IMPLEMENTATION OF AN ALGORITHM TO APPROXIMATE CONSTRAINED TETRAHEDRIZATIONS WITH PRE-SPECIFIED TRIANGULAR FACES

THESIS

Presented to the Graduate Council of
Southwest Texas State University
in Partial Fulfillment of
the Requirements

For the Degree of

Master of Science

By

Brian J. Collins, B.S.

San Marcos, Texas
August, 1997

To my parents whose constant love and
support means everything to me.

Acknowledgments

This thesis would not have been possible if it were not for the kind help of many people, including those specifically acknowledged here.

First, I would like to thank my mother who constantly prayed for me and continually prodded me to continue working on this thesis. Without her love, prayers, and support this whole project would probably never have been completed.

Next, I would like to thank Dr. Carol Hazlewood for being my thesis advisor and bringing me along into her wonderful area of expertise. After an undergraduate course in computational geometry, a graduate course in computational geometry, and now a thesis in computational geometry I now understand a lot more now than I used to about the subject. I am still dreaming of faceted objects in multidimensional space like she wanted us to in class.

Next, I would like to thank Dr. Donald Hazlewood who graciously let me use the office in the SWT Math Lab for my research plus space for my Sun 3/60. Also, I am grateful for allowing me to use the newly acquired Sun Sparc 20 and purchasing the Sparc Works C++ compiler which was a great help to the development of the code for this thesis.

I would also like to thank the singles groups of Live Oak Community Church in Austin, Texas and New Life Community Church in Cedar Rapids, Iowa for supporting me both in prayer and in life.

I also thank and acknowledge assistance from: Brian D. Hernandez (Lockheed Martin M&DS Valley Forge), Robert W. McBeth (Lockheed Martin Missiles & Space Austin Division), Gary Stokes (Lockheed Martin M&DS Valley Forge), David E. Seale (Rockwell Collins Avionics & Communications), Michael A. Miller (Rockwell Collins Avionics & Communications), and Wayne I. Hughes (Rockwell Collins Avionics & Communications).

TABLE OF CONTENTS

TABLES

FIGURES

ABSTRACT

IMPLEMENTATION OF AN ALGORITHM TO APPROXIMATE CONSTRAINED TETRAHEDRIZATIONS WITH PRE-SPECIFIED TRIANGULAR FACES

by

Brian James Collins, B.S.
Southwest Texas State University
August 1997

Supervising Professor: Dr. Carol Hazlewood

A tetrahedrization is a decomposition of a region in space into tetrahedra. It is not always possible to construct a tetrahedrization that contains prespecified facets. There is an unimplemented algorithm for producing a reasonably efficient approximate solution. Given points and triangles that intersect in (possibly empty) mutual faces, a binary space partition is used to define subregions of the convex hull of the input. The planar boundary faces of these subregions are triangulated with constraints, and the subregions are covered with tetrahedra that preserve the boundary triangles. The constraints are such that the set of tetrahedra is a tetrahedrization and the specified triangles are unions of facets of tetrahedra. An Object-Oriented analysis, an Object-Oriented design, and a C++ implementation of an algorithm to split the convex hull of a finite set of points by a plane is presented.

Chapter I

INTRODUCTION

1. Introduction

In Hazlewood's paper[17] a method is described for constructing a tetrahedrization in three-dimensional Euclidean space that includes specified triangular regions as the union of facets of tetrahedra. It is not always possible to construct a tetrahedrization in which specified triangles appear as facets and the current methods to determine if such a tetrahedrization exists is considered an NP-hard problem. NP-hard problems do not have a solution that completes in polynomial running time, $O(n^k)$ for any constant k, and an optimal solution is intractable.[9] In an earlier paper by Hazlewood[16] and Sun's thesis[29] a method is presented that uses an approximation algorithm to construct a tetrahedrization. An algorithm that returns near-optimal solutions in polynomial time (either worst case time or average case time) is considered an approximation algorithm.[9] Approximation algorithms or exponential running time algorithms (with small input sets) are both acceptable solutions for NP-hard problems. The algorithm presented in Hazlewood's paper[17] theoretically can construct such a tetrahedrization in polynomial time and space. This thesis describes the implementation of the algorithm presented in Hazlewood's paper.[17]

