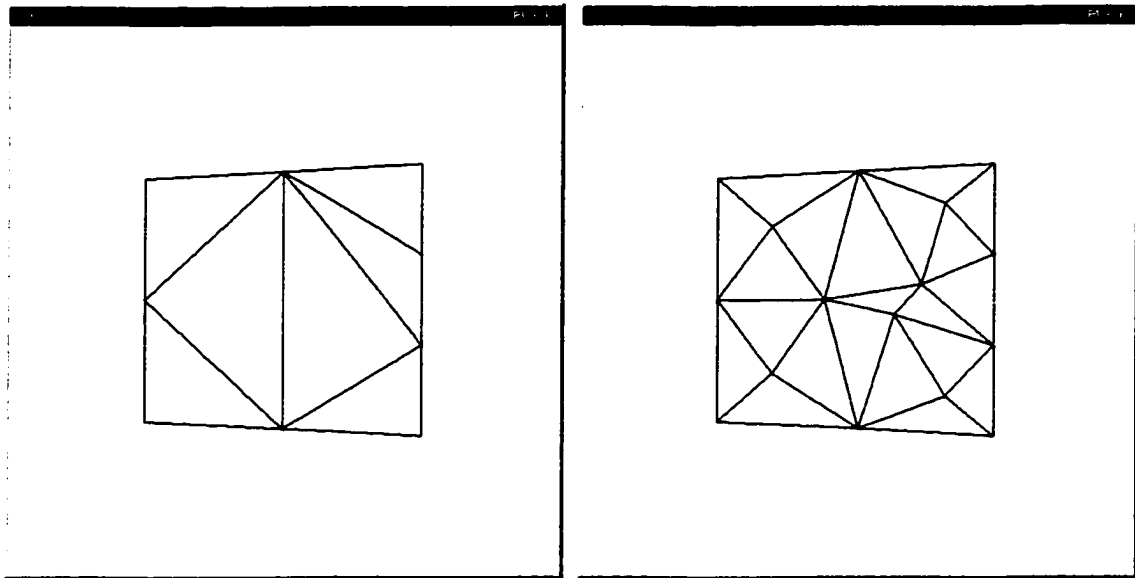


Figure 114: The face #12's triangulation in 2D parametric space.

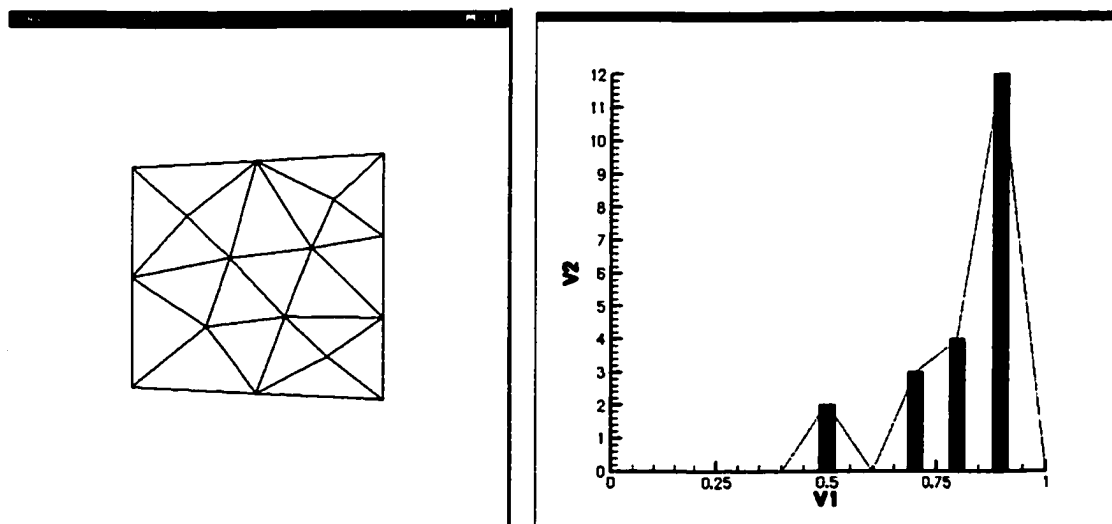


Figure 115: The face #12's smoothing and quality of the mesh (v1:ratio, v2:number of elements).

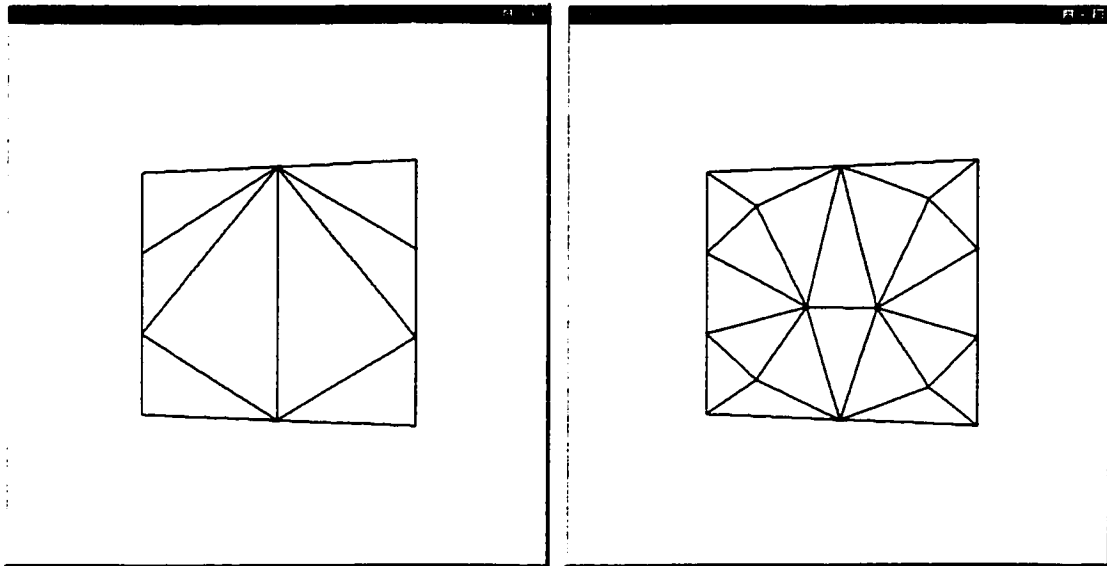


Figure 116: The face #13's triangulation in 2D parametric space.

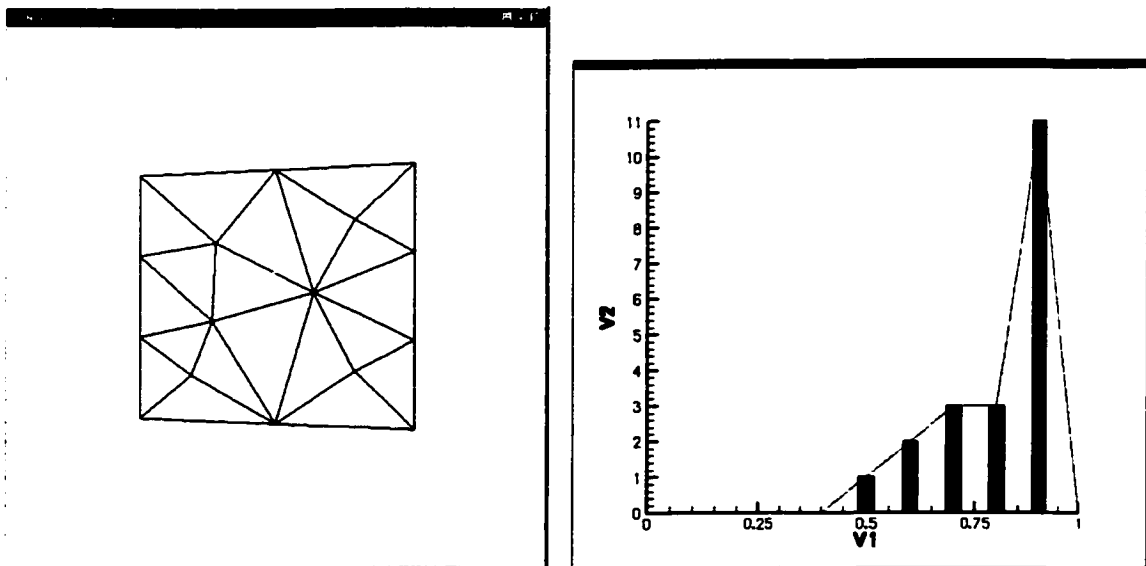


Figure 117: The face #13's smoothing and quality of the mesh ($v1$:ratio, $v2$:number of elements).

