CSG based automatic mesh generation using multiple element types

Richard H. Hall

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
CSG BASED AUTOMATIC MESH GENERATION
USING MULTIPLE ELEMENT TYPES

by

Richard H. Hall

A Thesis Submitted
in
Partial Fulfillment
of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Mechanical Engineering.

Approved by:

Prof. ___________________________________
Dr. Richard Budynas (Thesis Advisor)

Prof. ___________________________________
Dr. Joseph Török

Prof. ___________________________________
Mr. Guy Johnson

Prof. ___________________________________
Dr. Charles Haines (Department Head)

Department of Mechanical Engineering
College of Engineering
Rochester Institute of Technology
August, 1993
Title of Thesis -- "CSG Based Automatic Mesh Generation using Multiple Element Types"

I, Richard H. Hall, hereby grant permission to the Wallace Memorial Library of R.I.T. to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: 10/21/93

(Richard H. Hall)
Abstract

The objective of this thesis project is to explore a unique approach toward automatic mesh generation for finite element analysis. Current mesh generation algorithms are only applicable to a single type of domain. Countless mesh generators exist for meshing 2D regions with triangles and quadrilaterals, and mesh generators also exist which can mesh 3D regions with tetrahedra and other element types. However, not all structures are strictly "2D" or "3D", and not all structures are best modeled with a single type of element. An experienced finite element analyst typically uses many types of elements when modeling a real problem. This thesis addresses this approach to meshing in an automatic manner. However, at various stages, the user has the ability to change the course of the modeler.

In this thesis project, a program for automatic mesh generation has been developed on a constructive solid geometry (CSG) foundation. This program was written in object-oriented Pascal, and consists of well over 25,000 lines of code. The CSG system used was developed with PADL-2 as the guide, and allows complex geometries to be modeled as combinations of blocks and cylinders. This solid model is then broken into 1D, 2D and 3D regions, or "segments", using CSG-Tree segmentation logic. Each segment can then be meshed using an appropriate mesh generation technique. Thus, a single model can be meshed with multiple element types, just as an experienced analyst would do it.
Acknowledgments

This thesis has been a long time in the making, which has given many people a chance to contribute. I would like to thank everyone who made the writing of this thesis possible:

Dr. Richard Budynas, who has been envisioning a computer program to perform automatic mesh generator using multiple element types for a long, long time. I'm thankful to Dr. Budynas for having confidence in me, believing that I would be able to make his dream a reality, and for his continued support and encouragement. I only hope that what I have produced does justice to what he had in mind.

Committee members Dr. Joseph Török and Mr. Guy Johnson, who deserve extraordinary credit just for reading such a fat, boring book. Dr. Török's knowledge of finite element theory and Guy Johnson's knowledge of geometric modeling and programming were instrumental in writing the finished version of this thesis, and their contributions are greatly appreciated.

Mr. Steve Kurtz, who's course "Computer Graphics is Design" taught me object-oriented Pascal and, more importantly, enthusiasm for elegant programming. Without having picked up this enthusiasm from Steve, writing 25,000 lines of code would not have been possible.

Dr. Charles Haines, department head and academic advisor. As a BS/MS student, I'd like to thank Dr. Haines for making me feel like there was someone looking out for me.

The Gleason Society, for the honor and financial support of being selected as the 1991/92 Gleason Graduate Scholar.

Thanks to Megan, who has been an important part of my life and can never be anything less.

And finally, I would like to express my appreciation and gratitude for my parents, Harley and Sandy, for their never-ending patience and support. This thesis and every success in my life is a direct result of their unconditional love and constant understanding.
# Table of Contents

Abstract iii

Acknowledgments iv

List of Figures viii

List of Symbols xiii

1. Introduction 1
   1.1. The Finite Element Method in Mechanical Engineering 1
   1.2. History of the Finite Element Method 2
   1.3. Finite Element Theory 4
   1.4. Types and Uses of Finite Elements 7
      1.4.1. One-Dimensional Second-Order Equations 7
      1.4.2. One-Dimensional Fourth-Order Equations 12
      1.4.3. Two-Dimensional Scalar Valued Second-Order Equations 16
      1.4.4. Two-Dimensional Multi-Variable Equations 21
      1.4.5. Three-Dimensional Equations 22
   1.5. Modeling using Multiple Element Types 23
   1.6. Thesis Objective 24

   2.1. History of Mesh Generation 25
   2.2. Automatic Meshing of 1-D Regions 26
   2.3. Automatic Meshing of 2-D Regions 27
      2.3.1. Volume Triangulization Methods 27
      2.3.2. Element Extraction Methods 32
      2.3.3. Recursive Spatial Decomposition Methods 35
   2.4. Automatic Meshing of 3-D Regions 37
      2.4.1. Volume Triangulization Methods 37
      2.4.2. Element Extraction Methods 39
      2.4.3. Recursive Spatial Decomposition Methods 40
   2.5. Expert Systems for Automatic Mesh Generation 41

3. Geometric Modeling 44
   3.1. Modeling Techniques 44
      3.1.1. Sweep Representations 45
      3.1.2. Cell Decompositions 45
      3.1.3. Boundary Representation 46
      3.1.4. Constructive Solid Geometry 47
   3.2. Boundary Evaluation 48
   3.3. PADL-2 50
4. Expert Systems
   4.1. Expert System Structure 53
   4.2. Expert System Development 55

5. CSG Based Automatic Mesh Generation using Multiple Element Types 58
   5.1. "CSGMesh" Computer Program 59
      5.1.1. Program Overview 60
         5.1.1.1. Input File 60
         5.1.1.2. CSG Tree 66
         5.1.1.3. Segments 70
         5.1.1.4. Options 73
         5.1.1.5. Meshes 76
         5.1.1.6. Output File 83
      5.1.2. Solid Representation 85
         5.1.2.1. Solids 86
            5.1.2.1.1. Blocks 88
            5.1.2.1.2. Cylinders 88
            5.1.2.1.3. Other Primitives 89
            5.1.2.1.4. Unions 89
            5.1.2.1.5. Differences 90
            5.1.2.1.6. Intersections 90
         5.1.2.2. Boundary Representation 90
            5.1.2.2.1. Surfaces 92
            5.1.2.2.2. Edges 93
      5.1.2.3. Boundary Evaluation 94
   5.1.3. Segments 98
      5.1.3.1. BeamSegments 98
      5.1.3.2. PlateSegments 99
      5.1.3.3. CylinderPlateSegments 99
      5.1.3.4. BrickSegments 99
      5.1.3.5. CSG Segmentation Logic 100
         5.1.3.5.1. Blocks 103
         5.1.3.5.2. Cylinders 103
         5.1.3.5.3. Unions 103
         5.1.3.5.4. Differences 104
         5.1.3.5.5. Intersections 104
         5.1.3.5.6. Special Cases 104
         5.1.3.5.7. Using Surfaces to make Segments 106
         5.1.3.5.8. Combining Segments 106
   5.1.4. Segment Meshing Techniques 108
      5.1.4.1. Types of Meshes 108
      5.1.4.2. Meshing Beam Segments 109
      5.1.4.3. Meshing Plate Segments 109
      5.1.4.4. Meshing CylinderPlate Segments 111
      5.1.4.5. Meshing Brick Segments 112
      5.1.4.6. Editing Meshes 113
      5.1.4.7. Joining Meshes between Segments 113
5.2. Future Extensions to "CSGMesh"
  5.2.1. Further Primitive Types 114
  5.2.2. Implementing Mesh Generators 115
  5.2.3. Other possible additions to "CSGMesh" 120

6. Results: Examples of Geometries Meshed using "CSGMesh"
  6.1. Plate with Holes 127
    6.1.1. Plate using Plate Segment (2D Mesh) 129
    6.1.2. Plate using Brick Segment (3D Mesh) 130
  6.2. I-Beam 132
    6.2.1. I-Beam using Beam Segment 133
    6.2.2. I-Beam using Plate Segment 134
    6.2.3. I-Beam using Multiple Plate Segments 135
    6.2.4. I-Beam using Brick Segments 136
  6.3. Pipe 137
    6.3.1. Pipe using Beam Segment 138
    6.3.2. Pipe using Plate Segment 139
    6.3.3. Pipe using CylinderPlate Segment 140
    6.3.4. Pipe using Brick Segment 141
  6.4. Pipe with Holes 142
  6.5. Bracket 145

7. Discussion/Conclusion 150

8. References 151

9. Appendices:
   Appendix A. PADL-2 Source Files A-0
   Appendix B. PADL-2 Point Sets: Primitives, Halfspaces, and Edges B-0
   Appendix C. CSGMesh Object Hierarchy C-0
   Appendix D. CSGMesh Object Reference D-0
   Appendix E. CSGMesh Input (.CSG) File Syntax Diagrams E-0
   Appendix F. Algor Supersap Output File F-0
   Appendix G. ANSYS version 5.0 Output File G-0
   Appendix H. NASTRAN Output File H-0
# List of Figures

<table>
<thead>
<tr>
<th>Chapter 1:</th>
<th></th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1 Division of circle into triangles</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Figure 1.2 Generic Domain $\Omega$</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Figure 1.3 Strong and Weak Statements</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Figure 1.4 One-Dimensional Domain for Second-Order Equations</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>Figure 1.5 Typical One-Dimensional, Second-Order Element</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Figure 1.6 Interpolation Functions for 1D Two-Node Linear Element</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>Figure 1.7 One Dimensional Domain for Fourth-Order Equations</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>Figure 1.8 Interpolation Functions for 1D Two-Node Quadratic Element</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>Figure 1.9 Two-Dimensional Domain</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Figure 1.10 Two-Dimensional Elements</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>Figure 1.11 Interpolation Functions for 2D Three-Node Linear Triangle Elements</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>Figure 1.12 Four-Node Rectangular Element Local Coordinate System</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>Figure 1.13 Interpolation Functions for 2D Four-Node Linear Quadrilateral Elements</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>Figure 1.14 Three-Dimensional Elements</td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>Figure 1.15 Mesh using Different Element Types</td>
<td></td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 2:</th>
<th></th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1 Convex Hull of a Set of Points</td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>Figure 2.2 Starting Mesh for Delaunay Triangulation</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>Figure 2.3 Node Insertion in Delaunay Triangulation</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>Figure 2.4 Delaunay Triangulation of the Convex Hull of a Set of Points</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Figure 2.5 Various stages of a Paving Algorithm</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>Various Stages of a Paving Algorithm with Fronts by Inflation</td>
<td>33</td>
</tr>
<tr>
<td>Figure 2.7</td>
<td>Various Stages of a Paving Algorithm using Quadrilaterals</td>
<td>34</td>
</tr>
<tr>
<td>Figure 2.8</td>
<td>Various Stages of a Quadtree Algorithm</td>
<td>35</td>
</tr>
<tr>
<td>Figure 2.9</td>
<td>Mesh Generated by Quadtree Algorithm</td>
<td>36</td>
</tr>
<tr>
<td>Figure 2.10</td>
<td>Mesh Generated by 3D Delaunay Triangulation</td>
<td>38</td>
</tr>
<tr>
<td>Figure 2.11</td>
<td>Mesh Generated by Plastering Algorithm</td>
<td>39</td>
</tr>
<tr>
<td>Figure 2.12</td>
<td>Mesh Generated by Octree Algorithm</td>
<td>40</td>
</tr>
</tbody>
</table>

**Chapter 3:**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 3.1</td>
<td>Sweep Representation</td>
<td>45</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Boundary Representation (B-Rep)</td>
<td>46</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Constructive Solid Geometry (CSG) Representation</td>
<td>47</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Polygonal Representation of CSG Primitives</td>
<td>49</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Polygonal Representation of CSG Solid</td>
<td>49</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>PADL-2 Block</td>
<td>51</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>PADL-2 Cylinder</td>
<td>51</td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>PADL-2 Wedge</td>
<td>51</td>
</tr>
<tr>
<td>Figure 3.9</td>
<td>PADL-2 Cone</td>
<td>51</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>PADL-2 Sphere</td>
<td>51</td>
</tr>
<tr>
<td>Figure 3.11</td>
<td>PADL-2 Torus</td>
<td>51</td>
</tr>
</tbody>
</table>

**Chapter 4:**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 4.1</td>
<td>Expert System Structure</td>
<td>54</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Expert System Development</td>
<td>56</td>
</tr>
</tbody>
</table>
Chapter 6:

Figure 6.1 Plate with Holes -- Primitives
Figure 6.2 Plate with Holes -- Solid after Boundary Evaluation
Figure 6.3 Plate with Holes as Plate Segment
Figure 6.4 Mesh of Plate with Holes as Plate Segment
Figure 6.5 Plate with Holes as Brick Segment
Figure 6.6 Mesh of Plate with Holes as Brick Segment
Figure 6.7 I-Beam -- Primitives
Figure 6.8 I-Beam as Beam Segment
Figure 6.9 Mesh of I-Beam as Beam Segment
Figure 6.10 I-Beam as Plate Segment
Figure 6.11 Mesh of I-Beam as Plate Segment
Figure 6.12 I-Beam as Multiple Plate Segments
Figure 6.13  Mesh of I-Beam as Multiple Plate Segments  135
Figure 6.14  I-Beam as Brick Segment  136
Figure 6.15  Mesh of I-Beam as Brick Segment  136
Figure 6.16  Pipe -- Primitives  137
Figure 6.17  Pipe as Beam Segment  138
Figure 6.18  Mesh of Pipe as Beam Segment  138
Figure 6.19  Pipe as Plate Segment  139
Figure 6.20  Mesh of Pipe as Plate Segment  139
Figure 6.21  Pipe as CylinderPlate Segment  140
Figure 6.22  Mesh of Pipe as CylinderPlate Segment  140
Figure 6.23  Pipe as Brick Segment  141
Figure 6.24  Mesh of Pipe as Brick Segment  141
Figure 6.25  Pipe with Holes -- Primitives  142
Figure 6.26  Pipe with Holes -- Solid after Boundary Evaluation  143
Figure 6.27  Pipe with Holes -- Plate Segment Representation  143
Figure 6.28  Mesh of Pipe with Holes  144
Figure 6.29  Mesh of Pipe with Holes  144
Figure 6.30  Bracket -- Primitives  145
Figure 6.31  Bracket -- Solid after Boundary Evaluation  146
Figure 6.32  Bracket -- Plate Segment Representation  147
Figure 6.33  Bracket -- Plate Segment Representation (End View)  147
Figure 6.34  Bracket -- Mesh  148
Figure 6.35  Bracket -- Mesh (End View)  148
Figure 6.36  Bracket -- Mesh after Merging and Connecting  149
Figure 6.37  Bracket -- Mesh after Merging and Connecting (End View)  149
List of Symbols

**Bold** letters  Vector or Matrix

$\alpha$  unknown parameter in approximation function

A  Differential operator

$B(v,u)$  Bilinear functional of $u$ and $v$

BLO  Block primitive

CON  Cone primitive

CSG  Constructive Solid Geometry

CYL  Cylinder primitive

DIF  Difference operation

DOF  Degree of Freedom

e  Typical Element

e  Modulus of Elasticity (Young's Modulus)

EBC  Essential Boundary Condition

$f$  forcing function

$F^{(e)}$  Force Vector for element $e$

FEA  Finite Element Analysis

FEM  Finite Element Method

G  Shear Modulus

INT  Intersection operation

$I_{xx}$  Moment of Inertia about x-axis

$I_{xy}$  Product of Inertia

$I_{yy}$  Moment of Inertia about y-axis

J  Polar Moment of Inertia

$K^{(e)}$  Stiffness Matrix for element $e$

$l(v)$  Linear functional of $v$
M[X, S]  Classify X against S
MCR     Membership Classification Result
NBC     Natural Boundary Condition
PADL    Part and Assembly Description (or Definition) Language
RSD     Recursive Spatial Decomposition
SPH     Sphere primitive
TOR     Torus primitive
\( u \)  unknown function
UN      Union operation
WED     Wedge primitive
\( v \)  test function
\( \psi \) approximation function
1. Introduction

1.1. The Finite Element Method in Mechanical Engineering

In today's highly competitive world, products must be designed very carefully. Modern engineers must ensure that their designs will be functional, last when subject to hard use and extreme conditions, be attractive and pleasing to the user, achieve high standards in terms of safety, and will satisfy a multitude of other consumer demands. In addition to all of these requirements, designs must be cost-effective as well.

Fortunately, a modern design engineer is not required to produce a prototype of each of his design alternatives to submit to testing of all the necessary criteria. This surely would not be cost effective. There are several techniques available in which the engineer can represent his design mathematically to test various parameters. One of the most powerful, and certainly the most popular, is the Finite Element Method (FEM).

In the Finite Element Method, a complex geometry is broken down into a finite number of simple geometric shapes, called finite elements. The material properties and governing relationships (usually a set of differential equations) are expressed over these elements to yield a system of equations. These equations can be solved to give the approximate behavior of the structure.
1.2. History of the Finite Element Method

The idea of representing a given domain with a number of simple geometric shapes is not new. Ancient mathematicians estimated the value of pi accurate to 40 places by representing a circle with a large number of triangles as shown in figure 1.1.

\[
A = \sum_{e=1}^{n} A_e
\]

![Figure 1.1](image)

In modern times, the basic ideas of the finite element method originated in the aircraft industry, where wings and fuselages were represented as collections of strings, skins, and shear panels. In 1941, Hrenikoff presented the "frame-work method,"[1] in which plane elastic regions were modeled using a collection of bars and beams. The use of piecewise continuous functions dates to 1943, when Courant used a collection of triangular elements and the principle of minimum potential energy to study the St. Venant Torsion problem.[2] The formal presentation of the finite element method is attributed to Turner, Clough, Martin, and Topp, who in 1956 derived stiffness matrices for truss, beam, and other elements,[3] and to Argyris and Kelsey, who wrote their paper on Energy Theorems and Structural Analysis in 1960.[4] The term "finite element" was first coined and used by Clough in 1960.
In the early 1960's, engineers used the finite element method to find approximate solutions to problems in stress analysis, fluid flow, heat transfer, and other areas. The first book on finite elements by Ziekiewicz and Chung was published on 1967.\[5\] In the late 1960's and early 1970's, the finite element method was applied to non-linear problems and large deformations. A book on non-linear continua by Oden appeared in 1972.\[6\] Mathematical foundations were laid in the 1970's, including element development, convergence studies, and other related studies.

Since its inception, the literature on the finite element method has grown exponentially, and today there are numerous journals which are devoted primarily to the theory and application of the finite element method. A review of the historical developments and the basic theory of the method can be found in dozens of textbooks that are exclusively devoted to the introduction and application of the finite element method.
1.3. Finite Element Theory

As with any numerical technique, an understanding of the underlying principles of the FEM is necessary in order to use the method effectively. However, it would not be practical to include a thorough discussion of the theory of the finite element method in this thesis. Volumes upon volumes have been written about the theory and application of the finite element method. The purpose of this thesis is to explore the area of automatic mesh generation, and not to explain fully the theory and concepts of the finite element method. Thus, the subject of finite element theory will be limited to a brief summary discussion.

The Finite Element Method is a piecewise application of a variational method. A typical problem involves some domain Ω, defined by a boundary Γ, over which some mathematical relations hold. The objective of the analysis is to determine unknown functions which satisfy the mathematical relations (usually differential equations) over the domain. Figure 1.2 shows a generic domain Ω over which a set of differential equations describe the behavior.

![Figure 1.2](image)

**Governing Equation:** \( Au = f \)
- \( A = \) differential operator
- \( u = \) unknown function
- \( f = \) forcing function
In order to make use of the finite element method, the governing equations describing the behavior of the domain must be cast in "weak" (or variational) form. The differential equation $Au = f$ is said to be in the "strong" form, meaning that the equation represents an exact statement at every point in the domain. To obtain the weak form, test functions (represented by "v") must be chosen which are sufficiently differentiable and which take on the value zero at Essential Boundary Condition (EBC) locations. Both sides of the strong form are multiplied by the test function $v$ and then integrated over the domain, yielding

$$\int_{\Omega} (Au - f)v \, d\Omega = 0.$$

An analogy which helps to make the concepts of "strong" and "weak" forms more clear is that of some simple function, say $g(x)$. The statement $g(x) = 0$ for $0 < x < L$ is a strong statement. It says that at every point between 0 and L, the value of $g$ is identically zero. However, the statement $\int_{0}^{L} g(x) \, dx = 0$ is a much weaker statement. It allows the value of $g$ to be something other than zero between 0 and L, as long as the average value over the domain is zero. This forces the arbitrary function $g(x)$ to approximate zero as closely as possible between 0 and L. This is shown in figure 1.3.

![Figure 1.3](image-url)
After the differential equation is cast in weak form, the next step is to integrate by parts to transfer the differentiation from the dependent variable \( u \) to the test function \( v \). This serves to reduce the differentiation requirement on \( u \), which allows lower-order functions to be used to approximate the behavior of the system. In the process of integrating by parts, boundary terms are obtained which identify the nature of the boundary conditions in the solution. By setting \( v = 0 \) at Essential Boundary Condition (EBC) locations, and defining secondary variables at Natural Boundary Condition (NBC) locations, the boundary conditions become imposed into the functional.

The weak form becomes thus posed as: find \( u \) such that \( B(u,v) - \int v \) for all test functions \( v \) such that \( v = 0 \) at EBC locations, where \( B \) is bilinear functional representing the weak form and \( \int \) is linear functional representing the boundary terms.

This process will be shown more clearly as it is used to develop element equations in the next few sections.
1.4. Types and Uses of Finite Elements

The finite element method is applicable to countless problems posed on many domains. Each type of problem and each domain have their own unique set of equations. In this section a few problems will be considered to demonstrate the requirements for casting an equation in variational form over a domain.

1.4.1. One-Dimensional Second-Order Equations

Consider the problem of finding the function $u$ which satisfies the equation

$$-\frac{d}{dx} \left( a \frac{du}{dx} \right) - f = 0$$

applied over the domain $0 < x < L$, and the boundary conditions $u(0) = 0$ and $\left( a \frac{du}{dx} \right)_{x=L} = P$, where $a = a(x), f = f(x)$ and $P$ are given data of the problem. This equation arises in the axial deformation of a bar. The domain $\Omega=(0, L)$ of the problem, shown in figure 1.4(a), is divided into a set of line elements, called the finite element mesh, as shown in figure 1.4(b).

![Figure 1.4](image)
Since the governing differential equation is valid over the whole domain \( \Omega = (0,L) \), it is valid over each element of the finite element mesh. In particular, it is valid over generic element \( e \). Following the procedure described in section 1.3, the variational formulation of the governing differential equation can be constructed over element \( e \):

The strong form is given by \(-\frac{d}{dx}\left(a\frac{du}{dx}\right) - f = 0\) over the domain of element \( e \), \( \Omega_e = (x_A, x_B) \) as shown in figure 1.5.

![Figure 1.5](image)

Multiplying by the test function \( v \) and integrating over the domain yields the weak form:

\[
\int_{x_A}^{x_B} v \left[-\frac{d}{dx}\left(a\frac{du}{dx}\right) - f\right] dx = 0
\]

Integrating by parts to transfer the differentiation from the unknown function \( u \) to the test function \( v \) gives:

\[
\int_{x_A}^{x_B} \left(a\frac{dv}{dx}\frac{du}{dx} - vf\right) dx + \left[v \left(-a\frac{du}{dx}\right)\right]_{x_A}^{x_B} = 0
\]
Examining the boundary term in the above equation shows that the specification of $u$ at $x = x_A$ and $x = x_B$ constitute the essential boundary conditions, and the specification of $\left(-a \frac{du}{dx}\right)$ at $x = x_A$ and $x = x_B$ constitute the natural boundary conditions for the element. Thus, the basic unknowns at the element nodes are the primary variable $u$, which is the degree of freedom (DOF), and the secondary variable $\left(-a \frac{du}{dx}\right)$. To simplify the writing of the equations, let

$$u(x_A) \equiv u_1^{(e)} \quad u(x_B) \equiv u_2^{(e)}$$

$$\left(-a \frac{du}{dx}\right)_{x_A} \equiv P_1^{(e)} \quad \left(+a \frac{du}{dx}\right)_{x_B} \equiv P_2^{(e)}$$

Substituting this notation into the variational form gives

$$\int_{x_A}^{x_B} \left(a \frac{dv \ du}{dx} - vf\right)dx - P_1^{(e)} v(x_A) - P_2^{(e)} v(x_B) = 0$$

or $B(v, u) - l(v) = 0$ where $B(v, u)$ is the bilinear form given by $B(v, u) = \int_{x_A}^{x_B} \left(a \frac{dv \ du}{dx} \right)dx$

and $l(v)$ is the linear form given by $l(v) = \int_{x_A}^{x_B} vfdx + v(x_A) P_1^{(e)} + v(x_B) P_2^{(e)}$

To find an approximate solution to the above variational problem using the Galerkin method, the function $u$ is approximated over the element by

$$u_e(x) = \sum_{j=1}^{n} \alpha_j^{(e)} \psi_j^{(e)}(x)$$

where $\alpha_j$ are the parameters to be determined and $\psi_j(x)$ are the approximation functions. Substituting the above equation into the weak form gives
By defining the local stiffness matrix, $K$, and the local force vector, $F$, as follows,

$$
K_{ij}^{(e)} = B(\psi_i, \psi_j) = \int_{x_A}^{x_B} a \frac{d\psi_i}{dx} \frac{d\psi_j}{dx} \, dx - \int_{x_A}^{x_B} \psi_i f \, dx - P_1^{(e)} \psi_i(x_A) - P_2^{(e)} \psi_i(x_B) = 0
$$

the above equation can be written concisely in matrix form:

$$
[K^{(e)}][\alpha^{(e)}] = [F^{(e)}].
$$

All that remains is to construct the approximation functions, $\psi_i$. These functions are constructed using the conditions mentioned in section 1.3. Namely, the selected functions must be sufficiently differentiable and satisfy the essential boundary conditions of the element. They must also be linearly independent and complete. Three of these conditions are met if we choose a linear approximation of the form $u_e(x) = c_1 + c_2 x$. In order to satisfy the remaining requirement, we require $u_e$ to satisfy the EBC of the element. Thus,

$$
c_1 + c_2 x_A = u(x_A) \equiv u_1^{(e)}
$$

$$
c_1 + c_2 x_B = u(x_B) \equiv u_2^{(e)}
$$

Solving for $c_1$ and $c_2$ in terms of $u_1^{(e)}$ and $u_2^{(e)}$ yields

$$
c_1 = \frac{u_1^{(e)} x_B - u_2^{(e)} x_A}{x_B - x_A} \quad c_2 = \frac{u_2^{(e)} - u_1^{(e)}}{x_B - x_A}
$$
By substituting and collecting coefficients, it can be shown that $u_e(x) = \sum_{i=1}^{2} u_i^{(e)} \psi_i^{(e)}$

where $\psi_1^{(e)} = \frac{x_B - x}{x_B - x_A}$, $\psi_2^{(e)} = \frac{x - x_A}{x_B - x_A}$, and $x_A \leq x \leq x_B$. This expression satisfies the essential boundary conditions of the element, and the approximation functions ($\psi_i$) are continuous and linearly independent over the element. These interpolation functions for the two-node linear element are shown in figure 1.6. Using these interpolation functions to approximate the dependent variable, the following matrix equations are obtained:

$$
\begin{bmatrix}
K^{(e)}
\end{bmatrix} = \frac{a_e}{h_e} \begin{bmatrix}
1 & -1 \\
-1 & 1
\end{bmatrix}
$$

$$
\begin{bmatrix}
F^{(e)}
\end{bmatrix} = \frac{f_e h_e}{2} \begin{bmatrix}
1 \\
1
\end{bmatrix} + \begin{bmatrix}
P_1^{(e)} \\
P_2^{(e)}
\end{bmatrix}
$$

where $h_e$ is the length of the element ($x_B - x_A$). All that remains is to assemble the equations derived for each element into the global finite element formulation. This process is straightforward and is similar for all element types, so it will not be discussed here.

![Figure 1.6](image-url)
1.4.2. One-Dimensional Fourth-Order Equations

Consider the problem of finding the function $u$ which satisfies the equation

$$\frac{d^4}{dx^4} \left( b \frac{d^2 w}{dx^2} \right) + f = 0$$

applied over the domain $0 < x < L$, where $b = b(x)$ and $f = f(x)$ are the given data of the problem. This case arises in the bending of beams.

As in the second-order case, the domain is discretized into subintervals, as shown in figure 1.7.

![Figure 1.7](image)

The variational form over a typical element $e$ is given by

$$\int_{e} \left[ \frac{d^2}{dx^2} \left( b \frac{d^2 w}{dx^2} \right) + f \right] dx = 0$$
Integrating twice by parts to transfer half of the differentiation from \( w \) to \( v \) yields

\[
\int_{x_a}^{x_b} \left( \frac{d^2v}{dx^2} \frac{d^2w}{dx^2} + vf \right) dx + v \frac{d}{dx} \left( \frac{b d^2w}{dx^2} \right)_{x=x_a}^{x=x_b} - \frac{dv}{dx} \left. \frac{d^2w}{dx^2} \right|_{x=x_a}^{x=x_b} = 0.
\]

Inspection of the boundary terms indicates that the specification of \( w \) and \( \frac{dw}{dx} \) at \( x = x_A \) and \( x = x_B \) constitute the EBC, and the specification of \( \frac{d}{dx} \left( \frac{b d^2w}{dx^2} \right) \) and \( b \frac{d^2w}{dx^2} \) at \( x = x_A \) and \( x = x_B \) constitute the NBC for the element. Thus, the basic unknowns at the element nodes are the primary variables, which for notational convenience will be written as

\[
w_1^{(e)} \equiv w(x_A) \quad w_2^{(e)} \equiv w(x_B) \\
\theta_1^{(e)} \equiv -\frac{dw}{dx} \bigg|_{x=x_A} \quad \theta_2^{(e)} \equiv -\frac{dw}{dx} \bigg|_{x=x_B}
\]

and the secondary variables, which will be written as

\[
Q_1^{(e)} = \frac{d}{dx} \left( \frac{b d^2w}{dx^2} \right)_{x=x_A} \quad Q_2^{(e)} = \frac{d}{dx} \left( \frac{b d^2w}{dx^2} \right)_{x=x_B} \\
Q_3^{(e)} = b \frac{d^2w}{dx^2} \bigg|_{x=x_A} \quad Q_4^{(e)} = b \frac{d^2w}{dx^2} \bigg|_{x=x_B}
\]

In the case of the bending of beams, the primary variables, \( w \) and \( \theta \), represent displacement and rotation, which are the DOF of the element, while the secondary variables, \( Q_{1,3} \) and \( Q_{2,4} \), represent shear forces and bending moments. Substituting this notation into the variational formulation gives bilinear and linear forms as follows:
\[ B(v, w) = \int_{x_A}^{x_B} b \frac{d^2 v}{dx^2} \frac{d^2 w}{dx^2} \, dx \]

\[ l(v) = -\int_{x_A}^{x_B} v(x_A) Q_1^{(e)} + \left( \frac{dv}{dx} \right)_{x_A} Q_2^{(e)} - v(x_B) Q_1^{(e)} + \left( \frac{dv}{dx} \right)_{x_B} Q_4^{(e)} \]

The variational form requires that the interpolation functions be continuous with continuous derivatives up to order 3 (so that \( Q_1 \) and \( Q_3 \) are nonzero), and that they allow the approximation for \( w \) to satisfy the EBC. Since there are a total of four conditions in an element, a four parameter polynomial is selected for \( w_e \):

\[ w_e(x) = c_1 + c_2 x + c_3 x^2 + c_4 x^3. \]

Forcing the constraints (EBCs), we get the system of equations

\[
\begin{bmatrix}
1 & x_A & x_A^2 & x_A^3 \\
0 & -1 & -2x_A & -3x_A^2 \\
1 & x_B & x_B^2 & x_B^3 \\
0 & -1 & -2x_B & -3x_B^2
\end{bmatrix}
\begin{bmatrix}
c_1 \\
c_2 \\
c_3 \\
c_4
\end{bmatrix}
= \begin{bmatrix}
w_1 \\
\theta_1 \\
w_2 \\
\theta_2
\end{bmatrix}
\]

Solving for the \( c_i \)'s in terms of \( w_1, w_2, \theta_1, \) and \( \theta_2, \) and substituting the results back into \( w_e \) gives the interpolation functions:

\[
\begin{align*}
\psi_1^{(e)} &= 1 - 3 \left( \frac{x - x_A}{x_B - x_A} \right)^2 + 2 \left( \frac{x - x_A}{x_B - x_A} \right)^3 \\
\psi_2^{(e)} &= -(x - x_A) \left( 1 - \frac{x - x_A}{x_B - x_A} \right)^2 \\
\psi_3^{(e)} &= 3 \left( \frac{x - x_A}{x_B - x_A} \right)^2 - 2 \left( \frac{x - x_A}{x_B - x_A} \right)^3 \\
\psi_4^{(e)} &= -(x - x_A) \left[ \left( \frac{x - x_A}{x_B - x_A} \right)^2 - \frac{x - x_A}{x_B - x_A} \right]
\end{align*}
\]
These interpolation functions are shown in figure 1.8.

Using these interpolation functions, the following element matrices result:

\[
\begin{bmatrix}
6 & -3h & -6 & -3h \\
-3h & 2h^3 & 3h & h^2 \\
-6 & 3h & 6 & 3h \\
-3h & h^2 & 3h & 2h^2
\end{bmatrix}
\]

\[
\left[ K^{(e)} \right] = \frac{2b}{h^3} \left[ \begin{array}{c} 6 \\ -3h \\ -6 \\ -3h \\
2h^3 \\ 3h \\ 6 \\ 3h \\
h^2 \\ 3h \\ 2h^2 \end{array} \right]
\]

\[
\left[ F^{(e)} \right] = -\frac{f h}{12} \left[ \begin{array}{c} 6 \\ -h \\ 6 \\ h \end{array} \right] + \left[ \begin{array}{c} Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \end{array} \right]
\]

where the element displacement vector is given by \( \{ u^{(e)} \} = \left\{ \begin{array}{c} w_1 \\ \theta_1 \\ w_2 \\ \theta_2 \end{array} \right\} \)
1.4.3. Two-Dimensional Scalar Valued Second-Order Equations

Consider the problem of finding the equation $u$ which satisfies the second-order partial differential equation (PDE)

$$
-\frac{\delta}{\delta x}\left(a_{11}\frac{\delta u}{\delta x} + a_{12}\frac{\delta u}{\delta y}\right) - \frac{\delta}{\delta y}\left(a_{21}\frac{\delta u}{\delta x} + a_{22}\frac{\delta u}{\delta y}\right) + a_{00}u - f = 0
$$

applied over some 2-dimensional region $\Omega$, as shown in figure 1.9, where $a_{ij} = a_{ij}(x)$ and $f = f(x)$ are the given data of the problem. This equation arises in 2-dimensional heat transfer in an isotropic medium.

The variational form is given by

$$
\int_{\Omega} \left[ -\frac{\delta}{\delta x}\left(a_{11}\frac{\delta u}{\delta x} + a_{12}\frac{\delta u}{\delta y}\right) - \frac{\delta}{\delta y}\left(a_{21}\frac{\delta u}{\delta x} + a_{22}\frac{\delta u}{\delta y}\right) + a_{00}u - f \right] dxdy = 0.
$$

Integration by parts (with some help from the divergence theorem) yields:
where $nx$ and $ny$ are the $x$ and $v$ components of the unit normal $\mathbf{n}$ on the boundary $\Gamma$, and $ds$ is the arc length of an infinitesimal piece of the boundary.

Inspection of the boundary term shows that specification of $u$ constitutes the EBC and the specification of $\mathbf{n} \cdot \mathbf{u}$ constitutes the NBC of the formulation. Thus, $u$ is the primary variable and $\mathbf{n} \cdot \mathbf{u}$ is the secondary variable. In the case of heat transfer in an isotropic medium, $u$ would represent temperature and $\mathbf{n} \cdot \mathbf{u}$ would represent heat flux across the element boundary.

Using this notation, the variational form can be written as

$$
\int_{\Omega} \left( \frac{\partial u_e}{\partial x} \frac{\partial \mathbf{w}}{\partial x} - \frac{\partial u_e}{\partial v} \frac{\partial \mathbf{w}}{\partial v} \right) dx dy \quad \text{subject to} \quad \mathbf{n} \cdot \mathbf{u} = 0.
$$

The variational form indicates that $u$ may be approximated by $u_e = \mathbf{w} \cdot \mathbf{n}$, where $u_e$ and $\mathbf{w}$ are the values of $u$ at the point $(x, v)$, and $\mathbf{w}$ are linear interpolation functions. The specific forms of $\mathbf{w}$ depend on the type of element used.
As mentioned above, the form of the interpolation functions $\psi_j$ depend on the element type. For three-node triangles, three linearly independent terms are required, so the interpolation function could take on the form $u(x,y) = c_1 + c_2 x + c_3 y$. For a four-node quadrilateral, the form $u(x,y) = c_1 + c_2 x + c_3 y + c_4 xy$ could be used. Higher order functions, such as $u(x,y) = c_1 + c_2 x + c_3 y + c_4 (x^2 + y^2)$ and $u(x,y) = c_1 + c_2 x + c_3 y + c_4 xy + c_5 x^2 + c_6 y^2$ could be used for higher order elements, such as a quadrilateral with a fifth node at its center, or a six-node triangle with nodes at its corners and mid-sides.

Examples of 3, 4, 5, and 6 node 2D elements are shown in figure 1.10.

![3-node, 4-node, 5-node, 6-node elements](image)

**Figure 1.10**

By solving for the constants $c_i$ and substituting, the interpolation functions for three-node triangle elements are found to be:

$$
\psi_1 = \frac{1}{2 A_e} \left( (x_2 y_3 - x_3 y_2) + (y_2 - y_3) x + (x_3 - x_2) y \right)
$$

$$
\psi_2 = \frac{1}{2 A_e} \left( (x_3 y_1 - x_1 y_3) + (y_3 - y_1) x + (x_1 - x_3) y \right)
$$

$$
\psi_3 = \frac{1}{2 A_e} \left( (x_1 y_2 - x_2 y_1) + (y_1 - y_2) x + (x_2 - x_1) y \right)
$$

where $A_e$ is the area of the element, and $(x_i, y_i)$ are the coordinates of node $i$. These interpolation functions are shown in figure 1.11.
The interpolation functions for the quadrilateral element turn out to be

\[
\psi_1(\xi, \eta) = \left(1 - \frac{\xi}{a}\right) \left(1 - \frac{\eta}{b}\right)
\]

\[
\psi_2(\xi, \eta) = \frac{\xi}{a} \left(1 - \frac{\eta}{b}\right)
\]

\[
\psi_3(\xi, \eta) = \frac{\xi}{a} \frac{\eta}{b}
\]

\[
\psi_4(\xi, \eta) = \left(1 - \frac{\xi}{a}\right) \frac{\eta}{b}
\]

if we take \((\xi, \eta)\) to represent a local coordinate system on a master rectangular element with sides \(a\) and \(b\), as shown in figure 1.12.