A tetrahedrization is a decomposition of a region in space into tetrahedra. Tetrahedrization algorithms are important for applications involving surface modeling, applications involving modeling 3D objects with non-convex boundaries, and modeling 3D surfaces with discontinuities. The 2D analog to this problem has an O(n lg n) solution and a solution always exists, but the 3D problem does not always have a solution. A tetrahedrization of a finite set of points is a collection of tetrahedra that intersect in mutual (possibly empty) faces and cover the convex hull of the point set. It is not always possible to construct a tetrahedrization that contains prespecified facets. Presented in Hazlewood's paper[17] there is an unimplemented algorithm for producing a reasonably efficient approximate solution. Given points and triangles that intersect in (possibly empty) mutual faces, a binary space partition is used to define subregions of the convex hull of the input. The subregions are covered with tetrahedra in such a way that the result is a tetrahedrization in which the specified triangles are unions of facets of tetrahedra. The pseudo code for this algorithm is presented in Chapter 4. This thesis concentrated on

the Object-Oriented Analysis, the Object-Oriented Design, the portable C++ implementation of the algorithm to split the convex hull of a finite set of points by a plane, and visualization in the Geomview[15] program.

In Chapter 2 basic concepts and definitions are presented. In Chapter 3 a basic Object-Oriented analysis and a basic Object-Oriented design with classes designed to implement the algorithm is presented. In Chapter 4 a basic Object-Oriented programming implementation in C++ and sample program output in the Mathematica[36] and Geomview[15] programs is presented. In Chapter 5 detailed information about convex hulls is presented. In Chapter 6 detailed information about BSP trees is presented. In Chapter 7 physical program implementation issues are presented. In the appendix, a detailed C++ implementation is presented.

Chapter II

BACKGROUND

2. Basic Definitions

This chapter introduces the basic definitions and concepts of several key words that are used in chapter 3 through chapter 7.

Computational Geometry	The generation of efficient algorithms to solve geometric problems; or the systematic study of geometric algorithms.
Object	An abstraction of something in a problem domain, reflecting the capabilities of a system to keep information about it, interact with it, or both; an encapsulation of Attribute values and their exclusive Services. (synonym: an Instance)[7]
Class	A description of one or more Objects with a uniform set of Attributes and Services, including a description of how to create new Objects in the Class.[7]
Class-&-Object	A term meaning a Class and the Objects in that Class.[7]

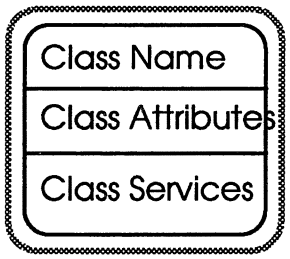


Figure 2-1 - Class-&-Object Structure Notation

Base Class	A base class or super class is a root class from which other classes are derived from. The newly derived classes inherit the base class functionality and add their own additional functionality.
Derived Class	A class that is derived from a base class or a super class. It has all of the inherited functionality from the parent class and it can add its own functionality.
OO	Object-Oriented. A methodology of analysis, design, and programming that subdivides a problem into classes that contain attributes and services that can act upon those classes. A class combines a data structure and the services to act upon that data structure into one reusable object. Object-Oriented systems encompass the ideas of: Encapsulation, Inheritance, Polymorphism, Dynamic Binding, and Generic Relationships.
OOA	Object-Oriented Analysis, establishes the classes in the problem domain. It specifies what each object in each class needs to know and do, and it specifies object interactions all in one model.[7]
OOD	Object-Oriented Design, adds design considerations to the OOA results. Mainly, OOD focuses on human interaction, data management, and task management classes. It includes additional detail on what each object in each class needs to know and do, and it specifies additional object interactions all in one model.[8]
OOP	Object-Oriented Programming, the implementation in a programming language of the classes developed through the OOA and OOD processes.[6]
Gen-Spec Structure	A half circle symbol in Coad/Yourdon design diagram which represents a generalization class - specialization class relationship. The Class-&-Object at the top of the half circle symbol is the generalization class and any Class-&-Object's at the bottom of the half circle symbol are the specialization classes. For example, a generalized "sensor" class might have a specialization "critical sensor" class and a specialization "standard sensor" class derived from it.