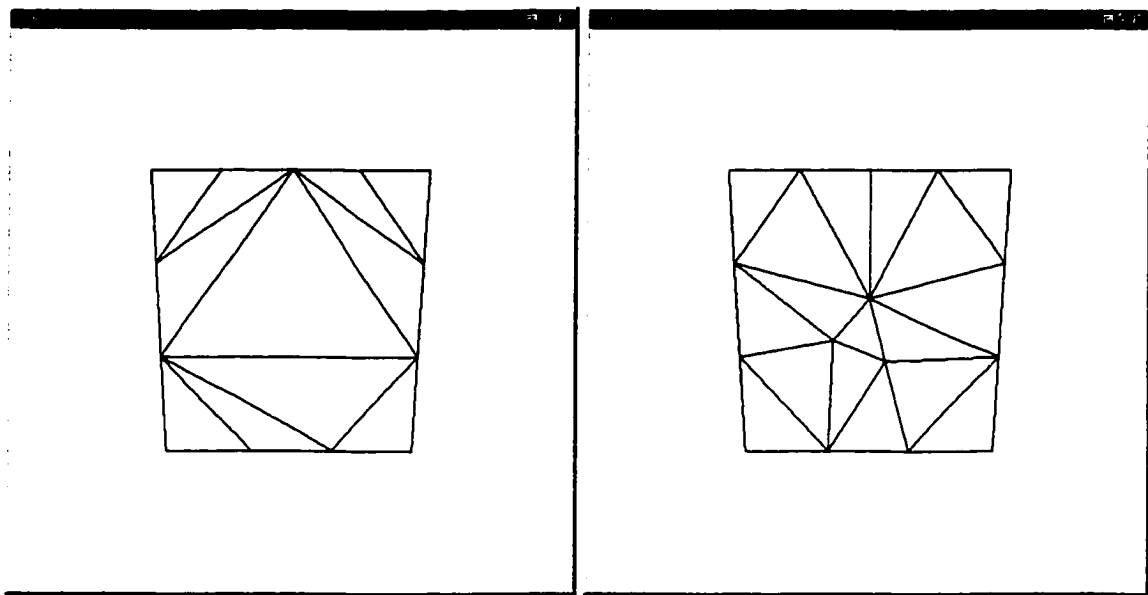


Figure 118: The face #14's triangulation in 2D parametric space.

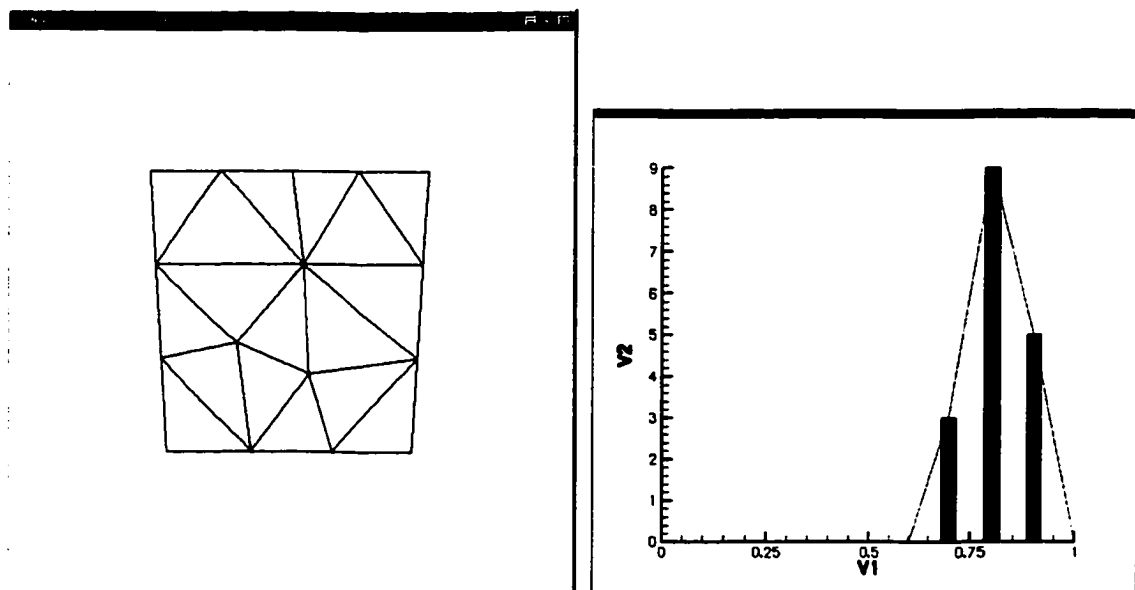


Figure 119: The face #14-15's smoothing and quality of the mesh (v1:ratio, v2:number of elements).

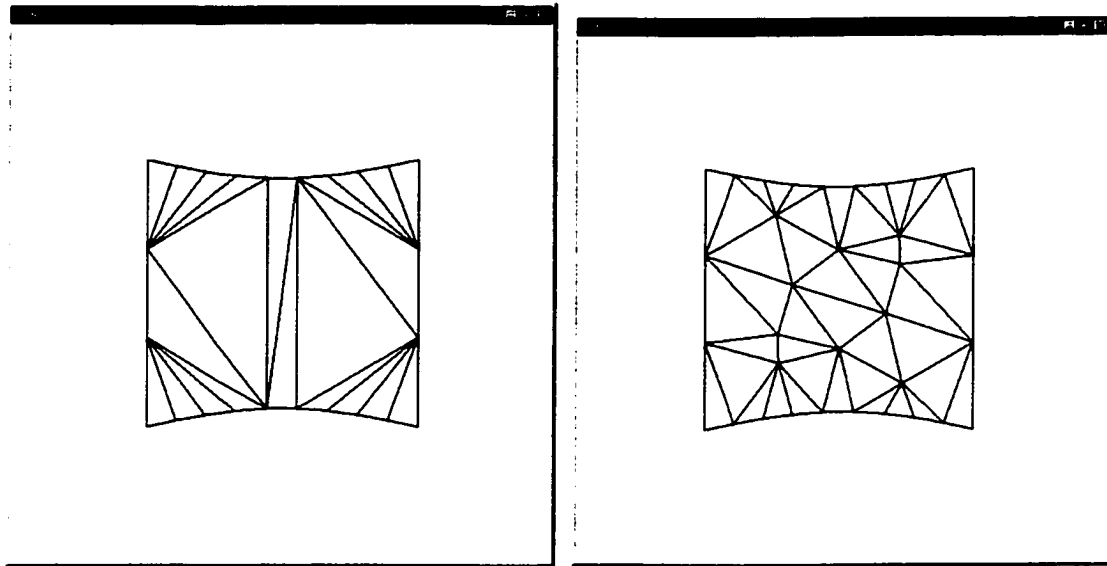


Figure 120: The face #16's triangulation in 2D parametric space.

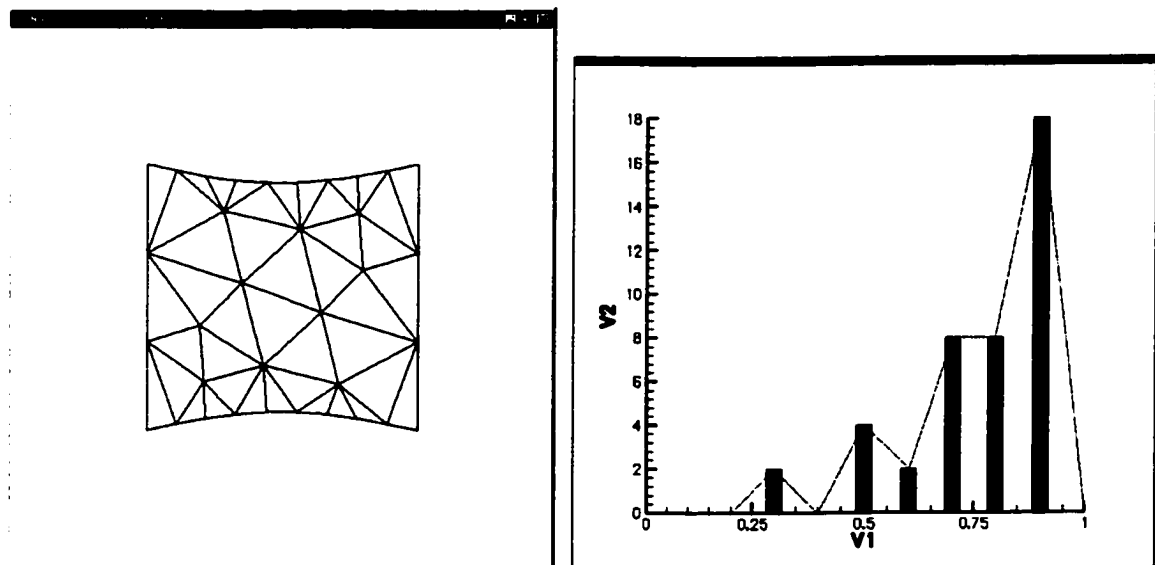


Figure 121: The face #16's smoothing and quality of the mesh (v1:ratio, v2:number of elements).

After completing the surface mesh of the model in the parametric space, the nodes on the faces are mapped through surface parametric equations to the 3D model to get the cartesian coordinates for each node.