---

**Figure 1.11**

**Figure 1.12**
The interpolation functions for a four-node quadrilateral element are shown in figure 1.13.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure113.png}
\caption{Figure 1.13}
\end{figure}

Computation of the stiffness matrices for 2 dimensional elements by exact integration is not easy. Generally, the element matrices are computed using numerical integration.
1.4.4. Two-Dimensional Multi-Variable Equations

In the previous section, the finite element analysis of second-order, two-dimensional problems that involved only one dependent unknown was considered. Often, an engineer must face a system of coupled partial differential equations in as many dependent variables as the number of equations. Examples of two-dimensional problems in which coupled differential equations arise include plane elastic deformation of a linear elastic solid, the flow of an incompressible viscous fluid, and the bending of elastic plates with transverse shear strains.\[7\] The equations describing the behavior of a plate under plane stress loading can be written as:

\[
-\frac{\delta}{\delta x}\left(\frac{E}{1-\nu^2}\frac{\delta u}{\delta x} + \frac{\nu E}{1-\nu^2}\frac{\delta v}{\delta y}\right)h - h\frac{E}{2(1+\nu)}\frac{\delta}{\delta y}\left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x}\right) - f_x = 0
\]

\[
-h\frac{E}{2(1+\nu)}\frac{\delta}{\delta x}\left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x}\right) - \frac{\delta}{\delta y}\left(\frac{\nu E}{1-\nu^2}\frac{\delta u}{\delta x} + \frac{E}{1-\nu^2}\frac{\delta v}{\delta y}\right)h - f_y = 0
\]

where \( h \) is the plate thickness, \( E \) is the modulus of elasticity, and \( \nu \) is the Poisson's ratio of the plate material. These equations would be slightly different for the plane strain case, and for the axisymmetric case.

Each node in the finite element mesh would have 2 degrees of freedom: translation in the \( x \) and \( y \) directions. Thus, a three-node triangle element would have 6 DOF, and a four-node quadrilateral would have 8 DOF. The element stiffness matrices in this case will be quite large, and like the 2D scalar valued case, are generally computed using numerical integration.
1.4.5. Three-Dimensional Equations

Consider the problem of finding the function \( u \) which satisfies the partial differential equation

\[
- \frac{\delta}{\delta x} \left( k_1 \frac{\delta u}{\delta x} \right) - \frac{\delta}{\delta y} \left( k_2 \frac{\delta u}{\delta y} \right) - \frac{\delta}{\delta z} \left( k_3 \frac{\delta u}{\delta z} \right) - f = 0
\]

where \( k_i = k_i(x, y, z) \) and \( f = f(x, y, z) \) are given functions of position in a three-dimensional domain \( \Omega \). The domain is divided into some three-dimensional elements, such as tetrahedrons, wedges, or bricks, shown in figure 1.14.

![Figure 1.14](image)

Figure 1.14

The element matrices require the use of interpolation functions that are at least linear in \( x, y, \) and \( z \). The assembly of equations, the imposition of boundary conditions, and the solution of the equations are completely analogous to those described in the previous sections.
1.5. Modeling using Multiple Element Types

The elements and equations derived in the previous sections are but a small fraction of those available to an engineer performing a finite element analysis. As was demonstrated, each element is painstakingly developed over a certain domain for a certain type of application. Therefore, it would be wise to use these elements in the manner for which they were developed. Structures which are to be analyzed using the FEM are often not representable by a single type of element. While certain regions of the structure may be best represented by a particular type of element, that element type may not be at all appropriate for other regions of the structure. For example, to model a simple table, it would be impossible to choose a single type of element to perform the analysis. If the entire table were modeled using 3D elements such as tetrahedra or bricks, the cost of performing the analysis in terms of computer time and storage would be astronomical. Instead, the analyst would choose beam elements to represent the table's legs and plate elements to represent its top, as shown in figure 1.15.

![Figure 1.15](image-url)
1.6. Thesis Objective

The objective of this thesis project is to develop a computer program to perform automatic meshing of structures using different element types where appropriate. Just as an analyst with common sense would apply beam elements in long, narrow sections of the structure and plate elements in flat sections of the structure, so should the proposed computer program.

An additional requirement of the proposed program is to be able to mesh the same structure in several different ways at the user's discretion. In order to generate beam, plate, and brick elements in a commercial program such as ANSYS, the user must define geometries using lines, areas, and volumes, respectively. There is no way for the user to define a geometry once, and then mesh it using different element types. If a model is designed to be meshed with 3D elements, and the analyst changes his mind and decides to use 2D elements, the entire model must be discarded and a new one begun which will allow the generation of the desired mesh. The program written in this thesis project should be able to mesh the same geometric model in different ways, thus saving valuable time and effort.
2. Automatic Mesh Generation: Review of Related Literature

Since the finite element method has become such an important tool in modern engineering, researchers are striving to make the best possible use of the method. Currently, the area of automatic mesh generation is being very heavily researched. Current research efforts into mesh generation focus on developing fully automatic mesh generation techniques. A fully automatic technique is one in which only the object geometry and topology and mesh attributes are required as input.\[8\]

2.1. History of Mesh Generation

In the early days of the FEM, analysts were required to manually create meshes. This involved defining each and every node and element in the model. Specifically, for each element is was necessary to specify shape (triangle, quadrilateral, tetrahedron, hexahedron, etc.), vertices (by node number), coordinates of vertices, physical attributes of vertices, edges, and surfaces, and sub-domain (element) number. Furthermore, the finite element analysis was a batch process, and no feedback was available to indicate errors during model construction. It was only after the analysis was run and the results became suspect that the analyst would go back and check the validity of the model. Thus, the finite element method could not be practical for large or complex problems.

Naturally, researchers tried to improve the process by providing a graphics interface during mesh generation and automating the mesh generation process. In the late 1960's, methods were suggested for automatically determining the coordinates of interior nodes based on interpolation schemes applied to the boundary nodes. By the early 1970's, pre-processors with graphics capabilities had emerged. The introduction of low-cost,
high-resolution machines in the late 1970's produced a dramatic change in the way meshes were generated and checked. Some of the earliest finite element modelers were PDA Engineering's "PATRAN," and SDRC's "GEOMOD/SUPERTAB." During the late 1970's and early 1980's, the MacNeal-Schwendler Corporation's "MSGMESH" gained popularity. This pre-processor, with which it was possible to create models for analysis using MSC/NASTRAN, included methods for generating nodes and elements by simple mapping techniques applied over linear quadrilateral and hexahedral "grid point fields."[9]

Most commercially available finite element programs today include pre-processors with interactive or semi-automatic mesh generation methods. To generate a mesh using these methods, the user must first divide the geometry into simple mapable regions, such as quadrilaterals or hexahedrons. The user must then insure that the mesh will be continuous across region boundaries. The individual regions would then be meshed using transport mapping techniques.

Today, breakthroughs are being made in fully automatic mesh generation. Several algorithms are available to mesh arbitrary 2D planar regions, and 3D mesh generation algorithms are continually becoming more robust.

2.2. Automatic Meshing of 1D Regions

The meshing of 1D (Beam-like) regions is trivial. All that is required is to divide the domain into a number of line segments, each segment being an element with a node at each end. No research is necessary in this area.
2.3. Automatic Meshing of 2D Regions

Robust automatic mesh generation algorithms for 2D regions are now widely available. Most of the fully automatic methods can be grouped into three families: Volume Triangulation methods, Element Extraction methods, and Recursive Spatial Decomposition methods. Other semi-automatic methods do exist which are quite elegant. For example, the generation of meshes by the solution of partial differential equations can produce truly beautiful meshes.[10] However, this method requires user interaction to define the equations, and therefore is not fully automatic. The goal of this thesis is automatic mesh generation, so only fully automatic methods will be considered.

2.3.1. Volume Triangulation Methods

Mesh generators in this category are typically referred to as Delaunay generators, because they use the principle of Delaunay Triangulation. Numerous authors, including Cavendish[11], Barnhill[12], Lawson[13], Green and Sibson[14], Lewis and Robinson[15], Lee[16], Watson[17], Bowyer[18], Coulomb[19], and Garg and Budynas[20] have investigated mesh generators based on this technique.

The method of Delaunay Triangulation meshes the convex hull of a set of points. The set of points is the collection of nodes in the model -- the Delaunay technique does not address the issue of node generation. Various researchers have developed numerous methods of defining nodes on which to perform Delaunay Triangulation. In two dimensions, the task of node definition is relatively simple compared with the much greater task of creating a valid mesh from these nodes.
The convex hull of a set of points (nodes) is the boundary of the smallest convex domain containing the set of points\textsuperscript{121}. Given the set of points in figure 2.1(a), the convex hull can be thought of as the boundary acquired by enclosing the set of points with a "rubber band," as shown in figure 2.1(b).

![Figure 2.1](image)

**Figure 2.1**

There are several types of algorithms for performing Delaunay Triangulation. One of these types, known as Incremental algorithms\textsuperscript{12, 13, 18}, construct the triangulation by starting with any node, and inserting nodes one at a time into the mesh. Another type, Divide and Conquer algorithms\textsuperscript{15, 16}, recursively split the set of data points into equally sized subsets until elementary subsets are obtained, and then merge the resulting pieces.

These algorithms can further be classified into one-step and two-step methods, based on whether they produce the final mesh in a single step, or whether they first produce an arbitrary triangulation which is then optimized in a second step.
The method utilized in the present work is a modification of Watson's algorithm\cite{Watson}. Watson's algorithm meshes a given set of nodes through the following steps:

**Step 1:** Three nodes (which will be referred to as "StartNodes") defining a triangle are created such that the triangle encloses the given set of nodes, as shown in figure 2.2(a). This triangle constitutes the original mesh, consisting of one element and three nodes (the StartNodes). The circum-circle of the triangle is computed and stored. (The circum-circle of a triangle is the circle which passes through each of the triangles vertices, as shown in figure 2.2(b).)

![original mesh (one triangle)](image)

**Figure 2.2**

**Step 2:** Insert nodes, one at a time, into the mesh. To insert a node:

(a) Determine which triangles in the mesh contain the node being inserted within their circum-circles, as shown in figure 2.3(a).
(b) Compute the bounding polygon of the set of triangles found in step ➀(a) as shown in figure 2.3(b) by removing all triangle sides shared by two triangles, keeping only those that are a part of only one triangle. Make a list of the nodes on this bounding polygon.

(c) Delete the triangles found in ➀(a) from the mesh.

(d) Create new triangles using the nodes found in ➀(b) and the node being inserted, as shown in figure 2.3(c), and add them to the mesh. Compute and store the circum-circles of these triangles.

Figure 2.3
Step ③: Repeat Step ② for all nodes.

Step ④: Delete triangles which contain one of the three StartNodes. The remaining triangles are the Delaunay Triangulation of the convex hull of the nodes, as shown in figure 2.4.

![Final Mesh of Convex Hull](image)

**Figure 2.4**

Watson’s algorithm has been extended in this work to be able to determine which triangles are inside and outside the geometry, so that non-convex domains with or without holes can be meshed. Although this result was accomplished in the work of [Garg21], the technique used in this thesis is completely different.

Meshes created by Delaunay Triangulation have the property of optimal equiangularity. This means that the mesh generated for a set of points by Delaunay Triangulation contains triangles which are as equiangular as can be achieved with the given points. Thus, the mesh contains the best shaped elements possible for the given set of nodes.
2.3.2. Element Extraction Methods

Element Extraction methods are also known as *Advancing Front* methods, or simply as *Paving* methods. This class of mesh generators has been investigated by George\textsuperscript{[22]}, Sadek\textsuperscript{[23]}, Lo\textsuperscript{[24]}, Bui and Hanh\textsuperscript{[25]}, and Blacker et al.\textsuperscript{[26]}, to name a few.

This method begins with the object's boundary and generates nodes and elements inward from the boundary until the entire domain is discretized into elements. This process can be broken down into the following steps.

Step ①: Initialize the front: the object's boundary is represented as a polygonal discretization of the actual boundary.

Step ②: Analyze the front:

(a) Determine the "departure zone," the region where new elements will be generated.

(b) Create internal points and elements.

Step ③: Update the front.

Step ④: Repeat Steps ② and ③ until front is null (entire domain has been meshed).

Figure 2.5 shows an object at various stages of meshing by a paving algorithm. Here, the zone of departure is determined by examining the entire front. Figure 2.6 shows the same object at various stages of a paving algorithm that determines fronts by inflation.
The paving technique can also be used to create meshes consisting of quadrilaterals. The paper of T. D. Blacker et al.\cite{26} describes one such algorithm. In the work by Blacker and his group, arbitrary geometries with or without holes can be meshed with quadrilaterals. The method computes fronts by inflation to create quadrilateral elements with as regular a shape as possible. An object at various stages of this algorithm is shown in figure 2.7

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure27.png}
\caption{Figure 2.7}
\end{figure}
2.3.3. Recursive Spatial Decomposition

In two dimensions, the technique of Recursive Spatial Decomposition is known as *quadtree*. This is due to the fact that the object under consideration is divided into a quaternary tree. This technique has been investigated by Yerri and Shephard\cite{27}, Cheng et al.\cite{28}, and Shephard et al.\cite{29}, among others.

The method proceeds as follows: a first quadrilateral is defined which encloses the entire domain to be meshed, as shown in figure 2.8(a). This quadrilateral is then split into four quadrilaterals, as shown in figure 2.8(b). Each of the quadrilaterals that the boundary of the object passes through are again split into four quadrilaterals. Quadrilaterals which are either entirely inside or entirely outside of the domain are not divided further. The process of quadrilateral division continues recursively until some limit is reached. A final quadtree is shown in figure 2.8(c).

![Figure 2.8](image)

**Figure 2.8**
The final stage is to convert the quaternary tree into a valid finite element mesh. This is done by simply breaking up internal quadrilaterals into elements, and applying rules to the quadrilaterals which the boundary of the object passes through, which is not so simple. A finite element mesh generated by the quadtree method is shown in figure 2.9.

![Figure 2.9](image)
2.4. Automatic Meshing of 3D Regions

The automatic meshing of three dimensional regions is substantially more difficult than that of two dimensional regions. Automatic meshing algorithms for 3D regions are just now becoming as robust as those for 2D regions have been. The currently available methods for meshing 3D regions can be broken into the same three families as those for 2D regions: Volume Triangulation methods, Element Extraction methods, and Recursive Spatial Decomposition methods. The 3D methods in these three families are completely analogous to the corresponding 2D methods.

2.4.1. Volume Triangulation

Delaunay Triangulation is possible in three dimensions as well as in two dimensions. In fact, a generic form of Delaunay Triangulation can be used to mesh a set of points in any dimension\[^{30}\]. The three-dimensional case of Delaunay Triangulation is completely analogous to the two dimensional case with triangles being replaced by tetrahedrons, and circum-circles being replaced by circum-spheres. 3D Mesh generators based on Delaunay Triangulation have been investigated by Cavendish et al.\[^{31}\], Cendes and Shenton\[^{32}\], Field and Frey\[^{33}\], Baker\[^{34,35}\], George et al.\[^{36}\], and Yuen et al.\[^{37}\].

The bottleneck in 3D Delaunay Triangulation is the creation of nodes. Recall from section 2.3.1 that Delaunay Triangulation acts on a pre-defined set of points. While the creation of this set of points is not a difficult problem in two dimensions, in three dimensions it is a considerable task to create a set of points that will give a good mesh. An approach in which nodes are defined according to the hierarchy -- vertex, edge, face, and solid, has been suggested by Yuen et al.\[^{37}\]. In this approach, nodes are defined naturally at the vertices. Nodes on the edges, faces, and inside the solid model are then
generated by recursive subdivision. This is just one of many methods which have been investigated for generating a set of nodes.

Once the nodes are generated the process of constructing a mesh by Delaunay Triangulation is very similar to the process described in section 2.3.1. First, a tetrahedron is constructed which encloses the entire set of points. The given nodes are then inserted one at a time into the mesh by finding the tetrahedrons which contain the new node within their circum-spheres. The union of the tetrahedrons gives a bounding polyhedron (similar to the bounding polygon in 2D). These tetrahedrons are deleted from the mesh and new ones are constructed using the nodes on the vertices of the bounding polyhedron along with the insertion node. This insertion process is repeated until all nodes have been added to the mesh. A mesh generated by 3D Delaunay Triangulation is shown in figure 2.10.

\[\text{Figure 2.10}\]
2.4.2. Element Extraction Methods

Element extraction methods in three dimensions are referred to as *Advancing Front* methods, just as in two dimensions. Three dimensional element extraction methods are also referred to as *Plastering* methods. This class of mesh generators has been investigated by Wördenweber\textsuperscript{[38]}, Woo and Thomasma\textsuperscript{[39]}, Löhner and Parikh\textsuperscript{[40, 41]}, Peraire et al.\textsuperscript{[42]}, and Blacker et al.\textsuperscript{[26]}, to name a few.

The plastering method begins with a surface mesh of the object. The object's boundary must be meshed with a collection of polygons. The mesh generation algorithm then extends the surface mesh into the interior of the geometry from the boundary. Typically, the original representation of the boundary is a collection of triangles, and the object is meshed with tetrahedral elements. The algorithm of Blacker et al.\textsuperscript{[26]} has had some success meshing simple geometries with hexahedral elements beginning with a surface mesh of quadrilaterals. A mesh generated by the plastering method is shown in figure 2.11.

![Figure 2.11](image-url)
2.4.3. Recursive Spatial Decomposition Methods

In three dimensions, Recursive Spatial Decompositions are called Octree methods. This method is being researched by Cheng et al.\cite{28}, Shephard et al.\cite{29}, Shephard et al.\cite{43}, Yerri and Shephard\cite{44}, Grice et al.\cite{45}, and Perucchio et al.\cite{46}.

This method is completely analogous to the quadtree method in two dimensions. First, the geometry to be meshed is enclosed in a box. This box is then subdivided into eight blocks, which are each recursively subdivided into eight blocks until some limit is reached. In the final step, the blocks are converted into tetrahedral finite elements. A mesh generated by the Octree technique is shown in figure 2.12.

![Figure 2.12](image-url)
2.5. Expert Systems for Automatic Mesh Generation

Even with fully automatic mesh generators at their disposal, analysts must make many decisions about how to model a structure for FEA. After all, the answers obtained from a finite element analysis are only as good as the model itself. A good model consists of much more than the subdivision of the domain into a finite element mesh. The analyst must make decisions about the dimensionality of his model -- whether it can best be represented by 1D, 2D, or 3D elements, or some combination of them. Once dimensionality is determined, the analyst must decide on element type. This decision requires recognition of the physical problem being investigated: failure modes expected, the presence of loads and boundary conditions, geometry of the domain, etc. In one dimension, the analyst must choose between truss, beam, or other element types. In two dimensions, triangles (3 or 6 node), quadrilaterals (4 or 8 node), or a multitude of other elements may be selected. The element selection will depend on the analysis type, whether it be static, dynamic, modal, thermal, etc. For static analysis, the analyst may choose plane stress, plane strain, axisymmetric, or other types. There are about 88 elements catalogued for plate bending alone. In three dimensions, the analyst has a choice of tetrahedron (4 or 10 node), hexahedron (8 or 20 node), and many more. There are also special elements for gaps, crack tips, hyperelastic materials, fluid flow, rigid elements, and others.

Once element type has been selected, the analyst must decide upon an appropriate mesh density. Too coarse a mesh will give unacceptable approximation errors. Too fine a mesh will be costly in terms of computer time and storage. An optimum mesh will have variable sized elements -- course in areas of low stress gradient and finer in areas with higher stress gradient or applied loads. The mesh must be refined at areas of material or geometric discontinuity. Still, the variation in element size between adjacent elements
must not be too great. The transition must be made gradually. The element shapes must not become too distorted -- this reduces the accuracy of the analysis.

It is evident from the foregoing discussion that a considerable amount of experience and engineering judgment is required in establishing an optimum finite element mesh. The Expert System approach, discussed in Chapter 4, provides a useful tool to bring together the accumulated knowledge of an experienced analyst and the computational efficiency of modern fully-automatic mesh generation methods. Several researchers have developed prototypes of systems employing the expert system approach:

One type of expert system that has been investigated is the expert advisor. This type of system looks at the problem under consideration and suggests a suitable finite element program on which to perform the analysis. It also recommends element types. Some of these systems can also suggest mesh densities in various parts of the domain. A few examples of this type of system are FACS\[47\] (Flexible Automatic Conversion System), PLASHTRAN\[48\] (Plates and Shells Structural Analysis), and the Rank-Babuska expert system\[49\].

Some expert systems have been developed which actually create meshes for finite element analysis. Among these are AMEKS\[50\] (Automated Meshing Knowledge System), which deals solely with automatic meshing of 2D objects, ADEPT\[51\] (Automated Design Expert), which can recommend the most appropriate analysis package and the element type best suited to the constraints, create a mesh of the object using the available elements, and prepare a suitable input file for analysis, INTELMESH\[52\], which can mesh planar domains and 3D axisymmetric structures with well-shaped triangular elements, and a system developed by Reichert et al.\[53\], which can optimally mesh 2D domains. Each of
these expert systems utilizes one of the automatic mesh generation algorithms discussed in the previous sections.
3. Geometric Modeling

In order to automatically generate a mesh for some structure, it is necessary to have the structure's geometry stored in a way that is meaningful to the computer. The generic term geometric modeling covers a wide range of techniques for representing objects mathematically.

3.1. Modeling Techniques

Present day geometric modeling schemes can be classified into four basic categories: Sweep Representations, Cell Decomposition, Boundary Representation (B-Rep), and Constructive Solid Geometry (CSG). By far the most popular and most utilized of these techniques are B-Rep and CSG. However, no single scheme can efficiently support a wide range of applications. Each technique has its own strengths and drawbacks.

In this thesis project, a combination of two techniques has been used: the Boundary Representation and Constructive Solid Geometry schemes. The specific details of the modeling system used in this project will be discussed in chapter 5.
3.1.1. Sweep Representations

The term sweep representation refers to the volume generated by "dragging" a 2D set of points (a surface or area) along a line or curve. Figure 3.1 shows a sweep representation. The mathematical definition of this representation scheme is much more complex than that of the CSG and B-Rep schemes. However, this technique is becoming more and more popular because of the ease with which complex parts can be described.

![Sweep Representation](image)

**Figure 3.1**

3.1.2. Cell Decomposition

In the cell decomposition method, solids are represented as collections of simple cells, such as tetrahedrons. While this technique is useful for mass-property calculations, general algorithms for creating and combining such representations are not available, and validation is not trivial.[54]
3.1.3. Boundary Representation

Boundary Representation (B-Rep) is one of the most widely recognized techniques for describing geometries. In this method, solids are defined in terms of their boundaries, or enclosing surfaces. These boundaries are in turn represented in terms of unbounded surfaces, curves, and points which together may be used to define "faces." Figure 3.2 shows the boundary representation of an object.

Figure 3.2

Boundary representations are good sources of information for graphics displays. Unfortunately, they are bulky, difficult to create, and costly to store and transmit. They are also expensive to validate.
3.1.4. Constructive Solid Geometry

In Constructive Solid Geometry (CSG), a complex object is constructed using a set of regularized operators on primitive objects of known geometric shape. This technique is reminiscent of children playing with toy blocks. In CSG, the "blocks" consist of geometric shapes known as primitives, such as cubes, cylinders, cones, spheres, etc. These primitives are combined using regularized Boolean set operations: Union, Intersection, and Difference. The union of two solids results in an object which contains the set of all points which belong to either of the two original solids. The intersection of two solids results in an object containing the set of points common to both solids. The difference of two solids is an object which contains the set of points that belong to the first solid but not to the second. An object represented by Constructive Solid Geometry is shown in figure 3.3.

![Figure 3.3](image)

CSG representations are easy to create, store, and transmit, and are efficient for many planning and design problems. Another positive aspect of CSG representations is that they always define a valid solid. However, this scheme is not an effective source of information for graphics displays.
3.2. Boundary Evaluation

Because no single representation scheme is ideal for all purposes, general purposes geometric modeling packages typically employ more than one scheme. CSG is often used as the primary representation because of its ease of creation, storage, and because it always produces a valid representation. In order to view the model graphically, however, another representation is required. Often the secondary representation is B-Rep because it is relatively easy to create a graphic display from a B-Rep model. This presents the problem of constructing the B-Rep from the CSG model. This process is known as boundary evaluation.

To evaluate the boundary of a CSG representation, it is necessary to classify each edge of each primitive against the solid being represented to determine which portions of each edge are in, on, and outside of the region bounded by the solid, which is by no means an easy process. The basic function for doing this is the set membership classification function\(^{56, 57}\). The set membership classification function, denoted M[X, S], operates on two sets, X and S. The set X is typically the edge which is being classified against the solid S. A call to M[X, S] breaks X into three sub-sets: X in S, X on S, and X out S, corresponding to the portions of the edge X which are inside the solid S, on the boundary of the solid S, and outside of the solid S. This result is known as a membership classification result (MCR). There are several algorithms which use the set membership classification function for boundary evaluation\(^{58}\), however, these algorithms are inefficient due to the large number of edges which must be classified for complex solids, and the difficulty in computing the membership classification results for each edge.

Research has been done to find more efficient ways of computing a B-Rep from a CSG solid model. One approach, investigated by Pilz and Kamel\(^{59}\), is to represent the
surfaces of CSG primitives as collections of polygons. The polygonal representations of three primitives, the block, cylinder and sphere, are shown in figure 3.4.

![Block, Cylinder, Sphere](image)

**Figure 3.4**

To represent a solid made up of Boolean operations of several primitives, the polygons representing each primitive are repeatedly subdivided until the resulting polygons contain the intersection curves between the CSG primitives, as shown in figure 3.5(a). Adjoining co-planar polygons are then collected and merged by deleting their common edges, as shown in figure 3.5(b). This method produces a consistent, non-redundant polygonal discretization of the boundary of the solid, which may by easily displayed by simple perspective plots. This polygonal representation would also be an excellent starting point for the creation of a finite element mesh.

![Discretized Solid](image)

**Figure 3.5**
3.3. PADL-2

PADL-2, which stands for Part & Assembly Description (or Definition) Language, version 2, is the forerunner of all Constructive Solid Geometry systems. The initial development effort took place between 1979 and 1981 at the University of Rochester by a team of approximately 15 programmers as the Production Automation Project. Development of the PADL-2 system was supported by ten industrial sponsors and by the National Science Foundation under grant DAR78-25359. In the 1980's, PADL-2 moved to Cornell University, where development was continued as the Cornell Programmable Automation (CPA) project. The development of optional modules usable with the core system has continued throughout the 1980's, and has been supported by the industrial associates of the CPA and by Cornell University. PADL-2 is still being used to this day as a basis for research into automatic mesh generation by recursive spatial decomposition\cite{461}.

The core of PADL-2 is a very large and robust solid modeler. It consists of 779 separate files, written in FLECS/FORTRAN 66 and FORTRAN 77, spread over 29 directories. A complete listing of the PADL-2 source files is given in Appendix A. The primary solid representation is CSG, and PADL-2 has the ability to perform boundary evaluation of the CSG model to get a B-Rep. Most of the procedures which perform the computational geometry and boundary evaluation are in the directories CGPAK.DIR, BEVAL.DIR, and BFILE.DIR. The CGPAK.DIR (Computational Geometry Package) directory\cite{601} contains the files for performing set membership classification, which are the most important files for this thesis.

PADL-2 supports six primitive types: Block, Cylinder, Wedge, Cone, Sphere, and Torus. Examples of these primitives are shown in figures 3.6 - 3.11. Formal definitions of the point sets defined by each primitive type are given in Appendix B.
Figure 3.6

Figure 3.7

Figure 3.8

Figure 3.9

Figure 3.10

Figure 3.11
An informationally complete representation of virtually any typical unsculptured industrial part can be constructed using these primitives. However, this CSG representation is not well suited to all types of applications, particularly graphics displays. Thus, PADL-2 has the ability to evaluate the boundary of the CSG representation and store it in a B-Rep.

The main procedure for boundary evaluation is in the file BVALMI.FLX in the BEVAL.DIR directory (see Appendix A). This procedure calls dozens of other subroutines. Boundary evaluation in PADL-2 is a very complicated process, however, it is also very robust. In can accurately evaluate the boundary of any solid made up of arbitrarily oriented planar, cylindrical, spherical, and conical halfspaces. (The orientation of the halfspaces is altered by applying a rigid motion to the primitive made up of the halfspaces.) The resulting boundary representation consists of surfaces and edges. PADL-2 supports five surface types, or halfspaces: the planar halfspace, cylindrical halfspace, spherical halfspace, conical halfspace, and toroidal halfspace. These are all of the surface types required to represent each of the six primitives. The formal definitions of these halfspaces are given in Appendix B. Also supported are eight edge types, or curve segments: line, ellipse, parabola, hyperbola, cedge (intersection of 2 cylindrical halfspaces), scedge (intersection of a spherical halfspace and a cylindrical halfspace), kkedgedge (intersection of 2 conical halfspaces), and torus profile. These are all of the curve segments which can be formed by the intersection of any two of the first four surface types. PADL-2 restricts the use of toroidal halfspaces. Loosely, primitives that intersect the torus must be "coaxial" with it. The formal definitions of each of the curve segments are given in Appendix B. For a more complete description of the PADL-2 point sets (low level geometric entities), References [64] and [65] are suggested.
4. Expert Systems

An Expert System is a computer program which contains knowledge about a specific field. It is used to assist people who do not have expertise in that specific field. Human experts in any field are typically in great demand, and unsolved problems far exceed the number of human experts who can solve them. The generation of optimum finite element meshes is such a problem. As was detailed in Section 2.5, the generation of a good finite element mesh requires a great deal of knowledge and experience. Thus, an expert system would be useful to inexperienced engineers performing a finite element analysis.

It is hoped that in the future the work done in the area of automatic mesh generation in this thesis project will be utilized in an Expert System environment. It was proposed at the beginning of this project that an expert system would be developed. However, the work in the areas of geometric modeling and automatic mesh generation proved to be so involved that research into expert systems was halted. Still, a brief discussion will be given here on the structure and development of expert systems.

4.1. Expert System Structure

There are no standard Expert Systems. There are a wide variety of techniques used to create these systems, and they differ as widely as the programmers who develop them and the innumerable problems they are designed to solve. Most expert systems, however, can be divided into three components: the Knowledge Base, the Inference Engine, and the User Interface. This structure is illustrated in figure 4.1.
The Knowledge Base is the component of the expert system that contains the information about the problem to be solved. It contains declarative knowledge, facts about objects, events, and situations, and procedural knowledge, consisting of information about courses of action. Although many knowledge representation schemes are available, the most prevalent is the rule based production system. In a rule based system, the knowledge is stored in the form of "if-then" rules, which apply to the system's problem domain. The "if" portion of a rule is known as the predicate, while the "then" portion is called the antecedent. Some of the rules, called meta-rules, pertain to other rules or even to themselves.

Access to a vast amount of knowledge does not make one an expert. It is necessary to know when and how to apply the appropriate knowledge. Similarly, having a knowledge base does not make an expert system. The system must have another component which directs the use of the knowledge. This component is referred to as the Inference Engine.

The Inference Engine is the key to the whole expert system. It is capable of using the stored knowledge in such a way as to make high level decisions that are typically made by experts. There are two methods by which inference engines make decisions. The first is called "forward-chaining." In forward-chaining, the inference engine examines the pool of known (given) information. It then scans through the knowledge base for rules who's
predicate is satisfied by the known information. When such rules are found, the antecedent of the rule is added to the pool of known information. It is hoped that by continually increasing the pool of known information in this manner, the desired fact or goal will eventually be deduced.

The second method by which inference engines reach decisions is called "backward-chaining." This method works backward from the desired fact or goal. The inference engine scans through the knowledge base for rules whose antecedents would determine this fact or goal. When such a rule is found, its predicate is checked against the known (given) information for truth. If the predicate is not satisfied, the inference engine finds rules whose antecedents would satisfy it. This continues until all the predicates of the rules needed to determine the desired fact or goal are satisfied by the pool of known information.

The third component of a typical expert system is the User Interface. This is the component of the expert system that communicates with the human user. The communication is two-way. The user must be able to define the problem to the expert system, by describing known information and the desired fact or goal. The expert system must be able to respond with its recommendations. There are several other tasks performed by the user interface. The user may question the system's decisions and ask for its reasoning. The system may prompt the user for more information.

4.2. Expert System Development

It takes two types of human experts to develop an expert system. One is the knowledge engineer. This is typically a computer scientist or programmer who is an expert in artificial intelligence and is skilled at developing expert systems. The other is the
domain expert, who is the human expert that the expert system is being designed to emulate. This person has significant expertise in the field under consideration, and is the resource for the knowledge of the system.

The development of an expert system can be thought of in five stages: Identification, Conceptualization, Formalization, Implementation, and Testing. A flow chart of these stages is shown in figure 4.2.

<table>
<thead>
<tr>
<th>Determine the characteristics of the problem</th>
<th>Find concepts to represent the knowledge</th>
<th>Design structures to organize the knowledge</th>
<th>Formulate rules that embody the knowledge</th>
<th>Validate the rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identification</td>
<td>Conceptualization</td>
<td>Formalization</td>
<td>Implementation</td>
<td>Testing</td>
</tr>
</tbody>
</table>

**Figure 4.2**

The first stage of the expert system development process, Identification, involves the determination of the characteristics of the problem. The knowledge engineer interviews each of the domain experts in order to develop a more general idea of the purpose of the expert system.

Once the problem is formally defined, it is analyzed further to ensure that its specific and general requirements are understood. Concepts are found to represent the knowledge of the domain expert. This is the Conceptualization of the problem. The knowledge engineer frequently creates a diagram of the problem to depict graphically the relationships between the objects and processes in the problem domain.
During the Formalization of the problem, structures are defined to organize the knowledge. The knowledge engineer begins to select the development techniques that are appropriate to the particular system. If a rule based system is being developed, the knowledge engineer develops a set of rules for review by the domain experts. Rules are revised until there is full agreement.

A prototype of the expert system is developed in the Implementation stage. This helps the knowledge engineer to determine if the correct techniques were chosen.

Once the prototype has been completed, Testing, the final stage, can begin. The knowledge engineer revises the structure and implementation of the expert system until the system provides solutions as valid as those of the domain experts.
5. Automatic Mesh Generation using Multiple Element Types

With the finite element method being such an important tool in modern engineering, it is necessary to eliminate the bottlenecks in performing a finite element analysis. Huge advances in computer speed and memory since the early days of the FEM have made the solution phase of finite element analyses a fast, straightforward process. At the same time, the development of the FEM into a complex and powerful tool has vastly increased the amount of expertise required to create a valid and appropriate finite element mesh. Present day finite element programs such as ANSYS and NASTRAN have element libraries consisting of hundreds of types of elements. Each of these elements is customized for a certain type of analysis over a certain domain. Thus, it requires a great deal of experience just to select the proper element type for a given problem. Generating a useful mesh with the selected elements is a much more difficult problem still. For these reasons, the bottleneck in performing a finite element analysis lies in the mesh generation. As a result, the topic of automatic mesh generation for finite element analysis is currently being very heavily researched.

As was demonstrated in Chapter 2, present day automatic mesh generators are limited in the domains that they can mesh. Mesh generation algorithms are classified as either 2-dimensional or 3-dimensional, and use only one type of element for the mesh. However, not all domains can be accurately modeled using only 2-dimensional or only 3-dimensional meshes. What is needed is a mesh generator that can apply the appropriate elements over different regions of the domain. That is what was attempted in the program "CSGMesh."
5.1. CSGMesh Computer Program

A computer program, called CSGMesh, was written as part of this thesis. This program was begun in the Fall of 1991, with no understanding of how large it would become. Over a period of a year and a half, CSGMesh has grown into a powerful program consisting of well over 25,000 lines of code. The source code is broken into about 70 separate units, each defining an object type or some aspect of program functionality. An object hierarchy containing 75 object types was created (see Appendix C for object hierarchy chart, and Appendix D for object field and method listings). The result of all of this is a powerful automatic mesh generation system.

CSGMesh was written in Borland's Turbo Pascal® version 6.0. This language was chosen for its support of an object-oriented environment, and because of the author's familiarity with Pascal. The user interface was written using Turbo Vision™, which creates a mouse-controlled, multi-window, menu driven system that is quite user friendly.

Under Borland's License Agreement, some restrictions apply to the use and distribution of CSGMesh. All CSGMesh executable and source code files may be freely distributed, but all Borland files may be distributed in executable form only (that is, as compiled .TPU files). The compiled CSGMesh program (executable file) may be used and distributed with no restrictions.

As complete a description as possible of CSGMesh will be given here, so that in the future it will be possible to use, modify, and expand this program to its fullest potential.
5.1.1. **Program Overview**

A typical session using CSGMesh can be divided into 4 stages, as follows:

1. Create and edit a text file which describes the geometry to be meshed.
2. Build a CSG Tree from text file. Make any desired modifications to the CSG Tree. Select materials, Segment Types (which define how regions will be meshed), and other options.
3. Break the CSG Tree into Segments. Modify the Segments as desired.
4. Mesh the Segments. Mesh data can then be written to output file.

Each of these stages will be discussed in detail in the sections to follow.

5.1.1.1. **Input File**

The first stage in a typical CSGMesh session is to define or specify a file which contains information describing the geometry to be meshed. CSGMesh looks for an ASCII input file to describe the part geometry. Existing files can be modified by selecting "File/Open." (i.e. Choosing the "Open" option from the "File" menu at the top of the CSGMesh screen as shown in figure 5.1.) New files can also be created by selecting "File/New." Typically, file names follow the format "filename.CSG."

The .CSG (dot-CSG) files contain definitions of primitives and Boolean operations on those primitives which describe the geometry to be meshed. Currently, two primitive types are fully supported: the Block (BLO) and the Cylinder (CYL). "Skeleton" units have been included for four additional primitive types: Wedges (WED), Cones (CON), Spheres (SPH), and Torii (TOR). These units allow for the simple addition of the definition of these primitives to CSGMesh in the future. Section 5.2.1 deals specifically with adding additional primitive definitions to CSGMesh, and describes the process in detail.
Defining primitives in the input file is straightforward. To define a primitive, a name is chosen, and that name is set equal to a solid defined by type, parameters, and rigid motion:

\[
\text{name} = \text{BLO}(X=x\text{-value}, \ Y=y\text{-value}, \ Z=z\text{-value}) \ \text{MOVEDBY} \ \text{rigid-motion};
\]

\[
\text{name} = \text{CYL}(R=r\text{-value}, \ H=h\text{-value}) \ \text{MOVEDBY} \ \text{rigid-motion};
\]

The solid's name is simply a string of characters that uniquely represents that solid. The solid's name may contain any character except a space. No two solids may have the same name. CSGMesh sometimes adds a back quote (') to the end of a solid's name to ensure that the name is unique.
The solid's type is given by its three letter code: BLO for blocks and CYL for cylinders. Each primitive type has its own parameters. To define a block, the X, Y, and Z dimensions must be specified. If any of these parameters are omitted from the description, they are set to the default value of one. If any parameter is specified more than once, the first value specified is the one that will be used. Blocks are defined in standard position by default. A block in standard position is shown in figure 5.2.