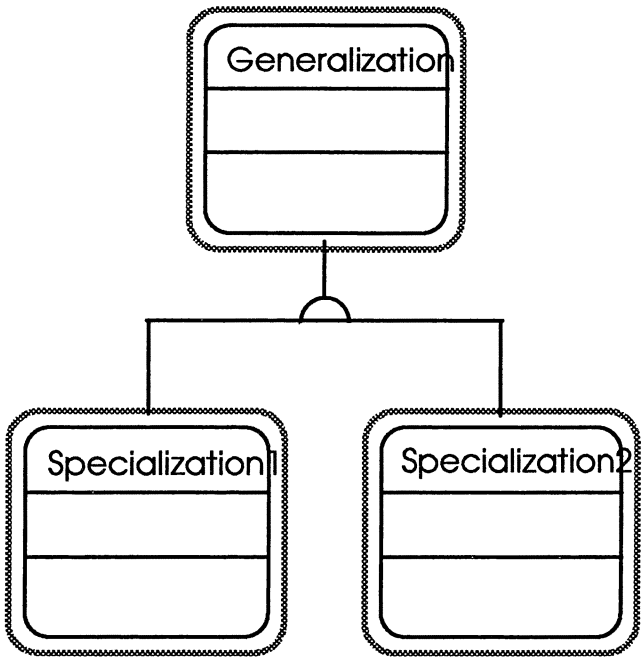


Figure 2-2 - Gen-Spec Structure Notation [7]

Whole-Part Structure

A triangle symbol in Coad/Yourdon design diagram which represents a whole class - part class relationship. The Class-&-Object at the top of the triangle symbol is the whole class and any Class-&-Object's at the bottom of the triangle symbol are the part classes. Numbers next to the whole class denote how many parts the whole class may have, and numbers next to the part classes denote how many whole classes the part class can belong to. For example, an "aircraft" class might have 1 or more engines, but an "engine" class would only belong to 1 aircraft.

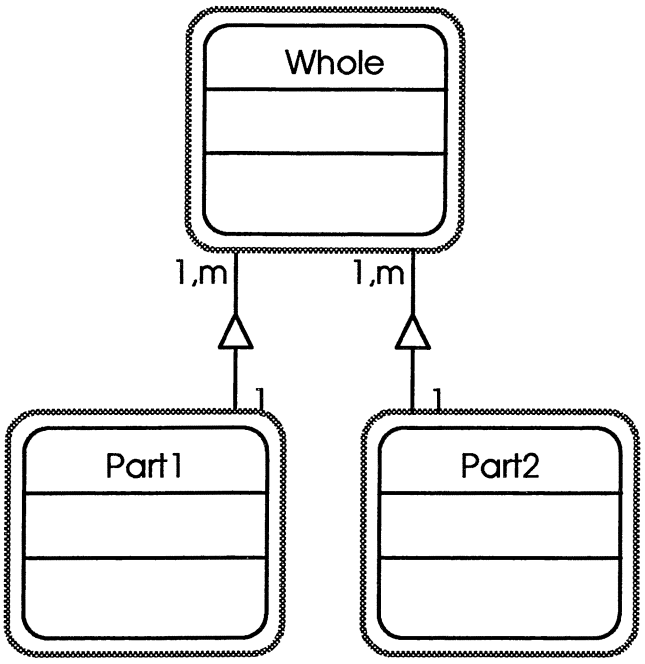


Figure 2-3 - Whole-Part Structure Notation [7]