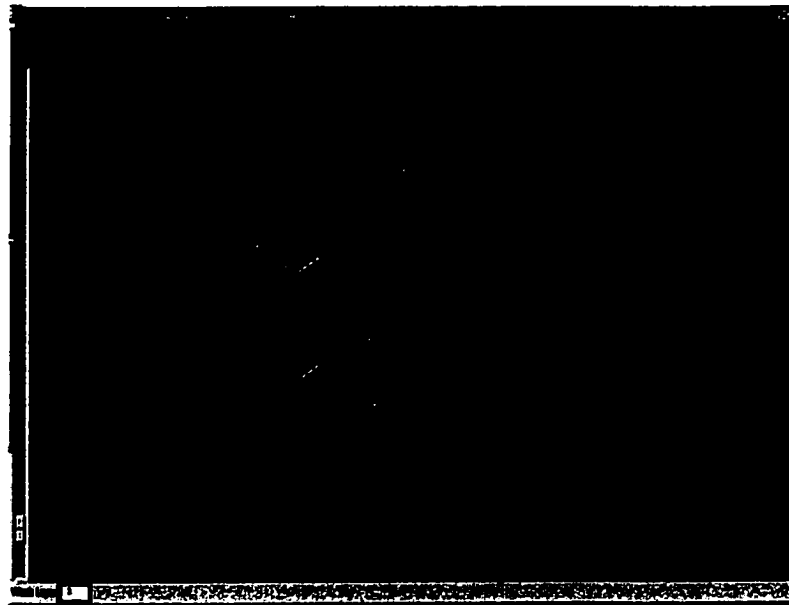


Figure 122: The node in the 3D model.

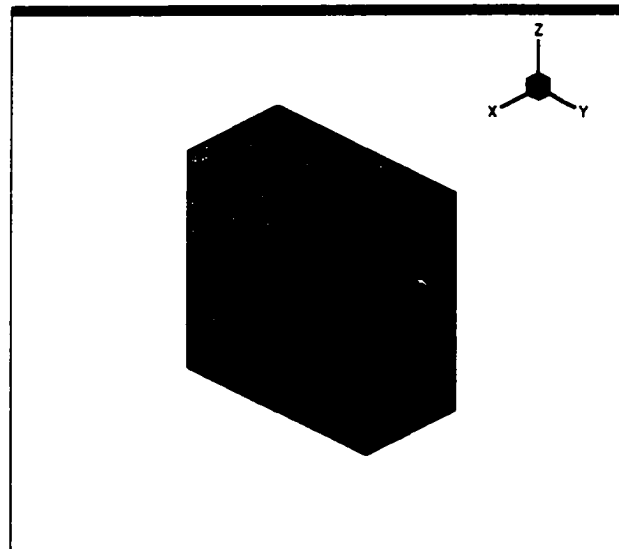


Figure 123: One surface mesh in 3D space.

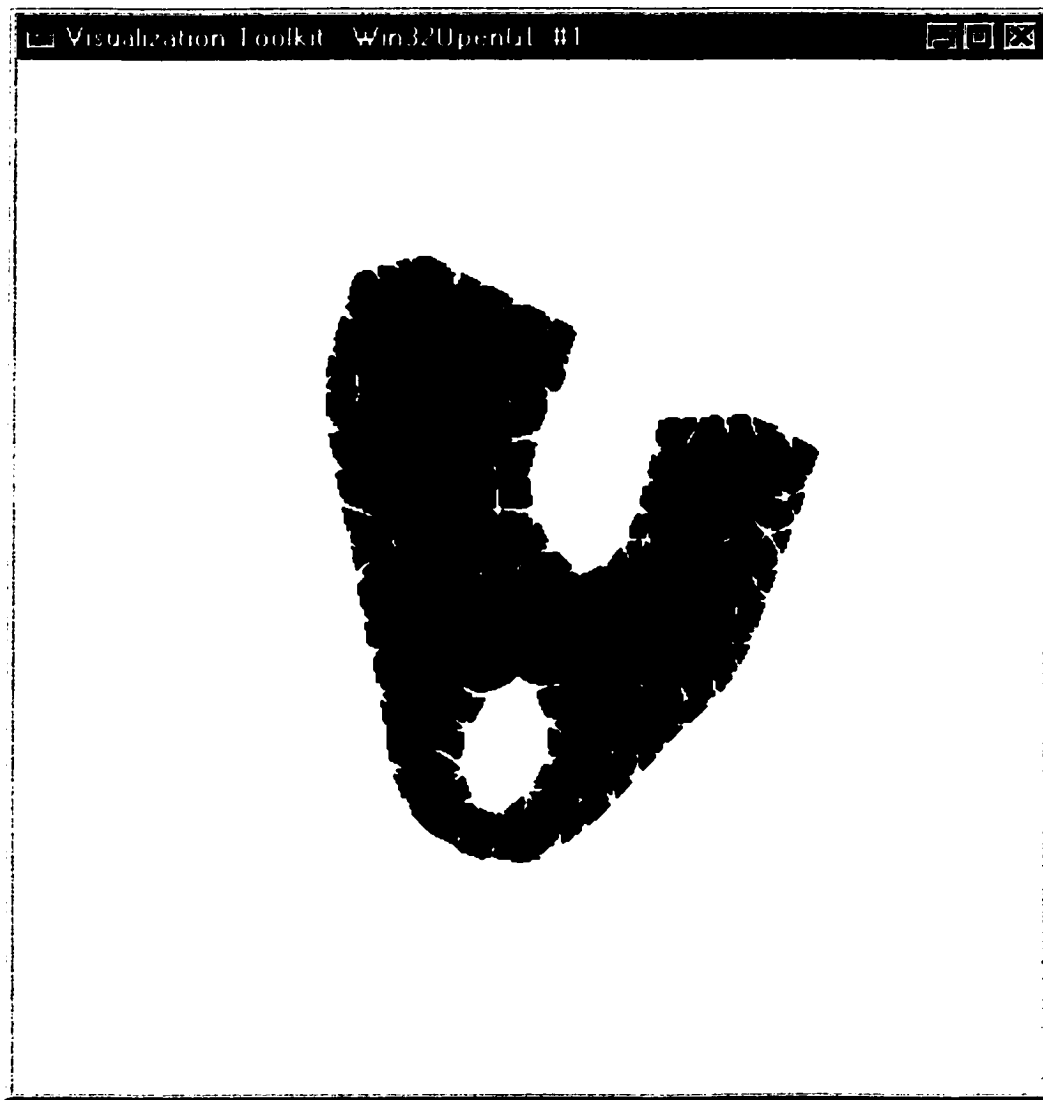


Figure 124: The final shrunk mesh for manifold case.

Once we have mapped all surface nodes into the 3D space, the initial triangulation is performed using these points. First tetrahedrons are highly deformed and have low stretch values. After inserting the nodes into the mesh, the final mesh is obtained as shown in Figure 124.

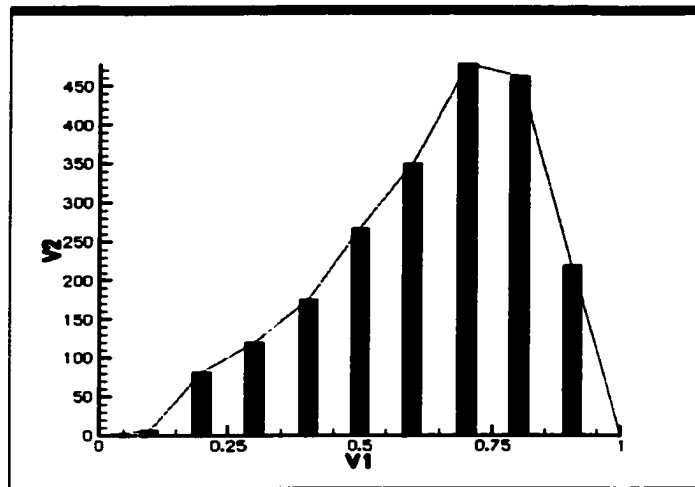


Figure 125: The stretch histogram of the final mesh. (v1: stretch, v2: number of elements).

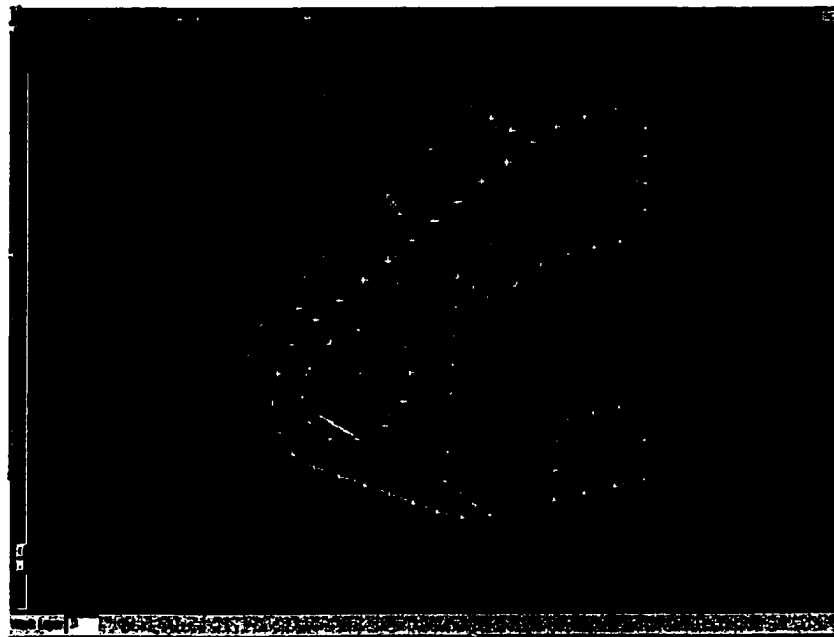


Figure 126: The solid view of a non-manifold model.