![Figure 5.2](image)

A block (or any solid) can be moved to a different position by specifying a rigid motion. Rigid Motions are defined after the solid definition after the word MOVEDBY. Rigid motions may be defined by a combination of any of the following:

- \( \text{MOVX}=\text{value} \)
- \( \text{MOVY}=\text{value} \)
- \( \text{MOVZ}=\text{value} \)
- \( \text{ROTX}=\text{value} \)
- \( \text{ROTY}=\text{value} \)
- \( \text{ROTZ}=\text{value} \)
- \( \text{DEGX}=\text{value} \)
- \( \text{DEGY}=\text{value} \)
- \( \text{DEGZ}=\text{value} \)

The MOV(i) motions translate the solid in the specified direction relative to the global coordinate system (i.e. along the global "i"-axis). For example, \( \text{MOVX}=1.5 \) would move the solid 1.5 units along the global X-axis.
The ROT(i) motions rotate the solid about the global "i"-axis an amount equal to value radians. For example, ROTY=1.570796327 would rotate the solid 1.570796327 radians (90 degrees) about the global Y-axis.

The DEG(i) motions are identical to the ROT(i) motions, except that the value is specified in degrees rather than radians. Thus, DEGY=90 would accomplish the same motion as ROTY=1.570796327.

As many rigid motions can be specified as will fit on the line of text. Rigid motions are applied in the order that they are written in the input file. Consequently, the solid defined by name = solid MOVEDBY MOVX=2, DEGZ=45; is not the same as the solid defined by name = solid MOVEDBY DEGZ=45, MOVX=2;. One must be careful when using rigid motions to specify the motion correctly, so that the instance of the solid will indeed be in the desired location.

To define a cylinder, the Radius (R), and Height (H) must be defined. If either R or H is omitted from the definition, the value of that parameter is set to the default value of one. When defining cylinders, the user has the option of specifying the Diameter rather than the Radius, by defining D=value instead of R=value. If both Radius and Diameter are specified, the value specified by Radius will be used. If any parameter is specified more than once, the first value will be used. A cylinder in its standard position is shown in figure 5.3. A cylinder can be moved to any position by specifying a rigid motion, as described above.
Three operations are also defined: the Union (UN), the Difference (DIF), and the Intersection (INT). Operations act on two solids. As their names imply, the Union takes the union of two solids (that is, the solid made up of all points in either solid), a Difference takes the difference of two solids (all the points in the first solid that are not in the second solid), and an Intersection takes the intersection of two solids (only those points contained in both solids). Operations (UN, DIF, INT) are defined by setting a name equal to a solid, \( \text{OPERATION} \), a second solid:

\[
\begin{align*}
\text{name} &= \text{solid1 UN solid2}, \\
\text{name} &= \text{solid1 DIF solid2}, \\
\text{name} &= \text{solid1 INT solid2};
\end{align*}
\]

Again, \( \text{name} \) is any collection of characters except space. Here, \( \text{solid} \) denotes either a pre-defined name, or a valid primitive definition. Figure 5.4 shows the Union, Difference, and Intersection of the two solids shown in figures 5.2 and 5.3. Note that the difference of two solids depends on which solid is specified first (before the DIF operation), and which solid is specified second (after the DIF operation). The solid defined by \( \text{name} = \text{solid1 DIF solid2} \); is not the same as the solid defined by \( \text{name} = \text{solid2 DIF solid1} \). Unions and Intersections do not depend on the order of input.
An example of a typical .CSG file is shown in figure 5.5. Complete syntax diagrams for .CSG files are given in Appendix E.

```plaintext
b1 = blo(x=4, y=3, z=0.5);
c1 = cyl(r=2.5, h=0.5) movedby movx=2.5;
b2 = blo(x=1, y=1, z=0.5) movedby deqz=45;
b3 = blo(x=1, y=1, z=0.5) movedby deqz=-45, movx=1.5;
thing = c1 un b1;
thing = thing dif b3;
surface = thing dif b2;
```
5.1.1.2. CSG Tree

Once the input file has been completed, CSGMesh can use it to construct a CSG Tree. To do this, the user must choose "CSG-Tree/Build Tree from File" as shown in figure 5.6. If there is a file open on the desktop, CSGMesh will use that file to construct the CSG Tree. If no file is open, CSGMesh will prompt the user to specify an existing .CSG file. The CSG Tree from the file shown in figure 5.5 is shown in figure 5.7.

Figure 5.6
Figure 5.7

CSGMesh shows this tree structure in the CSG-Tree window. CSGMesh also displays a list of solids in the SolidList window. Both of these windows are shown in figure 5.8.

Figure 5.8
The user has a lot of options for editing the solids in the CSG Tree. The most important option in terms of this thesis is the choice of segment type, or SegType. A solid's SegType determines how that solid will be meshed: 1D, 2D, or 3D. CSGMesh determines default SegTypes for each primitive and (where possible) for each operation, and displays them in the SolidList window. The user may change a solid's SegType by inspecting that solid. (A solid can be inspected by double-clicking on its name, either in the CSG-Tree window, or in the SolidList window, or by choosing "CSG-Tree/Inspect Solid.") Changing the SegType of an operation changes the SegTypes of all solids underneath that operation in the CSG-Tree. While inspecting a solid, the user also has the ability to completely re-define the solid. The parameters (X, Y, and Z for Blocks, and R and H for Cylinders) as well as the rigid motions can be changed. It is also possible to define materials for solids. If a material is chosen for an operation, all of the solids under that operation in the CSG Tree will automatically use that material. Materials can be named and their properties defined by choosing "Options/Define Material."

The user also has the ability to view the CSG Tree graphically. Selecting CSG-"Tree/Sketch CSG-Tree" displays all primitives in the CSG-Tree in graphics mode. The primitives in the CSG-Tree of figure 5.5 are displayed graphically in figure 5.9.
5.1.1.3. Segments

Once the user is satisfied with the state of the CSG Tree and the SolidList, he may select "Segments/Create Segments" from the pull down menus, as shown in figure 5.10. This breaks the CSG-Tree up into Segments, which are the regions of the structure that will be meshed. The process of Segmentation is the key to this thesis project; Section 5.2 deals with this topic at great length.

Figure 5.10
Segments may be altered in a number of ways. The fundamental Segment Type (i.e. Beam, Plate, or Brick) may be changed. Materials may be defined for segments. Mesh densities may be defined for segments. In actuality, "mesh density" is a misnomer. What is really specified by "mesh density" is node spacing. (Mesh density is by definition nodes per unit length, whereas in CSGMesh, "mesh density" defines distance between adjacent nodes -- that is, unit length per node.)

Segments can also be displayed graphically. The user has several options for drawing segments. Segments can, of course, be drawn as solids. Each segment has the portion of the CSG Tree that made it, and can be drawn as this solid. It is important to note that this solid is the true "segment" of the CSG Tree. All meshing is done on a simplified representation of this solid. During the CSG Tree segmentation, the root solids of segments are completely boundary-evaluated, so the drawing that results is an accurate wire frame of the segment's solid. Figure 5.11 shows the segment created from the CSG Tree in figure 5.8. This segment is drawn as a solid.

Figure 5.11
Another option when drawing segments is to draw their representation. It was mentioned above that meshing is done on a simplified representation of the solid. The representation of a Beam Segment is a 1D line, the representation of a Plate Segment is a 2D surface, and the representation of a Brick Segment is a 3D solid (the same solid as discussed above). The segment in figure 5.12 was constructed as a Plate segment. Therefore, its representation is a 2D surface. The program analyzes the solid's boundary representation to determine which direction is the "two-dimensional" direction, and creates a 2D surface representation at the center of the thickness of the solid in this direction. This representation is shown in figure 5.12.

Figure 5.12
5.1.1.4. Options

The user has several choices under the Options menu (shown in figure 5.13). He can choose an analysis type from Static, Modal, Thermal, Linear Transient Dynamic, Nonlinear Transient Dynamic, and more. However, only Static analysis is currently supported. The user can also choose the format in which he wants the output file to be written. Four formats are currently supported: ANSYS version 5.0, ANSYS version 4.4, NASTRAN, and Algor Supersap.

Figure 5.13
The Options menu includes selections for defining and editing materials. Materials may be assigned a material number, a name, and several properties: Modulus of Elasticity, Poisson's Ratio, Shear Modulus, Mass Density, Thermal Expansion Coefficient, and Structural Damping Coefficient. A material definition dialog box is shown in figure 5.14.

Figure 5.14
Although "Boundary Conditions" is included in the Options menu, no support of loading or boundary conditions has been included in this thesis project. It is possible to apply boundary conditions to a model by meshing the model using CSGMesh, outputting the mesh data to a file, and loading the file into a commercial finite element package, such as ANSYS, NASTRAN, or Supersap.
5.1.1.5. Meshes

The user of CSGMesh has a lot of flexibility when it comes to meshing. Each segment can be assigned its own mesh density. If a segment's mesh density is undefined, that segment will use the global mesh density, which can be assigned by selecting "Mesh/Mesh Density" as shown in figure 5.15. As was mentioned in section 5.1.1.3, "mesh density" is somewhat of a misnomer. CSGMesh defines mesh density as the distance between adjacent nodes. It would be more accurate to refer to "mesh density" as "node spacing."

![Image of CSGMesh interface]

**Figure 5.15**
The user also has the option of selecting meshing techniques. Currently, only one technique for each segment type is supported: Beam Division for Beam Segments, Delaunay Triangulation for Plate Segments, and 2½D Mesh Extrusion of 2D Mesh (Created by Delaunay Triangulation) for Brick Segments. It is sincerely hoped that others will extend CSGMesh in the future to include other meshing techniques.

Segments may be meshed individually by selecting "Mesh/Mesh a Segment," or all segments can be meshed simultaneously by selecting "Mesh/Auto Mesh all Segments." Figures 5.16 and 5.17 show a mesh of the segment in figure 5.11: Figure 5.16 as a plate segment and figure 5.17 as a brick segment. Once all segments are meshed, the individual meshes can be modified. Meshes created by Delaunay Triangulation (Plate Segments) have the ability to be smoothed. This moves each interior node to the average position of the nodes adjacent to it. Segments can also have their meshes deleted, and be remeshed. Figure 5.18 shows the Plate segment re-meshed with a finer mesh (smaller "mesh density" [node spacing]). Meshes can also be altered by adding nodes. Figure 5.19 shows the Plate segment with a courser mesh, and figure 5.20 shows this same mesh with a node added.
Figure 5.16

Figure 5.17
Figure 5.18
Figure 5.19

Figure 5.20
Once all Segments are meshed, the interfaces between the segments can be modified to ensure continuity between the individual Segment meshes. This is a two stage process. The first stage Merge the meshes. This is done by selecting "Mesh/Merge Meshes." Figure 5.21 shows the interface between two Plate Segments. These two segments were meshed with different mesh densities, and so the nodes do not line up at the interface. Merging the meshes adds a node to each mesh at the location of an interface node in the other mesh, so that the two meshes contain corresponding nodes along the interface, as shown in figure 5.22. Still, these nodes may not be coincident, because the thicknesses of the plate segment's solids may keep the representation of the plates from meeting at a single edge. By selecting "Mesh/Connect Meshes," the nodes on the interface are moved to line of intersection between the two plate segment's representations, as shown in figure 5.23.

Figure 5.21
Figure 5.22

Figure 5.23
5.1.1.6. Output File

The last stage of a CSGMesh session is writing the mesh data to an output file. Currently, four formats are supported: ANSYS version 5.0, ANSYS version 4.4, NASTRAN, and Algor Supersap. (Output Files for the mesh in figure 5.19 in Algor Supersap, ANSYS version 5.0, and NASTRAN formats are given in Appendices F, G, and H, respectively.) The output file is created so that the finite element model generated by CSGMesh can have loads and boundary conditions applied, and be analyzed by one of the finite element software packages listed. The model of the plate segment used throughout this chapter is shown in figures 5.24 and 5.25 after having been read into Supersap and ANSYS, respectively. In ANSYS, loads and boundary conditions were applied, and the analysis was run. The resulting stress in the plate is shown in figure 5.26.

Figure 5.24
Figure 5.25

Figure 5.26
5.1.2. Solid Representation

At the onset of this thesis project, geometric modeling systems were researched to determine the best solid modeling method in terms of the goals of the thesis project. PADL-2 was considered as a representation scheme. It was proposed that the work done in this thesis could use PADL-2 as a foundation and build upon it. However, once PADL-2 was thoroughly investigated, this idea was dropped. The solid modeling core of PADL-2 was written in the late 1970's and early 1980's in FLECS/FORTRAN 66 and FORTRAN 77. Complex data structures representing geometric entities were constructed out of integer and real arrays. While this may have been acceptable at the time, computer programming techniques have evolved so much since PADL-2 was written that by today's standards, PADL-2 is inflexible, non-extendible, unreadable, and inefficient. Modern data structures such as records and objects are able to handle the representations of solids and other geometric entities much more efficiently. Also, because of the sheer size of PADL-2, it would literally take years for a person unfamiliar with PADL-2 to learn enough to use it effectively as a basis to build upon.

For these reasons, it was concluded that it would be more economical to develop a new solid modeling system using modern programming techniques than it would to try to build upon the dubious foundation of PADL-2. Thus, the solid modeling system in CSGMesh was written from scratch with PADL-2's structure as a model.

The remainder of Section 5.1.2., which deals with the solid representations employed by CSGMesh, is broken up as follows: first, solids are discussed. Solids are what make up the CSG representation of an object. CSGMesh also performs boundary evaluation to acquire a B-Rep of the object. The B-Rep consists of surfaces and edges,
which are discussed next in the section on boundary representation. Finally, the actual procedures for performing the boundary evaluation are reviewed.

5.1.2.1. Solids

Solids are represented in CSGMesh by data structures known as objects. Object oriented programming is a relatively new approach to computer programming. It will be assumed for the purposes of this thesis that the reader has a working knowledge of object oriented programming.

The solid object type, tSolid (object types all begin with the letter "t"), has several fields: Name, Surface, Box, Material, SegType, Parent, Next, and Motion, which is inherited from ancestor tGeometricEntity. Name is a pointer to a string (pString), which stores the name defined in the .CSG input file. Surface is a pointer to a surface (pSurface), which is originally set to nil. When the solid's boundary is evaluated, the field Surface is used to store the boundary representation. Box is a pointer to the solid's bounding box (pBoundingBox), which is used to simplify the computation of set membership classifications. Material is an integer specifying the material out of which the solid is made. SegType is one of either BEAM, PLATE, or BRICK. The value of this field defines how the solid will be broken into a segment and eventually meshed. Parent is a pointer to the parent solid in the CSG tree (pSolid). Next points to the next solid in the linked list that is used to store all of the solids in CSGMesh. This linked list is held by the pSolid variable SolidList. TSolid also has 31 methods, not counting the ones inherited from its ancestors. These methods handle a variety of actions. Each of these fields and methods is described in detail in Appendix D.
Solids are linked to each other in two ways: in the \textit{SolidList}, and in the \textit{CSGTree}. Both of these are variables of type \textit{pSolid}. The variable \textit{SolidList} points to the first solid in that list. The variable \textit{CSGTree} points to the root solid of the CSG tree. Figure 5.27 shows graphically these relationships.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5_27}
\caption{SolidList}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5_28}
\caption{CSGTree}
\end{figure}

\textbf{Figure 5.27}
All primitive and operation solid types are derived from the ancestor \textit{tSolid} as shown in figure 5.28. Complete descriptions of each of the fields and methods are given in Appendix D, so only a brief description of each type will be given here.

![Diagram of object hierarchy]

**Figure 5.28**

5.1.2.1.1. Blocks

Blocks are based on the PADL-2 representation of the block primitive, as are all geometric entities in CSGMesh. A block has three real variable fields in addition to the fields inherited from \textit{tSolid}. These are \(x\), \(y\), and \(z\), which define the dimensions of the block. Blocks also have an integer field called \textit{SegAxis}, which determines which axis of the block will be the long axis if the block is used as a beam, or the surface normal axis if the block is used as a plate.

5.1.2.1.2. Cylinders

Cylinders have two real fields in addition to those inherited from \textit{tSolid}. These are \(r\) and \(h\), which define the cylinder's radius and height (or length). Again, this representation is based on the PADL-2 representation of the cylinder primitive.
5.1.2.1.3. Other Primitives

A framework has been supplied for the development of four additional primitive types: Wedge, Cone, Sphere, and Torus. These four, along with the two primitive types already mentioned, complete the set of primitives available in PADL-2.

No methods have been provided for any of these four primitives. They do, however, have fields based on their representations in PADL-2. The wedge primitive has real fields \( x, y, \) and \( z \), the cone primitive has real fields \( r \) and \( h \), the sphere primitive has real field \( r \), and the torus primitive has real fields \( R_{\text{min}} \) and \( R_{\text{maj}} \). The representations based on these values are described in Appendix B.

The steps necessary to develop these primitives fully are detailed in section 5.2.1. (Further Primitive Types).

5.1.2.1.4. Unions

Unions are by far the most used of the operations. They provide a way of building useful structures out of the block and cylinder primitives. \( t\text{Union} \) inherits two pointers to solids \( p\text{Solid} \) from its ancestor \( t\text{Operation} \). It is these pointer fields, \( \text{Right} \) and \( \text{Left} \), which define the relationships in the CSG tree. The structure of \( t\text{Union} \), and all \( t\text{Operation} \) derived solids (including \( t\text{Difference} \) and \( t\text{Intersection} \)) is shown graphically in figure 5.29. The extension of this figure to a full CSG tree is obvious.
5.1.2.1.5. Differences

With differences, it is possible to "cut holes" in solids. Difference is the second most often used operation after union. Like \textit{tUnion}, \textit{tDifference} inherits the \textit{pSolid} fields \textit{Left} and \textit{Right} from its ancestor \textit{tOperation}.

5.1.2.1.6. Intersections

Intersections are seldom used in the building of everyday solids, although they do have their uses. Like \textit{tUnion} and \textit{tDifference}, \textit{tIntersection} inherits the \textit{pSolid} fields \textit{Left} and \textit{Right} from its ancestor \textit{tOperation}.

5.1.2.2. Boundary Representation

While the object types for solids provide a way of defining an informationally complete, unambiguous representation of structures, this CSG representation is not ideal for all purposes. For example, it is not possible to easily produce accurate graphics displays of solids represented by a CSG tree. For this and other purposes, another representation scheme is needed. The scheme of choice is B-Rep, or boundary representation. This scheme describes an object by its bounding surfaces and edges.
Thus, object types used to define surfaces and edges have been developed in CSGMesh. These types are \textit{tSurface} and its descendants, and \textit{tEdge} and its descendants.

The boundary representation of each solid in the CSG tree can be evaluated and stored with that solid. The data structure used to handle this representation is a series of linked list. Each solid has a pointer to a surface (\textit{pSurface}), which is used to store the boundary representation of that solid. Each surface has a pointer to the next surface in the list (type \textit{pSurface}), and a pointer to its list of edges (type \textit{pEdge}). Thus, each solid with its boundary representation is a series of linked lists, as shown in figure 5.30. When one imagines the whole CSG tree and SolidList with boundary representations for each solid, the whole data structure representing the solid model becomes a forest of trees and lists.

![Diagram](image)

\textbf{Figure 5.30}
5.1.2.2.1. Surfaces

The surface object type, tSurface, has three fields, plus the motion inherited from its ancestor tGeometricEntity. Two of them have already been mentioned: the pointers to the edges and the pointer to the next surface in the list. The other field is an integer variable called Outside. This variable takes on the value +1 or -1. A positive value means that the surface bounds the solid such that the solid material is on the side away from the surface normal. A negative value means the opposite; that the solid material is on the side toward the surface normal. This can be illustrated with the example of a plane surface.

The surface normal of a plane surface points in the direction of the positive local z-axis, as shown in figure 5.31(a). A positive value for Outside indicates that the surface bounds material on the negative z-axis, as shown in figure 5.31(b), while a negative value for Outside indicates that the solid material is on the positive z-axis side as shown in figure 5.31(c). The use of the field Outside in cylinder surfaces determines whether the cylinder represents a solid rod or a hole by specifying which side of the cylinder surface the solid material is on.

![Figure 5.31](image-url)
There are five object types descended from tSurface: tPlaneSurf, tCylinderSurf, tConeSurf, tSphereSurf, and tTorusSurf. Of these five, only the first two are developed fully. The other three can never occur with only blocks and cylinders as primitives, and therefore are not developed. Just as with the non-supported primitives, a framework has been provided in CSGMesh so that these surface types may be developed in the future. Complete descriptions of the fields and methods for tSurface and all of its descendant object types are given in Appendix D.

5.1.2.2.2. Edges

The object type tEdge has been developed to represent edges of solids. This type is descended from type tGeometricEntity, from which it inherits a motion. Its fields include Next, a pointer to an edge (pEdge), which has already been mentioned, and two real values, t0 and t1. These variables specify the beginning and ending values for the parametric representation of the curve. TEdge has eight descendants: tLineEdge, tEllipseEdge, tCCEdge, tParabolaEdge, tHyperbolaEdge, tKKEdge, tSCEdge, and tTorusProfileEdge. Each edge type is based on the PADL-2 format, and descriptions of the parametric representation for all of these types can be found in Appendix B. Only the first three edge types are currently supported. These are the three which can be formed by the intersection of the two supported surface types: the intersection of a plane and a plane gives a LineEdge, the intersection of a plane and a cylinder gives and EllipseEdge, and the intersection of a cylinder and a cylinder gives a CCEdge. The remaining five edge types have frameworks provided so that they may be developed in the future. Complete field and method descriptions for all edge types are given in Appendix D.
5.1.2.3. Boundary Evaluation

Each type of solid has a method called *EvaluateBoundary* for performing boundary evaluation. For the primitives, this method is straightforward. Blocks and cylinders simply compute their surfaces and edges based on their parameters and motions, and store them with their *Surface* pointers.

Operations, on the other hand, are much more involved. Each operation type, *tUnion*, *tDifference*, and *tIntersection*, has its own method for evaluating its boundary.

Unions evaluate their boundaries in six stages. The first stage is to take all of the surfaces common to both of the union's *Left* and *Right* sub-solids and put them in a list, called *CommonList*. Each surface on *CommonList* includes all of the edges from both the *Left* and *Right* sub-solids which are associated with that surface. In the second stage, the surfaces on *CommonList* are then deleted from the *Left* and *Right* sub-solids, which leaves each sub-solid with unique surfaces. The third stage is to intersect each surface in *CommonList* with each of the other surfaces in *CommonList*. The resulting edges are classified with respect to both the *Left* and *Right* sub-solids, and the portions that are "ON" both sub-solids are kept. The fourth stage is to classify all the edges on all the remaining surfaces on both sub-solids with respect to the other sub-solid and keep those that classify as "OUT." In the fifth stage, each surface remaining on the *Left* sub-solid is intersected with surface remaining on the *Right* sub-solid. The resulting edges are classified with respect to both sub-solids, and those that are "ON" both sub-solids are added to both surfaces. The sixth and final stage is to put all surfaces from the *Left* sub-solid, the *Right* sub-solid, and *CommonList* together in the union's *Surface* pointer. Duplicate surfaces and edges are removed, and invalid surfaces and edges are deleted. This completes the boundary evaluation for a union.
This process can best be illustrated by an example. Consider the two solids in figure 5.32(a), labeled "Left" and "Right". The first stage in evaluating the boundary of the union of these two solids is to find all of the surfaces common to both solids and put them on the CommonList. These surfaces, which are shown shaded in figure 5.32(b), are deleted from Left and Right in the second stage. In the third stage, each surface in CommonList in intersected with each other surface in CommonList, and the resulting edges are classified against each sub-solid and the "ON" portions are kept. The resulting edges are shown in figure 5.32(c). The fourth stage serves to identify the edges in each sub-solid that are "OUT" of the other sub solid. These edges are shown in figure 5.32(d). In the fifth stage, the surfaces remaining on each sub-solid are intersected with the surfaces remaining on the other sub-solid, and the resulting edges are classified against both sub-solids. The edges that are "ON" both sub-solids are kept. These edges are shown highlighted in figure 5.32(e). Finally, the CommonList, Left, and Right surfaces and edges are combined to make the boundary evaluation of the union, as shown in figure 5.32(f).

Figure 5.32
Differences evaluate their boundaries in five stages, similar to unions. The first stage is to take all of the surfaces from the union's Left and Right sub-solids and put them in CommonList. Each surface on CommonList includes all of the edges from both the Left and Right sub-solids which are associated with that surface. The surfaces on Commonlist are then deleted from the Left and Right sub-solids. The second stage is to classify all of the edges on all of the surfaces remaining on the Left sub-solid with respect to the Right sub-solid. The edges that classify as "OUT" are kept. Also in the second stage, all of the edges on all of the surfaces remaining on the Right sub-solid are classified with respect to the Left sub-solid, and those that classify as "IN" are kept. The third stage is to intersect each surface in CommonList with each surface remaining on the Right sub-solid. Resulting edges are classified with respect to both Left and Right sub-solids, and those that are "ON" both sub-solid are added to both surfaces. In stage four, each surface remaining on the Left sub-solid is intersected with each surface remaining on the Right sub-solid. Resulting edges are classified with respect to both sub-solids, and those that are "ON" both sub-solids are added to both surfaces. In the fifth and final stage, all surfaces from the Left sub-solid, the Right sub-solid, and CommonList together in the difference's Surface pointer. Duplicate surfaces and edges are removed, and invalid surfaces and edges are deleted. This completes the boundary evaluation for a difference.

Intersections evaluate their boundaries in seven stages. The first stage is the same as that for difference; the common surfaces are put on CommonList and removed from the sub-solids. In the second stage, all the edges on all the surfaces in CommonList are classified with respect to both Left and Right sub-solids. The edges that classify as "ON" both sub-solids are kept. The third stage is to classify all edges on all remaining surfaces on both sub-solids with respect to the other sub-solid. The edges that classify as "IN" are kept. The fourth stage is to intersect each surface in CommonList with each surface remaining on the Right sub-solid. Resulting edges are classified with respect to both sub-
solids, and those that classify as "ON" both sub-solids are kept. The fifth stage is to intersect each surface in CommonList with each surface remaining on the Left sub-solid. Resulting edges are classified with respect to both sub-solids, and those that classify as "ON" both sub-solids are kept. In the sixth stage, each of the surfaces remaining on the Left sub-solid are intersected with each of the surfaces remaining on the Right sub-solid, and resulting edges are classified with respect to both sub-solids. The edges that classify as "ON" both sub-solids are added to both surfaces. The seventh and final stage is the same as the final stages for unions and differences; to combine all the surfaces from the Left sub-solid, the Right sub-solid, and CommonList together in the intersection's Surface pointer, and to get rid of duplicate and invalid surfaces and edges. This completes the boundary evaluation for intersections.

It should be noted that boundary evaluation typically begins with the root solid of the CSG tree and continues down the CSG tree recursively. In other words, in order to evaluate the boundary of the root solid of the CSG tree, which for non-trivial cases will be an operation, it is first necessary to evaluate the boundaries of the root solid's Left and Right sub-solids, so that the surfaces and edges will be there to use. If the sub-solids are operations, their Left and Right sub-solids will need to evaluate their boundaries to provide surfaces and edges. These solids will in turn need to have their Left and Right sub-solids' boundaries evaluated. This continues down the CSG tree until the primitives are reached, which evaluate their own boundaries without difficulty. Boundary evaluation then takes place back up the CSG tree until the root solid is reached.

The methods for performing boundary evaluation were written from scratch after a careful analysis of the PADL-2 boundary evaluation techniques. The only portions closely based on PADL-2 are the Edge-Surface classification functions (CHCLLP.FLX, CHCLEP.FLX, CGCLCP.FLX, CHCLLC.FLX, CHCLEC.FLX, and CHCLCC.FLX),
and the Surface-Surface intersection functions (SSINPP.FLX, SSINPC.FLX, and SSINCC.FLX).

5.1.3. Segments

The whole point of this thesis project is to mesh structures using different types of elements where appropriate. This is accomplished by breaking the CSG tree into Segments. CSGMesh recognizes four types of segments: BeamSegments, which are used for 1D regions of the structure, PlateSegments, which are used for 2D regions of the structure, and BrickSegments, which are used for 3D regions of the structure. The fourth type, CylinderPlateSegment, is used for things like hollow cylinders which will be meshed with 2D elements. All segment types are descended from the ancestor tSegment, from whom they inherit a pointer to a solid called Solid, a pointer to a segment (used for storing segments in a linked list) called Next, an integer field called Number for storing a unique reference number, a real field called MeshDensity, which is used when creating a mesh, a pointer to a Mesh, called Mesh, which is where the nodes and elements for each segment are stored, a boolean field called Selected, which is used for several purposes, and a word field called DrawOptions, which determines whether the segment will be drawn as a solid, a representation, and with or without nodes and elements.

Brief descriptions of each segment type will be given here. For a complete description, see Appendix D.

5.1.3.1. BeamSegments

BeamSegments are used to represent long, narrow portions of the structure being modeled. The representation of a BeamSegment is a simple line, which allows it to be
meshed with one-dimensional elements. This line is stored as a LineEdge which is at the center of gravity of the solid making up the BeamSegment. The field holding this representation is called CGAxis, and is of type pLineEdge. The type tBeamSegment also has real fields for storing the cross-sectional area, the x- and y-axis moments of inertia, and the x-y product of inertia.

5.1.3.2. PlateSegments

The representation of a PlateSegment is a surface, stored as a PlaneSurf in the field Surface, which is of type pPlaneSurf. This surface is at the mid-plane of the segment solid, and has edges defined where the surface intersects the solid. The type tPlateSegment also has a real field for storing the thickness of the solid. Two-dimensional portions of the model can be represented with plate segments.

5.1.3.3. CylinderPlateSegments

A CylinderPlateSegment is a special type of PlateSegment, from which it descends. It does not have any additional fields. However, the Surface field of type tCylinderPlateSegment points to a CylinderSurf. Using this segment type, hollow cylinders can be meshed with two dimensional elements.

5.1.3.4. BrickSegments

The type tBrickSegment does not have any fields beyond those inherited from ancestor tSegment. The representation of a BrickSegment is the solid which makes up that segment -- there is no simplified representation like there is for the other segment types. Brick segments are used for three-dimensional portions of the model.
5.1.3.5. CSG Segmentation Logic

The breaking up of a CSG tree into segments can be illustrated by the example of a simple table, like the one shown in figure 5.33. This table may be represented by the CSG tree shown in figure 5.34.

![Figure 5.33](image)

**Figure 5.33**

![Figure 5.34](image)

**Figure 5.34**
In this case, the CSG tree would be broken into five segments: one for each leg and one for the top. Each segment can be thought of as a different portion of the structure. This allows different parts of the model to be treated differently. Thus, the four segments representing the legs would be meshed with 1D (beam) elements, and the segment representing the top could be meshed with 2D (plate) elements.

The example of the table is a simple one because each segment consists of a single primitive. However, CSGMesh has the ability to create segments out of much more complicated portions of a CSG tree. For example, if each leg were formed by the union of three blocks to represent an I-beam, and the top had holes in it, CSGMesh could still break up the CSG tree into five segments as before, but each segment would be more complex.

This is accomplished with recursive calls to a function called, appropriately enough, "Segment." This function takes as its only argument a pointer to a solid (pSolid), and returns a pointer to a segment (pSegment). (The actual function definition in Pascal is "function Segment(Solid: pSolid): pSegment;".) When the function Segment is called with an operation as its argument, it first attempts to create segments from the operation's Left and Right sub-solids. It then tries to combine these two resulting segments into one segment. Thus, like boundary evaluation, the breaking up of a CSG tree into segments is recursive. The general procedure for breaking a CSG tree up into segments, using the table in figures 5.33 and 5.34 as an example, is as follows:

First, the function Segment is called with the CSGTree pointer as the argument (Segment(CSGTree)). In this case, the CSGTree pointer would point to the union "Table" in figure 5.34. Since this is an operation, the function Segment would be called with the Left and Right sub-solids of this operation as its arguments. So, there would be calls to
Segment(Legs) and Segment(Top). Legs is again an operation, so there would be calls to Segment(Legs1&2) and Segment(Legs3&4). Each of these would in turn call Segment(Leg#1), Segment(Leg#2), Segment(Leg#3), and Segment(Leg#4). The result of Segment(Leg#1) would be a BeamSegment, because the primitive block "Leg#1" fits the requirements for a beam (i.e. is long and narrow). The calls to Segment(Leg#2), Segment(Leg#3), and Segment(Leg#4) would also return BeamSegments. With these calls to Segment completed, the calls to Segment(Legs1&2) and Segment(Legs3&4) can now be considered. These are both identical, so only the first will be described. The two segments created from the sub-solids of the operation "Legs1&2" are both beam segments (Leg#1 and Leg#2), so CSGMesh tries to combine them into one segment. These two beam segments are parallel, but they do not overlap (touch each other), so they cannot be combined -- they are each separate portions of the structure. Thus, each of these segments are added to the SegmentList (similar to SolidList -- a linked list used for storing all segments), and the call to Segment(Legs1&2) returns a pointer to a NothingSegment.

The call to Segment(Legs3&4) also returns a NothingSegment for the same reason.

Now, the calls to Segment(Legs) and Segment(Top) can be considered. The sub-solids of the union "Legs" both returned NothingSegments, so these segments are disposed of and Segment(Legs) returns a NothingSegment of its own. The call to Segment(Top) returns a PlateSegment, because the primitive block "Top" fits the requirements for a plate (i.e. is flat and thin). Now the call to Segment(Table) is considered. The Left sub-solid returned a NothingSegment, and the Right sub-solid returned a PlateSegment, so the NothingSegment is disposed of and the call to Segment(Table) returns the PlateSegment created at the call to Segment(Top). Since "Table" is the root solid of the CSG tree, the segment resulting from this call is added to the SegmentList, and the segmenting terminates. When all is said and done, the SegmentList contains five segments: a BeamSegment for each leg, and a PlateSegment for the top.
Each type of solid is treated differently when segments are being created.

5.1.3.5.1. Blocks

The creation of a segment from a block primitive is a very simple process. If one dimension is 10 times or more greater than both of the other two dimensions, the block will be used as a beam segment with the axis along the long dimension. If any dimension is 10 times smaller than the other two dimensions, then the block will be used as a plate segment with the surface normal in the direction of the short axis. If neither of these two conditions are true, the block will be used as a brick segment.

5.1.3.5.2. Cylinders

The creation of a segment from a cylinder is also a very simple process. If the height of the cylinder is 10 times of more greater than its radius, it will be used as a beam segment. If the height is less than or equal to a fifth of the radius, it will be used as a plate segment. Otherwise, it will be used as a brick segment.

5.1.3.5.3. Unions

The creation of a segment from a union operation is the most complicated of all of the solid types. First, segments are created from the union's Left and Right sub-solids. Then extensive testing is done on these two segments to determine whether or not they are combinable. If they are, they are combined into a single segment. The methods by which segments are combined are discussed in section 5.1.3.5.8. If the segments cannot be combined, they are each added to the SegmentList separately, and the union is made into a NothingSegment.
5.1.3.5.4. Differences

The creation of a segment from a difference operator is not as difficult as it would seem. Two methods of creating such segments are used: the creation of a segment from a special case, and the creation of a segment based on surfaces. In order to create a segment from a difference operation, the difference is checked to see if it fits one of the special cases, described in section 5.1.3.5.6. Some special cases that use differences include hollow cylinders and angle-iron type constructs. If no special cases apply, the surfaces of the boundary representation of the difference are examined to determine whether the solid can be represented by beams, plates, or bricks. The creation of a segment based on surfaces is discussed in section 5.1.3.5.7.

5.1.3.5.5. Intersections

The creation of a segment from an intersection operation is similar to that for a difference operation. First, a check is made for special cases. If none apply, the segment is created based on the surfaces of the intersection's boundary representation.

5.1.3.5.6. Special Cases

Special cases allow solid models constructed using differences and intersections to be represented in different ways. There are currently several special cases defined in CSGMesh. Some of the more important are HollowCylinder, HollowBlock, C_Channel, and AngleIron.
The use of special cases can be illustrated through the example of the

*HollowBlock*. The *HollowBlock* is shown graphically in figure 5.35(a). The CSG tree

used to define a *HollowBlock* is shown in figure 5.35(b).

![HollowBlock](a)

![CSG Representation](b)

**Figure 5.35**

Since CSGMesh can recognize this special case, it is able to handle it in a variety
of ways. If the user has specified that he wanted 2D elements for this portion of the
structure, CSGMesh would redefine this difference to make the HollowBlock a union of
blocks as shown in figure 5.36. CSGMesh has the ability to redefine the structure of the
other special cases as well. The use of special cases is completely transparent to the user.

![Collection of 4 blocks](a)

**Figure 5.36**
5.1.3.5.7. Using Surfaces to make Segments

When no special cases apply to a difference or an intersection, CSGMesh must examine the surfaces of the solid's boundary representation in order to create a segment. CSGMesh checks to see if there are two surfaces which are parallel, and to which all of the other surfaces are perpendicular. If this is the case, the solid can be used as a beam (with the two parallel surfaces as the ends), as a plate (with the two parallel surfaces as the top and bottom) or as a brick. If the surfaces do not meet this requirement, the solid must be used as a brick.

5.1.3.5.8. Combining Segments

CSGMesh has extensive procedures for combining segments into one segment. Before combining segments, comprehensive tests are performed to ensure that the segments indeed may be combined.

To combine beam segments, CSGMesh checks to see that they are parallel and overlap, among other tests. If two beam segments are combinable, a new solid is created that is the union of the two solids making up the two beam segments. A new beam segment is created from this solid and the old segments are disposed of. The cross sectional area and moments of inertia of this resulting solid are computed.

To combine plate segments, CSGMesh checks to make sure that the two surfaces lie in the same plane, that they have the same thickness, and that they overlap, among other tests. If the two plate segments are combinable, a new solid is created that is the union of the two solids making up the two plate segments. A new plate segment is created from this solid and the old segments are disposed of.
The combination of two brick segments is similar. If two brick segments overlap in some area, the solids making up those segments are unioned together, and the resulting solid is made a new brick segment.

When CSGMesh adds any segment to the SegmentList, it checks that segment against each segment already in the SegmentList. If it can be combined with any of the existing segments, that combination is made, and the old segments are disposed of. The resulting new segment is then added to the SegmentList, again checking to see if it can be combined with any existing segments.
5.1.4. Segment Meshing Techniques

Once the CSG tree has been broken into segments, CSGMesh has the ability to mesh those segments. Currently, CSGMesh contains four methods for generating meshes. It is sincerely hoped that in the future others will extend CSGMesh to include more techniques for automatic mesh generation.