2D	2 Dimensional Space, E^2 also known as Euclidean space.
3D	3 Dimensional Space, E^3.
E^d	d Dimensional Space, E^d.
Convex Set	A set A is convex if for every pair of points p and q in set A, the line segment pq is in set A. See chapter 5 for a detailed explanation.
Convex Hull	The convex hull of a set of points P is the smallest convex set that contains P.[26] See section 3.2.3 and section 5.1 for a detailed explanation.
Tetrahedrization	A tetrahedrization of a finite set of points is a collection of tetrahedra that intersect in mutual (possibly empty) faces and cover the convex hull of the point set
BSP Tree	Binary Space Partition Tree - A BSP Tree is a data structure that represents a recursive, hierarchical subdivision of n-dimensional space into convex subspaces.[33] See section 3.2.4 and chapter 6 for a detailed explanation.

Plane Angle	The angle between two planes is the angle between the two normals to the planes. See section 3.1.1, figure 6, for a detailed explanation.
Coincident	Two planes that lie in the exact same space are coincident planes. The angle between the two planes is exactly 0 degrees. See section 3.1.1, figure 4, for a detailed explanation.
Spanning	Two planes that intersect each other are spanning planes. The intersection between two non-coplanar planes is a line. The angle between the two planes is greater than 0 degrees. See section 3.1.3, figure 6, for a detailed explanation.
Hyperplane	In E^d dimensional space, a hyperplane is a E^{d-1} dimensional object that is used to divide E^d space into two half-spaces. In E^3 a plane is a hyperplane, in E^2 a line is a hyperplane.

Chapter III

CLASS DESIGN

3. Class Design

The class design in this project followed common Object-Oriented (OO) design principles. All Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD) were performed using the Coad/Yourdon Object-Oriented Method.[6,7,8] The Object-Oriented Programming (OOP) implementation followed directly from the OOA and OOD processes and was done in the C++ programming language. The base classes of the problem domain were designed during the OOA phase of the project. Other classes were designed during the OOD phase of the project from the OOA base classes to solve the problem of implementing the Constrained Tetrahedrization algorithm as needed.

3.1 OOA Base Classes

The base classes were designed during the OOA phase of the project with the base objects of the problem in mind. The only true "base classes", by the Object-Oriented definition of the term, are the Point2 and Point3 classes which are self-contained classes and form the basis for all other class construction in the OOA phase of the project. The other classes that are built from the Point2 and Point3 classes are still called base classes for the terms of this project because they are classes that were designed during the OOA phase of the project and they are base objects for solving the problem in the Constrained Tetrahedrization algorithm.

The five major activities of OOA are:[7]

1. Subject Layer
2. Class-&-Object Layer
3. Structure Layer
4. Attribute Layer
5. Service Layer

A subject is a mechanism for guiding a reader through a large, complex model. Subjects are also helpful for organizing work packages based upon initial OOA investigation.[8] The subject layer for the problem domain was one simple task, so I decided to have one subject named Constrained Tetrahedrization which is the name of the program. The whole purpose of the program was to take in a set of points and a set of partitioning planes as input and to provide a set of partitioned convex hulls as output. The single purpose of the program only needed a single subject to represent it.

The Class-&-Object layer for the problem domain started out in the realm of 2D and 3D geometry objects. The natural classes that fell out of the 2D analysis were objects like point2 and line_segment2, which represented a point in E^2 and a line segment in E^2. The natural classes that fell out of the 3D analysis were objects like point3, line_segment3, and plane, which represented a point in E^3, a line segment in E^3, and a plane in E^3. Other classes that fell out of the analysis in this stage were objects like facet2, facet3, subfacet2, and subfacet3 which were subparts of a tetrahedron object. The tetrahedron object was not defined during the OOA phase of the project because it was a more complex object that needed further analysis and design.