9.3 CASE STUDY TWO

In this case, we consider a non-manifold solid part, and demonstrate union Boolean operation. The first part is a three-dimensional solid, and the second part is one-dimensional wireframe edge entity, as shown in Figure 127.

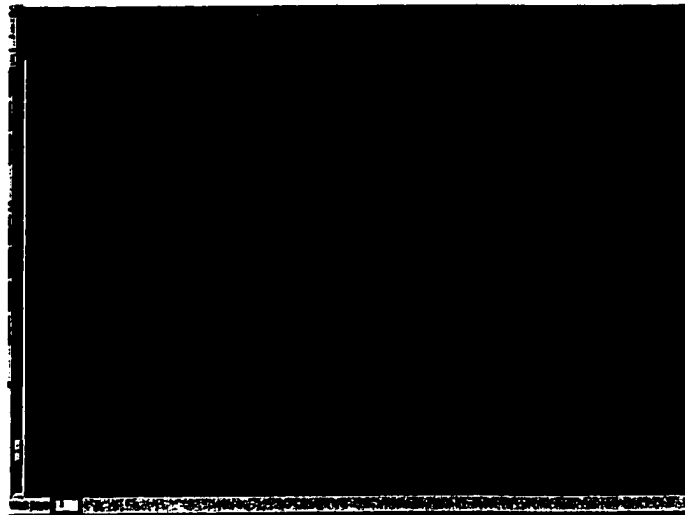


Figure 127: Before union Boolean operation.

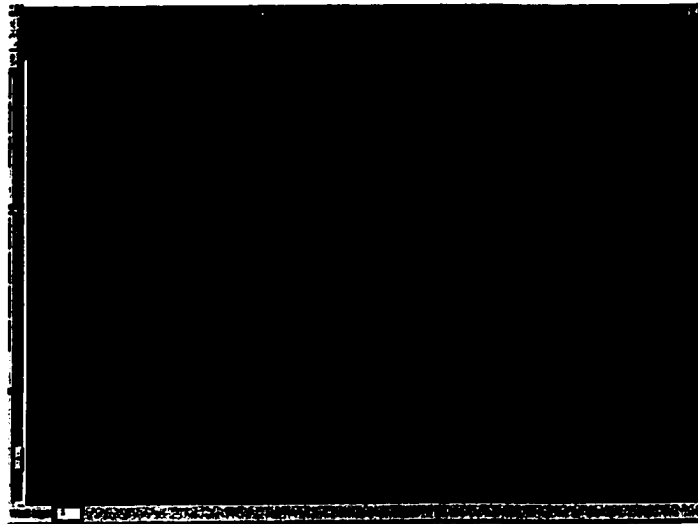


Figure 128: After intersection operation.

The intersection operation is performed between the edge and the corresponding face. Then a point is created at the intersection (Figure 128). The edge is divided into two parts. Because union Boolean operation is performed, the edge inside the solid is discarded, and the other part of the edge is merged with the solid, and a loop consists of single vertex is created within the face (Figure 129).

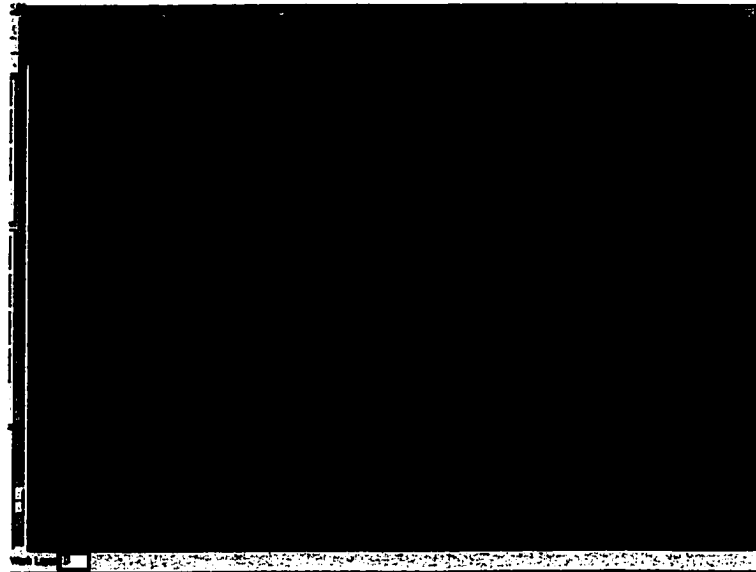


Figure 129: The merged edge and the solid.

After merging the edge, we create a face within the solid part (Figure 130).

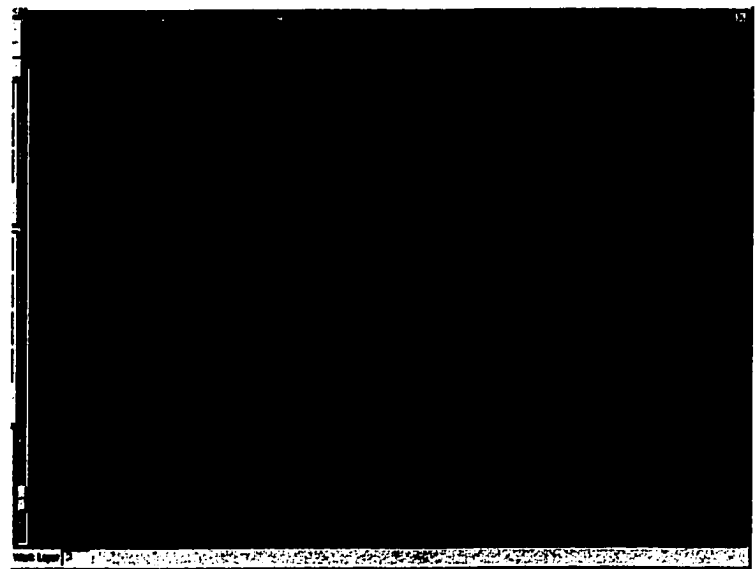


Figure 130: Before intersection operation.

The intersection operation is done among faces. Then the intersecting curve is found, and the face to be attached is split into two faces. An edge is created within the target face, and marked as coincident (Figure 131).

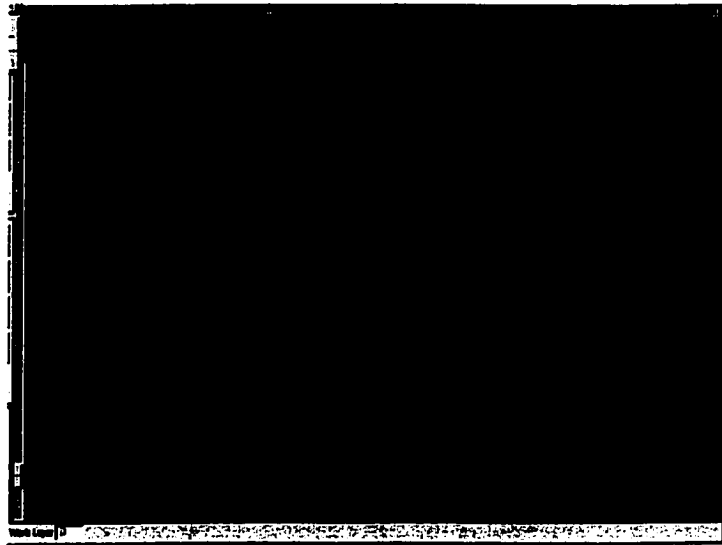


Figure 131: After intersection operation.

The face in the solid is discarded, and the other part is merged with the solid part. After finishing the topological reconstruction, Mesh generation process begins (Figure 132).

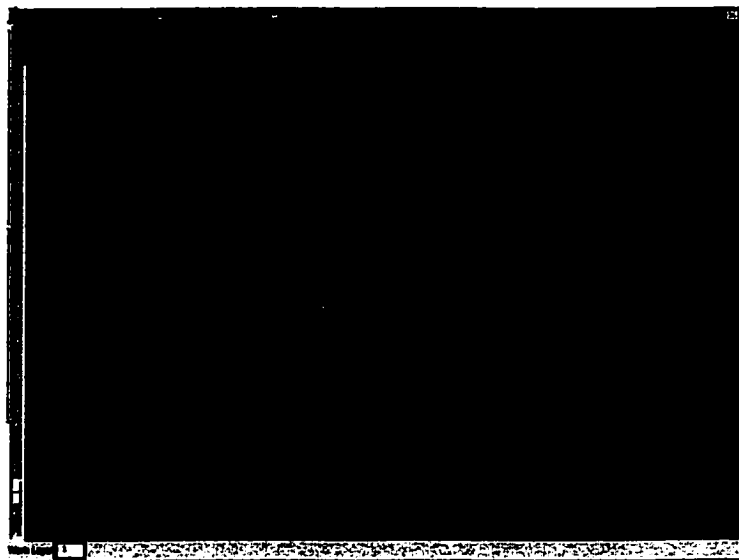


Figure 132: The boundary discretization for the non-manifold model.