5.1.4.1. Types of Meshes

In CSGMesh, a mesh is an object. CSGMesh has an object type called \textit{tMesh}, which is the common ancestor of all meshes. This type contains a field for a collection of nodes and a field for a collection of elements. (\textit{tCollection} is an object type defined in Turbo Vision\textsuperscript{TM}. For a description of this object type, see the Turbo Vision Guide\textsuperscript{[66].}) It also contains methods for drawing, writing itself to output files, etc. \textit{TMesh} has four descendants: \textit{tBeamMesh}, \textit{tAjayMesh}, \textit{tCylMesh}, and \textit{tBrickMesh}. Each of these types has its own methods for writing to an output file, describing itself, and more. A beam mesh is a mesh consisting of one dimensional elements. This is the mesh used for \textit{BeamSegments}. \textit{T AjayMesh} is based on the 1990 thesis work of Ajay Garg\textsuperscript{[21]}. This mesh consists of three-node triangle elements created by Delaunay Triangulation. \textit{T AjayMesh} is used for \textit{PlateSegments}. \textit{T CylMesh} is a descendant of \textit{T AjayMesh}. This type has been modified so that it can be used on \textit{CylinderPlateSegments}. Finally, \textit{T BrickMesh} consists of six-noded wedges created by extruding a 2D mesh created by Delaunay Triangulation through a 2½D solid. Each of these mesh types have a corresponding mesh generator object type, descended from the ancestor \textit{tMeshGenerator}. This object type has a method called \textit{CreateMesh}, which is what performs the actual mesh generation.
5.1.4.2. Meshing Beam Segments

The mesh generator used to create a BeamMesh is the type tBeamMeshGenerator. This object type meshes beam segments automatically by simple beam division. Evenly spaced nodes are created along the beam, the spacing based on the mesh density, and then the nodes are connected with elements.

5.1.4.3. Meshing Plate Segments

The mesh generator object type created for performing Delaunay Triangulation is tAjayMeshGenerator. The algorithms for Delaunay Triangulation are based on the work of Ajay Garg's 1990 Master's Thesis. However, substantial modifications have been made to fit within the CSGMesh solid modeling environment.

In the work of Garg, the structure to be meshed is represented by a simple polygon in two dimensions. In CSGMesh, the structure is represented by a combination of CSG and B-Rep in three dimensions. The existing procedures for performing set membership classification is CSGMesh were utilized for the creation of the set of nodes, as well as for determining whether triangles in the finished mesh were inside or outside the structure's domain. These modifications replaced rather dubious procedures based on the intersections of lines in two dimensions. Thus, the work of Garg was modified, extended, and improved in its inclusion in CSGMesh. Still, there is much room for improvement. Because the development of mesh generators was the last step taken in this thesis project, and because tAjayMeshGenerator was copied from pre-existing work, there are many opportunities to improve tAjayMeshGenerator, as well as all of the rest of the mesh generator object types.
The procedure for generating meshes in two-dimensional regions via the modified Ajay Garg extension of Watson's method of Delaunay Triangulation is as follows: First, nodes are generated on the boundary of the geometry. This is done by dividing the edges of the surface which represents the segment based on the mesh density parameter. Nodes are then generated in the interior of the geometry. This is done by finding the minimum and maximum y-coordinates of the boundary nodes and dividing this distance into steps based on the mesh density parameter, as shown in figure 5.37(c). LineEdges are then created which are on the surface representing the segment at these y-locations, as shown in figure 5.37(d). These LineEdges are then classified with respect to the segment's solid, and the portions that are "IN" are divided based on the mesh density parameter, and nodes are places at those locations, as shown in figures 5.37(e) and (f).

![Figure 5.37](image)

Once nodes have been generated, a mesh is created by Watson's method of Delaunay Triangulation. The final step is to determine which triangles are inside and which are outside the geometry. This is done by calculating the center of each triangle and classifying this point with respect to the segment's solid. If this point is "IN" the solid, then the triangle is inside the geometry, and if this point is "OUT" of the solid, then the triangle is outside of the geometry. Triangles which classify as "OUT" are discarded.
classification procedure is made more efficient by only classifying triangles in which all three vertices are boundary nodes. Triangles which contain at least one interior node are assumed to be inside the geometry, and are not classified. This drastically reduces the time required to check the mesh.

5.1.4.4. Meshing CylinderPlate Segments

The mesh generator developed to mesh hollow cylinders using two-dimensional elements, tCylMeshGenerator, is descended from tAjayMeshGenerator. The general procedure for creating a mesh is quite similar. First, boundary nodes are generated by dividing the edges of the surface representing the segment based on the mesh density, as shown in figure 5.38(c). Interior nodes are then generated by creating LineEdges around the circumference of the cylinder which are parallel to the axis of the cylinder and on the surface representing the cylinder, and dividing the portions of these edges which classify as "IN" the segment's solid, as shown in figures 5.38(d) - (f).

![Figure 5.38](image)
Once nodes are generated, a transformation is applied to all of the nodes to "unroll" the cylinder into a plane. This consists of redefining the nodes' coordinates from (r, θ, z) format to (x, y, z) format. With the nodes redefined in a plane, the mesh is created by Watson's method of Delaunay Triangulation. Once this is completed, the nodes are "rolled back up" into a cylinder by applying the inverse transformation; from (x, y, z) format back to (r, θ, z) format. The elements are then checked to determine which ones lie inside and which ones lie outside of the geometry. This check is identical to the one performed by tAjayMeshGenerator.

5.1.4.5. Meshing Brick Segments

Currently, only brick segments fitting certain requirements can be meshed by CSGMesh. In order to be meshed, brick segments must be what is commonly referred to as "2½D". Another way of saying this is that they may be defined by a sweep representation as defined in chapter 3. The test that CSGMesh performs to determine whether a brick segment can be meshed is to check the surfaces: if there are two surfaces which are parallel to each other, and all of the other surfaces in the boundary representation are perpendicular to those two surfaces, then the segment can be meshed. A solid which can be meshed by CSGMesh as a brick segment is shown in figure 5.39(a).

The meshing of brick segments which satisfy this criteria is very straightforward. It is accomplished by the mesh generator of type tBrickGenerator, which is a descendant of tMeshGenerator. First, the segment is treated like a PlateSegment, and meshed via Delaunay Triangulation, as shown in figure 5.39(b). This mesh is then extruded through the segment, which creates six-node wedges (pentahedrons) from the three-node triangles, as shown in figure 5.39(c).
5.1.4.6. Editing Meshes

Meshes of type \textit{tBeamMesh} and \textit{tAjayMesh} can be edited by adding nodes. This is a very effective method of locally modifying a mesh. \textit{AjayMeshes} also have the ability to be smoothed. In smoothing an \textit{AjayMesh}, each interior node is repositioned at the average of the positions of the nodes adjacent to it. Meshes of type \textit{tCylMesh} and \textit{tBrickMesh} cannot be modified.

5.1.4.7. Joining Meshes between Segments

Because each segment is meshed individually, it is necessary to join the meshes of individual segments together to get continuity across segment boundaries. CSGMesh can add nodes and reposition nodes that are on the interfaces between segments in order to accomplish this. Currently, CSGMesh can effectively join interfaces between beam segments and beam segments, between beam segments and plate segments, and between plate segments and plate segments. Because \textit{CylinderPlateSegment}'s and \textit{BrickSegment}'s meshes cannot be modified, meshes belonging to segments of these types cannot be joined to other meshes.
5.2. Future Extensions to "CSGMesh"

As large as CSGMesh has become over the past two years of work, there is still abundant room for improvement and extension. It is the hope of the author and of major advisor, Dr. Richard Budynas, that others will continue to develop CSGMesh in the future. It is envisioned that CSGMesh will grow into a powerful, robust tool for the automatic meshing of arbitrarily complex domains which will be considered for finite element analysis. CSGMesh is meant to be a program in which the user must simply specify how he wants the geometry meshed, and the software will perform the meshing to these specifications without difficulty.

Two major areas have been provided for in terms of extending CSGMesh to achieve this goal. The first is the inclusion of further primitive types, which will allow more complex domains to be considered. The second is the development of additional mesh generation techniques, which will allow for more efficient and accurate meshing of these domains. Each of these subjects will be dealt with in some detail to facilitate the further development of CSGMesh by others in the future.

In addition to these two major areas, there are a multitude of other improvements which could be made to CSGMesh. This list includes a wide range of things, including fixing minor bugs, adding certain non-essential niceties, re-writing portions of code, re-defining portions of program functionality, and even converting the entire source code from Pascal to C++. If all of these improvements had been included in this thesis work then CSGMesh would surely never have been completed. Although it leaves an uncomfortable feeling within the author to leave CSGMesh in an unfinished state, doing so is the only way that CSGMesh could have ever come to be.
5.2.1. Further Primitive Types

One way in which CSGMesh can be extended is by the inclusion of more primitive types in the solid modeling system. With this in mind, a framework has been provided for the development of the entire primitive set supported by PADL-2: block, cylinder, wedge, cone, sphere, and torus. Blocks and cylinders, of course, have already been developed. The development of the remaining four primitive types can be accomplished by filling out the "skeleton" units provided for each. This includes writing the following methods:

```
constructor Init;
function ReadValues(phrase: string): boolean; virtual;
procedure Copy(Solid: pSolid); virtual;
procedure Draw(Window: tRect); virtual;
function Inspect: word; virtual;
procedure ComputeBoundingBox; virtual;
procedure EvaluateBoundary; virtual;
procedure DefaultSegType; virtual;
```

Descriptions of these methods for blocks and cylinders can be found in the Object Reference (Appendix D). By using the CSGMesh source code as a reference, the development of these methods should be straightforward. Additional methods may be necessary for specific purposes for each primitive. Again, the existing source code and object reference will be helpful in this area.

When developing the sphere, cone, and torus, it will be necessary to develop new surface and edge types along with the primitive. This is because spheres, cones, and torii are made of surfaces other than plane and cylindrical surfaces. The intersections of these new surfaces result in all new edge types. The wedge primitive, however, is made up
solely of plane surfaces, so the development of this primitive will not include any surface or edge development.

The development of surfaces identical to that of solids. "Skeleton" units have been provided for all of the surface types supported in PADL-2, which can be filled out by the writing of the following methods:

```pascal
constructor Init;
procedure GetNormal(Point: t3DPoint; var Normal: tVector); virtual;
function ProfileEdges: pEdge; virtual;
function Intersection(Surface: pSurface): pEdge; virtual;
function ClassifyLineEdge(E: pLineEdge): pMCR; virtual;
function ClassifyEllipseEdge(E: pEllipseEdge): pMCR; virtual;
function ClassifyCCEdge(E: pCCEdge): pMCR; virtual;
{function Classify...Edge(E: p...Edge): pMCR; virtual;}
function Equals(Surface: pSurface): boolean; virtual;
function Parallel(Surface: pSurface): boolean; virtual;
function Perpendicular(Surface: pSurface): boolean; virtual;
procedure Copy(Surface: pSurface); virtual;
```

The coding of these methods should be straightforward using the object reference and the CSGMesh source code as guides. However, two methods deserve special consideration. The first is `function Intersection(Surface: pSurface): pEdge; virtual;`. In this method, the surface type must be able to compute the intersection edge with each of the surface types already developed. For example, if `tConeSurf` is the next surface type to be developed, the developer must insure that `tConeSurf.Intersection` can correctly compute the edge of intersection if the input parameter, `Surface`, is of type `tPlaneSurf` of type `tCylinderSurf`, because these two are already developed. To compute the edge of intersection with surface types that are not yet developed, the function should call the method of the
undeveloped surface type. Thus, in the preceding example, \texttt{tConeSurf.Intersection} would call \texttt{tSphereSurf.Intersection} if the input variable, \texttt{Surface}, were of type \texttt{tSphereSurf}, and would call \texttt{tTorusSurf.Intersection} if the input variable were of type \texttt{tTorusSurf}. This insures that each surface type will be able to interact with each other surface type as they are developed. The source code used to accomplish this for the example of type \texttt{tConeSurf} in Turbo Pascal would look something like this:

```pascal
function tConeSurf.Intersection(Surface: pSurface): pEdge; virtual;
  variable declarations;
begin
  if TypeOf(Surface) = TypeOf(tPlaneSurf) then
    begin
      Compute the intersection with the plane surface, because that type is already developed. The resulting edge could be of type tEllipseEdge, tParabolaEdge, tHyperbolaEdge, or tLineEdge.
    end
  else if TypeOf(Surface) = TypeOfftCylinderSurf) then
    begin
      Compute the intersection with the cylinder surface, because that type is already developed. The resulting edge would be a KKEdge.
    end
  else if TypeOf(Surface) = TypeOf(tConeSurf) then
    begin
      Compute the intersection with the cone surface, because that type is being developed. The resulting edge would be a KKEdge.
    end
  else if TypeOf(Surface) = TypeOf(tSphereSurf) then
    Intersection = Surface.Intersection(@Self)
  else if TypeOf(Surface) = TypeOf(tTorusSurf) then
    Intersection = Surface.Intersection(@Self)
  else Intersection = Surface.Intersection(@Self);
end;
```

By developing a surface in this way, it is not necessary to modify existing code so that pre-existing surface types can be intersected with surface types being developed. It is only
necessary to write new code, which is much easier than trying to modify someone else's code.

The second method that deserves special consideration is actually a collection of methods. The methods for classifying edges as "IN", "ON", or "OUT" must be written for each existing edge type. Thus, as further edge types are developed, it will be necessary to go back into the existing code for surface types and "teach" these surface types how the classify the new edge types.

The development of new edge types is similar to that for primitive solids and surfaces, type only difference being that in this case, existing code will have to be modified for the above mentioned reason. "Skeleton" units have been provided for each additional edge type, and filling them in requires the writing of the following methods:

constructor Init;
function tParameter(Point: t3DPoint): real; virtual;
procedure GetPoint(t: real; var Point: t3DPoint); virtual;
function Equal1(E: pEdge): boolean; virtual;
function Equal2(E: pEdge): boolean; virtual;
function EqualPoints(x,y: real): boolean; virtual;
procedure GetTangent(Point: t3DPoint; var Tangent: tVector); virtual;
function BoundingBox: pBoundingBox; virtual;
procedure Draw(Window: tRect); virtual;
procedure Copy(Surface: pSurface); virtual;

The writing of these methods should be easy using the object reference and CSGMesh source code as guides. As was mentioned, however, something further is required. To develop a new edge type, the code for all existing surface types must be extended. Each
existing surface type must have a method developed to classify the new edge type. These methods have been provided in the ancestor type, $tSurface$, as abstract methods, and their names clearly describe their purpose (i.e. function $ClassifyLineEdge$ for classifying an edge of type $tLineEdge$). If this step is omitted, when CSGMesh tries to classify a new edge type against an existing surface type, the function call will go to the abstract inherited method ($tSurface.Classify...Edge$), which will cause a run-time error (#208).

It is also possible to develop new primitives types beyond those supported by PADL-2. In order to do this, all of the above steps must be taken. Furthermore, modifications will have to be made to the unit containing the ancestor solid type, $tSolid$. This unit is called, appropriately enough, unit $Solids$. In this unit there are various global procedures and functions which distinguish between types of solids. Since the development of the solids defined in PADL-2 has been allowed for, these procedures know all about these solid types. In order to define a new solid type not foreseen, the following changes must be made to unit $Solids$: The uses portion of the implementation section must include the unit containing the new solid type. In effect, this tells unit $Solids$ where to look for the description of the new solid type. Also, global function $SolidOfType$ must include the new solid type. The reason for doing this and the way in which to do it will become apparent simply by looking at this function in the source code.

Finally, defining new primitive types will involve developing methods for creating segments from these primitives and solids made up of these primitives, and eventually meshing these segments. This is left up to the future developer.
5.2.2. Implementing Mesh Generators

Perhaps the greatest potential for extension of CSGMesh lies in the area of mesh generation. Chapter 1 of this work illustrated the many types and uses of finite elements available to an analyst. The second chapter dealt with the subject of automatic mesh generation and the many methods for performing it. It is the goal of this work to combine these two ideas; to create a platform for the automatic creation of finite element meshes consisting of multiple element types where appropriate. The ability to generate different meshes using different techniques is of primary importance to this goal.

Adding new mesh generation techniques to CSGMesh is not difficult. In CSGMesh, the algorithms for generating meshes are contained within objects. The object type \textit{tMeshGenerator} has been created as an ancestor for all mesh generator object types. This object type has two fields: \textit{Segment}, which is a pointer to the segment that the mesh generator is charged with meshing, and \textit{Mesh}, which is a pointer to the mesh it creates. Type \textit{tMeshGenerator} also has three methods, \textit{constructor Init}, for initialization, \textit{procedure CreateMesh}, for performing the actual meshing, and \textit{destructor Done}, for disposing of itself when through. All object types have constructors called "Init" and destructors called "Done", and there is nothing remarkable about these methods in type \textit{tMeshGenerator}. The procedure "CreateMesh", on the other hand, requires explanation.

Procedure \textit{CreateMesh} is meant to be overridden in each descendant of \textit{tMeshGenerator}. It is this method that contains the entire mesh generation algorithm. Thus, vast and complex mesh generation algorithms can be utilized by CSGMesh through this one simple object method call. This modularity of design makes the use of mesh generator objects simple and straightforward. Basically, the action of procedure \textit{CreateMesh} is as follows: The mesh generator object instantiates a new mesh object of
the appropriate type (generally, each mesh generator object type has a corresponding mesh object type). The mesh object has pointers to collections of nodes and elements as fields, which are empty at initialization. The mesh generator object then "creates" the mesh by defining nodes and elements and putting them into the mesh object's collections. When done, the mesh generator object sets the segment's mesh pointer to the mesh it created. The mesh generator object can then be disposed of, leaving the segment with a mesh.

In order to add a new mesh generation technique, it is necessary to create a new mesh generator object type descended from tMeshGenerator. This object type may have as many fields and methods as necessary to fully and efficiently encode the mesh generation algorithm, as long as the procedure CreateMesh is the "main" procedure. In effect, procedure CreateMesh can be thought of as a main program which may call other procedures and functions. For instance, the procedure tAjayMeshGenerator.CreateMesh calls other methods to perform certain tasks, tAjayMeshGenerator.CreateBoundaryNodes and tAjayMeshGenerator.CreateInteriorNodes being prime examples. The program unit defining mesh generators, unit MeshGen, is not used by any units except unit ThesisApp, which means that this unit can use all of the other program units. Essentially, the whole solid modeling system, including edges, surfaces, solids, and segments (all object types, in fact) are at the disposal of mesh generator objects. Thus, mesh generators can use high-level procedures in order to simplify the meshing process, as was done by using set membership classification to generate interior nodes in plate segments (see section 5.1.4.3.)

Mesh generators are called by procedure MeshSegments in unit ThesisApp. This procedure instantiates a new mesh generation object based on the segment type and the meshing technique chosen, and then calls its CreateMesh method. Once a new mesh generator has been developed, it can be used by including it in this procedure. New
meshing techniques must also be "registered" in unit *MeshTechniques*. This unit is meant to provide a method for the user to choose meshing techniques. With only one meshing technique currently available for each segment type, this unit is more of an example of what is possible that anything useful. However, with the development of additional mesh generation algorithms in the future, this unit will certainly evolve.

When developing a new mesh generator object, a new type of mesh will usually also be developed, descended from *tMesh*. In some cases, it will also be necessary to develop a new element type. These are descended from *tElement*. The development of these object types will become self-explanatory by examining the CSGMesh source code for the existing mesh-related object types.

5.2.3. Other possible additions to "CSGMesh"

As with any large computer program, (especially one written by a mechanical engineer and not a computer programmer) there are bugs to be fixed and other possibilities for improvement in CSGMesh. What follows is a list of the improvements the author would like to have made to CSGMesh if time had permitted. The list is organized roughly in order of scale: first come bugs which are relatively minor and don't require much in the way of re-writing. Some of these bugs, however, are quite elusive and would require a good deal of investigation. Next come niceties, or little extras that would make CSGMesh more pleasant to use, but are not essential to the operation of the program. Finally, some major changes are suggested which would greatly improve the operation of the program. These would require moderate to vast amounts of re-writing.

There are probably many bugs in CSGMesh which have not been discovered. The ones that are known about have been fixed wherever possible. Still, a few bugs remain.
One of the most elusive has to do with deleting segments. Occasionally, after all segments are deleted, the program will lock up while trying to create new segments. This is probably a result of incomplete disposal of the previous segments, or inadequate handling of pointers. The source of this bug has been hunted down numerous times through the entire source code, but has yet to be found. The results of this bug are quite minor. All that is required is to restart the program. This bug can be entirely avoided by simply exiting and re-entering CSGMesh whenever a new model is to be used, rather than deleting the old model and loading in the old one.

Another very elusive bug appears while drawing segments. Occasionally, the program will crash with run-time error 004: "Duplicate Identifier". The Turbo Pascal programmer's guide describes this error as occurring when an identifier has already been used within the current block. This bug has popped up very recently, even after most of this document has been written, while attempting to get print-outs for Chapter 6. So far, no reason for this bug has been discovered.

The next bug is somewhat more than a simple bug. The procedures to solve polynomials don't work. The solving of polynomials comes into play when classifying EllipseEdges and CCEdges against cylinder surfaces. These procedures -- found in unit PolyZero -- were taken from PADL-2 file PZERO.FOR. The main procedure for solving polynomials is as follows:

\[
\textbf{procedure } \text{FindZeros}(\text{NA: integer; } A: \text{RealArray; var } \text{NZ: integer; var } Z: \text{RealArray});
\]

where \( \text{NA} \) is the number of terms in the polynomial (polynomial order plus one), \( A \) is an array of real numbers representing the coefficients of the polynomial, \( \text{NZ} \) is the number of distinct, real roots found, and \( Z \) is a real array containing those roots. The fact that this
procedure does not give good results is not a major concern, because of the modularity of the program allows the simple replacement of *procedure FindZeros*. The correction of this problem will allow arbitrarily oriented cylinders to be combined. Currently, cylinders can be combined only if they are co-axial.

Although CSGMesh can evaluate the boundary of the majority of CSG representations, sometimes it comes across a solid for which it cannot correctly evaluate the boundary. This is not a programming bug, but a logic bug. The boundary evaluation routines in CSGMesh were developed from scratch, only loosely following the PADL-2 technique. Unlike the PADL-2 boundary evaluator, CSGMesh does not use the concept of neighborhoods. By revising the boundary evaluation routines to include neighborhoods, it is possible to make boundary evaluation much more robust.

There are many features which would make CSGMesh a more professional program. The development of a lot of these things was anticipated, but since they would be the "icing on the cake", they were not implemented in preference to the "cake". The list of such features includes:

A material library -- it would be nice to have a list of materials to choose from, such as steel, aluminum, etc. Once a material is picked, CSGMesh could look up the values for that material's properties. This would save the user the hassle of looking up and entering material properties.

Context-sensitive help -- CSGMesh has grown into such a large program that it is difficult to tell what everything in the pull-down menus is for. Turbo Pascal and Turbo Vision provide a system for context-sensitive help, so that the program can explain each of it's features.
More Special Cases -- Developing Special Cases is the only real way to allow for different meshes in solids made using difference and intersection operations. Special cases such as half-round, quarter-round, and C-Channel have been anticipated. The program unit containing special cases contains some instructions for adding more cases.

Way of applying boundary conditions -- Currently, there is no way to specify boundary conditions or loads on the model. It is a simple task to do these things in to commercial finite element software (ANSYS, NASTRAN, or Supersap) which reads the input file. However, it would be nice to make CSGMesh as complete as possible, including some provisions for boundary conditions and loading.

Improved method of input -- With this one, we're beginning to get into the changes that will require extensive re-writing. It would certainly be nice for CSGMesh to have an interactive method of defining geometries -- perhaps a graphics-based system where the user can pick primitives and combine them as if they were toy blocks.

Better graphics interface -- The graphics interface used now is somewhat sloppy, and the transition between text and graphics modes is akward. A system in which the different graphics windows, including the drawing window, the segment icon window, the message line window at the bottom of the screen, and the change-view windows were objects themselves, similar to the Turbo Vision text window objects, would be ideal. Conceivable, the entire user interface could be accomplished in graphics mode.

Output for other analysis types -- Currently, output files can only be written for static analysis. By extending the methods of nodes, elements, meshes, and segments, it would be possible to have output files written for other analysis types as well.
At this point, we're getting into some pretty serious overhauling of CSGMesh. The following items are more than just nice additions to the program. They are fundamental changes to the way CSGMesh operates, and would serve to make CSGMesh a much more flexible, robust system. These changes are certainly beyond the scope of the current work, but eventually they may become possible.

Rather than mesh each segment individually and then try to joing the individual meshes together to achieve continuity between segment interfaces, it would be more elegant to discover these interfaces prior to meshing, and insure that the meshes generated would be continuous across those segment boundaries. This idea was seriously investigated, and code was written to attempt it. This attempt included dividing segments into multiple segments, and functions which tested whether edges and surfaces were shared. Remnants of this attempt still remain in the CSGMesh source code. In the interest of ever finishing this thesis, it was decided that post-meshing segment interfacing would suffice.

Originally, it was hoped that CSGMesh would involve more in the way of expert systems and artificial intelligence. However, it was determined that the project was already large enough without that requirement. Still, hopes remain that much of the functionality of CSGMesh could be used within an expert system shell. This, of course, would be more than an extension of CSGMesh, it would essentially be a new beginning, using bits and pieces of CSGMesh where appropriate. It is also hoped that CSGMesh will be converted from Pascal to C++. Perhaps these two goals could be combined, and an entirely new program could be developed using CSGMesh as the prototype. The possibilities are limitless.
6. Results: Examples of Geometries Meshed using "CSGMesh"

CSGMesh has been used to generate numerous meshes. In this chapter, a sample of these various meshes will be shown to illustrate the flexibility of CSGMesh. In each case, the input file will be given, along with figures of the resulting solid, the segment representations, and finally the mesh.

6.1. Plate with Holes

The first example demonstrates the ability of CSGMesh to generate meshes in arbitrary non-convex planar domains, with or without holes. This small area of automatic mesh generation has been heavily investigated throughout the 1980's. In fact, this is probably the most thoroughly researched area of automatic mesh generation of all. Therefore, it was deemed necessary for CSGMesh to have the capability of creating valid meshes in such domains.

The geometry of the planar domain for this example was defined via the following input file:

```
floor = blo(x=1.414213562, y=1.414213562, z=0.1);
roof  = blo(z=0.1) movedby degz=-45, movy=1.414213562;
window = blo(x=0.2, y=0.15, z=0.1) movedby movx=0.65, movy=0.8;
door  = blo(x=0.25, y=0.5, z=0.1) movedby movx=0.25;
attic = cyl(r=0.1, h=0.1) movedby movx=0.7071, movy=1.5;
chimney = door movedby movx=0.8, movy=1.5;
house = floor un roof dif window dif door dif attic un chimney;
```

The primitives which this file defines are shown in figure 6.1, and the resulting solid after boundary evaluation is shown in figure 6.2.
Figure 6.1

Figure 6.2
6.1.1. Plate using Plate Segment (2D Mesh)

By specifying the use of a PlateSegment for this solid, CSGMesh creates the PlateSegment shown in figure 6.3. This representation is a planar surface, located at the middle of the solid's thickness.

Figure 6.3

The mesh of this domain is shown in figure 6.4.
6.1.2. Plate using Brick Segment (3D Mesh)

This same solid model can be used in more than one way. Figure 6.5 shows the solid represented as a brick segment. Note that this is identical to figure 6.2, because the representation of a brick segment is the actual solid making up that segment -- there is no simplified representation. Figure 6.6 shows the three-dimensional mesh generated on this brick segment.
6.2. I-Beam

CSGMesh has the ability to mesh the same model in a variety of ways. This allows
the user to use the same solid model for many different analyses, rather than defining a
new model of a different dimensionality for each analysis. The I-beam is a good example
of this. The input file defining the I-beam is as follows:

\[
\begin{align*}
top &= \text{blo}(x=8, y=1, z=50) \text{ movedby } \text{movx}=-4, \text{movy}=3.99; \\
middle &= \text{blo}(x=1, y=8, z=50) \text{ movedby } \text{movx}=-0.5, \text{movy}=-4; \\
bottom &= \text{top movedby } \text{movy}=-8.98; \\
i-beam &= \text{top un middle un bottom};
\end{align*}
\]

The primitives defined by this file are shown in figure 6.7.

![Figure 6.7](image-url)
6.2.1. I-Beam using Beam Segment

Here, the I-beam is represented as a beam. CSGMesh has computed the x and y moments of inertia, as well as the xy product of inertia and cross sectional area of the beam. Thus, the resulting elements can be written to an output file with these parameters.

Figure 6.8

Figure 6.9
6.2.2. I-Beam using Plate Segment

The use of a single plate segment across the cross section of the I-beam would be useful for a plane strain or plane stress analysis. The I-beam represented in this way is shown in figure 6.10, and the mesh is shown in figure 6.11.

Figure 6.10

Figure 6.11
6.2.3. I-Beam using Multiple Plate Segments

A more accurate analysis could be performed by representing the I-beam as a collection of plates. This method of representation and the resulting mesh are shown in figures 6.12 and 6.13.
6.2.3. I-Beam using Brick Segments

Finally, the option of representing the I-beam as a single solid can be considered. This representation and mesh are shown in figures 6.14 and 6.15.

Figure 6.14

Figure 6.15
6.3. Pipe

The example of a hollow pipe illustrates many things, including the 
HollowCylinder special case and the use of CylinderPlate segments. Like the I-beam, the 
hollow pipe can be represented in many ways. The input file used to define the pipe is as 
follows:

\[
\begin{align*}
\text{outside} &= \text{cyl}(d=0.5, h=0.625); \\
\text{inside} &= \text{cyl}(d=0.25, h=0.625); \\
\text{pipe} &= \text{outside dif inside};
\end{align*}
\]

The primitives defined by this file are shown in figure 6.16.

Figure 6.16
6.3.1. Pipe using Beam Segment

Here, the pipe is represented as a beam. Just as with the I-beam, CSGMesh has computed the x and y moments of inertia, as well as the xy product of inertia and cross sectional area of the pipe. Thus, the resulting elements can be written to an output file with these parameters.

Figure 6.17

Figure 6.18
6.3.2. Pipe using Plate Segment

The use of a single plate segment across the cross section of the pipe would be useful for a plane strain or plane stress analysis. The pipe represented in this way is shown in figure 6.19, and the mesh is shown in figure 6.20.
6.3.3. Pipe using CylinderPlate Segment

The pipe gives a good example of CylinderPlate segments. If the user selects a plate segment for the pipe, CSGMesh first discovers that the solid is a hollow cylinder (this is done by the use of the HollowCylinder special case), and asks the user whether he wants a plate segment or a CylinderPlate segment. The CylinderPlate representation and mesh are shown in figures 6.21 and 6.22.

**Figure 6.21**

**Figure 6.22**
6.3.4. Pipe using Brick Segment

Finally, the option of representing the pipe as a single solid can be considered.

This representation and mesh are shown in figures 6.23 and 6.24.

Figure 6.23

Figure 6.24
6.4. Pipe with Holes

The use of CylinderPlate segments can be extended to include hollow cylinders with holes cut in them, and with other modifications. This allows a great deal of flexibility in modeling and meshing curved surfaces. The input file for such a solid is as follows:

```plaintext
outside = cyl(r=1.25, h=5);
inside = cyl(r=1, h=5);
pipe = outside dif inside;
hole = blo(x=1, y=5, z=1) moved by movz=2, movy=-2, movx=-0.3;
thing = pipe dif hole;
```

The primitives defined by this file are shown in figure 6.25. Figure 6.26 shows this solid after boundary evaluation.

![Diagram of a pipe with holes](image-url)
The representation of this solid as a plate segment (CylinderPlate, in fact, although this is transparent to the user) is shown if figure 6.27.
Figures 6.28 and 6.29 show two views of the mesh of the pipe with holes.
6.5. Bracket

The next example shows a structure made up of multiple segments, in this case three plate segments. This example will show how individual meshes are joined together.

The input file describing the structure is as follows:

```
plate1 = blo(x=5, y=5, z=0.25);
plate2 = blo(x=5, y=0.25, z=5);
plates = plate1 un plate2;
rib1 = blo(x=0.25, y=4.75, z=4.75) movedby movx=2.325, movy=0.25, movz=0.25;
rib2 = blo(x=0.25, y=9, z=9) movedby movx=2.325, degx=-45, movz=5, movy=0.25
rib = rib1 dif rib2;
bracket = plates un rib;
```

The primitives defined by this file are shown in figure 6.30. The resulting solid after boundary evaluation is shown in figure 6.31.

![Figure 6.30](image-url)
Figure 6.31
The representation of three plates is shown in figure 6.32. An end view is shown in figure 6.33 to show how the plates do not exactly line up at the edges, because of the thickness of the solid.

**Figure 6.32**

**Figure 6.33**
The meshes of these plates are shown in figure 6.34. Notice in figure 6.35 that the meshes do not align at the edges.

Figure 6.34

Figure 6.35
After merging and connecting the meshes, the meshes do align at the edges, as shown in figures 6.36 and 6.37.

Figure 6.36

Figure 6.37
7. Discussion/Conclusion

As stated in the first chapter, the objective of this thesis was to develop a computer program capable of two things: first, performing automatic meshing of structures using different element types where appropriate, and second, meshing the same structure in several different ways at the user's discretion.

Despite the fact that CSGMesh is rough around the edges, it does meet these two goals. CSGMesh does indeed show that it is possible to automatically create meshes containing multiple element types. The meshes created with CSGMesh shown in the previous chapter were generated quickly and easily. The approach of breaking up a CSG-tree into segments prior to meshing appears to be a good method of meshing complex structures using simple element types where appropriate, as an experienced analyst would do.

Also, each of the models in the preceding chapter was meshed in a variety of ways. With CSGMesh, there is no need to redefine the model in order to create a mesh consisting of a different type of element, or even elements of a different dimension. A single CSG model can be represented and meshed as one-dimensional, two-dimension, three-dimensional, or a combination thereof, at the user's discretion.

With the addition of further primitive types to the CSG representation, and further mesh generation algorithms utilizing different element types in different domains, CSGMesh has the potential to grow into a very powerful, robust, intelligent system for the automatic generation of finite element meshes using multiple element types.
References


[27] Yerri, M. A. and M. S. Shephard: "A modified quadtree approach to finite
1, pp. 39-46, Jan/Feb 1983.

"Quadtree/Octree meshing with adaptive analysis," Numerical Grid Generation in
Computational Fluid Mechanics'88, Miami, FL, USA, 1988

[29] Shephard, M. S., F. Guerinone, J. E. Flaherty, R. A. Ludwig, P. L. Baehmann: 
"Adaptive solutions of the Euler equations using finite quadtree and octree grids," 


[31] Cavendish, J. C., D. A. Field, and W. H. Frey: "An approach to automatic three-

generation using Delaunay Tessellation," IEEE Trans. Magnetics, vol 21, pp. 2535-
2538, 1985.

Research Publication GMR-4967, General Motors Research Labs, Warren, MI, 
1985.

[34] Baker, T. J.: "Three dimensional mesh generation by triangulation of arbitrary

using a constrained Delaunay Triangulation," Eng. Comp., vol 5, pp. 161-175, 
1989.

domains of any shape," Impact of Computing in Science and Engineering, vol 2, 

automatic finite element mesh generation," Int. J. for Num. Meth. in Eng., vol 32, 

[38] Wördenweber, B.: "Automatic mesh generation of 2- and 3-dimensional curvilinear
manifolds," Ph. D. dissertation (Technical Report No. 18), Cambridge University,
UK, 1981.


Appendix A
PADL-2 Source Files

Since CSGMesh's solid modeling system was modeled after PADL-2, a listing of the PADL-2 source code files is included here so that any person wishing to modify or extend CSGMesh can look to the PADL-2 code on which it is based. Of primary interest and importance are the files in the CGPAK (Computational Geometry Package) directory, found on page 1 and 2 of this appendix.
Total of 22 files.

Directory [PADL2.SOURCE.BEVAL]

BEINIT.FLX;1 BEWIRE.FLX;1 BFDISP.FLX;1 BFRGET.FLX;1
BFRKIL.FLX;1 BFRMOV.FLX;1 BLDISP.FLX;1 BOXHSP.FLX;1
BVALMI.FLX;1 PRFMAK.FLX;1 VISCOM.CMN;1 VISLOC.FLX;1
VISPLX.FLX;1

Total of 13 files.

Directory [PADL2.SOURCE.BFILE]

B2INIT.FLX;1 B2NUMA.FLX;1 B2NUMS.FLX;1 BADEL.FLX;1
BAGATR.FLX;1 BAGBOX.FLX;1 BAGFLG.FLX;1 BAGINT.FLX;1
BAGMOV.FLX;1 BAGNAM.FLX;1 BALIST.FLX;1 BALIVE.FLX;1
BANEW.FLX;1 BANOS.FLX;1 BANUMS.FLX;1 BAPATR.FLX;1
BAPBOX.FLX;1 BAPFLG.FLX;1 BAPINT.FLX;1 BAPMOV.FLX;1
BAPNAM.FLX;1 BAPUTS.FLX;1 BAPINT.FLX;1 BEFIND.FLX;1
BEGALL.FLX;1 BEGATR.FLX;1 BEGBND.FLX;1 BEGBOX.FLX;1
BEGFLG.FLX;1 BEGIN.FLX;1 BEGIP.FLX;1 BEGNAM.FLX;1
BEGNHB.FLX;1 BEGPF.C.FLX;1 BEGPLN.FLX;1 BEGREP.FLX;1
BEGSD.FLX;1 BELINK.FLX;1 BELIST.FLX;1 BELIVE.FLX;1
BEPASS.FLX;1 BEP ALL.FLX;1 BEPATR.FLX;1 BEPBND.FLX;1
BEPBOX.FLX;1 BEPEDG.FLX;1 BEPFLG.FLX;1 BEPINT.FLX;1
BEPF.FLX;1 BEPNA.M.FLX;1 BEPINT.FLX;1 BEPINT.FLX;1
BEPPLN.FLX;1 BEPNAM.FLX;1 BEPINT.FLX;1 BESPINT.FLX;1
BEPREP.FLX;1 BEPNAM.FLX;1 BESPINT.FLX;1 BESHR.FLX;1
BFDEL.FLX;1 BFGALL.FLX;1 BFGATR.FLX;1 BFGBOX.FLX;1
BFGLN.FLX;1 BFGLN.FLX;1 BFGATR.FLX;1 BFG BOX.FLX;1
BFGLN.FLX;1 BFGLN.FLX;1 BFGINT.FLX;1 BFGNAM.FLX;1
BFGLN.FLX;1 BFGLN.FLX;1 BFGINT.FLX;1 BFGNAM.FLX;1
BFGLF.C.FLX;1 BFGREP.FLX;1 BFGTYP.FLX;1 BFLIST.FLX;1
BFLIVE.FLX;1 BFNEW.FLX;1 BFNOE.FLX;1 BFNUME.FLX;1
BFPALL.FLX;1 BFPATR.FLX;1 BFPBOX.FLX;1 BFPFLG.FLX;1
BFPINT.FLX;1 BFPNAM.FLX;1 BFPNOR.FLX;1 BFPFC.FLX;1
BFPREP.FLX;1 BFPINT.FLX;1 BSFAR.FLX;1 BNXEOA.FLX;1
BNXEOF.FLX;1 BNXEOS.FLX;1 BNXFOA.FLX;1 BNXFO.FLX;1
BNXSQ2.FLX;1 BSCOPY.FLX;1 BSDELF.LX;1 BSGATR.FLX;1
BSGFL.FLX;1 BSGFLG.FLX;1 BSGINT.FLX;1 BSGNUM.FLX;1
BSGTY.P.FLX;1 BSLIST.FLX;1 BSLIVE.FLX;1 BSMOVE.FLX;1
BSNEW.FLX;1 BSNOF.FLX;1 BSNUME.FLX;1 BSNUMF.FLX;1
BSPATR.FLX;1 BSBBOX.FLX;1 BSSFLG.FLX;1 BSPINT.FLX;1
BSPLN.FLX;1 BSPTYP.FLX;1 BSREAL.FLX;1 BXCURN.FLX;1

Total of 112 files.