First, we create node on the vertices, then the edges using the topology. As explained in mesh generation procedure used in this research, the initial Delaunay triangulation with boundary nodes is performed as shown in Figure 133,134.

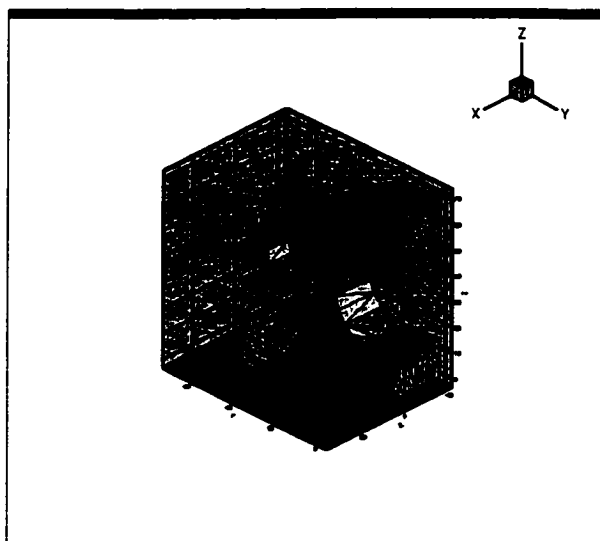


Figure 133: The boundary triangulation of the non-manifold model.

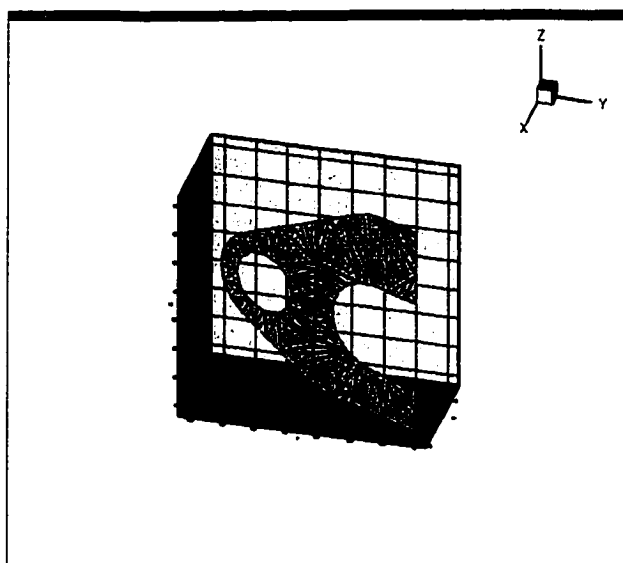


Figure 134: The front face triangulation in absolute coordinates.

Since the boundary triangulation is completed, the nodes inside the faces are generated incrementally, as shown in figure 135-137.

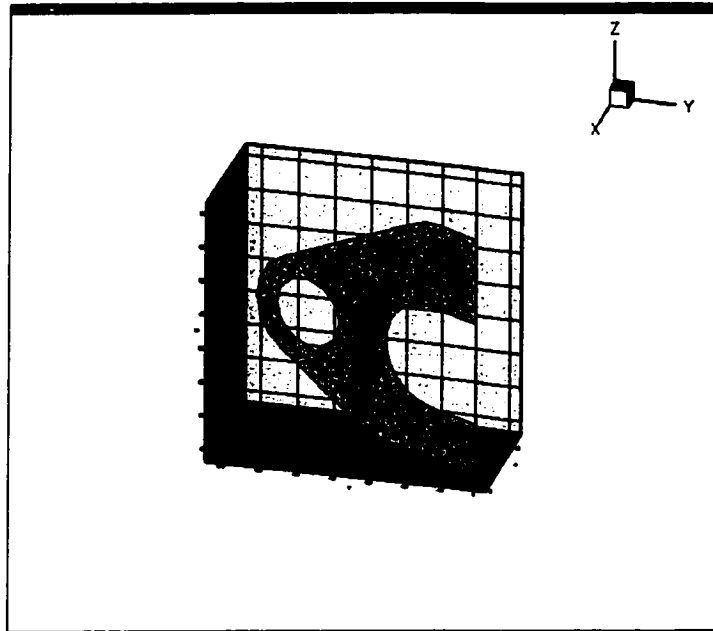


Figure 135: The front face triangulation after internal nodes are inserted.

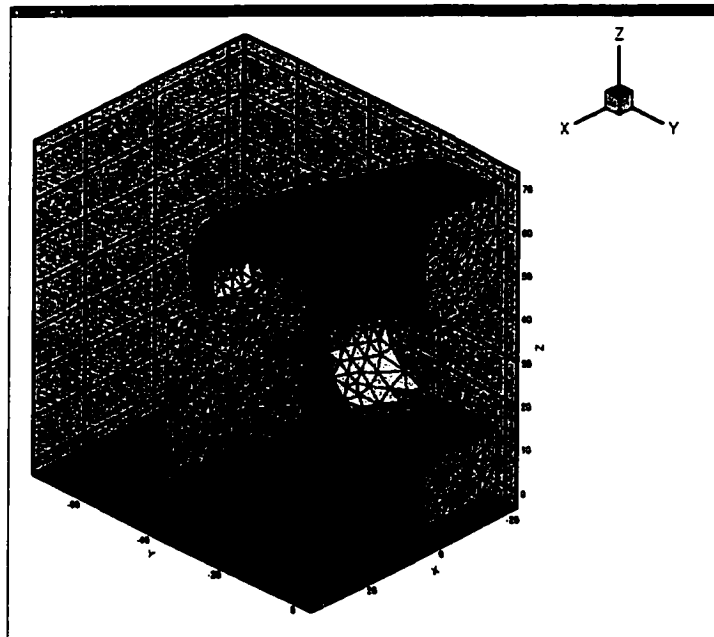


Figure 136: Boundary triangulation of the model.

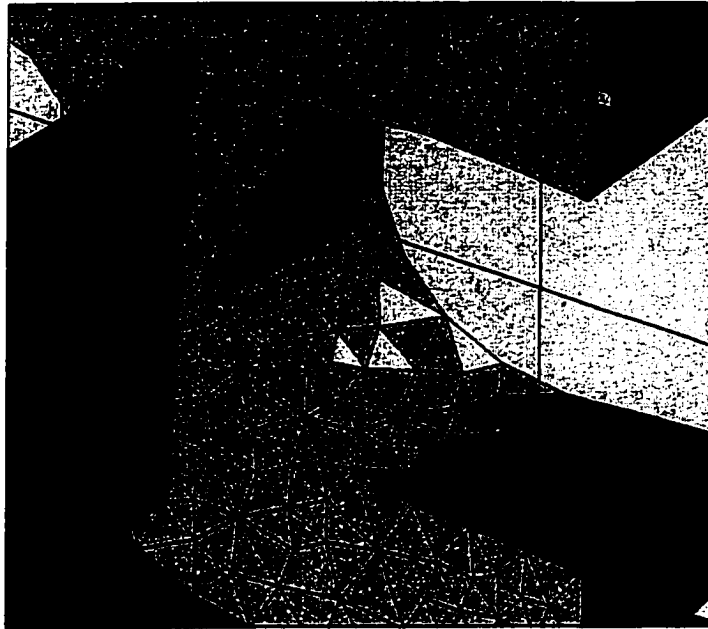


Figure 137: Zoomed view of the interface.

Having all face node generated, nodes inside the solid part are created, as shown in Figure 138-140.

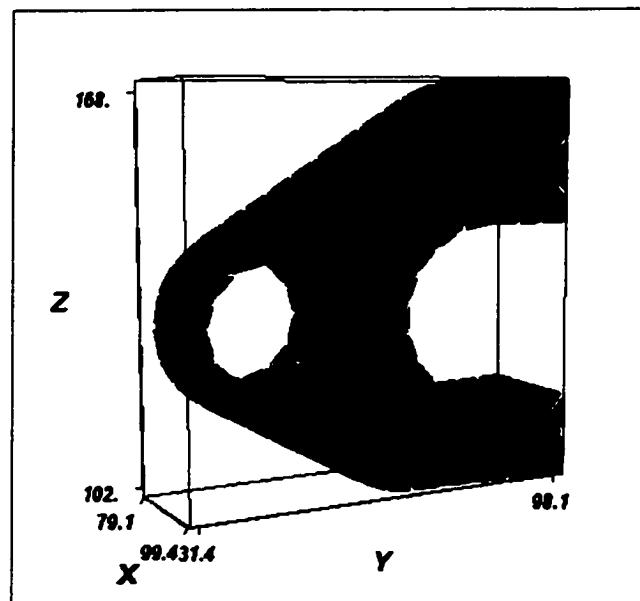


Figure 138: Shrunk tetrahedrons in the solid part of the model.