Directory [PADL2.SOURCE.CGPAK]

ALGEBRA.FLX;1 APPLY.FLX;1 BSORT.FLX;1 CGCDAT.CMN;1
CGCLAS.CMN;1 CGCMNV.CMN;1 CGCON.CMN;1 CGERR.FLX;1
CGGE.FLX;1 CGGEGE.FLX;1 CGGGSE.FLX;1 CGINTL.FLX;1
CGK.FLX;1 CGMKON.FLX;1 CGMSGE.FLX;1 CGPAK.FLX;1
al of 140 files.

Directory [PADL2.SOURCE.COMPAK]

COMPAK.INC;1 EQDIRA.FLX;1 EQMAGA.FLX;1 EQMAGR.FLX;1
REQA.FLX;1 REQR.FLX;1 RZEROA.FLX;1 RZEROR.FLX;1
VANG.FLX;1 VEOA.FLX;1 VEQR.FLX;1 VERRPT.FLX;1
VNORM.FLX;1 VZEROA.FLX;1

Total of 14 files.

Directory [PADL2.SOURCE.GRAPAK]

CONGET.FLX;1 CONREP.FLX;1 GRABKD.FLX;1 GRACL.R.FLX;1
GRACOM.CMN;1 GRACRE.FLX;1 GRAEVA.FLX;1 GRAGAR.FLX;1
GRAGET.FLX;1 GRAINT.FLX;1 GRAIXA.FLX;1 GRAIXL.FLX;1
GRAIPI.FLX;1 GRAMIF.FLX;1 GRANO.FLX;1 GRAPFI.FLX;1
GRAPOP.FLX;1 GRAPOP.FLX;1 GRAPOS.FLX;1 GRAPUS.FLX;1
GRARAT.FLX;1 GRASTT.FLX;1 GRAREP.FLX;1 GRARES.FLX;1
GRASTG.CMN;1 GRASTT.FLX;1 GRATPL.CMN;1 GRIARC.FLX;1
GRIBOX.FLX;1 GRICMT.FLX;1 GRICLY.FLX;1 GRIDN.FLX;1
GRIEVA.FLX;1 GRIFND.FLX;1 GRIMET.FLX;1 GRIMOT.FLX;1
GRINDO.FLX;1 GRIPAR.FLX;1 GRIRAT.FLX;1 GRIRNA.FLX;1
GRISNA.FLX;1 GRIVPD.FLX;1 LPRINT.FLX;1

Total of 43 files.

ectory [PADL2.SOURCE.PACPAK]
Total of 41 files.

Directory [PADL2.SOURCE.PARSER]
FNOPEN.FLX;1 INTPT2.FLX;1 INTPT3.FLX;1 LRDATA.CMN;1 LWRITE.FLX;1
LRDIM.CMN;1 LRTERM.CMN;1 P2.BNF;1 P2.GMR;1
P2.LIS;1 P2INIT.FLX;1 PARSER.FOR;1 PPUT.FLX;1
PRSCOM.CMN;1 PRSINI.FLX;1 PRSSTG.CMN;1 READMED.LIS;1
SCAN.FLX;1 SYNTH.FLX;1 SYNTH2.FLX;1 TEXTIO.CMN;1

Total of 20 files.

Directory [PADL2.SOURCE.PGPAK]
DISPFS.FLX;1 PGBEGN.FLX;1 PGCLT.CMN;1 PGDCS.FLX;1
PGDEDG.FLX;1 PGDLST.FLX;1 PGDPRM.FLX;1

Total of 7 files.

Directory [PADL2.SOURCE.PICPAK]
AINI.FLX;1 DTCLAS.FLX;1 DTCLCC.FLX;1 DTCLLK.FLX;1
DTC0MP.FLX;1 DTCLLS.FLX;1 DTCLLT.FLX;1 DTCOMB.FLX;1
DTC0MP.FLX;1 DTDPFL.FLX;1 DTEHCL.FLX;1 DTINFL.FLX;1
DTLCY.FLX;1 DTLKIL.FLX;1 DTMCRK.FLX;1 DTNTL.FLX;1
DTRSR.FLX;1 DTUNL.FLX;1 INSPFC.FLX;1 MRGFLS.FLX;1
PICCOM.CMN;1 PICK.FLX;1 PICKFC.FLX;1

Total of 23 files.

Directory [PADL2.SOURCE.QUALIF]
IGES.CMN;1 OCTQUA.CMN;1 PROQUA.CMN;1 QUAGET.FLX;1
QUASV.FLX;1 QUASV.FLX;1 QUAS2.CMN;1 QUASEAT.FLX;1
QUASV.FLX;1 TXTURE.CMN;1

Total of 10 files.

Directory [PADL2.SOURCE.SDIR3D]
LST001.FOR;1 LSTAD01.FOR;1 LSTADE.FOR;1 LSTADF.FOR;1
LST01.FOR;1 LSTC0M.FLX;1 LSTCOP.FOR;1 LSTDLE.FOR;1
LSTDUP.FOR;1 LSTINT.LF;1 LSTKIL.FOR;1 LSTLK2.FOR;1
LSTLST2.FOR;1 LSTNXT.FLX;1 LSTPAK.FOR;1 LSTREP.FOR;1
LSTUNI.FOR;1 SD001.FOR;1 SDADD.FOR;1 SDB00L.FOR;1
"DP.COMN;1 SDB0G.FOR;1 SDDEL.FOR;1 SDETE.FOR;1
"ETI.FOR;1 SDGETF.FOR;1 SGTE1.FOR;1 SDGTI.FOR;1

A-3
Total of 30 files.

Directory [PADL2.SOURCE.SGPAK]

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IT.FLX;1</td>
<td>BXUNBX.FLX;1</td>
</tr>
<tr>
<td>S Pane.EGN.FLX;1</td>
<td>SGBKD.CMN;1</td>
</tr>
<tr>
<td>SGCOM.CMN;1</td>
<td>SGDAT.FLX;1</td>
</tr>
<tr>
<td>SGERR.FLX;1</td>
<td>SGERRS.CMN;1</td>
</tr>
<tr>
<td>SGM4MU.FLX;1</td>
<td>SGMOVE.FLX;1</td>
</tr>
<tr>
<td>SGSADV.CMN;1</td>
<td>SGSAVE.FLX;1</td>
</tr>
<tr>
<td>SGTST.FLX;1</td>
<td></td>
</tr>
</tbody>
</table>

Total of 25 files.

Directory [PADL2.SOURCE.SHADER]

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BXUNBX.FLX;1</td>
<td>SDRBOX.CMN;1</td>
</tr>
<tr>
<td>SDRDAT.FLX;1</td>
<td>SDRDOT.FLX;1</td>
</tr>
<tr>
<td>SDRGSC.FLX;1</td>
<td>SDRIN1.FLX;1</td>
</tr>
<tr>
<td>SDRPPS.FLX;1</td>
<td>SDRRPC.FLX;1</td>
</tr>
<tr>
<td>SDRTRA.CMN;1</td>
<td>SHADE.FLX;1</td>
</tr>
</tbody>
</table>

Total of 18 files.

Directory [PADL2.SOURCE.SMPAK]

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISSTAT.FLX;1</td>
<td>ISSTAT.FLX;1</td>
</tr>
<tr>
<td>SMAOUT.FLX;1</td>
<td>SMBLOC.FLX;1</td>
</tr>
<tr>
<td>SMGET.FLX;1</td>
<td>SMCMOV.FLX;1</td>
</tr>
<tr>
<td>SMDUMP.FLX;1</td>
<td>SMFTCH.FLX;1</td>
</tr>
<tr>
<td>SMMSTO.FLX;1</td>
<td>SMRMOV.FLX;1</td>
</tr>
<tr>
<td>SMSTAT.FLX;1</td>
<td>SMSTOW.FLX;1</td>
</tr>
</tbody>
</table>

Total of 30 files.

Directory [PADL2.SOURCE.SPAPAK]

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDIOSP.FLX;1</td>
<td>EDOUSP.FLX;1</td>
</tr>
<tr>
<td>SPASTT.FLX;1</td>
<td>SPAUSE.CMN;1</td>
</tr>
<tr>
<td>SPINBX.FLX;1</td>
<td>SPINSF.FLX;1</td>
</tr>
<tr>
<td>SPMOVE.FLX;1</td>
<td>SPPRIM.FLX;1</td>
</tr>
</tbody>
</table>

Total of 15 files.

Directory [PADL2.SOURCE.STGPAKT]

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALFTST.FLX;1</td>
<td>ALPHEQ.FLX;1</td>
</tr>
<tr>
<td>CONCAT.FLX;1</td>
<td>COPY.FLX;1</td>
</tr>
<tr>
<td>FSTCHR.FLX;1</td>
<td>IASCII.FLX;1</td>
</tr>
<tr>
<td>INTOST.FLX;1</td>
<td>LENGTH.FLX;1</td>
</tr>
<tr>
<td>READ.FLX;1</td>
<td>RETOST.FLX;1</td>
</tr>
<tr>
<td>STGPAK.FLX;1</td>
<td>STGPRO.FLX;1</td>
</tr>
<tr>
<td>STGTST.FLX;1</td>
<td>STGUPTF.FLX;1</td>
</tr>
<tr>
<td>STTST.FLX;1</td>
<td>SUBSTR.FLX;1</td>
</tr>
<tr>
<td>TRIMZ2.FLX;1</td>
<td>WRITE.FLX;1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A'-4</td>
<td></td>
</tr>
</tbody>
</table>

A-4
Total of 3 files.

Directory [PADL2.SOURCE.STGPAKV]

<table>
<thead>
<tr>
<th>ALFTST.FLX;1</th>
<th>ALPEQ.FLX;1</th>
<th>ALPHG1.FLX;1</th>
<th>ALPHLT.FLX;1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCAT.FLX;1</td>
<td>COPY.FLX;1</td>
<td>DUPSTG.FLX;1</td>
<td>EREAD.FLX;1</td>
</tr>
<tr>
<td>SHR.FLX;1</td>
<td>IASCI1.FLX;1</td>
<td>INDEXSTG.FLX;1</td>
<td>INDEXXX.FLX;1</td>
</tr>
<tr>
<td>INT.FLX;1</td>
<td>INTOST.FLX;1</td>
<td>LENGTH.FLX;1</td>
<td>LSTCHR.FLX;1</td>
</tr>
<tr>
<td>PRINT.FLX;1</td>
<td>READ.FLX;1</td>
<td>RETOST.FLX;1</td>
<td>RLSE.FLX;1</td>
</tr>
<tr>
<td>STGC1.SCMN;1</td>
<td>STGINDEX.FLX;1</td>
<td>STGPAK.FLX;1</td>
<td>STGPRO.FLX;1</td>
</tr>
<tr>
<td>STGSE.FLX;1</td>
<td>STGSTT.FLX;1</td>
<td>STGSTT.FLX;1</td>
<td>STGUPR.FLX;1</td>
</tr>
<tr>
<td>STREAD.FLX;1</td>
<td>STTOAR.FLX;1</td>
<td>STTORE.FLX;1</td>
<td>STTOST.FLX;1</td>
</tr>
<tr>
<td>SUBSTR.FLX;1</td>
<td>TEST.FLX;1</td>
<td>TRIM2.FLX;1</td>
<td>TRIMLF1.FLX;1</td>
</tr>
<tr>
<td>TRIMR.FLX;1</td>
<td>TRIMZ2.FLX;1</td>
<td>TRIMZL.FLX;1</td>
<td>TRIMZ2R.FLX;1</td>
</tr>
<tr>
<td>WRITE.FLX;1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total of 41 files.

Directory [PADL2.SOURCE.SUPPORT]

<table>
<thead>
<tr>
<th>PLYCON.SCMN;1</th>
<th>PLYPAK.FLX;1</th>
<th>PZERO.FOR;1</th>
<th>RMPAK.FLX;1</th>
</tr>
</thead>
<tbody>
<tr>
<td>STKPAK.FLX;1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total of 5 files.

Directory [PADL2.SOURCE.SYSTEM]

<table>
<thead>
<tr>
<th>CENTER.FLX;1</th>
<th>CONST.SCMN;1</th>
<th>CPUS1C.FLX;1</th>
<th>HELP.FOR;1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKERR.FLX;1</td>
<td>OCTOBJ.SCMN;1</td>
<td>P2D1T.FLX;1</td>
<td>P2OPS.SCMN;1</td>
</tr>
<tr>
<td>PADL2.FLX;1</td>
<td>PD1ERR.FLX;1</td>
<td>PFLFORM.SCMN;1</td>
<td>PFSOUT.FLX;1</td>
</tr>
<tr>
<td>PFL1ST.SCMN;1</td>
<td>PFL1ST.FLX;1</td>
<td>PFLOPN.FLX;1</td>
<td></td>
</tr>
<tr>
<td>PFSRED.FLX;1</td>
<td>STUBS.FLX;1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total of 18 files.

Directory [PADL2.SOURCE.VGPAK]

<table>
<thead>
<tr>
<th>FRDS1D.DIR;1</th>
<th>PSF.DIR;1</th>
<th>QDV.DIR;1</th>
<th>REG1S.DIR;1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST1.DIR;1</td>
<td>T1K.DIR;1</td>
<td>VGBEGN.FLX;1</td>
<td>VGCLOS.FLX;1</td>
</tr>
<tr>
<td>VGCOLR.FLX;1</td>
<td>VGCOM.SCMN;1</td>
<td>VGCRSF.FLX;1</td>
<td>VGERAS.FLX;1</td>
</tr>
<tr>
<td>VG1RAW.FLX;1</td>
<td>VGER.FLX;1</td>
<td>VGMOVF1.FLX;1</td>
<td>VGETCT.FLX;1</td>
</tr>
<tr>
<td>VG1L1N.FLX;1</td>
<td>VG1LINE.FLX;1</td>
<td>VGMOVF2.FLX;1</td>
<td>VGPORF.FLX;1</td>
</tr>
<tr>
<td>VG1PTCT.FLX;1</td>
<td>VG1PTRN.FLX;1</td>
<td>VGR1SZ.FLX;1</td>
<td>VGSCOL.FLX;1</td>
</tr>
<tr>
<td>VG1SDV.FLX;1</td>
<td>VG1SMP.FLX;1</td>
<td>VGS1TY.FLX;1</td>
<td>VGSTAT.FLX;1</td>
</tr>
<tr>
<td>VGUSER.SCMN;1</td>
<td>VSWIND.DIR;1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total of 30 files.

Directory [PADL2.SOURCE.VGPAK.FRDS1D]

<table>
<thead>
<tr>
<th>CLSFRD.FLX;1</th>
<th>ERAFRD.FLX;1</th>
<th>PTFNFRD.FLX;1</th>
<th>S1DVFRD.FLX;1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total of 4 files.

Directory [PADL2.SOURCE.VGPAK.PS1F]

<table>
<thead>
<tr>
<th>CLS1PSF.FLX;1</th>
<th>ERAPS1F.FLX;1</th>
<th>LINPS1F.FLX;1</th>
<th>PTFNPS1F.FLX;1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCLPSF.FLX;1</td>
<td>SDVPSF.FLX;1</td>
<td>SSTPS1F.FLX;1</td>
<td></td>
</tr>
</tbody>
</table>

Total of 7 files.
Directory [PADL2.SOURCE.VGPAK.QDV]
QDVDVR.FOR;1  QPLAYER.FOR;1

Total of 2 files.

Directory [PADL2.SOURCE.VGPAK.REGIS]
CLSRGS.FLX;1  CSRSTI.FLX;1  ERARGS.FLX;1  LINRGS.FLX;1
MOVRGS.FLX;1  PTNRGSTI.FLX;1  SDVRGS.FLX;1

Total of 7 files.

Directory [PADL2.SOURCE.VGPAK.STI]
CLRSTI.FLX;1  CLSSTI.FLX;1  CSRSTI.FLX;1  DRWSTI.FLX;1
ERASTI.FLX;1  LINSTI.FLX;1  MOVSTI.FLX;1  PNSTSTI.FLX;1
RSZSTI.FLX;1  SCLSTI.FLX;1  SDSTSTI.FLX;1  SSTSTI.FLX;1
STSTUBS.FLX;1

Total of 13 files.

Directory [PADL2.SOURCE.VGPAK.TEK]
CSRTEK.FLX;1  DRWTEK.FLX;1  ERATEK.FLX;1  LINTEK.FLX;1
MOVTEK.FLX;1  PRTTEK.FLX;1  PTNTEK.FLX;1  SAVTEK.FLX;1
SCLTEK.FLX;1  SDVTEK.FLX;1  TEKDRW.FLX;1

Total of 11 files.

Directory [PADL2.SOURCE.VGPAK.VSWIND]
CSRVSW.FOR;1  CSRVSFW.FOR;1  CSRVSW.FOR;1  ERAVSFW.FOR;1
CLSVSW.FOR;1  PTNVSFW.FOR;1  RSVVSFW.FOR;1  SAVSRN.FLX;1
SAVSVSW.FOR;1  SCLVSFW.FOR;1  SDVVSFW.FOR;1  SSVSTVSFW.FOR;1
VSCLOS.FOR;1  VSSIZE.FOR;1  VSWIND.CMN;1  VSSTUBS.FOR;1

Total of 16 files.

Directory [PADL2.SOURCE.VGPAKVS]
AAAREAD.ME;1  SAVERS.FLX;1  TEKDRW.FLX;1  VGBEGN.FLX;1
VGCOLS.FLX;1  VGCOLR.FLX;1  VGCOM.CMN;1  VGCRSR.FLX;1
VGDAIT.FLX;1  VGDECMN;1  VGDRAW.FLX;1  VGEND.FLX;1
VGERAS.FLX;1  VGECMNTV;1  VGLIM.FLX;1  VGLINE.FLX;1
VGMOVE.FLX;1  VGPORTE.FLX;1  VGPRTRN.FLX;1  VGPSTCV.FLX;1
VGRSIZ.FLX;1  VGSLOAD.FLX;1  VGDECMN;1  VGSSTY.FLX;1
VGSTAT.FLX;1  VGUSER.CMN;1  VSCLOS.FLX;1  VSSIZE.FLX;1

Total of 28 files.

Grand total of 29 directories, 779 files.
Appendix B
PADL-2 Point Sets: Primitives, Halfspaces, and Edges

This appendix contains mathematical descriptions of the PADL-2 geometric entities. It was these geometric entities (primitives, halfspaces, and edges) that the CSGMesh primitives, surfaces, and edges were based upon. The descriptions in this appendix were taken from "Representations in the PADL-2.0/N processor: Low level geometric entities," [Reference 65], and are pretty much self-explanatory. For further information, see reference 65.
**BLOCK**

TYPE: "BLOCK"

CONFIGURATION PARAMETERS: \( X_0, Y_0, Z_0 \)

VALIDITY CONSTRAINTS: \( X_0, Y_0, Z_0 > 0 \)

POINT SET: \( \{(x,y,z) : 0 \leq x \leq X_0, \text{ and} \)
\( 0 \leq y \leq Y_0, \text{ and} \)
\( 0 \leq z \leq Z_0 \} \).

---

**WEDGE**

TYPE: "WEDGE"

CONFIGURATION PARAMETERS: \( X_0, Y_0, Z_0 \)

VALIDITY CONSTRAINTS: \( X_0, Y_0, Z_0 > 0 \)

POINT SET: \( \{(x,y,z) : 0 \leq x \leq X_0, \text{ and} \)
\( 0 \leq y \leq Y_0, \text{ and} \)
\( 0 \leq z \leq Z_0, \text{ and} \)
\( y \cdot Z_0 + z \cdot Y_0 \leq Z \cdot Y_0 \} \).

---

**CYLINDER**

TYPE: "CYLND"

CONFIGURATION PARAMETERS: \( R, H \)

VALIDITY CONSTRAINTS: \( R, H > 0 \)

POINT SET: \( \{(x,y,z) : x^2 + y^2 \leq R^2, \text{ and} \)
\( 0 \leq z \leq H \} \).

---

**SPHERE**

TYPE: "SPHERE"

CONFIGURATION PARAMETERS: \( R \)

VALIDITY CONSTRAINTS: \( R > 0 \)

POINT SET: \( \{(x,y,z) : x^2 + y^2 + z^2 \leq R^2 \} \).
CONE

TYPE: "CONE"
CONFIGURATION PARAMETERS: R, H
VALIDITY CONSTRAINTS: R, H > 0

POINT SET: \{(x, y, z) : x^2 + y^2 <= (z * \tan(R/H))^2, \text{ and } 0 <= z <= H \}.

TORUS

TYPE: "TORUS"
CONFIGURATION PARAMETERS: Rmin, Rmaj
VALIDITY CONSTRAINTS: Rmin > 0, Rmaj > 0, Rmaj > Rmin

POINT SET:
\{(x, y, z) : (x^2 + y^2 + z^2 - Rmaj^2 - Rmin^2)^2 - 4 * Rmaj^2 (Rmin^2 + z^2) \}.
PLANAR HALFSPACE

TYPE: "PSURF"  \(^1\)
CONFIGURATION PARAMETERS: none
VALIDITY CONSTRAINTS: none

POINT SET: \( \{(x,y,z) : z \leq 0\} \).  \(^2\)
CYLINDRICAL HALFSPACE

TYPE: "CSURF"
CONFIGURATION PARAMETERS: R
VALIDITY CONSTRAINTS: R > 0

POINT SET: \{ (x, y, z) : x^2 + y^2 < R^2 \}.
SPHERICAL HALFSPACE

TYPE: "SSURF"
CONFIGURATION PARAMETERS: \( R \)
VALIDITY CONSTRAINTS: \( R > 0 \)

POINT SET: \( \{ (x, y, z) : x^2 + y^2 + z^2 \leq R^2 \} \).
CONICAL HALPSPACE

"KSURF"

CONFIGURATION PARAMETERS: \( \alpha = \tan \left( \frac{\theta}{2} \right) \)

VOLIDITY CONSTRAINTS: \( 0 < \theta < \pi \)

POINT SET: \((x, y, z): x^2 + y^2 \leq (\alpha z)^2\)
TOROIDAL HALFSpace

TYPE: "TSURF"
CONFIGURATION PARAMETERS: Rmin, Rmaj
VALIDITY CONSTRAINTS: Rmin > 0, Rmaj > 0, Rmaj > Rmin

POINT SET:
\[
\{(x, y, z) : (x^2 + y^2 + z^2 - Rmaj^2 - Rmin^2)^2 \leq 4 \cdot Rmaj^2 (Rmin^2 + z^2) \}. \\
\]

or, in some places

\[
\{(x, y, z) : (x^2 + y^2 + z^2 + Rmaj^2 - Rmin^2)^2 \leq 4 \cdot Rmaj^2 (x^2 + y^2) \}. \\
\]
LINE SEGMENT

TYPE: "LINE"
CONFIGURATION PARAMETERS: to, tl
VALIDITY CONSTRAINTS: \(-\infty \leq t_0 < t_1 \leq \infty\)

POINT SET: 
\[
\{(x,y,z) : x = 0, \text{ and } y = 0\} = \\
\{(x,y,z) : x = 0, y = 0, z = t, t_0 \leq t \leq t_1\}\}
\]
ELLIPSE SEGMENT

TYPE: "ELLIPS"
CONFIGURATION PARAMETERS: Ru, Rv, t0, t1
VALIDITY CONSTRAINTS:

1) Ru, Rv > 0.
2) -π ≤ t0 ≤ t1 ≤ π.

POINT SET: \[
\{ (x,y,z) : \frac{x^2}{Ru^2} + \frac{y^2}{Rv^2} + \frac{z^2}{2} = 0 \} = \\
\{ (x,y,z) : x = Ru \cos(t), \quad y = Rv \sin(t), \quad z = 0, \quad t0 \leq t \leq t1 \}
\]
TYPE: "PARAB"
CONFIGURATION PARAMETERS: 4p, t0, tl
VALIDITY CONSTRAINTS:

1) 4p > 0.
2) -∞ <= t0 <= tl <= ∞.

POINT SET: \{ (x,y,z) : \text{x}^2 = 4p \cdot y, \text{and} \ z = 0 \} =
\{ (x,y,z) : x = t, y = t^2/4p, z = 0, t0 <= t <= tl \}.
HYPERBOLIC SEGMENT

TYPE: "HYPERB"
CONFIGURATION PARAMETERS: b/2, a/2, t0, t1
VALIDITY CONSTRAINTS:

1) a/2, b/2 > 0.
2) t0 ≤ t ≤ t1 ≤ ∞

POINT SET: \{ (x, y, z) : \frac{y^2}{b^2} - \frac{x^2}{a^2} = 1,
and z = 0,
where x > 0 and y > 0 \} =

\{ (x, y, z) : x = \frac{a}{2}(t-1/t),
y = \frac{b}{2}(t+1/t),
z = 0,
t0 ≤ t ≤ t1 \}. 

CYLINDER/CYLINDER EDGE SEGMENT

TYPE: "CCEDGE"
CONFIGURATION PARAMETERS: R1, R2, h, a, b, sign1, sign2, t0, t1

VALIDITY CONSTRAINTS:

1) R1, R2 > 0.
2) h >= 0.
3) a**2 + b**2 = 1, a non-zero.
4) sign1, sign2 = +1 or -1.
5) h - R2 < R1.
6) if (h+R2)/R1 > 1, tmin = 0
   otherwise tmin = arccos((h+R2)/R1): <4>
   if (h-R2)/R1 < -1, tmax = π
   otherwise tmax = arccos((h-R2)/R1): <4>
   tmin <= t0 < t1 <= tmax.

POINT SET: \{ (x,y,z) : x**2 + y**2 = R1**2, and 
          (x-h)**2 + (az-by)**2 = R2**2, and 
          sign2*y >= 0, and 
          sign1*(z-b*y/a) >= 0 \} = 

\{ (x,y,z) : x = R1*cos(t), 
      y = sign2*R1*sin(t), 
      z = b*sign2*R1*sin(t)/a + 
        sign1*sqrt(rad(t)), 
      t0 <= t <= t1, where 
      rad(t) = (R2**2 - (R1*cos(t) - h)**2)/a**2 \}.

(See Remark Below)

-------

<4> Arccos is taken in principal value, i.e. between 0 and π
CCEDGE Local Coordinate System and Configuration Parameters
SPHERE/CYLINDER EDGE SEGMENT

TYPE: "SCEDGE"

CONFIGURATION PARAMETERS: Ro, Rs, h, sign, t0, tl

VALIDITY CONSTRAINTS:

1) Ro, Rs > 0.
2) h > 0.
3) N < D, where N = h^2 + Ro^2 - Rs^2, and
   \[ D = 2Ro^2h. \]
4) sign = 1 or -1.
5) if N/D < -1, then \( t_{\text{min}} = -\pi \), \( t_{\text{max}} = \pi \).
   if N/D >= -1, then \( t_{\text{max}} = \arccos(N/D) \), \( t_{\text{min}} = -t_{\text{max}} \).

POINT SET:

1. \( (x,y,z) : x^2 + y^2 = Ro^2 \), and
2. \( (x-h)^2 + y^2 + z^2 = Rs^2 \), and
3. \( \text{sign}z \geq 0 \}

\( \{ (x,y,z) : x = Ro\cos(t), y = Ro\sin(t), z = \text{sign}\sqrt{\text{rad}(t)}, t0 = t = tl, \text{ where} \)
\( \text{rad}(t) = -N + D\cos(t) \} \).

(See Remark Below)
SCEDGE Local Coordinate System and Configuration Parameters
CONE/CONE EDGE SEGMENT

TYPE: "KKEDGE"

CONFIGURATION PARAMETERS: a, b, c, d, e, f, g, sign, t0, t1

VALIDITY CONTRAINTS

1) 0 ≤ t ≤ π / 2
2) -π ≤ t0 < t1 ≤ π

POINT SET: { (x,y,z) : x = ±S(t)*cos(t),
               y = ±S(t)*sin(t),
               z = S(t),
               t0 ≤ t ≤ t1, where
               S(t) = -B(t)/A(t) + sign*sqrt( B(t)^2 - A(t)*C)/A(t)
               A(t) = a + 2*b*c(t) + c*(sin(t))^2
               B(t) = d + e*o(t) + f*sin(t)
               C = g } <5>

(See Remark Below)

--------

<5> The CONE/SPHERE edge and the CONE/CYLINDER edge also have this parametric form. The implicit form of the point sets for these edges have not been given.
KKEDGE Local Coordinate System and Configuration Parameters
TORUS PROFILE EDGE SEGMENT

TYPE: "TPROF"
CONFIGURATION PARAMETERS: a, b, c, d, e, Rmaj, Rmin, sign, t0, t1

VALIDITY CONTRAINTS

1) Rmin ≤ Rmaj
2) -π ≤ t0 < t < t1 ≤ π

POINT SET: \{ (x, y, z) : x = (Rmaj + Rmin*F(t))*cos(t), y = (Rmaj + Rmin*F(t))*sin(t), z = Rmin*G(t), t0 ≤ t ≤ t1, \\
where \[ F(t) = \frac{p^*2 - q^*2}{p^*2 + q^*2}, \quad G(t) = \frac{2pq}{p^*2 + q^*2} \]
and \[ p(t) = -B + \text{sign} \ast \sqrt{B^*2 + A^*2 - C^*2}, \quad q(t) = C - A \]
and \[ A(t) = a + b*cos(t) + c*sin(t), \quad B = d, \quad C = e \}

TPROF Local Coordinate System
Appendix C
CSGMesh Object Hierarchy

Objects marked with a single asterisk (*) are Turbo Vision™ objects. Their descriptions, uses, fields, and methods may be found in the Turbo Vision Guide. They will not be included in this work.

Objects marked with a double asterisk (**) are not currently supported in CSGMesh. These are the "skeleton" objects. Frameworks, or "skeletons", have been provided for these object types so that they may be developed with ease in the future.
Appendix D
CSGMesh Object Reference

This appendix contains an alphabetical listing of all the CSGMesh object types, with explanations of their general purposes and usage, their fields, and their methods. To find information on a specific object, keep in mind that many of the properties of the objects in the hierarchy are inherited from ancestor objects. Rather than duplicate all that information endlessly, this appendix only documents fields and methods that are new or changed for a particular object.
t3DPoint:

type t3DPoint = object(tObject)

It is used to represent a point in three dimensional space. It is used for a wide variety of purposes, including drawing and computational geometry. t3DPoint is defined in unit Points.

Fields:

x: real; The x location of the point.
y: real; The y location of the point.
z: real; The z location of the point.

Methods:

constructor Init;
    Calls tObject.Init and sets the x, y, and z fields to zero.

procedure Assign(a,b,c: real);
    Sets the fields x, y, and z equal to the values of a, b, and c, respectively.

function Equals(p: t3DPoint): boolean;
    True if Self is within Epsilon of the point "p", otherwise False.

procedure Move(direction: t3DPoint; magnitude: real);
    Moves the point magnitude units along the vector specified by direction.

procedure Apply(Matrix: tMatrix);
    Multiplies the point vector (x,y,z) by the matrix specified by Matrix.

procedure ApplyInverse(Matrix: tMatrix);
    Multiplies the point vector (x,y,z) by the inverse of the matrix specified by Matrix.

procedure ConvertToScreen(Window: tRect);
    Converts a point in three dimensional space (x,y,z) to a point on the computer screen (i,j,0) by applying the graphics transformation matrix Trans (defined in unit Graphics), among other mathematical operations.

procedure ConvertToWorld(Window: tRect);
    Converts a point on the computer screen, given by (i,j,0), to the corresponding point in three dimensional space (x,y,z) by applying the inverse of the graphics transformation matrix Trans, among other mathematical operations.

procedure Draw(Window: tRect);
    Draws the point on the computer screen as a small circle. Must be in graphics mode.

end; {t3DPoint}
tAjayMesh:

tAjayMesh = object(tMesh)

tAjayMesh is based on the work that Ajay Garg did for his Master's Thesis in 1990, thus the name. This type was copied pretty much word for word from Ajay's code, with some alterations and extensions to make it fit within the CSGMesh solid modeling environment. Because of this, it is not as elegantly programmed as it could or should be, is not as efficient, and is not as well debugged.

A great deal of the functionality for creating a mesh by Delaunay Triangulation is contained within this object type. Also, a lot of the functionality is contained within the associated element type, tTriangle3. Ideally, this functionality and some fields and methods should be given to the associated mesh generator, tAjayMeshGenerator and taken away from type tAjayMesh and type tTriangle3.

tAjayMesh is defined in unit AjayMesh.

Fields:
MaxCoords: XYZExtPoint;  The maximum coordinates (in Local two-dimensional space) of the set of points to be meshed.
MinCoords: XYZExtPoint;  The minimum coordinates (in Local two-dimensional space) of the set of points to be meshed.
BoundingTriangle: pTriangle3;  The first triangle in the mesh into which all nodes are inserted.
StartNode1: pNode;  The first of the three nodes of the BoundingTriangle.
StartNode2: pNode;  The second of the three nodes of the BoundingTriangle.
StartNode3: pNode;  The third of the three nodes of the BoundingTriangle.
BoundaryNodes: pCollection;  The nodes on the boundary of the structure being meshed.
InteriorNodes: pCollection;  The nodes in the interior of the structure being meshed.
TriangleList: pCollection;  A temporary list of Triangles to discard.
TempList: pCollection;  Another temporary list of Triangles to discard.
NewNode: pNode;  The Node currently being added to the mesh.

Methods:
constructor Init;
   Calls tMesh.Init, then initializes new collections for the fields BoundaryNodes and InteriorNodes, TempList, and TriangleList.

procedure CreateBoundingTriangle;
   Creates the first triangle of the mesh, which contains all of the nodes to be inserted into the mesh. The vertices of this triangle are StartNode1, StartNode2, and StartNode3, which are created in this procedure, and the resulting triangle is stored in field BoundingTriangle.

procedure SetMaxMinCoords;
   Finds the minimum and maximum coordinates of the set of nodes, and sets the fields MinCoords and MaxCoords equal to these coordinates.
procedure AddNode(Node: pNode);
    Adds the node Node to the mesh by Watson's method of Delaunay Triangulation.

function AddNodeAt(Pos: XYZExtPoint): pNode; virtual;
    Creates a new node at the position specified by Pos, and adds it to the mesh by Watson's method of Delaunay Triangulation.

procedure DeleteNode(Node: pNode); virtual;
    Currently does nothing, and is not used. This is one of the possible extensions to CSGMesh -- the interactive remeshing of geometries by adding, moving and deleting nodes.

procedure ConvertTriangleList;
    This procedure takes the list of triangles which have been identified as having the new Node contained within their circum-circles (these triangles are stored in field TriangleList by procedure DiskContainsNode). From this list of triangles, the bounding polygon is generated. New triangles are added to the mesh based on the nodes of the bounding polygon by calling procedure MakeNewTriangles, and the old triangles are deleted.

procedure CreateMesh; virtual;
    Adds each node to the mesh, one at a time.

procedure DiskContainsNode(theTriangle: pTriangle3);
    Tests each triangle in the mesh against the node contained in field NewNode to see if the triangle contains the node within its circum-circle. If it does, the triangle is deleted from the mesh (field Elements) and is added to the temporary list (field TriangleList).

procedure MakeNewTriangles(aTriangle: pTriangle3);
    Creates new triangles based on the bounding polygon of the triangles containing the new node within their circum-circles. Called from within procedure ConvertTriangleList.

function ScrutinizeTriangle(theTriangle: pTriangle3): boolean;
    Checks a triangle to determine whether it should be kept or thrown away, based on whether it contains a node which is outside of the MinCoords and MaxCoords. This method serves no purpose other than to get rid of triangles which contain one of the three StartNodes as a vertex, thus resulting in the mesh of the convex hull of the set of points.

procedure Adjacent2aNode(centerNode: pNode;
    var aTriList, NodeList: pCollection);
    Finds all triangles which contain the node centerNode as a vertex, and puts these triangles in the collection aTriList. All of the vertices of these triangles are then placed (once each) in the collection NodeList. This is how bounding triangles are determined.
tAjayMesh

procedure ShiftNode(theNode: pNode);
   Moves the node specified by theNode to the average position of the adjacent nodes, as found
   by procedure Adjacent2aNode.

function TypeString: string; virtual;
   Returns the string "Delaunay Triangulation"

function Smooth: boolean; virtual;
   Performs procedure ShiftNode on each of the InteriorNodes.

procedure AjayDraw(Window: tRect; Motion: tMatrix;
   DrawBoundNodes, DrawInterNodes, DrawElements: Boolean);
   Draws the mesh in the portion of the screen specified by Window, first applying the matrix
   specified by Motion to each of the nodes (this allows nodes in Local coordinate system to be
   drawn in Global coordinates). Draws BoundaryNodes, InteriorNodes, and Elements based on the
   values of the boolean input parameters DrawBoundNodes, DrawInterNodes, and DrawElements.

procedure Debug(Window: tRect; Motion: tMatrix); virtual;
   Draws the entire mesh in grey, then draws each element one at a time in red, pausing between
   each for a key-press, so that the user can see each element in order to determine if there was an
   error in meshing.

procedure Write; virtual;
   Writes the mesh to an output file.

destructor Done; virtual;
   Disposes of all fields defined by type tAjayMesh, then calls destructor tMesh.Done to dispose
   of all inherited fields.

end; {tAjayMesh}

tAjayMeshGenerator:

type tAjayMeshGenerator = object(tMeshGenerator)

   tAjayMeshGenerator is the mesh generator object type developed to generate meshes by
   Delaunay Triangulation. There was no corresponding object type is the work of Ajay Garg, on
   which the technique of Delaunay Triangulation was based. Thus, much of the functionality that
   should belong to this type actually belongs to the corresponding mesh and element object types,
tAjayMeshGenerator

tAjayMesh, and tTriangle3, as it did in the work of Garg. Ideally, if future revisions to CSGMesh, this functionality will be transferred from these types to tAjayMeshGenerator, making a more efficient, elegant code. tAjayMeshGenerator is defined in unit AjayGenerator.

Fields:
None

Methods:

function NodeTooCloseToBoundary(Node: pNode): boolean;
Tests whether the node specified by Node is too close to an existing boundary node, which would result in poor element shape. If so, the node is deleted.

procedure GenerateNodes(Edge: pEdge;
    frontward: boolean;
    aList: pCollection);
Generates equally spaced nodes along the edge specified by Edge based on the mesh density parameter. The boolean argument frontward tells whether nodes should be generated from the edge's beginning to end, or from the end to the beginning. This is important when generating boundary nodes around the perimeter of the domain, where nodes cannot be generated at the beginning and end of each line, which would result in two nodes at each vertex of the perimeter.

The resulting nodes are stored in the collection aList.

procedure CreateBoundaryNodes;
Creates nodes on each of the edges of the plate segment's plane surface representation by calling procedure GenerateNodes for each edge, first determining the proper value for the boolean argument frontward.

procedure CreateInteriorNodes;
Creates equally spaced LineEdges throughout the domain, and classifies them with respect to the solid. The portions of the LineEdges that are "IN" the geometry get nodes generated along them at a spacing determined by the mesh density parameter by calling procedure GenerateNodes.

procedure DeleteExternalTriangles;
Calculates the center point of each triangle in the mesh, and classifies this point with respect to the segment's solid. By doing so, the triangles which are outside of the domain can be identified and deleted.

procedure CreateMesh; virtual;
Initializes a new Mesh object of type tAjayMesh, creates boundary nodes and interior nodes, and then calls tAjayMesh.CreateMesh to generate elements by Delaunay Triangulation.

end; {tAjayMeshGenerator}
tAttributeList:

```pascal
tAttributeList = object(tObject)
```

This type is not used for anything. It was a part of the work of Ajay Garg's 1990 thesis, but was not used there either. This object type, or something like it, could provide a way of defining loads and boundary conditions.

**Fields:**

```pascal
{   fForceVect: XYZExtPoint,   }{   fForceMag: real;   }{   fFreedom: integer;   }
```

**Methods:**

```pascal
{   constructor Init;   }
```

---

tBeamElement:

```pascal
tBeamElement = object(tElement)
```

This element type can be used to represent one-dimensional domains. Currently, it can be used as a beam element, although it would be possible to extend it to be used as a truss element, a heat-conducting rod, or some other element type. `tBeamElement` descends from the ancestor `tElement`, and is defined in unit `BeamElements`.

**Fields:**

N1, N2: pNode; The end nodes of the element
N3: pNode; This node is needed to define Orientation

**Methods:**

constructor Init;

Calls `tElement.Init`, then sets three node fields to nil.

function Description: string; virtual;

Returns the string "2-Node Beam Element"

procedure Draw(Window: tRect; Motion: tMatrix); virtual;

Draws the element in the portion of the screen specified by `Window`, first applying the matrix specified by `Motion` to the nodes, which allows elements (nodes) defined in local coordinate systems to be drawn in the global coordinate system.
tBeamElement

procedure DrawNodes(Window: tRect; Motion: tMatrix); virtual;
   Calls tNode.Draw for each of the three nodes.

function ANSYSTypeString: string; virtual;
   Returns the string "BEAM4" if output format is ANSYS version 5.0, or simply "4" if the output format is ANSYS version 4.4.

function Supported: boolean; virtual;
   Tells whether element type is supported given the values of OutputFormat and AnalysisType.

procedure Write(var OutFile: Text); virtual;
   Writes the element to the output file for future analysis by commercial analysis package.

destructor Done; virtual;
   Sets the three node pointers to nil before calling tElement.Done.

end; {tBeamElement}

tBeamMesh:

type tBeamMesh = object(tMesh)

   tBeamMesh is the mesh object corresponding to the mesh generator tBeamMeshGenerator. It is a descendant of type tMesh, and is defined in unit BeamMesh.

Fields:
   None

Methods:
   function TypeString: string; virtual;
      Returns the string "Divided Beam"

   procedure Draw(Window: tRect; Motion: tMatrix;
                   DrawNodes, DrawElements: boolean); virtual;
      Draws the mesh in the portion of the screen specified by Window, first applying the matrix specified by Motion to each of the nodes (this allows nodes defined in local coordinate systems to be drawn in global coordinates). Draws Nodes and Elements based on the values of the boolean input parameters DrawNodes and DrawElements.
procedure Write; virtual;
    Writes the mesh to the output file.

function AddNodeAt(Pos: XYZExtPoint): pNode; virtual;
    Adds a node to the mesh at the position specified by Pos.

procedure DeleteNode(N: pNode); virtual;
    Deletes the specified node from the mesh.

end; {tBeamMesh}

tBeamMeshGenerator: 

type tBeamMeshGenerator = object(tMeshGenerator)

    This is the object type used to create beam meshes. It has but one method -- CreateMesh.
    tBeamMeshGenerator descends from type tMeshGenerator, and is defined in unit BeamGenerator.

    Fields:
    None

    Methods:
    procedure CreateMesh; virtual;
        Generates equally spaced nodes along the LineEdge representation of the beam stored in the
        CGAxis field of the beam segment, and connect them with beam elements.

end; {tBeamMeshGenerator}

tBeamSegment: 

type tBeamSegment = object(tSegment)

    A beam segment is used to represent one-dimensional regions of the geometry to be meshed.
    This is one of three main segment types, the others being tPlateSegment for two-dimensional
    regions and tBrickSegment for three-dimensional regions. Each of these are descendants of
    tSegment, from which they inherit a pointer to a solid called Solid (which holds the portion of the
    CSG tree representation which makes up the segment) as well as some other fields described
    under type tSegment. tBeamSegment is defined in unit BeamSegments.
Fields:
CGAxis: pLineEdge; The simplified representation of the BeamSegment. This LineEdge is at
the center of gravity of the solid making up the beam segment.
Area: real; The cross-sectional area of the beam segment.
Ixx: real; The x-axis moment of inertia of the beam segment.
Iyy: real; The y-axis moment of inertia of the beam segment.
Ixy: real; The xy product of inertia of the beam segment.

Methods:
constructor Init;
Calls tSegment.Init, then initializes a new LineEdge for the field CGAxis. Sets Area, Ixx, Iyy,
and Ixy to zero.

procedure Copy(S: pSegment); virtual;
Copies the CGAxis and the values of Area, Ixx, Iyy, and Ixy from the segment specified by S.

procedure GetMotion(var Motion: tMatrix); virtual;
Returns the motion of the CGAxis, which is the representation of the segment.

function Inspect: word; virtual;
Creates a segment inspection dialog box for the editing of the segment parameters. The
resulting word is equal to the command that closes the dialog box (cmOk or cmCancel), so that
the procedure calling tBeamSegment.Inspect will know whether the edited values of the
parameters are to be saved or not. Note: see Turbo Vision Guide for a description of dialog
boxes and commands.

function ReDefine: word; virtual;
Creates a segment definition dialog box which allows the segment representation (CGAxis)
can be redefined manually.

procedure Draw(Window: tRect); virtual;
Draws the segment in the portion of the screen specified by Window.

procedure DrawRep(Window: tRect); virtual;
Draws the representation of the segment (CGAxis) in the portion of the screen specified by
Window.

procedure DrawIcon(Window: tRect); virtual;
Draws the segment icon in the portion of the screen specified by Window.

procedure TurnAround; virtual;
Turns the segment end for end, so that two beam segments can be combined if they go in
opposite directions.
procedure MoveStartTo(t: real); virtual;
    Moves the beginning of the beam segment \((t_0\) in the \(CGAxis\)) to the value of \(t\), and changes the solid making up the segment to reflect this new length.

procedure MoveEndTo(t: real); virtual;
    Moves the end of the beam segment \((t_1\) in the \(CGAxis\)) to the value of \(t\), and changes the solid making up the segment to reflect this new length.

procedure RotateAxis(Angle: real); virtual;
    Rotates the \(CGAxis\) LineEdge around its axis by an angle specified by \(Angle\). This allows beam segments in different orientations to be combined into a single beam segment, and insures that the values computed for Area, \(I_{xx}\), \(I_{yy}\), and \(I_{xy}\) will be correct.

function CombinesWith(Segment: pSegment): boolean; virtual;
    Tests whether the segment specified by \(Segment\) can be combined with Self into a single beam segment.

function AddNodeAt(p: t3DPoint): pNode; virtual;
    Adds a node to the segment's mesh at a point specified by \(p\). If \(p\) is not on the segment's \(CGAxis\) representation, this method computes the corresponding closest point on that representation and adds a node there.

procedure AddGraphicsNode(p: t3DPoint); virtual;
    Adds a node to the segment's mesh by converting a point on the screen to the closest point on the segment's \(CGAxis\) representation, and adding the node there.

function CheckOutputFormat: word; virtual;
    Checks to see if mesh's element type is supported for current \(OutputFormat\) and \(AnalysisType\).

procedure WriteANSYSParameters(var OutFile: Text); virtual;
    Writes the element type and parameters (including Area, \(I_{xx}\), \(I_{yy}\), and \(I_{xy}\)) to the output file.

procedure WriteNASTRANParameters(var OutFile: Text); virtual;
    Writes the NASTRAN beam property card (PBAR card) to the output file.

procedure WriteSUPERSAPPParameters(var OutFile: Text); virtual;
    Writes the Supersap element control data line, Material property data, area property data, element load factors, fixed-end force data, intermediate load data, and element data lines to the output file.

destructor Done; virtual;
    Disposes of the \(CGAxis\), then calls tSegment.\Done.


tBeamSegment

destructor Kill, virtual;
   Disposes of the CGAxis, then calls tSegment.Kill.

end; {tBeamSegment}

tBlock:

type tBlock = object(tPrimitive)

   This is the block primitive, modeled after the PADL-2 block primitive. Type tBlock descends from type tPrimitive, which in turn descends from type tSolid, from which tBlock inherits a multitude of fields and methods. It is defined in unit Blocks.

   Fields:
   x: real; The x-dimension of the block.
   y: real; The x-dimension of the block.
   z: real; The x-dimension of the block.
   SegAxis: integer; The axis which will be the long axis for a beam segment or short axis for a plate segment (0=undefined, 1=x, 2=y, 3=z)

   Methods:
   constructor Init;
      Calls tPrimitive.Init, then sets SegAxis, x, y, and z equal to zero.

   function ReadValues(phrase: string): boolean; virtual;
      Reads the values of x, y, and z from the string phrase. This is part of the cheesy method for reading the CSG tree from a text file.

   procedure Copy(Solid: pSolid); virtual;
      Copies the solid (if it is a block) specified by Solid.

   procedure Draw(Window: tRect); virtual;
      This procedure draws the block in the portion of the screen specified by Window.

   procedure PrintInfo; virtual;
      This procedure was originally written for debugging, before there was any way of drawing or inspecting the block. Subsequently, this method is never used, but still remains as sort of a vestigial method.

   function Inspect: word; virtual;
      Creates a solid inspection dialog box for the editing of the block's parameters.
procedure ComputeBoundingBox; virtual;
    Computes the bounding box of the block, and stores it with pointer field Box (inherited from tSolid). Bounding boxes greatly simplify set membership classification.

procedure BeamPosition;
    Re-defines the block so that the long axis is the z-axis, which will be the axis of the beam segment created from this block.

procedure PlatePosition;
    Re-defines the block so that the short axis is the z-axis, which will be the surface normal of the plate segment created from this block.

procedure SetLength(L: real); virtual;
    Re-defines block so that length is $L$. Used for creating segments.

procedure TurnAround; virtual;
    Re-defines block so that block is turned end for end. Used for creating beam segments.

function AlignWith(B: pBlock): boolean; virtual;
    Re-defines block so that motion is the same as the motion of block specified by $B$. Used for creating beam segments.

procedure EvaluateBoundary; virtual;
    Evaluates the boundary of the block (surfaces and edges), and stores them in the Surface pointer (inherited from type tSolid).

procedure DefaultSegAxis;
    Computes the default segment axis based on the block's dimensions.

procedure DefaultSegType; virtual;
    Computes the default segment type based on the block's dimensions.

function IsA(what: string): boolean; virtual;
    Sets value equal to tPrimitive.IsA(what), then makes value true if what equals 'BLOCK' or 'BLO'.

end; {tBlock}
tBoundingBox:

```
tBoundingBox
  type tBoundingBox = object(tObject)

  Fields:
  MinX, MaxX: real;  // The minimum and maximum x-coordinates.
  MinY, MaxY: real;  // The minimum and maximum y-coordinates.
  MinZ, MaxZ: real;  // The minimum and maximum z-coordinates.

  Methods:
  constructor Init;
    Calls tObject.Init, and sets all fields (MinX, MaxX, MinY, MaxY, MinZ, and MaxZ) equal to zero.

  procedure Assign(x0, x1, y0, y1, z0, z1: real);
    Sets the values of the fields MinX, MaxX, MinY, MaxY, MinZ, and MaxZ to x0, x1, y0, y1, z0, and z1, respectively.

  function Overlaps(Box: pBoundingBox): boolean;
    Returns true if Self overlaps Box, otherwise false.

  procedure Copy(Box: pBoundingBox);
    Copies the values of MinX, MaxX, MinY, MaxY, MinZ, and MaxZ from Box.

  procedure Expand(amount: real);
    Increases MaxX, MaxY, and MaxZ by amount, and reduces MinX, MinY, and MinZ by amount.

  procedure Draw(Window: tRect);
    Draws the bounding box in the portion of the screen specified by Window.
```

tBrickGenerator:

```
tBrickGenerator
  type tBrickGenerator = object(tMeshGenerator)

  This is the object type used to create three-dimensional meshes (actually 2½D meshes). It has but one method -- CreateMesh. tBrickGenerator descends from type tMeshGenerator, and is defined in unit BrickGenerator.
```
**tBrickGenerator**

**Methods:**
procedure CreateMesh; virtual;

Creates a mesh in a 2½D brick segment by first creating a 2D mesh by Delaunay Triangulation (using *tAjayMeshGenerator*), and then extruding this mesh through the solid by layers, creating 6-node wedge elements from the 3-node triangle elements.

end; {tBrickGenerator}

**tBrickSegment:**

```

type tBrickSegment = object(tSegment)

A brick segment is used to represent three-dimensional regions of the geometry to be meshed. This is one of three main segment types, the others being tBeamSegment for one-dimensional regions and tPlateSegment for two-dimensional regions. Each of these are descendants of tSegment, from which they inherit a pointer to a solid called Solid (which holds the portion of the CSG tree representation which makes up the segment) as well as some other fields described under type tSegment. tBrickSegment is defined in unit BrickSegments.

**Fields:**
Top, Bottom: pSurface; If the segment is 2½D, then these pointers point to the surfaces belonging to the segment's solid which are the "top" and "bottom" of the solid, otherwise they are nil.

**Methods:**
constructor Init,
Calls tSegment.Init, then sets Top and Bottom to nil.

procedure Copy(S: pSegment); virtual;
Copies the segment specified by S.

procedure GetMotion(var Motion: tMatrix); virtual;
Returns the motion of the Solid, which is the representation of the segment.
```
function Inspect: word; virtual;
    Creates a segment inspection dialog box for the editing of the segment parameters. The resulting word is equal to the command that closes the dialog box (cmOk or cmCancel), so that the procedure calling tBrickSegment.Inspect will know whether the edited values of the parameters are to be saved or not. Note: see Turbo Vision Guide for a description of dialog boxes and commands.

procedure Draw(Window: tRect); virtual;
    Draws the segment in the portion of the screen specified by Window.

procedure DrawRep(Window: tRect); virtual;
    Draws the representation of the segment (Solid) in the portion of the screen specified by Window.

procedure DrawIcon(Window: tRect); virtual;
    Draws the segment icon in the portion of the screen specified by Window.

function CombinesWith(Segment: pSegment): boolean; virtual;
    Tests whether the segment specified by Segment can be combined with Self into a single brick segment.

function Meshable: boolean; virtual;
    Tests to see if the segment is in fact 2½D and meshable by the extrusion of a 2D mesh by examining the surfaces of the boundary representation of the solid.

function AddNodeAt(p: t3DPoint): pNode; virtual;
    Currently does nothing, simply returns a nil pointer.

procedure AddGraphicsNode(p: t3DPoint); virtual;
    Currently does nothing.

function Thickness: real; virtual;
    Returns the distance between the Top and Bottom surfaces if the segment in 2½D.

function CheckOutputFormat: word; virtual;
    Checks to see if mesh's element type is supported for current OutputFormat and AnalysisType.

procedure WriteANSYSParameters(var OutFile: Text); virtual;
    Writes the element type and parameters to the output file.

procedure WriteNASTRANParameters(var OutFile: Text); virtual;
    Writes the NASTRAN solid property card (PSOLID card) to the output file.
procedure WriteSUPERSAPParameters(var OutFile: Text), virtual;
   Writes the Supersap element control data line, material property data, distributed surface
   loads, element load factors, and element data lines to the output file.

destructor Done; virtual;
   Sets Top and Bottom to nil, then calls tSegment.Done.

destructor Kill; virtual;
   Sets Top and Bottom to nil, then calls tSegment.Kill.
end; {tBrickSegment}

tCCEdge:

type tCCEdge = object(tEdge)

   Type tCCEdge represents the edge formed by the intersection of two cylinder surfaces. This
   type is based on the type CCEDGE of PADL-2, and has the same parameters. tCCEdge descends
   from tEdge, from which it inherits the real fields $t0$ and $t1$, which are the endpoints of the
   parametric form of the curve. The set of points defined by a CCEdge are given parametrically in
   Appendix B.
   Type tCCEdge is defined in unit CCEdges.

Fields:
   $r1$, $r2$, $h$, $a$, $b$: real; Real parameters used to define the curve.
   sign1, sign2: integer; Integer parameters used to define the curve.

Methods:
constructor Init;
   Calls tEdge.Init, then sets all real parameters ($r1$, $r2$, $h$, $a$, and $b$) to zero, and sets sign1 and
   sign2 to +1.

function tParameter(Point: t3DPoint): real; virtual;
   Returns the value of the parameter $t$ in the parametric form of the curve of the point specified
   by Point.

procedure GetPoint(t: real; var Point: t3DPoint); virtual;
   Computes the coordinates of the point on the curve corresponding to the value $t$ in the
   parametric representation of the curve in global $(x,y,z)$ coordinates, and returns these coordinates
   in the argument Point.
function Equal1(E: pEdge): boolean; virtual;
   True if the edge has the same motion and parameters \((r1, r2, h, a, b, sign1 \text{ and } sign2)\) as \(E\), but may have different end points \((t0 \text{ and } t1)\).

function Equal2(E: pEdge): boolean; virtual;
   True if the edge has the same motion, parameters, and endpoints as \(E\), that is, the edge is identical to \(E\).

function EqualPoints(x, y: real): Boolean; virtual;
   True if the points corresponding to t-parameters of \(x\) and \(y\) are at the same point.

procedure GetTangent(Point: t3DPoint; var Tangent: tVector); virtual;
   Returns the tangent at point \(Point\) in the argument \(Tangent\).

function BoundingBox: pBoundingBox; virtual;
   Computes and returns the bounding box of the edge.

procedure Draw(Window: tRect); virtual;
   Draws the edge in the portion of the screen specified by \(Window\).

procedure Copy(Edge: pEdge); virtual;
   Calls \(tEdge.Copy\), then copies the CCEdge parameters from the edge specified by \(Edge\).

end; \{tCCEdge\}

tCone:

type tCone = object(tPrimitive)

   The cone primitive is not yet supported in CSGMesh. This object type has been included to facilitate the inclusion of the cone primitive in the future. Type tCone can be found in unit Cones.

Fields:
r: real;  The radius of the cone.
h: real;  The length of the cone.

Methods:
None

end; \{tCone\}
tConeSurf:

type tConeSurf = object(tSurface)

Type tConeSurf is not yet supported in CSGMesh. This object type has been included to facilitate the inclusion of the cone primitive and it's related surface type in the future. Type tConeSurf can be found in unit ConeSurfaces.

Fields:
alpha: real; The configuration parameter \( \alpha = \tan(\theta/2) \) of the cone surface.

Methods:
None

end; {tConeSurf}

tCSGInterior:

type tCSGInterior = object(tScroller)

Type tCSGInterior was developed as the interior of the CSGTree window. This object type draws the CSGTree and handles the event when a solid name is clicked on. Type tCSGInterior descends from the Turbo Vision object type tScroller, and is defined in unit ShowTree.

Fields:
SolidArray: array[1..MaxROW, 0..MaxCOL div 6 + 1] of pSolid; An array of pointers to solids which tells the window interior where to put each solid's name on the screen.

Methods:
constructor Init(var Bounds: tRect; aHScrollBar, aVScrollBar: pScrollBar);
    Calls tScroller.Init, then sets up the SolidArray.

procedure Draw; virtual;
    Draws the interior of the CSGTree window.

procedure HandleEvent(var Event: tEvent); virtual;
    Brings up the solid's inspection dialog box when that solid's name is double-clicked, and makes a solid the CurrentlySelectedSolid when that solid's name is single clicked.

end; {tCSGInterior}
tCSGWindow:

```pascal
type tCSGWindow = object(tWindow)

  Type tCSGWindow is the descendant of Turbo Vision type tWindow which makes the CSGTree window. It is defined in unit ShowTree.

Fields:
None

Methods:
constructor Init(Bounds: tRect);
  Calls tWindow.Init, then creates an interior of type tCSGInterior.

procedure ReDraw; virtual;
  Makes sure the title of the window correctly reflects the root solid of the CSG tree before calling tWindow.ReDraw.

procedure MakeInterior(Bounds: tRect);
  Creates a new interior of type tCSGInterior.

procedure HandleEvent(var Event: tEvent); virtual;
  Calls tWindow.HandleEvent, then handles the special event if the left or right arrow are pressed, which results in the narrowing or widening of the solid names.

end; {tCSGWindow}
```

tCylinder:

```pascal
type tCylinder = object(tPrimitive)

  This is the cylinder primitive, modeled after the PADL-2 cylinder primitive. Type tCylinder descends from type tPrimitive, which in turn descends from type tSolid, from which tCylinder inherits a multitude of fields and methods. It is defined in unit Cylinders.

Fields:
r, h: real;

Methods:
constructor Init;
  Calls tPrimitive.Init, then sets r and h equal to zero.
```
function ReadValues(phrase: string): boolean; virtual;
    Reads the values of $r$ and $h$ from the string *phrase*. This is part of the cheesy method for reading the CSG tree from a text file.

procedure Copy(Solid: pSolid); virtual;
    Copies the solid (if it is a cylinder) specified by *Solid*.

procedure GetAxis(var Axis: tVector);
    Returns the axis of the cylinder in the argument *Axis*.

procedure Draw(Window: tRect); virtual;
    This procedure draws the cylinder in the portion of the screen specified by *Window*.

procedure PrintInfo; virtual;
    This procedure was originally written for debugging, before there was any way of drawing or inspecting the cylinder. Subsequently, this method is never used, but still remains as sort of a vestigial method.

function Inspect: word; virtual;
    Creates a solid inspection dialog box for the editing of the cylinder's parameters.

procedure ComputeBoundingBox; virtual;
    Computes the bounding box of the cylinder, and stores it with pointer field *Box* (inherited from *tSolid*). Bounding boxes greatly simplify set membership classification.

procedure SetLength(L: real); virtual;
    Re-defines cylinder so that length ($h$) is $L$. Used for creating segments.

procedure TurnAround; virtual;
    Re-defines cylinder so that cylinder is turned end for end. Used for creating beam segments.

procedure EvaluateBoundary; virtual;
    Evaluates the boundary of the cylinder (surfaces and edges), and stores them in the *Surface* pointer (inherited from type *tSolid*).

procedure DefaultSegType; virtual;
    Computes the default segment type based on the cylinder's dimensions.

function IsA(what: string): boolean; virtual;
    Sets value equal to tPrimitive.IsA(what), then makes value true if *what* equals 'CYLINDER' or 'CYL'.

end; {tCylinder}
tCylinderPlateSegment:

tCylinderPlateSegment = object(tPlateSegment)

A cylinder-plate segment is used to represent curved two-dimensional regions of the geometry to be meshed. This is a special case of type tPlateSegment, from which it descends and inherits a real field called Thickness and a pointer to a surface, called Surface, which is the representation of the segment. The representation of a PlateSegment is a PlaneSurf, while the representation of a CylinderPlateSegment is a CylinderSurf. The type tPlateSegment is one of three main segment types, the others being tBeamSegment for one-dimensional regions and tBrickSegment for three-dimensional regions. Each of these are descendants of tSegment, from which they inherit a pointer to a solid called Solid (which holds the portion of the CSG tree representation which makes up the segment) as well as some other fields described under type tSegment.

Fields:
None

Methods:
constructor Init;
   Calls tSegment.Init, initializes a new CylinderSurf for field Surface, and sets Thickness to zero.

procedure Copy(S: pSegment); virtual;
   Copies the segment specified by S.

function SetDrawOptions: word; virtual;
   Calls tSegment.SetDrawOptions (to override tPlateSegment's ability to draw from face on).

procedure ViewFaceOn; virtual;
   Does nothing (to override tPlateSegment's ability to draw from face on).

procedure DefaultValueSettings; virtual;
   Calls tSegmentDefaultValueSettings (to override tPlateSegment's ability to draw from face on).

procedure DrawRep(Window: tRect); virtual;
   Draws the representation of the segment (Surface, in this case type tCylinderSurf) in the portion of the screen specified by Window.

function CombinesWith(Segment: pSegment): boolean; virtual;
   Tests whether the segment specified by Segment can be combined with Self into a single CylinderPlate segment.
function AddNodeAt(p: t3DPoint): pNode; virtual;
   Currently does nothing, simply returns a nil pointer.

procedure AddGraphicsNode(p: t3DPoint); virtual;
   Currently does nothing.

end; {tCylinderPlateSegment}

tCylinderSurf:

tCylinderSurf = object(tSurface)

Type tCylinderSurf is used to represent cylindrical surfaces. It descends from type tSurface, from which it inherits a pointer to a list of edges called Edge, a integer field called Outside, and a pointer called Next used for storing surfaces in linked lists. Type tCylinderSurf has one additional field -- the real field r, which represents the radius of the cylindrical surface. The length of the cylindrical surface is infinite. The CylinderSurf is based on the PADL-2 "CSURF" halfspace, which is described in Appendix B. Type tCylinderSurf is defined in unit CylSurface.

Fields:
r: real; The radius of the cylindrical surface.

Methods:
constructor Init;
   Calls tSurface.Init and sets r equal to zero.

procedure GetNormal(Point: t3DPoint; var Normal: tVector); virtual;
   Returns the surface normal at the point specified by Point in the variable Normal.

function ProfileEdges: pEdge; virtual;
   Returns a pointer to a list of profile edges based on the current viewing parameters. For a cylinder surface, the profile edges are the sides of the cylinder. These are not actual edges of the cylinder, and their apparent position depends on the viewpoint from which the cylinder is being observed.

function Intersection(Surface: pSurface): pEdge; virtual;
   Returns the edge (or edges) of intersection of Self with the surface specified by Surface.

function ClassifyLineEdge(E: pLineEdge): pMCR; virtual;
   Classifies the line edge E with respect to Self, determining which portions are "IN", "ON", and "OUT" of the cylindrical surface.
function ClassifyEllipseEdge(E: pEllipseEdge): pMCR; virtual;
    Classifies the ellipse edge E with respect to Self, determining which portions are "IN", "ON", and "OUT" of the cylindrical surface.

function ClassifyCCEdge(E: pCCEdge): pMCR; virtual;
    Classifies the CCEdge E with respect to Self, determining which portions are "IN", "ON", and "OUT" of the cylindrical surface.

function Equals(Surface: pSurface): boolean; virtual;
    Returns true if the surface specified by Surface is equal to Self.

function Parallel(Surf: pSurface): boolean; virtual;
    Returns true if the surface specified by Surface is parallel to Self. (This function only returns true for coaxial cylindrical surfaces.)

function Perpendicular(Surf: pSurface): boolean; virtual;
    Returns true if the surface specified by Surface is perpendicular to Self. (This function only returns true for plane surfaces who's normal is parallel to the cylindrical surface's axis.)

procedure Copy(Surface: pSurface); virtual;
    Copies the surface specified by Surface.

end; {tCylinderSurf}

tCylMesh:

type tCylMesh = object(tAjayMesh)

    Type tCylMesh is a descendant of tAjayMesh, from which it inherits most of it's fields and methods. This is possible because the associated mesh generator, tCyMeshGenerator, first generates nodes on a cylindrical surface, then "unrolls" these nodes into a plane which is meshed by Delaunay Triangulation just as tAjayMesh. This type is not very well developed, and much could be done to improve it, such as methods for adding nodes and otherwise editing the mesh.

    Fields:
    None

    Methods:
    function TypeString: string; virtual;
        Returns the string "Cylindrical Plate"

end; {tCylMesh}
tCylMeshGenerator:  

\[
t\text{CylMeshGenerator} = \text{object(tAjayMeshGenerator)}
\]

\[\text{tCylMeshGenerator is the mesh generator object type developed to generate meshes on CylinderPlate segments. tCylMeshGenerator descends from tAjayMeshGenerator, and is defined in unit CylGenerator. This mesh generator object type generates a mesh by first defining nodes on the cylindrical surface, then "unrolling" those nodes into a plane, and performing Delaunay Triangulation on the re-defined nodes. The mesh is then "rolled" back into a cylinder. The methods for performing the Delaunay Triangulation in the plane are inherited from type tAjayMeshGenerator.}\]

Fields:
None

Methods:

- procedure CreateInteriorNodes;
  
  Creates equally spaced LineEdges around the circumference of the cylinder surface parallel to the cylinder's axis, and classifies them with respect to the solid. The portions of the LineEdges that are "IN" the geometry get nodes generated along them at a spacing determined by the mesh density parameter by calling procedure GenerateNodes.

- procedure UnRollNodes;
  
  Re-defines the nodes so that they can be thought of as being in a plane. Nodes are originally defined in rectangular coordinates \((x, y, z)\). Each node's location is converted to cylindrical coordinates \((r, \theta, z)\). The coordinates of the node are then computed in a new cylindrical/rectangular system as follows: \(x = \theta, \ y = z, \ z = 0\).

- procedure RollNodes;
  
  Re-defines the nodes back into regular rectangular \((x, y, z)\) coordinates from the coordinates defined in procedure \text{UnRollNodes} as follows: \(x = r \cos(\theta), \ y = r \sin(\theta), \ z = z\).

- procedure CreateMesh; virtual;
  
  Initializes a new Mesh object of type \text{tCylMesh}, creates boundary nodes and interior nodes, Un-Rolls those nodes, calls \text{tCylMesh.CreateMesh} (inherited from \text{tAjayMesh}) to generate elements by Delaunay Triangulation, and then Rolls the mesh back up into a cylinder.

end; \{tCylMeshGenerator\}
**tDifference:**

```type
tDifference = object(tOperation)
```

This is the Difference operation, one of the three boolean set operations available for creating complex geometries from primitive solids (the other two are Union and Intersection). Type `tDifference` descends from `tOperation`, which in turn descends from type `tSolid`, from which `tDifference` inherits a multitude of fields and methods. From `tOperation`, `tDifference` inherits two pointers to solids, `Left` and `Right`, which hold the solids on which the difference operation acts. `TDifference` is defined in unit Differences.

**Fields:**
None

**Methods:**

- **procedure Rename; virtual;**
  Creates a new name for the Difference (The `Name` field is a pointer to a string inherited from `tSolid`). This name is equal to the `Left` solid's name + "-" + the `Right` solid's name. Thus, if the `Left` solid were named "L" and the `Right` solid were named "R", the new name of the Difference would be "L-R"

- **procedure PrintInfo; virtual;**
  This procedure was originally written for debugging, before there was any way of drawing or inspecting the difference. Subsequently, this method is never used, but still remains as sort of a vestigial method.

- **procedure ComputeBoundingBox; virtual;**
  Computes the bounding box of the difference (equal to the bounding box of the `Left` solid), and stores it with pointer field `Box` (inherited from `tSolid`). Bounding boxes greatly simplify set membership classification.

- **procedure EvaluateBoundary; virtual;**
  Evaluates the boundary of the difference (surfaces and edges), and stores them in the `Surface` pointer (inherited from type `tSolid`).

- **function ClassifyEdge(E: pEdge): pMCR; virtual;**
  Classifies an edge with respect to the difference, determining which portions of the edge are "IN", "ON", and "OUT" of the difference solid. This is done by classifying the edge with respect to the `Left` and `Right` solids, and combining the resulting MCRs into a single MCR.

- **procedure DefaultSegType; virtual;**
  Computes the default segment type based on the `Left` and `Right` solids' segment types.
function IsA(what: string): boolean; virtual;
Sets value equal to tOperation.IsA(what), then makes value true if what equals 'DIFFERENCE' or 'DIF'
end; {tDifference}

tDuplicateNodeMarker:

tDuplicateNodeMarker

A DuplicateNodeMarker is used to mark coincident nodes that are part of different segment's meshes. The merging and connecting of individual segment meshes leaves duplicate nodes at the interfaces of the individual segments. This object type provides a method for dealing with these nodes. More methods could be added in the future to elegantly eliminate duplicate nodes. This type could also be used to handle node offsets, which arise in the merging and connecting of beam segment meshes. Type tDuplicateNodeMarker is defined in unit Nodes.

Fields:
Node1, Node2: pNode; The two coincident nodes.

Methods:
constructor Init(N1, N2: pNode);
Creates a new DuplicateNodeMarker, and sets the Node1 and Node2 fields to N1 and N2 respectively.
end; {tDuplicateNodeMarker}

tEdge:

tEdge

Type tEdge is the common ancestor of all edge types. It's t0 and t1 fields represent the endpoints in the edge's parametric representation. (Each edge type has a unique representation, based on the PADL-2 representation of the edge. These representations are given in Appendix B.) The Next field is used for storing edges in linked lists. TEdge inherits a Motion and methods for dealing with this motion from ancestor tGeometricEntity. TEdge is defined in unit Edges.
tEdge

Fields:
t0: real; The beginning value of \( t \) in the parametric equation of the edge.
t1: real; The ending value of \( t \) in the parametric equation of the edge.
Outside: integer; Not used. Was meant for a "neighborhood" type function.
Next: pEdge; Pointer to the next edge in a linked list.

Methods:
constructor Init;
   Calls tGeometricEntitity.Init, sets \( t0 \) and \( t1 \) to zero, sets Outside to +1, and sets Next to nil.

function Length: real; virtual;
   Abstract method. This method must be overridden in descendant object types.

function tParameter(Point: t3DPoint): real; virtual;
   Abstract method. This method must be overridden in descendant object types.

procedure GetPoint0(var Point: t3DPoint), virtual;
   Calls GetPoint(t0, Point).

procedure GetPoint1(var Point: t3DPoint), virtual;
   Calls GetPoint(t1, Point).

procedure GetPoint(t: real; var Point: t3DPoint); virtual;
   Abstract method. This method must be overridden in descendant object types.

function Equal1(E: pEdge): boolean; virtual;
   Abstract method. This method must be overridden in descendant object types.

function Equal2(E: pEdge): boolean; virtual;
   Abstract method. This method must be overridden in descendant object types.

function EqualPoints(a,b: real): Boolean; virtual;
   Abstract method. This method must be overridden in descendant object types.

procedure GetTangent(Point: t3DPoint; var Tangent: tVector); virtual;
   Abstract method. This method must be overridden in descendant object types.

function BoundingBox: pBoundingBox; virtual;
   Abstract method. This method must be overridden in descendant object types.

procedure Copy(Edge: pEdge); virtual;
   Copies \( t0, t1, \) Outside, and Motion from edge specified by \( Edge \).
procedure Complement; virtual;
   Sets Outside equal to -Outside. This method is never used.

destructor Done; virtual;
   Disposes of Next, if not equal to nil, then calls tGeometricEntity.Done.

end; {tEdge}

tElement:

type tElement = object(tObject)

   Type tElement is the common ancestor for all element types. It has no fields, but provides abstract methods for many things. It is defined in unit Elements.

Fields:
None

Methods:
constructor Init;
   Calls tObject.Init, updates clock and heap views on desktop, and checks for sufficient memory.

function Description: string; virtual;
   Returns the string "Generic Element"

procedure Draw(Window: tRect; Motion: tMatrix); virtual;
   Abstract method. This method must be overridden in descendant object types.

procedure DrawNodes(Window: tRect; Motion: tMatrix); virtual;
   Abstract method. This method must be overridden in descendant object types.

function ANSYSTypeString: string; virtual;
   Abstract method. This method must be overridden in descendant object types.

function Supported: boolean; virtual;
   Abstract method. This method must be overridden in descendant object types.

procedure Write(var OutFile: Text); virtual;
   Abstract method. This method must be overridden in descendant object types.
destructor Done; virtual;
   Calls tObject.Done.

end; {tElement}

tEllipseEdge:

tEllipseEdge
type tEllipseEdge = object(tEdge)

Type tEllipseEdge is used to represent ellipses. This type is based on the type "ELLIPS" of
PADL-2, and has the same parameters. tEllipseEdge descends from tEdge, from which in inherits
the real fields to and tl, which are the endpoints of the parametric form of the curve. The set of
points defined by an ellipse edge are given parametrically in Appendix B.

Type tEllipseEdge is defined in unit EllipseEdges.

Fields:
Ru: real; The local x-axis radius of the ellipse.
Rv: real; The local y-axis radius of the ellipse.

Methods:
constructor Init;
   Calls tEdge.Init, then sets Ru and Rv equal to zero

function Length: real; virtual;
   Returns the Length of the curve.

function tParameter(Point: t3DPoint): real; virtual;
   Returns the value of the parameter t in the parametric form of the curve of the point specified
by Point.

procedure GetPoint(t: real; var Point: t3DPoint); virtual;
   Computes the coordinates of the point on the curve corresponding to the value t in the
parametric representation of the curve in global (x,y,z) coordinates, and returns these coordinates
in the argument Point.

function Equal1(E: pEdge):boolean; virtual;
   True if the edge has the same motion and parameters (Ru and Rv) as E, but may have different
end points (t0 and tl).
function Equal2(E: pEdge): boolean; virtual;
    True if the edge has the same motion, parameters, and endpoints as E, that is, the edge is identical to E.

function EqualPoints(a, b: real): Boolean; virtual;
    True if the points corresponding to t-parameters of x and y are the same point.

procedure GetTangent(Point: t3DPoint; var Tangent: tVector); virtual;
    Returns the tangent at point Point in the argument Tangent.

procedure Draw(Window: tRect); virtual;
    Draws the edge in the portion of the screen specified by Window.

function BoundingBox: pBoundingBox; virtual;
    Computes and returns the bounding box of the edge.

procedure Copy(Edge: pEdge); virtual;
    Calls tEdge.Copy, then copies the ellipse parameters from the edge specified by Edge.

end; {tEllipseEdge}

tGeometricEntity:

tGeometricEntity

type tGeometricEntity = object(tObject)

    Type tGeometricEntity is the common ancestor of all Edges, Surfaces, and Solids. This type provides a rigid motion, stored in the matrix field Motion, as well as several methods for dealing with this rigid motion. A rigid motion is a (3x4) matrix which defines a local coordinate system as follows: the first column represents the local x-axis, the second column represents the local y-axis, the third column represents the local z-axis, and the fourth column represents the origin of the local coordinate system.

    Having the common ancestor tGeometricEntity allows all edges, surfaces and solids to handle rigid motions in a uniform, straightforward fashion, which greatly simplifies writing and debugging code. Type tGeometricEntity is defined in unit GeometricEntities.

Fields:
Motion: tMatrix; A matrix representing a rigid motion.