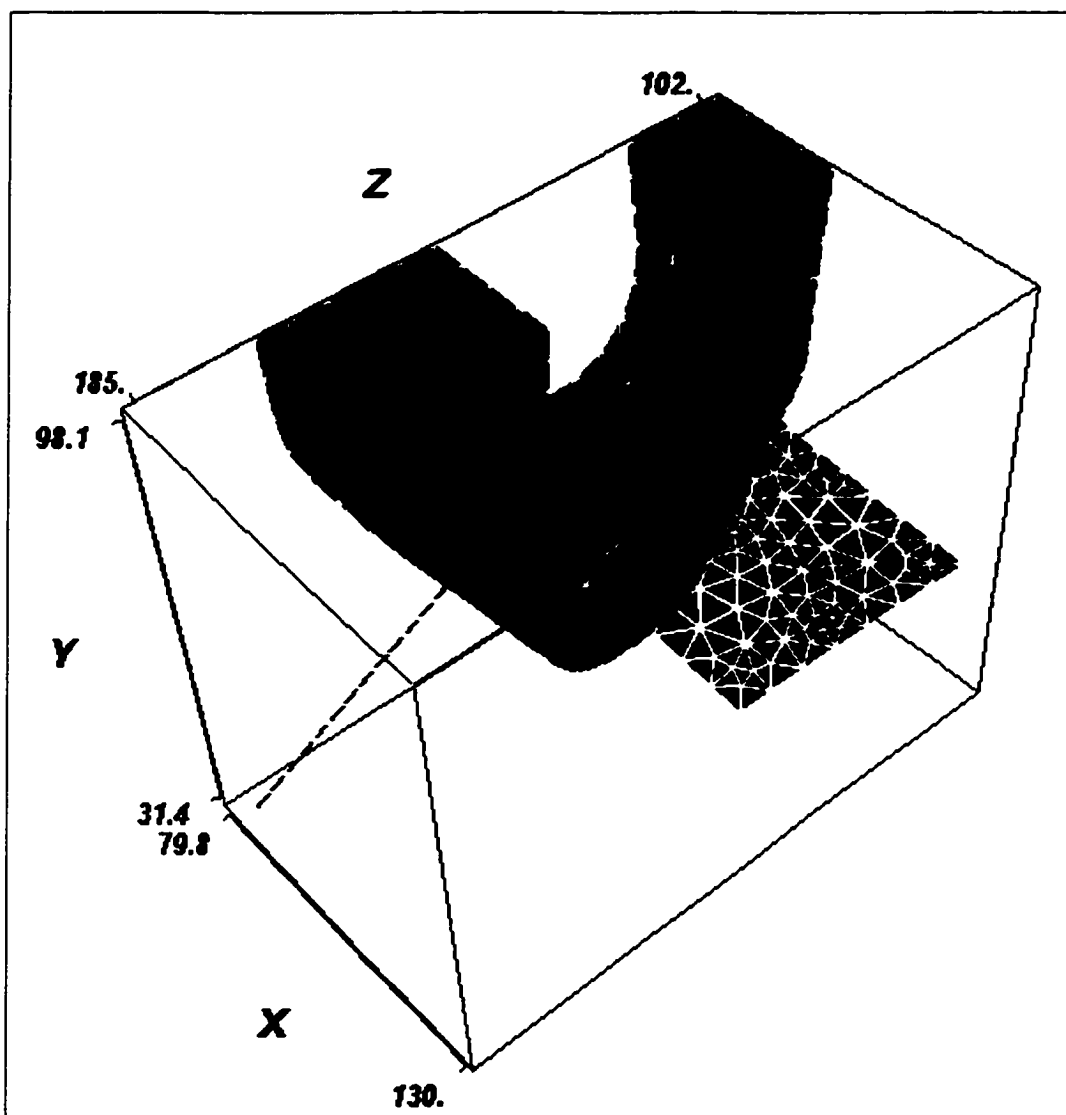


Figure 139: Shrunk tetrahedrons in the non-manifold model.

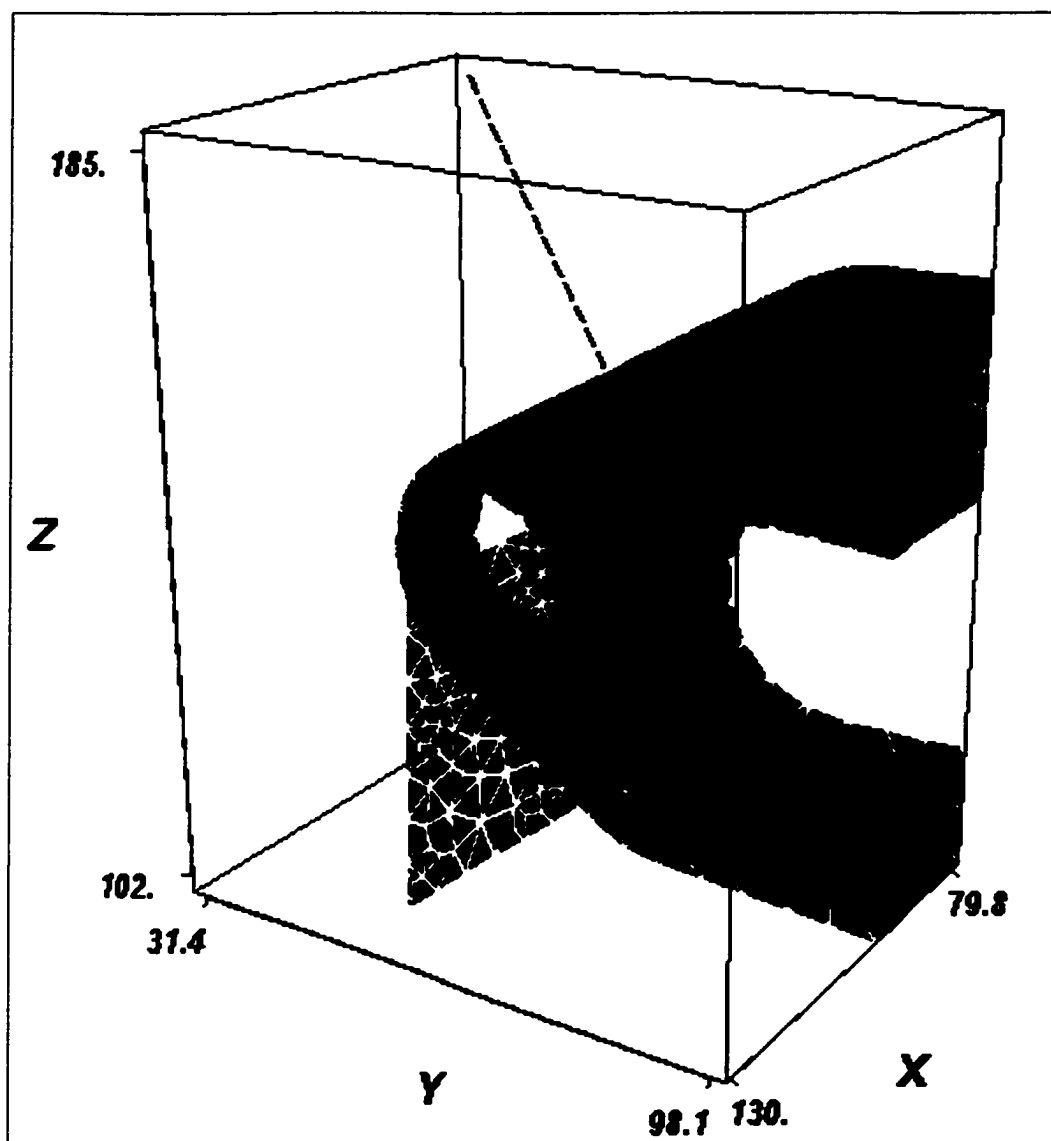


Figure 140: Shrunk tetrahedrons in the non-manifold model.

9.4 CASE STUDY THREE

In this case study, Boolean operation of subtraction type is demonstrated to construct a non-manifold geometry consists of inner face where the solid is splitted (Crack effect). At the beginning, a face is introduced to cut the solid part.

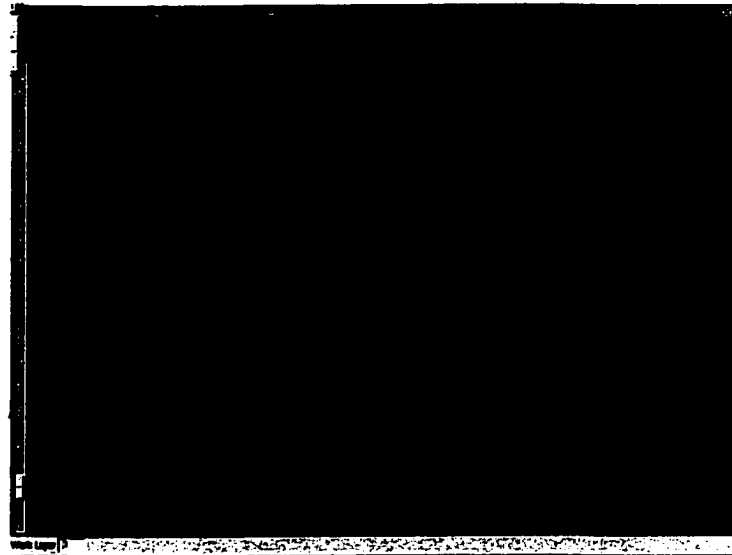


Figure 141: A face cutting a solid.

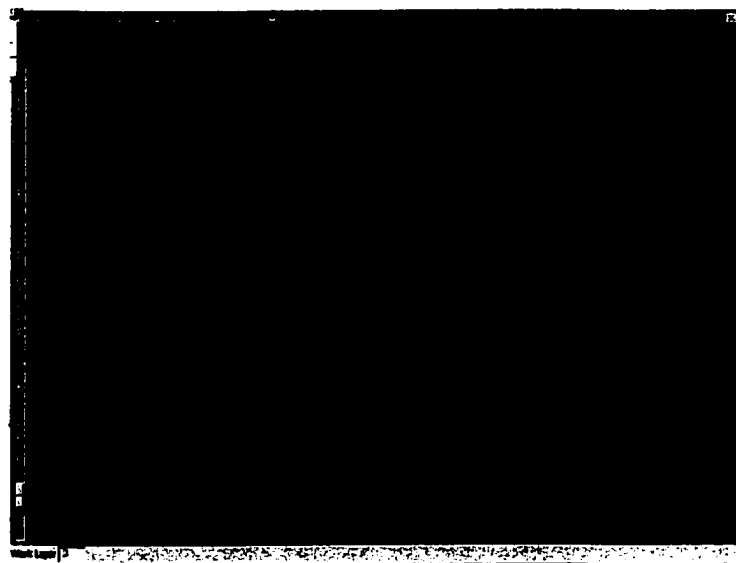


Figure 142: Intersection curves created by Boolean operation process.

The intersection operation is performed among all faces. The resultant's three curves are created. The first one is generated in such a way that it forms a closed circuit. This curve later will be called the crack tip edge. The intersecting curve splits the face on the top of the model into two new faces. Then the other two new faces are created on the crack connected to the new top faces to complete the topology. After the Boolean operation is completed, non-manifold model becomes as shown in Figure 143.

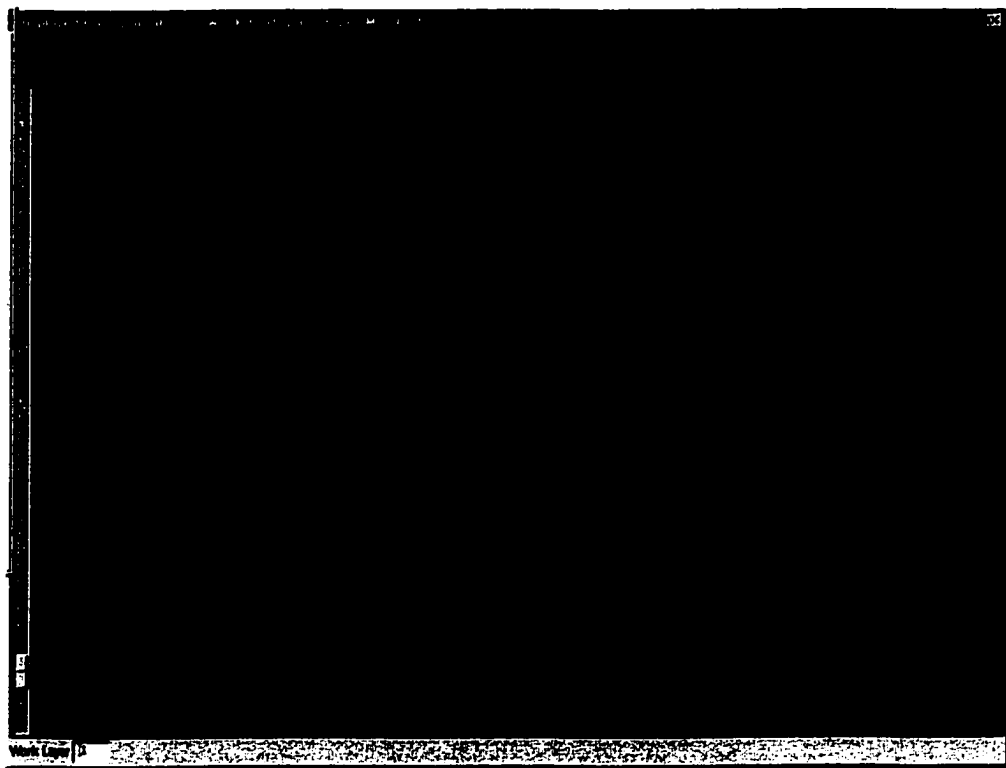


Figure 143: Boundary discretizing in subtraction Boolean operation.

Since the Boolean operation is completed, we discretize the boundary starting from vertices, and edges. Then the mesh on the faces is generated, as shown in Figure

141. The node insertion into the solid is performed following the same procedure explained before. The result is shown in Figures 144 and 145.

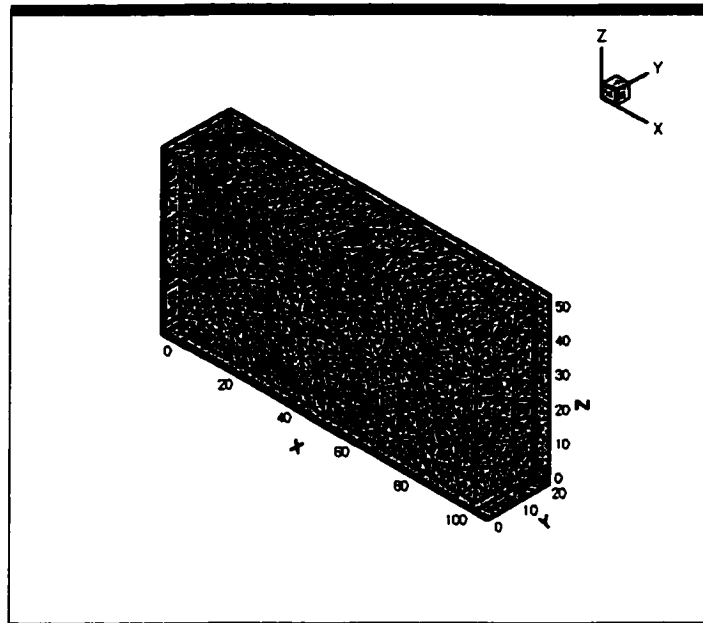


Figure 144: Boundary triangulation in subtraction Boolean operation.

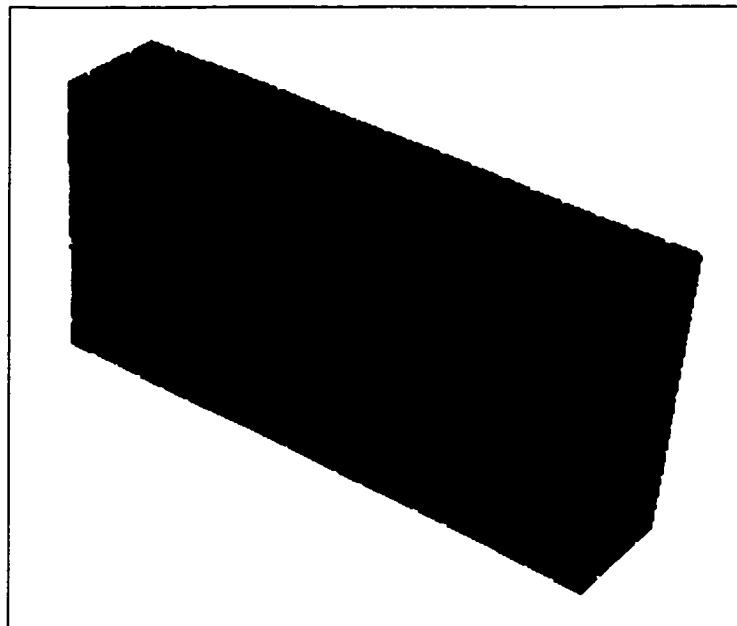


Figure 145: Shrunk tetrahedrons in the non-manifold model.

Chapter 10

Conclusions and Future Work

10.1 CONCLUSIONS

In this study, a mesh generation process for non-manifold geometries has been developed. For the implementation it required more than 10,000 lines of code in C and Fortran languages. The process has been implemented in three phases. During the first phase, to represent the non-manifold topology, a special version of the radial edge structure has been constructed in a module name “NMT”. The structure is also capable of representing finite element modeling information to have easy transition between geometric and finite element modeling. The module employs this non-manifold structure by gathering the connectivity and finite element information among the topological elements. Such implementation has allowed a new concept to be introduced in the performing Boolean operations. This concept reduces the computational time dramatically by dividing the information extraction task into two parts. In addition, Boolean operation of subtraction type has been developed for non-manifold geometries. This operation gives a way to define cracks in solid models, a way that is a unique tool for fracture mechanics of various types of analysis, such as stress distribution around a crack and crack propagation.

In the last phase of the process, mesh generation task has been employed on non-manifold geometry. Delaunay triangulation in 2D and 3D with incremental node insertion method is primarily used since it allows constrained nodes to be used in every stage of

mesh generation process. Newly developed local mesh smoothing method has been applied in the mesh generation algorithm. The method simply improves the efficiency of selecting target triangles/tetrahedrons in the process and decreases the number of the steps in the global smoothing.