Methods:
constructor Init;
    Calls tObject.Init, then sets the Motion equal to the identity matrix (putting the geometric entity at the global coordinate system origin in standard position). This method also updates the
clock and heap views on the desktop, and checks to see if the available space on the heap is getting low. If so, it calls the application's `OutOfMemory` method.

procedure Draw(Window: tRect); virtual;
   Abstract method. This method must be overridden in descendant object types.

procedure GetMotion(var u,v,w,p: t3DPoint);
   Sets the argument \( u \) equal to the local x-axis (first column of the rigid motion matrix), \( v \) equal to the local y-axis (second column), \( w \) equal to the local z-axis (third column), and \( p \) equal to the origin of the local coordinate system (fourth column of the rigid motion matrix).

procedure PutMotion(u,v,w,p: t3DPoint);
   Sets the matrix field `Motion` equal to the rigid motion defined by the vectors \( u \), \( v \), \( w \), and \( p \), where \( u \) is the local x-axis, \( v \) is the local y-axis, \( w \) is the local z-axis, and \( p \) is the origin of the local coordinate system.

procedure GetOrigin(var p: t3DPoint);
   Returns the origin of the local coordinate system (fourth column of the matrix field `Motion`) in the argument \( p \).

procedure PutOrigin(p: t3DPoint);
   Sets the origin of the local coordinate system (fourth column of the matrix field `Motion`) equal to the vector specified by \( p \).

procedure GetU(var u: t3DPoint);
   Returns the x-axis of the local coordinate system (first column of the matrix field `Motion`) in the argument \( u \).

procedure GetV(var v: t3DPoint);
   Returns the y-axis of the local coordinate system (second column of the matrix field `Motion`) in the argument \( v \).

procedure GetW(var w: t3DPoint);
   Returns the z-axis of the local coordinate system (third column of the matrix field `Motion`) in the argument \( w \).

procedure Move(direction: tVector; magnitude: real); virtual;
   Moves the origin of the local coordinate system (fourth column of the matrix field `Motion`) \( \text{magnitude} \) units along the vector specified by \( \text{direction} \).

end; {tGeometricEntity}
tHeadingBox (unit Materials):

type tHeadingBox = object(tView)

Type tHeadingBox is used for the column headings at the top of certain windows (in this case, the Material window -- the heading box gives labels to the Number and Name columns).

Fields:
None

Methods:
constructor Init(Bounds: tRect);
   Initializes the HeadingBox.

procedure Draw; virtual;
   Draws the HeadingBox.

end; {tHeadingBox (unit Materials)}

---

tHeadingBox (unit Segments):

type tHeadingBox = object(tView)

Type tHeadingBox is used for the column headings at the top of certain windows (in this case, the Segments window -- the heading box gives labels to the Solid Name, Type, Material, and Mesh columns).

Fields:
None

Methods:
constructor Init(Bounds: tRect);
   Initializes the HeadingBox.

procedure Draw; virtual;
   Draws the HeadingBox.

end; {tHeadingBox (unit Segments)}
tHeadingBox (unit Solids):

type tHeadingBox = object(tView)

Type tHeadingBox is used for the column headings at the top of certain windows (in this case, the Solids window -- the heading box gives labels to the Name, Type, Matl, and SegType columns).

Fields:
None

Methods:
constructor Init(Bounds: tRect);
    Initializes the HeadingBox.

procedure Draw; virtual;
    Draws the HeadingBox.

end; {tHeadingBox (unit Solids)}

tHyperbolaEdge:

type tHyperbolaEdge = object(tEdge)

This type is not currently supported in CSGMesh. It was included to facilitate the inclusion of the HyperbolaEdge, which can result from the intersection of a plane surface (tPlaneSurf) and a ConeSurface (tConeSurf -- not yet supported). This object type inherits real fields $t0$ and $t1$ from parent tEdge, which denote the endpoints of the edge. tHypbolaEdge is defined in unit HyperbolaEdges.

Fields:
a, b: real;   Parameters defining the hyperbola. For a mathematical description, see Appendix B.

Methods:
None

end; {tHyperbolaEdge}
This is the Intersection operation, one of the three boolean set operations available for creating complex geometries from primitive solids (the other two are Union and Difference). Type tIntersection descends from tOperation, which in turn descends from type tSolid, from which tIntersection inherits a multitude of fields and methods. From tOperation, tIntersection inherits two pointers to solids, Left and Right, which hold the solids on which the intersection operation acts. TIntersection is defined in unit Intersections.

Fields:
None

Methods:

procedure Rename; virtual;

Creates a new name for the Intersection (The Name field is a pointer to a string inherited from tSolid). This name is equal to the Left solid's name + "*" + the Right solid's name. Thus, if the Left solid were named "L" and the Right solid were named "R", the new name of the Difference would be "L*R"

procedure PrintInfo; virtual;

This procedure was originally written for debugging, before there was any way of drawing or inspecting the intersection. Subsequently, this method is never used, but still remains as sort of a vestigial method.

procedure ComputeBoundingBox; virtual;

Computes the bounding box of the intersection (equal to the intersection of the bounding box of the Left solid and the bounding box of the Right solid), and stores it with pointer field Box (inherited from tSolid). Bounding boxes greatly simplify set membership classification.

procedure EvaluateBoundary; virtual;

Evaluates the boundary of the intersection (surfaces and edges), and stores them in the Surface pointer (inherited from type tSolid).

function ClassifyEdge(E: pEdge): pMCR; virtual;

Classifies an edge with respect to the intersection, determining which portions of the edge are "IN", "ON", and "OUT" of the intersection solid. This is done by classifying the edge with respect to the Left and Right solids, and combining the resulting MCRs into a single MCR.

procedure DefaultSegType; virtual;

Computes the default segment type based on the Left and Right solids' segment types.
function IsA(what: string): boolean; virtual;
    Sets value equal to tOperation.IsA(what), then makes value true if what equals 'INTERSECTION' or 'INT'.
end; {tIntersection}

tKKEdge:  
tKKEdge

type tKKEdge = object(tEdge)

    This type is not currently supported in CSGMesh. It was included to facilitate the inclusion of the KKEdge, which can result from the intersection of two cone surfaces (tConeSurf -- not yet supported). This object type inherits real fields t0 and t1 from parent tEdge, which denote the endpoints of the edge. tKKEdge is defined in unit KKEdges.

    Fields:
    a, b, c, d, e, f, g, alpha: real;  Real parameters defining KKEdge. For a mathematical description, see Appendix B.
    sign: integer;  Integer parameters defining KKEdge. For a mathematical description, see Appendix B.

    Methods:
    None
end; {tKKEdge}

tLine (unit Lines):  
tLine (unit Lines)

type tLine = object(tObject)

    Type tLine was developed to represent a line, primarily for drawing purposes in graphics mode.

    Fields:
    p1, p2: t3DPoint;  The endpoints of the line.

    Methods:
    procedure Assign(x1,y1,z1,x2,y2,z2: real);
        Sets the fields p1 and p2 to the coordinates (x1, y1, z1) and (x2, y2, z2), respectively.
procedure Apply(Matrix: tMatrix);
    Applies the motion specified by the matrix Matrix to each of the endpoints, p1 and p2.

procedure Draw(Window: tRect);
    Draws the line in the portion of the screen specified by Window.

end; {tLine (unit Lines)}

tLine (unit uLine):
    type tLine = object(tObject)

    This object type, tLine, comes from the work done by Ajay Garg. Because of the changes made in representation scheme, this type has no real use. Therefore, although the fields and methods are listed, no descriptions will be given.

    Fields:
    fl, f2, fv: XYZExtPoint;   { 2 Endpoints and Direction  }

    Methods:
    constructor Init;
    constructor ILine(A,B: XYZExtPoint);
    procedure SetEnds(A,B: XYZExtPoint);
    function XZPlaneInt(X, Z: real; var ISN: XYZExtPoint; var U: real): boolean;
    function YZPlaneInt(Y, Z: real; var ISN: XYZExtPoint; var U: real): boolean;
    function XYPlaneInt(X,Y: real; var ISN: XYZExtPoint; var U: real): boolean;
    function MinDist(p: XYZExtPoint; var U,V: real): real;
    procedure SolveIndirect(p: XYZExtPoint; var Found: XYZUVPoint);
    procedure SolveDirect(u,v: real; var Result: XYZExtPoint);

end; {tLine (unit uLine)}

tLineEdge:
    type tLineEdge = object(tEdge)

    Type tLineEdge is used to represent lines. This type is based on the type "LINE" of PADL-2, and has the same parameters. tLineEdge descends from tEdge, from which it inherits the real
fields $t_0$ and $t_1$, which are the endpoints of the parametric form of the curve. The set of points defined by an line edge are given parametrically in Appendix B.

Type tLineEdge is defined in unit LineEdges

Fields:
None

Methods:
function Length: real; virtual;
   Returns the Length of the line.

function tParameter(Point: t3DPoint): real; virtual;
   Returns the value of the parameter $t$ in the parametric form of the line of the point specified by Point.

procedure GetPoint(t: real; var Point: t3DPoint); virtual;
   Computes the coordinates of the point on the line corresponding to the value $t$ in the parametric representation of the line in global $(x,y,z)$ coordinates, and returns these coordinates in the argument Point.

function Equal1(E: pEdge): boolean; virtual;
   True if the edge has the same motion and parameters ($Ru$ and $Rv$) as $E$, but may have different end points ($t_0$ and $t_1$).

function Equal2(E: pEdge): boolean; virtual;
   True if the edge has the same motion, parameters, and endpoints as $E$, that is, the edge is identical to $E$.

function EqualPoints(a,b: real): Boolean; virtual;
   True if the points corresponding to t-parameters of $x$ and $y$ are the same point.

procedure GetTangent(Point: t3DPoint; var Tangent: tVector); virtual;
   Returns the tangent at point Point in the argument Tangent.

procedure Draw(Window: tRect); virtual;
   Draws the edge in the portion of the screen specified by Window.

function BoundingBox: pBoundingBox; virtual;
   Computes and returns the bounding box of the edge.

procedure Copy(Edge: pEdge); virtual;
   Calls tEdge.Copy, then copies the ellipse parameters from the edge specified by Edge.
function DistanceToPoint(p: t3DPoint): real; virtual;
Computes and returns the shortest distance from the line to the point specified by p.

end; {tLineEdge}

tMaterial:

type tMaterial = object(tObject)

Type tMaterial is used to store material property information for the solid model. In addition to being a simple record of physical properties, this object type has the ability to display the information of the screen in a dialog box, and to write the properties to an output file in a variety of formats.

Currently, only structural properties are supported for materials. In the future, it is hoped that Materials are one of the portions of CSGMesh that will be extended. Possibilities exist for including other types of properties, such as thermal, magnetic, etc., which would be used for different types of analyses.

Materials are stored in the global list MaterialList. Type tMaterial is defined in unit Materials.

Fields:
Number: integer; The material's unique identification number. Must be greater than zero and less than 100.

Name: pString; The name of the material.

{ Structural Properties: }
E: real; { Young's Modulus (Modulus of Elasticity) }
NU: real; { Poisson's Ratio }
G: real; { Shear Modulus }
DENS: real; { Mass Density }
ALPHA: real; { Thermal Expansion Coefficient }
DAMP: real; { Damping Coefficient }

{ Thermal Properties: } (not yet supported)
{ C: real; } { Specific Heat }
{ ENTH: real; } { Enthalpy }
{ HF: real; } { Convection or Film Coefficient }
{ K: real; } { Heat Conduction Coefficient }
{ EMIS: real; } { Emissivity }

Next: pMaterial; The next material in the MaterialList.
tMaterial

Methods:
constructor Init;
    Sets the material's Number to the first available material number, sets the Name to "" (blank), and sets all properties equal to zero.

function Prev: pMaterial;
    Returns the previous material in the MaterialList.

function ZeroValues: boolean;
    Returns true if all of the properties' values are equal to zero and the Name is equal to "" (blank).

function Inspect: word, virtual;
    Creates a material inspection dialog box for the editing of the material's parameters.

procedure Add, virtual;
    Adds the material to the MaterialList after checking to make sure that the material's number is unique.

procedure RemoveFromList; virtual;
    Removes the material from the MaterialList.

procedure Write(var OutFile: Text);
    Writes the material's properties to the file specified by OutFile in the currently selected OutputFormat (ANSYS, NASTRAN, or Supersap).

destructor Done; virtual;
    Disposes of the Next material in the MaterialList, then disposes of Self.

end; {tMaterial}

tMaterialListInterior:

tMaterialListInterior

type tMaterialListInterior = object(tScroller)

    Type tMaterialListInterior was developed as the interior of the tMaterialListWindow window. This object type draws the MaterialList and handles the event when a material name is clicked on. Type tMaterialListInterior descends from the Turbo Vision object type tScroller, and is defined in unit Materials.
**tMaterialListInterior**

**Fields:**

- `MaterialAt: array[1..99] of pMaterial;`  
  An array containing the `MaterialList`. Note that the index of each material in the array is not necessarily that material's `Number`, but instead the position of that material in the `MaterialList`. This array makes it possible to convert a mouse click on some line of the `MaterialListWindow` to the proper material in the `MaterialList`.

**Methods:**

- `constructor Init(Bounds: tRect; aScrollBar: pScrollBar);`  
  Initializes the `tMaterialListInterior`.

- `procedure Draw; virtual;`  
  Draws the `tMaterialListInterior` by simply listing the materials in the `MaterialList`.

- `procedure HandleEvent(var Event: tEvent); virtual;`  
  If a material name is clicked on, makes that material the `CurrentlySelectedMaterial`. If a material name is double clicked on, calls that material's `Inspect` method.

**tMaterialListWindow:**

**type tMaterialListWindow = object(tWindow)**

Type `tMaterialListWindow` is the descendant of Turbo Vision type `tWindow` which makes the `MaterialList` window. It is defined in unit Materials.

**Fields:**

- None

**Methods:**

- `constructor Init;`  
  Creates a `HeadingBox` and a `MaterialListInterior` and inserts them into the newly initialized window.

end; {tMaterialListWindow}
tMCR:

```pascal
type tMCR = object(tObject)

MCR stands for Membership Classification Result. Type tMCR stores the result when an edge is classified against a surface or a solid. An MCR has three lists: In, On, and Out. When an edge is classified against a surface or solid, the edge is broken up into pieces that are inside, outside, and on the surface or solid. These three lists store the appropriate pieces of the edge.

Type MCR is defined in unit MCRs.

Fields:
InEdge: pEdge;      The list of edge portions that are inside the surface or solid.
OnEdge: pEdge;      The list of edge portions that are on the surface or solid's boundary.
OutEdge: pEdge;      The list of edge portions that are outside the surface or solid.

Methods:
constructor Init;
  Sets InEdge, OnEdge, and OutEdge to nil.

procedure AddIn(E: pEdge);
  Adds the edge specified by E to the InEdge list.

procedure AddOn(E: pEdge);
  Adds the edge specified by E to the OnEdge list.

procedure AddOut(E: pEdge);
  Adds the edge specified by E to the OutEdge list.

procedure Draw(Window: tRect);
  Draws the edges in all three lists in different colors in the portion of the screen specified by Window. This procedure was written for debugging purposes, and is not used in the finished version of CSGMesh.

destructor Done; virtual;
  Disposes of any edges in the three lists, then disposes of self.

end; {tMCR}
```
type tMesh = object(tObject)

The ancestor of all mesh types is tMesh. This object type contains collections for nodes and elements, a real field for storing the mesh's density (node spacing), and a boolean field called NodesLocal, who's value tells whether the nodes are defined in the local coordinate system of the segment owning the mesh (a true value), or whether they are defined in global coordinates (a false value). Most of tMesh's methods are abstract, and must be overridden in the object types descended from tMesh. Type tMesh is defined in unit Meshes.

Fields:
Density: real; The density at which the mesh was generated. In CSGMesh, "mesh density" refers to the node spacing, or the approximate distance between adjacent nodes.
Elements: pCollection; A collection for storing the elements in the mesh.
Nodes: pCollection; A collection for storing the nodes in the mesh.
NodesLocal: boolean; A true value means that the nodes are defined in the local coordinate system of the segment owning the mesh. A false value means that the nodes are defined in global coordinates.

Methods:
constructor Init;
Sets Density equal to zero, and initializes new collections for Nodes and Elements.

procedure Draw(Window: tRect; Motion: tMatrix;
DrawNodes, DrawElements: boolean); virtual;
Draws the mesh in the portion of the screen specified by Window. If the argument DrawNodes is true, then nodes will be drawn, otherwise they will not. If the argument DrawElements is true, then elements will be drawn, otherwise they will not. The matrix passed in the argument Motion is sent along as an argument to the node and element drawing methods. If the nodes are defined in a local coordinate system, passing this coordinate system in the Motion argument allows the mesh to be drawn in the global coordinate system.

function Description: string; virtual;
Returns a string describing the number of nodes and elements in the mesh, as well as the type of element. For example: "142 3-Node Triangle Elements / 438 Nodes."

function TypeString: string; virtual;
Returns the string "Generic Mesh." This method is meant to be overridden in descendant object types.

function Smooth: boolean; virtual;
Returns the value FALSE and creates a message box stating that a mesh of this type cannot be smoothed. Descendant meshes which can be smoothed must override this method.
function ClosestNode(Pos: XYZExtPoint): pNode; virtual;
   Returns the closest node in the mesh to the point specified by Pos.

function AddNodeAt(Pos: XYZExtPoint): pNode; virtual;
   Abstract method. This method must be overridden in descendant object types.

procedure DeleteNode(N: pNode); virtual;
   Abstract method. This method must be overridden in descendant object types.

procedure Write; virtual;
   Abstract method. This method must be overridden in descendant object types.

procedure Debug(Window: tRect; Motion: tMatrix); virtual;
   Draws each element in the mesh one at a time.

destructor Done; virtual;
   Disposes of Elements, Nodes, and finally Self.

end; {tMesh}

tMeshGenerator:

tMeshGenerator

tMeshGenerator = object(tObject)

   TMeshGenerator is the ancestor type of all mesh generators. The use of type tMeshGenerator
   makes adding new mesh generation algorithms to CSGMesh quite easy. A MeshGenerator object
   type works as follows: first, it is given a segment to mesh. Then, it is told to mesh the segment.
   The entire mesh generation algorithm is contained in the one method - CreateMesh. It then
   "gives" the mesh it created to the segment (setting the segment's Mesh pointer to the mesh), and
   disposes of itself.

Fields:
   Segment: pSegment; The segment to be meshed.
   Mesh: pMesh; The resulting mesh.

Methods:
   constructor Init(aSegment: pSegment);
      Initializes the mesh generator by setting the Segment field equal to the segment specified by
      aSegment and setting Mesh equal to nil.
procedure CreateMesh; virtual;
    Abstract method. This method must be overridden in descendant object types.
    The method CreateMesh belonging to any descendant of tMeshGenerator must contain the
total mesh generation algorithm for the appropriate segment type.

destructor Done; virtual;
    Sets the Segment and Mesh pointers equal to nil before disposing of self. This destructor does
NOT get rid of the associated segment or mesh.

end; {tMeshGenerator}

tMovement:

type tMovement = object(tCollection)

    Type tMovement is basically a collection of nodes that has been moved to align segment
meshes at the boundaries. The nodes in the collection have all been moved by the vector stored in
the Vector field in order to make the mesh continuous across a segment boundary. This is used
when the "Connect Meshes" command is given is CSGMesh. The nodes can be moved and
moved back again by calling the methods Move and UnMove. Movements are stored in the global
collection MovementList. tMovement is defined in unit Movements.
    Basically, the whole notion of movements was a quick fix to the problem of discontinuities
between segment meshes. In was hastily thrown together, and has not been thoroughly debugged.
This is one of the areas of CSGMesh where vast improvements can be made.

Fields:
    Vector: tVector;  The vector between the nodes' old positions and new positions.
    Moved: boolean;  Tells whether the nodes have been moved or not, or unmoved.

Methods:
    constructor Init(V: pVector);
        Calls tCollection.Init, sets the Vector field equal to V, sets Moved equal to false, and adds the
movement to the MovementList.

    procedure Insert(Item: Pointer); virtual;
        Adds a node (specified by Item) to the movement.

    procedure Move;
        Moves each node in the collection along the Vector and sets the field Moved equal to true.
procedure UnMove;
    Moves each node in the collection backwards along Vector and sets Moved to false.

procedure Draw(Window: tRect);
    Draws all of the nodes in the collection in red in the portion of the screen specified by Window.

destructor Done; virtual;
    Disposes of Self without deleting any of the nodes in the collection.

end; {tMovement}

tNode:

tNode = object(tObject)

Type tNode is used to represent nodes. It was copied from the work of Ajay Garg, so it does not fit in to the CSGMesh solid modeling system very well (i.e. based on XYZPoint instead of t3DPoint). It would be better to re-do type tNode to utilize the type t3DPoint, however, it is quite straightforward as it is. Type tNode is defined in unit Nodes.

Fields:
fPos: XYZExtPoint;  The position of the node.
fAtt: pAttributeList;  The node's attributes. Currently not used.

Methods:
constructor Init;
    Sets the node's position to (0,0,0).

procedure Draw(Window: tRect; Motion: tMatrix);
    Draws the node in the portion of the screen specified by Window after applying the matrix Motion, which allows nodes defined in local coordinate systems to drawn in the global coordinate system.

procedure Write(var OutFile: Text); virtual;
    Writes the node and it's location values to the file specified by OutFile in the current OutputFormat, be in ANSYS, NASTRAN, or Supersap.

procedure DeleteDuplicateNodeMarkers;
    Deletes any DuplicateNodeMarkers which refer to this (Self) node.
tNode

destructor Done; virtual;
   Calls DeleteDuplicateNodeMarkers, then disposes of Self.
end; {tNode}

tNothingSegment:

tNothingSegment

type tNothingSegment = object(tSegment)

   A NothingSegment is just that -- nothing. When two segments of different type or two un-
combineable segments are combined, the resulting segment is nothing, denoted by a
NothingSegment. TNothingSegment is defined in unit Segments.

Fields:
None

Methods:
None

end; {tNothingSegment}

tOperation:

tOperation

type tOperation = object(tSolid)

   Type tOperation is an intermediate type between tSolid and the three operation types, tUnion,
tDifference, and tIntersection. TOperation was created so that the three operation types could
inherit a good deal of their functionality from a common ancestor, and because it was easier to
type each of the methods contained here once that in would have been to type them all three
times, once for each operation type. Type tOperation is defined in unit Solids.

Fields:
Right: pSolid; The solid on the operation's right in the CSG tree -- that is, the right child.
Left: pSolid; The solid on the operation's left in the CSG tree -- that is, the left child.

Methods:
constructor Init;
   Calls tSolid.Init and sets Right and Left to nil.
procedure Add; virtual;
    Adds the Right and Left solids to the SolidList if they are not already there, and then calls tSolid.Add.

function Inspect: word; virtual;
    Creates a solid inspection dialog box for the editing of the operation's parameters.

procedure Copy(Solid: pSolid); virtual;
    Copies the solid specified by Solid.

procedure Move(direction: tVector; magnitude: real); virtual;
    Calls tSolid.Move, then invokes Right.Move and Left.Move to move Right and Left solids.

function ReadMotion(sentence: string): boolean; virtual;
    Calls tSolid.ReadMotion, then invokes Right.ReadMotion and Left.ReadMotion.

procedure SetSegType(ST: SegmentTypes); virtual;
    Calls tSolid.SetSegType and then invokes Right.SetSegType and Left.SetSegType.

procedure SetMaterial(M: integer); virtual;
    Calls tSolid.SetMaterial and then invokes Right.SetMaterial and Left.SetMaterial.

procedure SetLength(L: real); virtual;
    Invokes Right.SetLength and Left.SetLength.

procedure TurnAround; virtual;
    Invokes Right.TurnAround and Left.TurnAround.

destructor Done; virtual;
    Disposes of Right and Left, then disposes of Self by calling tSolid.Done.

function IsA(what: string): boolean; virtual;
    Sets value equal to tSolid.IsA(what), then makes value true if what equals 'OPERATION" or 'OPER'.

decl {tOperation}
**tParabolaEdge:**

```pascal
type tParabolaEdge = object(tEdge)

This type is not currently supported in CSGMesh. It was included to facilitate the inclusion of the ParabolaEdge, which can result from the intersection of a plane surface (tPlaneSurf) and a ConeSurface (tConeSurf -- not yet supported). This object type inherits real fields \( t0 \) and \( t1 \) from parent \( tEdge \), which denote the endpoints of the edge. \( tParabolaEdge \) is defined in unit ParabolaEdges.

**Fields:**
- \( p \): real; Parameter defining the parabola. For a mathematical description, see Appendix B.

**Methods:**
- None

**tPlaneSurf:**

```pascal
type tPlaneSurf = object(tSurface)

Type \( tPlaneSurf \) is used to represent planar surfaces. It descends from type \( tSurface \), from which it inherits a pointer to a list of edges called \( Edge \), a integer field called \( Outside \), and a pointer called \( Next \) used for storing surfaces in linked lists. Type \( tPlaneSurf \) has no additional fields -- it is defined as the x-y plane of the local coordinate system defined by the \( Motion \) field inherited from \( tGeometricEntity \). The length and width of the planar surface are infinite. The PlaneSurf is based on the PADL-2 "PSURF" halfspace, which is described in Appendix B. Type \( tPlaneSurf \) is defined in unit PlaneSurfaces.

**Fields:**
- None

**Methods:**
- procedure GetNormal(Point: t3DPoint; var Normal: tVector); virtual;
  Returns the surface normal at the point specified by \( Point \) in the variable \( Normal \).

- function ProfileEdges: pEdge; virtual;
  Returns a nil pointer -- a planar surface has no profile edges.

- function Intersection(Surface: pSurface): pEdge; virtual;
  Returns the edge (or edges) of intersection of Self with the surface specified by \( Surface \).
function PlaneIntersection(Surface: pPlaneSurf): pEdge; virtual;
    Returns the edge (or edges) of intersection of Self with the planar surface specified by Surface.

function CylinderIntersection(Surface: pSurface): pEdge; virtual;
    Returns the edge (or edges) of intersection of Self with the cylindrical surface specified by Surface.

function ClassifyLineEdge(E: pLineEdge): pMCR; virtual;
    Classifies the line edge E with respect to Self, determining which portions are "IN", "ON", and "OUT" of the planar surface.

function ClassifyEllipseEdge(E: pEllipseEdge): pMCR; virtual;
    Classifies the ellipse edge E with respect to Self, determining which portions are "IN", "ON", and "OUT" of the planar surface.

function ClassifyCCEdge(E: pCCEdge): pMCR; virtual;
    Classifies the CCEdge E with respect to Self, determining which portions are "IN", "ON", and "OUT" of the planar surface.

function Equals(Surface: pSurface): boolean; virtual;
    Returns true if the surface specified by Surface is equal to Self.

function Parallel(Surf: pSurface): boolean; virtual;
    Returns true if the surface specified by Surface is parallel to Self.

function Perpendicular(Surf: pSurface): boolean; virtual;
    Returns true if the surface specified by Surface is perpendicular to Self.
end; \{tPlaneSurf\}

tPlateSegment:

tPlateSegment

A plate segment is used to represent two-dimensional regions of the geometry to be meshed. This is one of three main segment types, the others being tBeamSegment for one-dimensional regions and tBrickSegment for three-dimensional regions. Each of these are descendants of tSegment, from which they inherit a pointer to a solid called Solid (which holds the portion of the CSG tree representation which makes up the segment) as well as some other fields described under type tSegment. tPlateSegment is defined in unit PlateSegments.
tPlateSegment

Fields:
Surface: pSurface; The surface representing the two-dimensional representation of the segment.
thickness: real; The thickness of the segment.

Methods:
constructor Init;
Calls tSegment.Init, then initializes a new tPlaneSurf for field Surface, and sets thickness equal to zero.

procedure Copy(D: pSegment); virtual;
Copies the segment specified by D.

procedure GetMotion(var Motion: tMatrix); virtual;
Returns the motion of the Surface, which is the representation of the segment.

function Inspect: word; virtual;
Creates a segment inspection dialog box for the editing of the segment parameters. The resulting word is equal to the command that closes the dialog box (cmOk or cmCancel), so that the procedure calling tPlateSegment.Inspect will know whether the edited values of the parameters are to be saved or not. Note: see Turbo Vision Guide for a description of dialog boxes and commands.

function ReDefine: word; virtual;
Creates a segment re-definition dialog box to allow the user to manually alter the segment representation Surface.

function SetDrawOptions: word; virtual;
Identical to tSegment.SetDrawOptions with the addition of the option of viewing the PlateSegment from face-on.

procedure ViewFaceOn; virtual;
Defines the global graphics parameters Projection and ViewUp, computes the new view point, and calls MakeTransformationMatrix (defined in unit Graphics) so that the graphics drawings of the plate segment will be from face-on.

procedure DefaultViewSettings; virtual;
Calls tSegment.DefaultViewSettings, then, if the DrawOptions include drawing from face-on, calls ViewFaceOn.

procedure Draw(Window: tRect); virtual;
Draws the segment in the portion of the screen specified by Window.
procedure DrawRep(Window: tRect); virtual;
   Draws the representation of the segment (Surface) in the portion of the screen specified by Window.

procedure DrawIcon(Window: tRect); virtual;
   Draws the segment icon in the portion of the screen specified by Window.

function CombinesWith(Segment: pSegment): boolean; virtual;
   Tests whether the segment specified by Segment can be combined with Self into a single plate segment.

procedure CreateEdges; virtual;
   Creates the edges of the Surface representation by intersecting each of the surfaces in the segment's solid with the representation Surface, and keeping the edge portions that classify as "ON" the solid.

function AddNodeAt(p: t3DPoint): pNode; virtual;
   Adds a node to the mesh at the location specified by p, returns the added node.

procedure AddGraphicsNode(p: t3DPoint); virtual;
   Adds a node at the point on the screen specified by p.

function CheckOutputFormat: word; virtual;
   Checks to see if mesh's element type is supported for current OutputFormat and AnalysisType.

procedure WriteANSYSParameters(var OutFile: Text); virtual;
   Writes the element type and parameters to the output file.

procedure WriteNASTRANParameters(var OutFile: Text); virtual;
   Writes the NASTRAN solid property card (PSOLID card) to the output file.

procedure WriteSUPERSAPParameters(var OutFile: Text); virtual;
   Writes the Supersap element control data line, material property data, distributed surface loads, element load factors, and element data lines to the output file.

destructor Done; virtual;
   Disposes of Surface, then calls tSegment.Done.

destructor Kill; virtual;
   Disposes of Surface, then calls tSegment.Kill.
end; {tPlateSegment}
tPrimitive:

type tPrimitive = object(tSolid)

  Type tPrimitive is an intermediate type between tSolid and the six primitive types, tBlock, tCylinder, tWedge, tCone, tSphere, and tTorus. TPrimitive was created so that the three operation types could inherit some of their functionality from a common ancestor. Type tPrimitive is defined in unit Solids.

Fields:
None

Methods:

function ClassifyEdge(E: pEdge): pMCR; virtual;
  Classifies the edge E against the primitive by classifying the edge against each of the primitive's surfaces. This method will work for all non-concave (that is, convex) primitives.

function IsA(what: string): boolean; virtual;
  Sets value equal to tSolid.IsA(what), then makes value true if what equals 'PRIMITIVE" or 'PRIM'.

end; {tPrimitive}

tSCEdge:

type tSCEdge = object(tEdge)

  This type is not currently supported in CSGMesh. It was included to facilitate the inclusion of the SCEdge, which can result from the intersection of a sphere surface (tSphereSurf -- not yet supported) and a cylinder surface (tCylinderSurf). This object type inherits real fields t0 and t1 from parent tEdge, which denote the endpoints of the edge. tSCEdge is defined in unit SCEdges.

Fields:
Rc, Rs, h: real; Real parameters defining SCEdge. For a mathematical description, see Appendix B.
sign: integer; Integer parameter defining SCEdge. For a mathematical description, see Appendix B.

Methods:
None

end; {tSCEdge}
tSegment:

Type tSegment is the common ancestor of all segment types -- tBeamSegment, tPlateSegment, and tBrickSegment. These segment types are used to represent different portions of the CSG tree. Thus, different parts of the structure can be thought of as unique segments, and can be meshed differently with different element types. Segments are stored in the global list SegmentList. TSegment is defined in unit Segments.

Fields:

Solid: pSolid; The portion of the CSG tree out of which the segment is made.
Next: pSegment; The next segment in the SegmentList.
Number: integer; The unique number of the segment -- equal to the segment's position in the SegmentList.
MeshDensity: real; The node spacing to be used meshing the segment.
Mesh: pMesh; A pointer to the segment's mesh.
Selected: boolean; Boolean field indicating whether segment is selected or not.
DrawOptions: word; Tells how the segment is to be drawn -- as a solid or a representation, with or without nodes and elements.

Methods:

constructor Init;
Sets Solid to nil, Next to nil, Number to zero, MeshDensity to 0.0, Mesh to nil, Selected to false, and DrawOptions to GlobalDrawOptions.

function Prev: pSegment;
Returns the previous segment in the SegmentList.

procedure Copy(S: pSegment); virtual;
Copies the Solid, MeshDensity, and DrawOptions from the segment specified by S.

procedure GetMotion(var Motion: tMatrix); virtual;
Abstract method. This method must be overridden in descendant object types.

function Inspect: word; virtual;
Abstract method. This method must be overridden in descendant object types.

function ReDefine: word; virtual;
Puts a message box on the screen that says the segment cannot be redefined. Must be overridden for segment types which can be redefined.

function SetDrawOptions: word; virtual;
Creates a Draw-options dialog box.
procedure DefaultViewSettings; virtual;
   Calls Solid.DefaultViewSettings.

procedure Draw(Window: tRect); virtual;
   Draws the segment according to the DrawOptions in the portion of the screen specified by Window.

procedure DrawSolid(Window: tRect); virtual;
   Draws the segment's Solid in the portion of the screen specified by Window.

procedure DrawRep(Window: tRect); virtual;
   Abstract method. This method must be overridden in descendant object types.

procedure DrawIcon(Window: tRect); virtual;
   Draws the segment icon in the portion of the screen specified by Window.

function Overlaps(Segment: pSegment): boolean; virtual;
   Returns true if the segment specified by Segment overlaps Self in some region of space. That is, if the two segments touch, the result is true.

function CombinesWith(Segment: pSegment): boolean; virtual;
   Abstract method. This method must be overridden in descendant object types.

function Meshable: boolean; virtual;
   Returns true.

procedure ConvertNodesToLocal;
   Calls GetMotion and applies the inverse of the resulting motion to each of the nodes in the Mesh.

function AddNodeAt(p: t3DPoint): pNode; virtual;
   Abstract method. This method must be overridden in descendant object types.

procedure AddGraphicsNode(p: t3DPoint); virtual;
   Abstract method. This method must be overridden in descendant object types.

function CheckOutputFormat: word; virtual;
   Checks to see if the type of element in the Mesh is supported for writing to output file.
procedure WriteANSYSParameters(var OutFile: Text); virtual;
    Writes a heading for an ANSYS output file to file specified by OutFile.

procedure WriteNASTRANParameters(var OutFile: Text); virtual;
    Abstract method. This method must be overridden in descendant object types.

procedure WriteSUPERSAPParameters(var OutFile: Text); virtual;
    Abstract method. This method must be overridden in descendant object types.

procedure RemoveFromList; virtual;
    Removes the segment from the SegmentList.

procedure DeleteOverlapMarkers;
    Deletes any OverlapMarkers referring to the segment.

destructor Done; virtual;
    Calls RemoveFromList and DeleteOverlapMarkers, disposes of Solid and Mesh, and calls tObject.Done.

destructor Kill; virtual;
    Calls DeleteOverlapMarkers, disposes of Next, disposes of Solid and Mesh, then calls tObject.Done.

end; {tSegment}

tSegmentListInterior:                     tSegmentListInterior

    type tSegmentListInterior = object(tScroller)

    Type tSegmentListInterior was developed as the interior of the tSegmentListWindow window. This object type draws the SegmentList and handles the event when a material name is clicked on. Type tSegmentListInterior descends from the Turbo Vision object type tScroller, and is defined in unit Segments.

    Fields:
    SegmentAt: array[1..MaxSegments] of pSegment;  An array containing the SegmentList. This array makes it easy to convert a mouse click on some line of the SegmentListWindow to the proper segment in the SegmentList.
tSegmentListInterior

Methods:
constructor Init(Bounds: tRect; aScrollBar: pScrollBar);
   Initializes the tSegmentListInterior.

procedure Draw; virtual;
   Draws the tSegmentListInterior by simply listing the segments in the SegmentList.

procedure HandleEvent(var Event: tEvent), virtual;
   If a segment name is clicked on, makes that segment the CurrentlySelectedSegment. If a segment name is double clicked on, calls that segment's Inspect method. Also, defines the following shortcut keys: "R" redefines CurrentlySelectedSegment; "I" inspects CurrentlySelectedSegment; "O" sets DrawOptions for CurrentlySelectedSegment; "T" sets segment type for CurrentlySelectedSegment; "D" draws CurrentlySelectedSegment; "M" meshes CurrentlySelectedSegment; "U" unmeshes (deletes mesh for) CurrentlySelectedSegment; and "X" deletes CurrentlySelectedSegment.

end; {tSegmentListInterior}

tSegmentListWindow:

type tSegmentListWindow = object(tWindow)

   Type tSegmentListWindow is the descendant of Turbo Vision type tWindow which makes the SegmentList window. It is defined in unit Segments.

Fields:
None

Methods:
constructor Init;
   Creates a HeadingBox and a SegmentListInterior and inserts them into the newly initialized SegmentListWindow.

end; {tSegmentListWindow}
tSingleSurface:

tSingleSurface = object(tSolid)

Type tSingleSurface is not used in the finished version of CSGMesh. It was initially created to divide segment's prior to meshing. In the current version of CSGMesh, segment meshes are merged after meshing is complete, so it was not necessary to divide segments prior to meshing.

Fields:
None

Methods:
constructor Init(Surf: pSurface);
procedure EvaluateBoundary; virtual;
function ClassifyEdge(E: pEdge): pMCR; virtual;

end; {tSingleSurface}

tSolid:

tSolid = object(tGeometricEntity)

Type tSolid is the common ancestor of all CSG solids: the primitives (tBlock, tCylinder, tWedge, tCone, tSphere, and tTorus), and the operations (tUnion, tDifference, and tIntersection). The ancestor of tSolid is tGeometricEntity, from which tSolid inherits a Motion matrix as well as several methods for dealing with this rigid motion. TSolid is one of the most important objects in all of CSGMesh -- it is this object type that defines the solid modeling representation scheme. Solids are stored in the global list SolidList. Type tSolid is defined in unit Solids.

Fields:
Name: pString; The solid's name. Each solid has a unique name.
Surface: pSurface; Pointer to the solid's Boundary Representation, which is a list of surfaces, each of which has a list of its edges.
Box: pBoundingBox; Pointer to the solid's bounding box, a box oriented parallel to the global coordinate system which encloses the entire solid.
Material: integer; Integer reference number of the material out of which the solid is made.
SegType: SegmentTypes; One of either BEAM, PLATE, BRICK, or NOTHING, which determines how the solid will be broken into segments and eventually meshed.
Parent: pSolid; A pointer to the solid's parent solid in the CSG tree.
Next: pSolid; The next solid in the SolidList.
Methods:

constructor Init;

Calls tGeometricEntity_Init to initialize Motion, sets Name, Surface, Parent, Next, and Box equal to nil, sets Material equal to 1, and sets SegType equal to NOTHING.

procedure Add; virtual;

Adds the solid to the SolidList, first checking that the solid's Name is unique.

procedure Copy(Solid: pSolid); virtual;

Copies the Motion, Surface list, and SegType from the solid specified by Solid. Makes the Name equal to Solid's name plus the apostrophe character, for example, the solid copied from a solid named "Henry" would be named "Henry'". This ensures that each solid's name is unique.

function Prev: pSolid; virtual;

Returns the previous solid in the SolidList.

procedure Rename; virtual;

Abstract method. This method must be overridden in descendant object types.

function IsA(what: string): boolean; virtual;

This was meant as an easy method of determining what type of solid a solid is. The solid checks the string what, and if it is equal to "SOLID" or "SOL", a true value is returned, otherwise a false value is returned. Each descendant of tSolid has its own version of this method, and each solid first checks its parent's IsA method, then checks for its individual identity.

procedure DefaultViewSettings;

Computes the appropriate view settings to make the solid just fit in the graphics window.

procedure Draw(Window: tRect); virtual;

Draws the solid in the portion of the screen specified by Window. This method draws the solid by drawing each edge of each surface, which is not very efficient. It is suggested that this method be overridden in descendant objects (especially primitives) to make it more efficient.

procedure Erase(Window: tRect); virtual;

Sets the drawing color to black and draws the solid.

procedure Move(direction: tVector; magnitude: real); virtual;

Calls tGeometricEntity_Move, then moves each of the surfaces in the Surface list.

function ReadValues(sentence: string): boolean; virtual;

Abstract method. This method must be overridden in descendant object types.
function ReadMotion(sentence: string): boolean; virtual;
   Reads a rigid motion from a line of a .CSG input file, for example "MOVX=1.5". For more
   on the format of the .CSG file, see Appendix E.

procedure PrintInfo; virtual;
   This procedure was originally written for debugging, before there was any way of drawing or
   inspecting the cylinder. Subsequently, this method is never used, but still remains as sort of a
   vestigial method.

function Inspect: word; virtual;
   Abstract method. This method must be overridden in descendant object types.

procedure AddSurface(NewSurface: pSurface);
   Adds the surface specified by NewSurface to the end of the Surface list.

procedure DeleteSurface(DeadSurface: pSurface);
   Removes the surface specified by DeadSurface from the Surface list.

procedure ComputeBoundingBox; virtual;
   Adds the surface specified by NewSurface to the end of the Surface list.

procedure Complement; virtual;
   Complements each surface in the Surface list.

procedure EvaluateSurfaces; virtual;
   Evaluates the boundary of the solid, the deletes all of the edges.

procedure EvaluateBoundary; virtual;
   Abstract method. This method must be overridden in descendant object types.

procedure CountSurfaces(var nPl, nCy, nCo, nSp, nTo: integer);
   Counts all of the surfaces in the Surface list by type.

function ClassifyEdge(E: pEdge): pMCR; virtual;
   Abstract method. This method must be overridden in descendant object types.

procedure DefaultSegType; virtual;
   Abstract method. This method must be overridden in descendant object types.

procedure SetSegType(ST: SegmentTypes); virtual;
   Sets the field SegType equal to the value of ST.
procedure SetMaterial(M: integer); virtual;
    Sets the Material field equal to the value of M.

procedure SetLength(L: real); virtual;
    Abstract method. This method must be overridden in descendant object types.

procedure TurnAround; virtual;
    Abstract method. This method must be overridden in descendant object types.

procedure Debug(Window: tRect); virtual;
    Draws the solid surface by surface and edge by edge.

procedure RemoveFromList; virtual;
    Removes the solid from the SolidList.

destructor Done; virtual;
    Calls RemoveFromList, disposes of Name, Surface, and Box, and calls tGeometricEntity.Done.

destructor Kill;
    Disposes of Next, disposes of Name, Surface, and Box, and calls tGeometricEntity.Done.

end; {tSolid}

tSolidListInterior:

  type tSolidListInterior = object(tScroller)

  Type tSolidListInterior was developed as the interior of the tSolidListWindow window. This object type draws the SolidList and handles the event when a material name is clicked on. Type tSolidListInterior descends from the Turbo Vision object type tScroller, and is defined in unit Solids.

  Fields:
  SolidAt: array[1..MaxSolids] of pSolid;
      An array containing the SolidList. This array makes it easy to convert a mouse click on some line of the SolidListWindow to the proper Solid in the SolidList.
tSolidListInterior

Methods:
constructor Init(Bounds: tRect; aScrollBar: pScrollBar);
   Initializes the tSolidListInterior.

procedure Draw; virtual;
   Draws the tSolidListInterior by simply listing the solids in the SolidList.

procedure HandleEvent(var Event: tEvent); virtual;
   If a solid name is clicked on, makes that solid the CurrentlySelectedSolid. If a solid name is
double clicked on, calls that Solid's Inspect method.

end; {tSolidListInterior}

tSolidListWindow:

type tSolidListWindow = object(tWindow)

   Type tSolidListWindow is the descendant of Turbo Vision type tWindow which makes the
SolidList window. It is defined in unit Solids

Fields:
   None

Methods:
constructor Init;
   Creates a HeadingBox and a SolidListInterior and inserts them into the newly initialized
SolidListWindow.

end; {tSolidListWindow}

tSphere:

type tSphere = object(tPrimitive)

   The sphere primitive is not yet supported in CSGMesh. This object type has been included to
facilitate the inclusion of the sphere primitive in the future. Type tSphere can be found in unit
Spheres.
tSphere

Fields:
r: real; The radius of the sphere.

Methods:
None

end; {tSphere}

tSphereSurf:

type tSphereSurf = object(tSurface)

Type tSphereSurf is not yet supported in CSGMesh. This object type has been included to facilitate the inclusion of the sphere primitive and its related surface type in the future. Type tSphereSurf can be found in unit SphereSurfaces.

Fields:
r: real; The radius of the spherical surface.

Methods:
None

end; {tSphereSurf}

tSurface:

type tSurface = object(tGeometricEntity)

Type tSurface is the common ancestor of all surface types. It's Edge field holds a list of the surface's edges. The Outside field is used to tell which side of the surface the stuff is on, so to speak. For example, a cylindrical surface can either be the boundary of a rod, or the boundary of a hole. The Outside field is used to tell which is the case. The Next field is used for storing surfaces in linked lists. TSurface inherits a Motion and methods for dealing with this motion from ancestor tGeometricEntity. TSurface is defined in unit Surfaces.
tSurface

Fields:
Edge: pEdge; Pointer to a linked list of the surface's edges.
OutSide: integer; Has a value of +1 or -1 to tell which side of the surface the "stuff" is on. A value of +1 indicates that surface normal points in the standard, outward direction (away from the interior of the solid), and a value of -1 indicates that the surface normal points inward (toward the "stuff").
Next: pSurface; A pointer to the next surface in the linked list.

Methods:
constructor Init;
    Calls tGeometricEntity.Init, sets Edge and Next equal to nil, and sets Outside equal to +1.

procedure GetNormal(Point: t3DPoint; var Normal: tVector); virtual;
    Abstract method. Must be overridden in descendant object types.

function BoundingBox: pBoundingBox; virtual;
    Computes the bounding box by taking the union of the bounding boxes of all of the edges.

procedure Draw(Window: tRect); virtual;
    Draws the surface in the portion of the screen specified by Window by drawing all of the edges in the Edge list.

procedure AddEdge(NewEdge: pEdge);
    Adds the edge specified by NewEdge at the end of the Edge list.

procedure RemoveEdge(GoneEdge: pEdge);
    Removes the edge specified by GoneEdge from the Edge list without disposing of the edge.

procedure DeleteEdge(DeadEdge: pEdge);
    Removes the edge specified by DeadEdge from the Edge list and disposes of it.

procedure CountEdges(var nLi, nEl, nCC, nPa, nHy, nSC, nKK, nTP, Total: integer);
    Counts all of the edges in the Edge list by type.

function ValidEdges: boolean; virtual;
    Does some simple tests to make sure the edges in the Edge list make a valid surface. There is a great deal of room for improvement in this method.

procedure RemoveDuplicateEdges;
    Removes any duplicate edges from the Edge list.
procedure CombineBrokenEdges;
    Goes through the \textit{Edge} list to find any edges that start where another edge ends. If two such edges are found, they are combined into a single edge.

function ProfileEdges: pEdge; virtual;
    Abstract method. Must be overridden in descendant object types.

function Intersection(Surface: pSurface): pEdge; virtual;
    Abstract method. Must be overridden in descendant object types.

function ClassifyEdge(E: pEdge): pMCR; virtual;
    Calls the appropriate method based on the type of edge \textit{E}. For example, if \textit{E} were a \textit{tLineEdge}, this method would call \textit{ClassifyLineEdge}. If \textit{E} were a \textit{tEllipseEdge}, it would call \textit{ClassifyEllipseEdge}. And so on.

function ClassifyLineEdge(E: pLineEdge): pMCR; virtual;
    Abstract method. Must be overridden in descendant object types.

function ClassifyEllipseEdge(E: pEllipseEdge): pMCR; virtual;
    Abstract method. Must be overridden in descendant object types.

function ClassifyCCEdge(E: pCCEdge): pMCR; virtual;
    Abstract method. Must be overridden in descendant object types.

function ClassifyParabolaEdge(E: pParabolaEdge): pMCR; virtual;
    Abstract method. Must be overridden in descendant object types.

function ClassifyHyperbolaEdge(E: pHyperbolaEdge): pMCR; virtual;
    Abstract method. Must be overridden in descendant object types.

function ClassifySCEdge(E: pSCEdge): pMCR; virtual;
    Abstract method. Must be overridden in descendant object types.

function ClassifyKKEdge(E: pKKEdge): pMCR; virtual;
    Abstract method. Must be overridden in descendant object types.

function ClassifyTorusProfileEdge(E: pTorusProfileEdge): pMCR; virtual;
    Abstract method. Must be overridden in descendant object types.

function Equals(Surface: pSurface): boolean; virtual;
    Abstract method. Must be overridden in descendant object types.
function Parallel(Surf: pSurface): boolean; virtual;
   Abstract method. Must be overridden in descendant object types.

function Perpendicular(Surf: pSurface): boolean; virtual;
   Abstract method. Must be overridden in descendant object types.

procedure Copy(Surface: pSurface); virtual;
   Copies the Motion, Edge list, and Outside from the surface specified by Surface.

procedure Move(direction: tVector; magnitude: real); virtual;
   Calls tGeometricEntity.Move, then moves each of the edges in the Edge list.

procedure Complement; virtual;
   Sets Outside equal to -Outside, and complements each of the edges in the Edge list.

destructor Done; virtual;
   Disposes of Next, disposes of Edge, and calls tGeometricEntity.Done.

tTetrahedron4:

tTetrahedron4

type tTetrahedron4 = object(tElement)

   Type tTetrahedron4 represents a four-noded tetrahedron element. These elements are not
generated directly from any of the current mesh generators in CSGMesh, but can be created by
breaking 6-node wedge elements into tetrahedrons. Type tTetrahedron4 is defined in unit
TetrahedronElements.

Fields:
N1, N2, N3, N4: pNode;   The four nodes of the tetrahedron element.

Methods:
constructor Init;
   Calls tElement.Init and sets the four nodes to nil.

function Description: string; virtual;
   Returns the string "4-Node Tetrahedron Element"

procedure Draw(Window: tRect; Motion: tMatrix); virtual;
   Draws the Tetrahedron element in the portion of the screen specified by Window.
procedure DrawNodes(Window: tRect; Motion: tMatrix), virtual;
   Draws the four nodes in the portion of the screen specified by Window.

function ANSYSTYPEString: string, virtual;
   Returns the string "SOLID45", which is the ANSYS four-node structural tetrahedron element.

function Supported: boolean, virtual;
   Returns true or false depending on whether tetrahedron elements are supported for the current OutputFormat and AnalysisType.

procedure write(var OutFile: Text), virtual;
   Writes the element to the output file specified by OutFile in the current OutputFormat.

end; {tTetrahedron4}

tTetrahedron10:

type tTetrahedron10 = object(tTetrahedron4)

Type tTetrahedron10 represents a ten-noded tetrahedron element. These elements are not generated directly from any of the current mesh generators in CSGMesh, but can be created by breaking 6-node wedge elements into tetrahedrons. Type tTetrahedron10 is defined in unit TetrahedronElements.

Fields:
{N1, N2, N3, N4,} N5, N6, N7, N8, N9, N10: pNode;  Nodes 1-4 are inherited from ancestor tTetrahedron4. Nodes 5-10 are defined here.

Methods:
constructor Init;
   Calls tElement.Init and sets the ten nodes to nil.

function Description: string, virtual;
   Returns the string "10-Node Tetrahedron Element"

procedure DrawNodes(Window: tRect; Motion: tMatrix), virtual;
   Draws the ten nodes in the portion of the screen specified by Window.
function ANSYSTypeString: string; virtual;
    Returns the string "SOLID92", which is the ANSYS ten-node structural tetrahedron element.

function Supported: boolean; virtual;
    Returns true or false depending on whether tetrahedron elements are supported for the current OutputFormat and AnalysisType.

procedure write(var OutFile: Text); virtual;
    Writes the element to the output file specified by OutFile in the current OutputFormat.

end; {tTetrahedron10}

tTesisApplication:

tTesisApplication

tTesisApplication

tTesisApplication

tTesisApplication

tTesisApplication

function ANSYSTypeString: string; virtual;
    Returns the string "SOLID92", which is the ANSYS ten-node structural tetrahedron element.

function Supported: boolean; virtual;
    Returns true or false depending on whether tetrahedron elements are supported for the current OutputFormat and AnalysisType.

procedure write(var OutFile: Text); virtual;
    Writes the element to the output file specified by OutFile in the current OutputFormat.

end; {tTetrahedron10}

tTesisApplication:

tTesisApplication

tTesisApplication

tTesisApplication

tTesisApplication

The tTesisApplication object is the "application object." Basically, this object type represents the CSGMesh computer program (application). For a complete description and explanation of application objects, see the Turbo Pascal Turbo Vision Guide [Reference 66].

Fields:
Clock: pClockView;       The clock in the upper right hand corner.
Heap: pHeapView;          The available heap space display in the lower right hand corner.

Methods:
constructor Init;
    Initializes the application object.

destructor Done; virtual;
    Disposes of the application object.

procedure Idle; virtual;
    This method is called repeatedly while the application object is waiting for an event (a mouse click or a key press).

procedure HandleEvent(var Event: TEvent); virtual;
    This method tells the application what to do when a mouse click or key-press in encountered.

procedure InitMenuBar; virtual;
    Describes the menu bar at the top of the screen.
procedure InitStatusLine; virtual;
   Describes the status line at the bottom of the screen.

procedure OutOfMemory; virtual;
   Tells the application what to do when the heap is used up.

end; {tThesisApplication}

tTorus:

type tTorus = object(tPrimitive)

   The torus primitive is not yet supported in CSGMesh. This object type has been included to facilitate the inclusion of the torus primitive in the future. Type tTorus can be found in unit Torii.

Fields:
Rmin: real;   The minor radius of the torus.
Rmaj: real;   The major radius of the torus.

Methods:
None

end; {tTorus}

tTorusProfileEdge:

type tTorusProfileEdge = object(tEdge)

   This type is not currently supported in CSGMesh. It was included to facilitate the inclusion of the TorusProfileEdge. This object type inherits real fields t0 and t1 from parent tEdge, which denote the endpoints of the edge. tTorusProfileEdge is defined in unit TorusProfileEdges.

Fields:
a, b, c, d, e, Rmaj, Rmin: real;   Real parameters defining the torus profile edge. For a mathematical description, see Appendix B.
sign: integer;   Integer parameter defining the torus profile edge. For a mathematical description, see Appendix B.
Methods:
None
end; {tTorusProfileEdge}

tTorusSurf:
tTorusSurf

type tTorusSurf = object(tSurface)

Type tTorusSurf is not yet supported in CSGMesh. This object type has been included to facilitate the inclusion of the torus primitive and its related surface type in the future. Type tTorusSurf can be found in unit TorusSurfaces.

Fields:
Rmin: real; The minor radius of the torus surface.
Rmaj: real; The major radius of the torus surface.

Methods:
None
end; {tTorusSurf}

tTriangle3:
tTriangle3

type tTriangle3 = object(tElement)

Type tTriangle3 represents a three-noded triangle element. These elements are generated by the AjayMeshGenerator object. They are currently the standard two-dimensional element in CSGMesh. Type tTriangle3 was copied from the work of Ajay Garg, so it contains a lot of extraneous stuff not needed in CSGMesh. Type tTriangle3 is defined in unit TriangleElements.

Fields:
N1, N2, N3: pNode; The three nodes of the triangle elements.
E12, E23, E13: boolean; Edge flags used for computing bounding polygons.
fCircumCenter: XYZExtPoint; The circumcenter of the triangle.
fSquaredRadius: real; The value of the radius of the triangle's circum-circle squared.
Methods:
constructor Init;
  Calls tElement.Init, sets the three nodes to nil, sets the three edge flags to true, and sets the CircumCenter to (0,0,0) and the SquaredRadius to 0.

function Description: string; virtual;
  Returns the string "3-Node Triangle Element"

procedure Draw(Window: tRect; Motion: tMatrix); virtual;
  Draws the Triangle element in the portion of the screen specified by Window.

procedure DrawNodes(Window: tRect; Motion: tMatrix); virtual;
  Draws the three nodes in the portion of the screen specified by Window.

procedure DrawCircumCircle(Window: tRect; Motion: tMatrix);
  Draws the element's CircumCircle in the portion of the screen specified by Window. This method is never used in the finished version of CSGMesh. It was developed for debugging.

function ANSYSTypeString: string; virtual;
  Returns either the string "SHELL63" or the string "PLANE42", which are the ANSYS three-node structural triangle elements for plate and plane stress elements.

function Supported: boolean; virtual;
  Returns true or false depending on whether triangle elements are supported for the current OutputFormat and AnalysisType.

procedure write(var OutFile: Text); virtual;
  Writes the element to the output file specified by OutFile in the current OutputFormat.

destructor Done; virtual;
  Disposes of the triangle element without disposing of the three nodes.

end; {tTriangle3}

tTriangle6:

type tTriangle6 = object(tElement)

  Type tTriangle6 represents a six-noded triangle element. These elements are not generated directly from any of the current mesh generators in CSGMesh, but can be created by changing 3-
node triangle elements into 6-node triangle elements. Type tTriangle6 is defined in unit TriangleElements.

**Fields:**
N1, N2, N3, N4, N5, N6: pNode; The six nodes of the triangle element.

**Methods:**

- **constructor Init;**
  - Calls tElement.Init, and sets the six nodes to nil.

- **function Description: string; virtual;**
  - Returns the string "6-Node Triangle Element"

- **procedure Draw(Window: tRect; Motion: tMatrix); virtual;**
  - Draws the Triangle element in the portion of the screen specified by Window.

- **procedure DrawNodes(Window: tRect; Motion: tMatrix); virtual;**
  - Draws the six nodes in the portion of the screen specified by Window.

- **function ANSYSTypeString: string; virtual;**
  - Returns either the string "SHELL93" or the string "PLANE2", which are the ANSYS six-node structural triangle elements for plate and plane stress elements.

- **function Supported: boolean; virtual;**
  - Returns true or false depending on whether triangle elements are supported for the current OutputFormat and AnalysisType.

- **procedure write(var OutFile: Text); virtual;**
  - Writes the element to the output file specified by OutFile in the current OutputFormat.

- **destructor Done; virtual;**
  - Disposes of the triangle element without disposing of the six nodes.

end; {tTriangle6}

tUnion:

tUnion

type tUnion = object(tOperation)

This is the Union operation, one of the three boolean set operations available for creating complex geometries from primitive solids (the other two are Difference and Intersection). Type
tUnion
descends from tOperation, which in turn descends from type tSolid, from which
tUnion inherits a multitude of fields and methods. From tOperation, tUnion inherits two pointers
to solids, Left and Right, which hold the solids on which the union operation acts. TUnion is
defined in unit Unions.

Fields:
None

Methods:
procedure Rename; virtual;
    Creates a new name for the Intersection (The Name field is a pointer to a string inherited from
tSolid). This name is equal to the Left solid's name + "+" + the Right solid's name. Thus, if the
Left solid were named "L" and the Right solid were named "R", the new name of the Difference
would be "L+R".

procedure PrintInfo; virtual;
    This procedure was originally written for debugging, before there was any way of drawing or
inspecting the intersection. Subsequently, this method is never used, but still remains as sort of a
vestigial method.

procedure ComputeBoundingBox; virtual;
    Computes the bounding box of the union (equal to the union of the bounding box of the Left
solid and the bounding box of the Right solid), and stores it with pointer field Box (inherited from
tSolid). Bounding boxes greatly simplify set membership classification.

procedure EvaluateBoundary; virtual;
    Evaluates the boundary of the union (surfaces and edges), and stores them in the Surface
pointer (inherited from type tSolid).

function ClassifyEdge(E: pEdge): pMCR; virtual;
    Classifies an edge with respect to the union, determining which portions of the edge are "IN",
"ON", and "OUT" of the union solid. This is done by classifying the edge with respect to the Left
and Right solids, and combining the resulting MCRs into a single MCR.

procedure DefaultSegType; virtual;
    Computes the default segment type based on the Left and Right solids' segment types.

function IsA(what: string): boolean; virtual;
    Sets value equal to tOperation.IsA(what), then makes value true if what equals 'UNION' or
'UN'

end; {tUnion}
**tVector:**

```pascal
type tVector = object(t3DPoint)

Type tVector is used to represent vectors. It is very similar to its ancestor, t3DPoint, from which it inherits three real fields, x, y, and z. These three fields are used to represent the head end of the vector. It is assumed that the vector begins at the origin.

**Fields:**
None

**Methods:**

function Length: real;
Returns the length of the vector \(\sqrt{x^2 + y^2 + z^2}\).

procedure Normalize;
Normalizes the vector (so that Length = 1).

procedure Orthogonalize(var u, v: tVector);
Alters the argument vectors \(u\) and \(v\) so that they form a right-handed ortho-normal triple with \(Self\) being the \(w\) vector.

procedure ScalarMultiply(Scalar: real);
Multiplies \(x\), \(y\), and \(z\) by \(Scalar\).

procedure Sum(Vector1, Vector2: t3DPoint);
Sets the value of \(Self\) to the vector sum of \(Vector1\) and \(Vector2\).

procedure Diff(Vector1, Vector2: t3DPoint);
Sets the value of \(Self\) to the vector difference of \(Vector1\) minus \(Vector2\).

function Dot(Vector: t3DPoint): real;
Returns the dot product of \(Self\) and \(Vector\).

procedure Cross(Vector1, Vector2: t3DPoint);
Sets the value of \(Self\) equal to the cross-product of \(Vector1\) and \(Vector2\).

function Angle(Vector: tVector): real;
Returns the angle between \(Self\) and \(Vector\).

function Parallel(Vector: t3DPoint): boolean;
Returns true if \(Self\) is parallel to \(Vector\) and pointed in the same direction, otherwise false.

function AntiParallel(Vector: t3DPoint): boolean;
Returns true if \(Self\) is parallel to \(Vector\) and pointed in the opposite direction, otherwise false.
function Perpendicular(Vector: t3DPoint): boolean;
    Returns true if Self is perpendicular to Vector.
end; {tVector}

tWedge: tWedge

type tWedge = object(tPrimitive)

    The wedge primitive is not yet supported in CSGMesh. This object type has been included to facilitate the inclusion of the wedge primitive in the future. Type tWedge can be found in unit Wedges

Fields:
    x, y, z: real; The dimensions of the Wedge.

Methods:
    None

end; {tWedge}

tWedgeElement: tWedgeElement

type tWedgeElement = object(tElement)

    Type tWedgeElement represents a six-noded wedge element. These elements are generated by the BrickGenerator object. They are currently the standard three-dimensional element in CSGMesh. Type tWedgeElement is defined in unit WedgeElements.

Fields:
    N1, N2, N3, N4, N5, N6: pNode; The six nodes of the wedge element.

Methods:
    constructor Init;
        Calls tElement.Init, and sets the six nodes to nil.

    function Description: string; virtual;
        Returns the string "6-Node Wedge Element".
procedure Draw(Window: tRect; Motion: tMatrix); virtual;
    Draws the Triangle element in the portion of the screen specified by Window.

procedure DrawNodes(Window: tRect; Motion: tMatrix); virtual;
    Draws the six nodes in the portion of the screen specified by Window.

function ANSYSTypeString: string; virtual;
    Returns either the string "SOLID45", which is the ANSYS eight-node structural brick element, but that can be used as a six-node wedge.

function Supported: boolean; virtual;
    Returns true or false depending on whether wedge elements are supported for the current OutputFormat and AnalysisType.

procedure write(var OutFile: Text); virtual;
    Writes the element to the output file specified by OutFile in the current OutputFormat.

end; {tWedgeElement}

tWedgeMesh: tWedgeMesh

type tWedgeMesh = object(tMesh)

    tWedgeMesh is the mesh object corresponding to the mesh generator tBrickGenerator. It is a descendant of type tMesh, and is defined in unit WedgeMesh.

Fields:
Layer: pCollection; A collection of collections. Each collection is a collection of nodes on a layer of the extruded mesh.

Methods:
constructor Init;
    Calls tMesh.Init and initializes a new Layer collection.

function TypeString: string; virtual;
    Returns the string "Extruded 2-D Mesh"

procedure Write; virtual;
    Writes the mesh to the output file.
tWedgeMesh

destructor Done; virtual;
    Disposes of the mesh.

end; {tWedgeMesh}
Appendix E
CSGMesh Input (.CSG) File Syntax Diagrams

To read a syntax diagram, follow the arrows. Alternative paths are often possible; paths that begin at the left and end with an arrow on the right are valid. A path traverses boxes that hold the names of elements used to construct that portion of the syntax. The names in italics stand for constructions. Names in regular type -- reserved words, operators, and punctuation -- are the terms to be used in the input file. A dot on a line indicates a space.

.CSG files consist of a series of .CSG Input Lines. The syntax diagram for a .CSG Input Line is given immediately below, after which it is dissected into its constituent parts.
union


difference

intersection

block parameters


cylinder parameters

rigid motion

\[ \text{solid} \rightarrow \text{UN} \rightarrow \text{solid} \]

\[ \text{solid} \rightarrow \text{DIF} \rightarrow \text{solid} \]

\[ \text{solid} \rightarrow \text{INT} \rightarrow \text{solid} \]

\[ \begin{array}{c}
X= \\
Y= \\
Z=
\end{array} \]

\[ \begin{array}{c}
R= \\
H= \\
D=
\end{array} \]

\[ \text{X}, \text{Y}, \text{Z} \rightarrow \text{MOVEDBY} \]

\[ \begin{array}{c}
X= \\
Y= \\
Z=
\end{array} \]

\[ \begin{array}{c}
\text{MOV} \\
\text{ROT} \\
\text{DEG}
\end{array} \]
Appendix F
Algor Supersap Output File

Output file generated for figure 5.19, page 80 (see section 5.1.1.6. Output File).
<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>178</td>
<td>44</td>
<td>63</td>
<td>43</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.5000000</td>
<td>0.0</td>
</tr>
<tr>
<td>179</td>
<td>71</td>
<td>44</td>
<td>45</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.5000000</td>
<td>0.0</td>
</tr>
<tr>
<td>180</td>
<td>45</td>
<td>39</td>
<td>78</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.5000000</td>
<td>0.0</td>
</tr>
<tr>
<td>181</td>
<td>71</td>
<td>45</td>
<td>78</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.5000000</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.3</td>
<td>0.3</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1.0000000</td>
<td>0.0</td>
<td>0.0</td>
<td>0.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix G
ANSYS version 5.0 Output File

Output file generated for figure 5.19, page 80 (see section 5.1.1.6. Output File).
% ELEMENTS

!------------------------- ELEMENTS ------------------------!

Type, 1

Mat. 1! steel
E, 52, 49, 53, 53,
E, 53, 49, 50, 50,
E, 53, 50, 54, 54,
Appendix H
NASTRAN Output File

Output file generated for figure 5.19, page 80 (see section 5.1.1.6. Output File).
<table>
<thead>
<tr>
<th>N</th>
<th>Description</th>
<th>Grid Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Segment 1. SURFACE</td>
<td></td>
</tr>
</tbody>
</table>

**GRID (Node Definition) Cards:**

<table>
<thead>
<tr>
<th>N</th>
<th>Description</th>
<th>Grid Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SURFACE</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>D3Mesh</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Mesh of SURFACE</td>
<td>prepared by D3Mesh</td>
</tr>
<tr>
<td>4</td>
<td>in Saturday, 9/15/1985</td>
<td></td>
</tr>
</tbody>
</table>

**Material Definition Cards:**

<table>
<thead>
<tr>
<th>N</th>
<th>Description</th>
<th>Grid Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MAT,L 3.00000000000000E+007</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3.00000000000000E+007</td>
<td></td>
</tr>
</tbody>
</table>

**Parameter Definition Cards:**

<table>
<thead>
<tr>
<th>N</th>
<th>Description</th>
<th>Grid Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**GRID (Node Definition) Cards:**

<table>
<thead>
<tr>
<th>N</th>
<th>Description</th>
<th>Grid Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GRID

60, 5

500167?6057473E+0000

2

5C0065605

61,0

9. 377644-6730544E-0001
4 9664 67 060967 3E+0000

2

GRID

1

-1

868 32 8 57 001163E*5000

2

55000000660556E-

3 2 31

GRID

62,0

1.4 9664 67 060 967 3E+0000

-1

323035-1301156E*6655

2

56000005668000E-

2qn

GRID

63,0

1.49664 67 060 967 3E+0000

-3

64,0

1.49664 67 060967 3E+0000

5

47142356553669E-3051
35535714 395094E-CO61

2

GRID

9

58000000000066E-

; 651

GRID
GRID
GRID
GRID
3RID

65,5

1.4 9664 67 060967 3E*0000

1

07642S57199759EJ-0005

2

55650565566555E-

5 551

66,6

1

50060600655865E-

GRID
GRID
GRID

7 2,0

1.

GRID

73,0

.

55732142993-

36555E-

50000000566555E.

5 55 1

,

3,
3,

5',

,

i

5 3 51

49664670609673E+06 30
1.49664670669673E+0008

r

3-E-C 55;

2

67, 5

2

533214 29 5 3 3986E-5006

2

56600006008560E-

3 3 5' 1

,

63,0

1.49664

67 860967

3E+0008

2

2

3 35 1

,

69

5

1. 995528 9414610 9E+0C00

-1

51313714306175E*:555
96531196515662E*3365

50CC00000CC000E-

2

50600006600565E-

56 3 1

5

7

5,5

47205089055751E+6055

2

500000C0008000E-

0061

2

1

1. 395528941 4610 9E +0008
1. 995528941461 3 9E+0C00

-1

7

-9

837399150C8391E-5551

2

56560600550565E-

655 1

39552994146159E+0000

9

964 23153166688E-5561

2

50006805555C55E-

655 1

1. 9955289414610 9E+0C0C
1. 9955289414610 9E +0005

1

43731736487593E+6000

2

50CCOC000000

33E-

5 501

I

9382115-858334E*3500

2

500003S6S0000CE-

3 551

2

50000000000060E-

5 651

,

-r

2

50000800000000E-

5561

t

5,

,

5,

,

,

GRID

74,3
GRID 7 5,5'

.

1. 9955289414610 9E+0000

2

49910578829076E+1000

j

55 1

76,0

2.49441117682727E+0000

-1

GRID

7 7,0

2. 49441117682727 E+0000

-1

45989806927537E+5000

2

50000006000000E-

GRID

78,0

2.4 94 41117 6827 27E+0000

-9

39850227410716E-0001

2

50000000600000E-

3061

GRID

7 9,6

2.49441117682727E+0OO0

9

1908438306E-

2

50000000000000E-

5001

D

80,5

2.4 94 41117 6327 2-E+0000

45188143132373E+0000

2

50000000000000E-

5661

,

GRID

81,0

96792095421915E+0000

2

5000000000000CE-

6001

,

GRID

32,0

2.49441117682727E+0000
2.49441117682727E+0000

2

48396047715776E+0000

2

GRID

83,0

2. 9932 934 121 934 5E+0000

-1

955317 41 90004 9E+0000

C-

GRID

,84,0

2. 9 932 934 1 21 93 4 5E+ 0000

-1

4597 85 67 710154E+0000

GRID

85,0

2. 9932934121934 5E*0000

-9

GRID

86,0

2. 9 932 934121934 5E+0000

GRID

87,0

2. 9932 934 121 934 5E+0008

GRID

e8,o

jRI

J-

3584

8001

f

3,

GRID

97994591114194E+0000

'-

6 651

,

-

j,

5001
5000C008000500E-

3501

2

50000000000000E-

3001

6425393520075 9E-0001

2

50000000000000E-

5001

-4

68722193299982E-5001

2

500000000000CCE-

5501

,

5

,

5

22341290500663E-0001

9

50000000000060E-

6061

,

5

,

5

,

2. 9 932 934121934 5E+0000

1

0178730324 02 3 5E +0000

2

50000000000000E50008600000000E-

3801

50000000000000E-

3801

5661

3
,

5,

GRID

89,0

2. 9 932 93412193 4 5E+0000

1

5134047743C131E+0000

9

GRID

90, 0

2.9932 934121934 5E+0000

2

08893651620208E+0000

2

GRID

91,0

2. 9932 934 121 934 5E+0000

2

50446825810286E+0000

2

500000000000GOE-

6 661

,

3

,

GRID

92,0

3.4 9217 564 7 55600E+0000

-1

81335163127369E+0000

2

50000000000000E-

0001

,

5

,

GRID

93,0

3.4 9217 564 7 55600E+0000

-1

33201646814632E+0000

2

50000000000000E-

0001

,0,

50000000000000E-

0051

,

0001

,0,

0001

,0,

0001

,0,

GRID

,94,0

GRID

,

,

95,0

GRID , 96,0

3.49217564755600E+0000

-8

50681305018952E-0001

2

3.4 9217 564 7 55600E+0000

-3

69346141891583E-0001

2

50000000000000E50000000000000E-

,

p

3,

3.4 9217 564 7 55600E+0000
3.4 9217 564 7 55600E+0000

1

11989021235786E-0001

2

GRID , 97,0

5

93324184363155E-0001

2

GRID

3.4 9217 564 7 55600E+0000

1

07 465934749052E+0000

2

50000000000080E-

0001

,0,

3.49217564755600E+0000

1

55599451061789E+0000

1

50000000000000E-

0001

,0,

98,0

GRID , 99,0

,

2.03732967374526E+000 3,
2.51866483687263E+000 3,

2.50000000000000E -0001,0,

3,
3,
L,
3,
L,
L,
3,
3,
L,
3,
L,
3,

2.5O0OO0OO0OO0OOE -0001,0,

GRID 100, 3,
GRID ,101, 3,

3.49217564755600E+000 3,
3.49217564755600E+000 3,

GRID

3. 99105788292218E+000

3,"

3.99105788292218E+000

3,"

L. 00534 036560930E + 000

3. 99105788292218E+000

3,-

5.04672819908592E-000

3.99105788292218E+000

3,"

1.00527420788421E-000

102, 3,
GRID 103, 5,
GRID ,104, 3,
GRID 105, 3,
GRID 106, 3,
GRID 107, 3,
GRID 108, 3,
GRID 109, 3,
GRID 110, 3,
GRID 111, 3,
GRID 112, 3,
GRID 113, 3,

3.99105788292218E+000

3,
3. 99105788292218E+000 3,
3.99105788292218E+000 3,

L.50600791131001E+000

1. 96662271 4 92823E-000

3.97329817193531E-000
L.4 97 997 3628 9424E+000

4.48994011828472E+000

3,"

L.00888019895137E+000

4.4899401182S472E+000

3,-

5.04440099475687E-000

3,
4.48994011828472E+000 3,
4.48994011828472E+000 3,
4.48994011828472E+000

3.00000000000000E+000
5.04440099475687E-000
L. 0088801 98 95137E+000

$ Element Definition Cards:
#1:

$ Segment

SURFACE

CTRIA3, 1,.L, 52, 49, 53,
CTRIA3,2, L, 53, 49, 50,
CTRIA3,3,JL, 53, 50, 54,
CTRIA3,4,]L,54,50,51,
CTRIA3,5,]L, 54, 51, 55,
CTRIA3,6,]L, 55, 51, 56,
CTRIA3,7,]L, 57, 46, 58,
CTRIA3,8,]

.,58,46,47,

CTRIA3,9,1

.,58,47,59,

CTRIA3,10, 1,59,47,48,
CTRIA3,11 1,60,59,48,
CTRIA3,12 1,61,52,62,
CTRIA3,13 1,62,52,53,
CTRIA3,14 1,62,53,63,
CTRIA3,15 1,63,53,54,
CTRIA3,16 1,64,56,57,
ctri;\3,17 1,64,57,65,

H-2

2.50000000000000E -0001,0,
2.50000000000000E -0001,0,
2.50000000000000E -0001,0,
2.50000000000000E -0001,0,
2.50000000000000E -0001,0,

2.50000000000000E -0001,0,
2.50000000000000E -0001,0,
2.50000000000000E -0001,0,

2.50000000000000E -0001,0,
2.50000000000000E -0001,0,
2.50000000000000E -0001,0,

2.50000000000000E -0001,0,


| CTRIA3, 93, 1, 112, 137, 113, 112, 137, 113, |
| CTRIA3, 94, 1, 113, 107, 108, 113, 107, 108, |
| CTRIA3, 95, 1, 113, 108, 113, 108, 113, |
| CTRIA3, 96, 1, 113, 112, 113, 112, 113, |
| CTRIA3, 97, 1, 137, 113, 137, 113, |
| CTRIA3, 98, 1, 2, 3, 113, |
| CTRIA3, 99, 1, 4, 113, 112, |
| CTRIA3, 100, 1, 3, 4, 112, 113, 100, 1, 3, 4, 112, |
| CTRIA3, 101, 1, 4, 111, 112, 111, 101, 1, 4, 111, 112, |
| CTRIA3, 102, 1, 5, 111, 112, 111, 102, 1, 5, 111, 112, |
| CTRIA3, 103, 1, 9, 10, 11, 10, 11, |
| CTRIA3, 104, 1, 9, 10, 11, |
| CTRIA3, 105, 1, 10, 11, 12, 12, 11, 10, |
| CTRIA3, 106, 1, 8, 10, 11, 10, 11, |
| CTRIA3, 107, 1, 8, 10, 11, |
| CTRIA3, 108, 1, 9, 10, 11, |
| CTRIA3, 109, 1, 9, 10, 11, |
| CTRIA3, 110, 1, 10, 11, |