Additionally, the removal of triangles/tetrahedrons corresponding to the empty regions in the mesh is another contribution to the algorithms used in both 2D and 3D, before any internal node insertion is performed inside the boundary, for preventing unnecessary node insertion.

On the other hand, the mesh generation in non-manifold geometries is a perfect tool in finite element modeling of structures with welds. Especially for large structures, the volumes outside welding zone do not need fine mesh. Although, coarse mesh is used in those volumes, the computation time could be significant as stated in [42]. However, the model can be reduced to a non-manifold geometry. The volumes with uniform thickness outside the weld region could be reduced to areas that are meshed by shell elements so that the computational time can be reduced tremendously.

Another implementation could be the remeshing capability that would be needed in various manufacturing simulations such as in welding simulation where the torch moves along. Since the undo and canceling operations are possible in NMT, and do not take much time, any of the topological elements could be modified continuously for updating the mesh generated. This property is useful in welding simulation because the remeshing is needed as the torch moves along.

As a result, the methods implemented in this research create triangles/tetrahedrons in non-manifold geometries with acceptable distortion. Additional algorithms could be implemented to improve the efficiency and the environment as part of future work.

10.2 FUTURE WORK

As a continuation of this research, it is highly suggested that the development of a non-manifold geometric kernel that is independent of any other application area is necessary. This could be achieved by employing NURBS for representation of curves and surfaces in constructing the model.

Next, in addition to the Delaunay triangulation method, the use of other methods such as advancing front, quadtree, and octree mesh generation should be investigated, so that the mesh capability could be extended to cover a wide range of different element types.

References

- [1] Sugihara, K., and Inagaki, I., “Why is the 3D Delaunay triangulation difficult to construct?”, *Information Processing Letters*, v54, pp.275-280, 1995.
- [2] Frey, W.H., “Selective Refinement: A new strategy for automatic node placement in graded triangular meshes”, *Int. Journal for Numerical Methods in Engineering*, V24, pp.2183-2200, 1987.
- [3] Joe, B., and Simpson R.B., “Triangular meshes for region of complicated shape”, *International Journal for Numerical Methods in Engineering*, v23, pp.751-778, 1996.
- [4] Weatherill, N.P., and Hassan, O., “Efficient three-dimensional Delaunay triangulation with automatic point creation and imposed boundary constraints”, *International journal for Numerical Methods in Engineering*, v37, pp.2005-2039, 1994.
- [5] Masuda, H., “Topological operators and Boolean operations for complex-based non-manifold geometric models”, *Computer Aided Design*, v25, pp.119-129, 1992.
- [6] Golias, N.A., and Tsiboukis, “An approach to refining three-dimensional tetrahedral meshes based on Delaunay transformations”, *International Journal for Numerical Methods in Engineering*, v37, pp.793-812, 1994.
- [7] Sapidis, N., and Perucchio, R., “Delaunay triangulation of arbitrarily shaped planar domains”, *Computer Aided Geometric Design*, v8, pp.421-437, 1991.

- [8] Chen, J., Gursoz E.L., and Prinz F.B., "Integration of parametric geometry and non-manifold topology in geometric modeling", 2nd ACM Solid Modeling, v5, pp.53-64, 1993.
- [9] Le-Ho, K., "Finite element mesh generation methods: a review and classification", Computer Aided Design, v20, pp.27-38, 1988.
- [10] Shin, Y., and Han, S., "Data Enhancement for sharing of ship design models", Computer Aided Design, v30, pp.931-941, 1998.
- [11] Charlesworth, W.W., and Anderson, D.C., "Application of non-manifold topology", Proceeding of the Computers in Engineering Conference and the Engineering Database Symposium ASME, pp.103-112, 1995.
- [12] Weiler, K., "Topological Structures for geometric modeling", PhD. Thesis, Rensselaer Polytechnic Institute, August 1986.
- [13] Weiler, K., "Bounday graph operators for non-manifold geometric modeling representations", in: IFIP WG5.2 Working Conference on Geometric Modeling for CAD Applications, 12-14, pp.3-36, 1986.
- [14] Weiler, K., "The radial edge structure: A topological representation for non-manifold geometric boundary modeling", in: IFIP WG5.2 Working Conference on Geometric Modeling for CAD Applications, 12-14, pp.37-66, 1986.
- [15] Crocker, G.A., "An editable non-manifold boundary representation", IEEE Computer Graphics & Application, v1, pp.39-51, 1991.
- [16] Shephard, M.S., and Georges M.K., "Reliability of automatic 3D mesh generation", Computer Methods in Applied Mechanics and Engineering, v101, pp.443-462, 1992.

- [17] Frey W.H., and Field D.A., "Mesh relaxation: a new technique for improving triangulations", *International Journal for Numerical Methods in Engineering*, v31, pp.1121-1133, 1991.
- [18] Gursoz, E.L., Choi, Y., and Prinz, F.B., "Boolean set operations on non-manifold boundary representation objects", *Computer Aided Design*, v23, pp.33-39, 1991.
- [19] Kondo, K., "Three-dimensional finite element meshing by incremental node insertion", *International Journal for Numerical Methods in Engineering*, v39, pp.3519-3534, 1996.
- [20] Unigraphics Open API reference Version 15.0, Electronic Data Systems Corporation Unigraphics Division, Volume 1-7, 1996.
- [21] Watson, D.F., "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes", *The Computer Journal*, v24/2, pp.167-172, 1981.
- [22] Boyer, A., "Computing Diriclet tessellations", *The Computer Journal*, v24/2, pp.162-166, 1981.
- [23] Watson, D.F., and Philip, G.M., "Systematic triangulation", *Computer Vision, Graphics, and Image Processing*, v26, pp.217-223, 1984.
- [24] Du, C., "An algorithm for automatic Delaunay triangulation of arbitrary planar domains", *Advances in Engineering Software*, v27, pp.21-26, 1996.
- [25] Chew, L.P., "Constrained Delaunay triangulations", *Algorithmica*, v4, pp.97-108, 1989.
- [26] Aurenhammer, F., "Voronoi Diagrams: a survey of a fundamental geometric data structure", *ACM Computing Surveys*, v23/3, pp.345-411, 1991.

- [27] Sugihara, K., and Iri, M., "A robust topology-oriented incremental algorithm for Voronoi diagrams", *International Journal of Computational Geometry & Applications*, v4/2, pp.179-228, 1994.
- [28] Kemper A., and Wallrath, M., "An analysis of geometric modeling in database systems", *ACM Computing Surveys*, v19/1, pp.47-91, 1987.
- [29] Edelsbrunner, H., Preparata, P., and West, D.B., "Tetrahedrizing point sets in three dimensions", *J. Symbolic Computation*, v10, pp.335-347, 1990.
- [30] Golias, N.A., and Dutton, R.W., "Delaunay triangulation and 3D adaptive mesh generation", *Finite Element in Analysis and Design*, v25, pp.331-341, 1997.
- [31] Mantyla, M., "Introduction to Solid Modeling", Computer Science Press Inc., 1988
- [32] Boumgart, B. G., "Geometric modeling for computer vision", Stanford Artificial Intelligence Research Memo No.249 Computer Science Department, Stanford University, Palo Alto, Oct. 1974.
- [33] Lawson, C. L., "Properties of n-dimensional triangulations, *Computer Aided Design*, v3, pp.231-246.
- [34] Paul-Louis G., and Borouchaki, H., "Delaunay Triangulation and Meshing", *Application to Finite Elements*, Hermes, France, pp.413
- [35] Braid, I. , "Notes on a Geometric Modeler", CAD Group Document No:101, University of Cambridge Computer Laboratory, June 1979
- [36] Requicha, A., "Mathematical models of rigid solid objects", *Production Automation Project Tech. Memo 28*, University of Rochester, Nov. 1977.

- [37] Person, H., "Topological operators and Boolean operations for complex-based non-manifold geometric models.", *Computer Aided Design*, v25,2, pp.119-129.
- [38] Guyder, M. K. , "Automating the optimization of 2^{1/2} Axis milling", *Computer in Industry*, v15, pp.163-168.
- [39] Suh, Y.S. and Lee, K. , " NC Milling tool path generation for arbitrary pockets defined by sculptured surfaces", *Computer Aided Design*, v14,3 ,pp.10-18.
- [40] Held, M., Andor, L. and Lucas, G. , "Pocket machining based on contour-parallel tool paths generated by means of proximity maps", *Computer Aided Design*, v26,3 ,pp.189-203.
- [41] Southwood, J.S., and Chang, T.C. , "A single-feature approach to automatic NC code generation for 2^{1/2} dimensional prismatic features", *JDM*, v4,2 ,pp115-127.
- [42] SYSWELD Welding Simulation Software User Manual, SYSTUS International Inc., France.

Vita

Ergun Nart was born August 1, 1969 in Istanbul, Turkey. He obtained a Bachelor's of Science degree, in Mechanical Engineering from the Marmara University, Istanbul in 1992. After one year, he continued his education at Marmara University Graduate Institute of Science and Technology. During the first year, he got a scholarship from Turkish Government to continue his education at Lehigh University where he has specialized in Geometric Modeling, Mesh Generation, Finite Element Analysis, and their applications as working toward a Ph.D. degree in Mechanical Engineering.