The Structure layer for the problem domain ended up being standard Geometric representations for the most part. In 3D, for example, a line segment is composed of two points, a plane is composed of three points, a facet has three points, and a subfacet has two points. In 2D, for example, a line segment is composed of two points, a facet has two points, and a subfacet has one point.

The Attribute layer for the classes in the problem domain only consisted of the standard Geometric representations for the most part. In 3D, for example, a point had 3 attributes X, Y, and Z. In 2D, for example, a point had 2 attributes X and Y.

The Service layer for the classes in our problem domain only consisted of the standard constructors, destructors, and accessors at this point in the analysis which meant that there were no services defined in the design because these standard services are not denoted in the Coad/Yourdon object model. The next sections describe the classes built during the OOA phase of the project on a class by class basis.

3.1.1 Point2 Class

The Point2 class is a representation of a point in 2D or Euclidean space. Internally it is represented with 2 double length floating point numbers labeled X and Y. Common constructors, destructors, and accessors are provided with the class to make the class as open and re-usable as possible, but they are not shown using the Coad/Yourdon object model. Shown in Figure 3-1 is the object that this class represents and the Object-&-Class diagram:

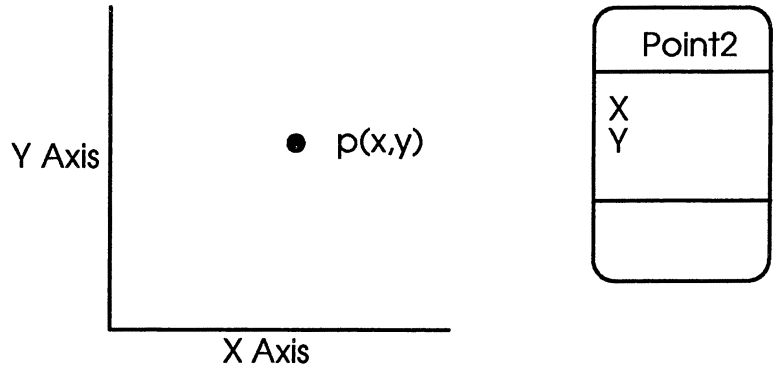


Figure 3-1 - Point2 Class Representation and Object-&-Class Diagram

3.1.2 Point3 Class

The Point3 class is a representation of a point in 3D space. Internally it is represented with 3 double length floating point numbers labeled X, Y, and Z. Common constructors, destructors, and accessors are provided with the class to make the class as open and re-usable as possible, but they are not shown using the Coad/Yourdon object model. Shown in Figure 3-2 is the object that this class represents and the Object-&-Class diagram:

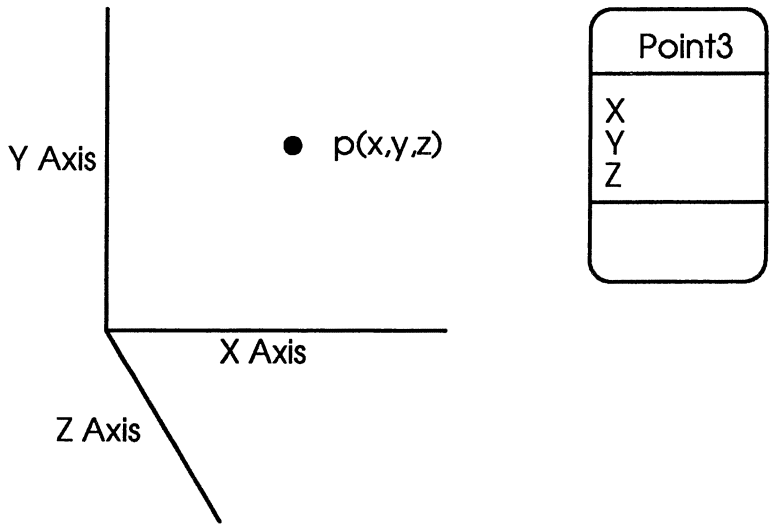


Figure 3-2 - Point3 Class Representation and Object-&-Class Diagram

3.1.3 Plane Class

The Plane class is a representation of a plane in 3D space. Internally it is represented with 3 points labeled P1, P2, and P3. Internally it is also is represented with 4 double length floating point numbers labeled A, B, C, and D. The plane coefficients (Ax+By+Cz+D=0) are calculated using Newell's Method.[5,30] Common constructors,

destructors, and accessors are provided with the class to make the class as open and re-usable as possible, but they are not shown using the Coad/Yourdon object model. Shown in Figure 3-3 is the object that this class represents and the Object-&-Class diagram:

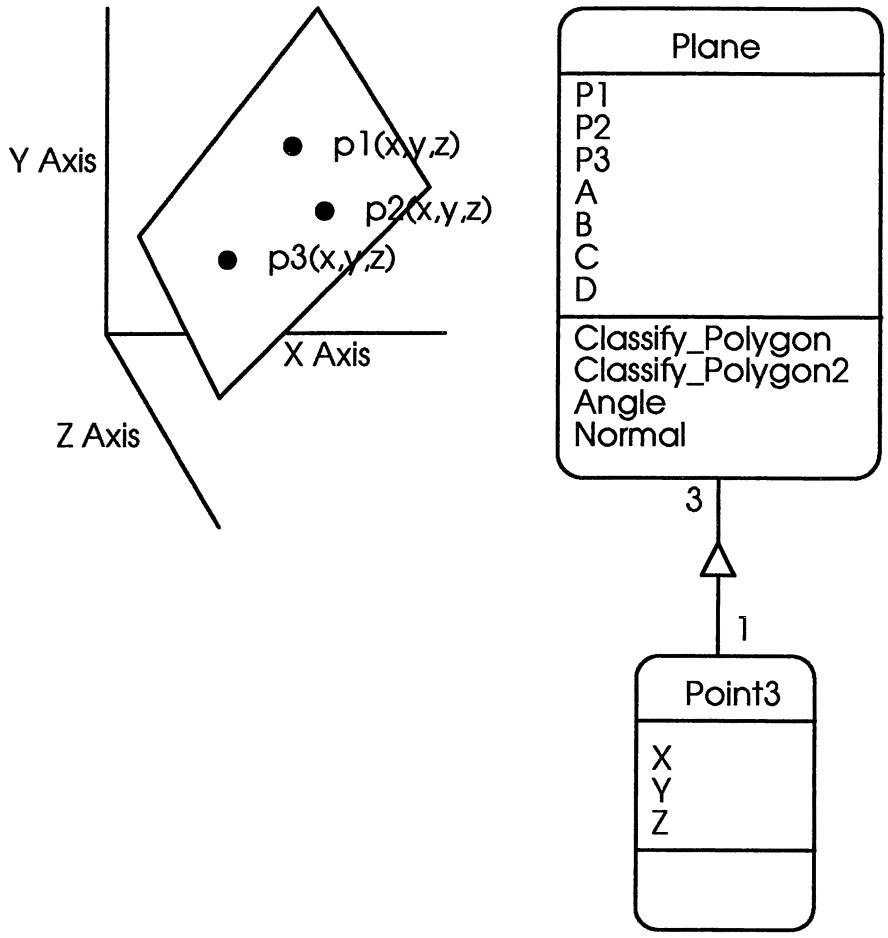


Figure 3-3 - Plane Class Representation and Object-&-Class Diagram

The services that are shown in this class design were added during the OOD phase of the design. The Classify_Polygon service in this class takes the current plane object and a plane that is passed in as a parameter and it determines whether the parameter plane is coincident, in back of, in front of, or spanning the current plane. Shown in Figure 3-4 are examples of coincident, in back of / in front of, and spanning planes:

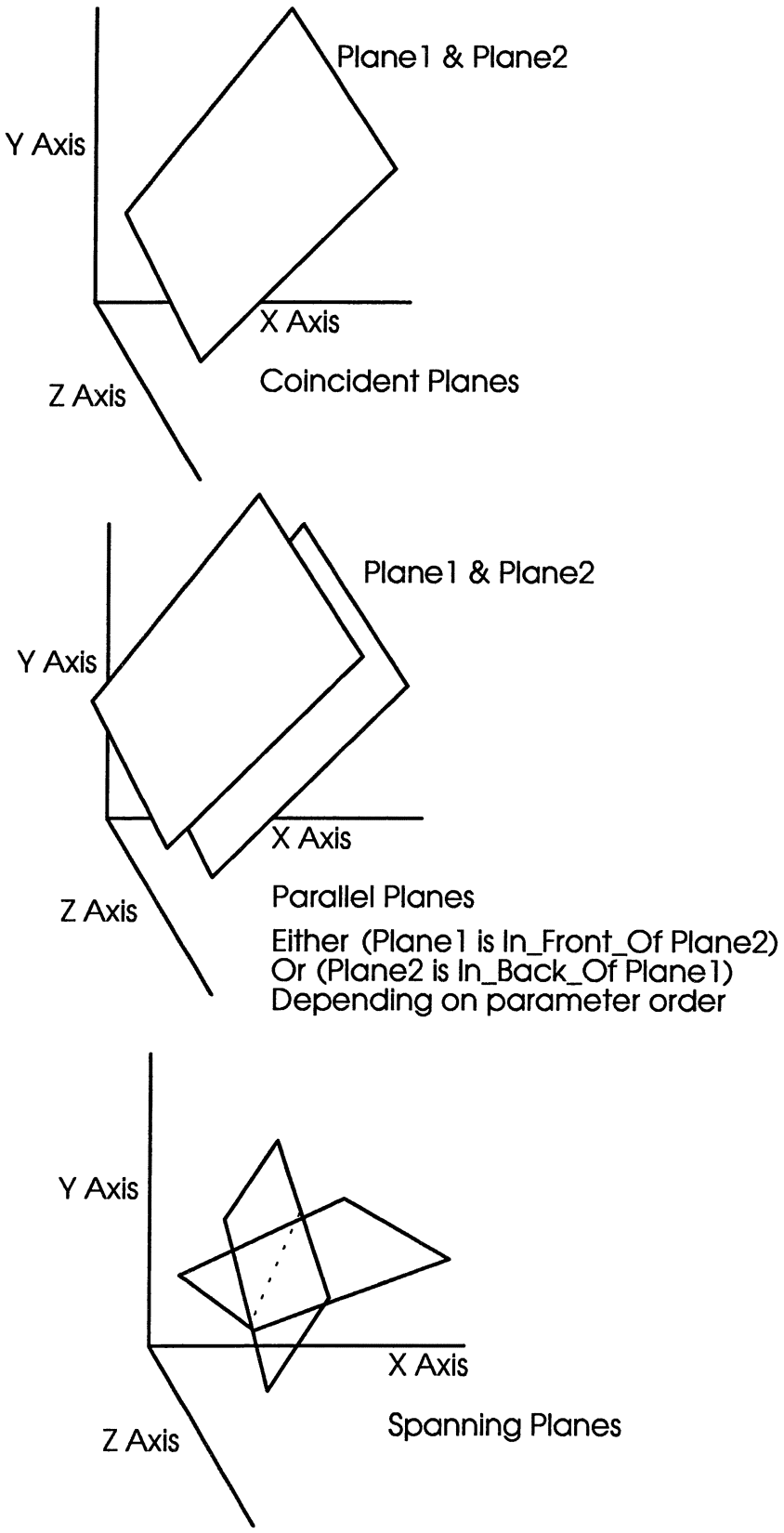


Figure 3-4 - Examples of Coincident, In Front Of / In Back Of, and Spanning Planes

The Classify_Polygon2 service in this class takes the current plane and a point that is passed in as a parameter and it returns whether the point is coincident, in back of, in front of, or spanning the current plane. Shown in Figure 3-5 are examples of a point coincident to a plane, a point in back of a plane, and a point in front of a plane:

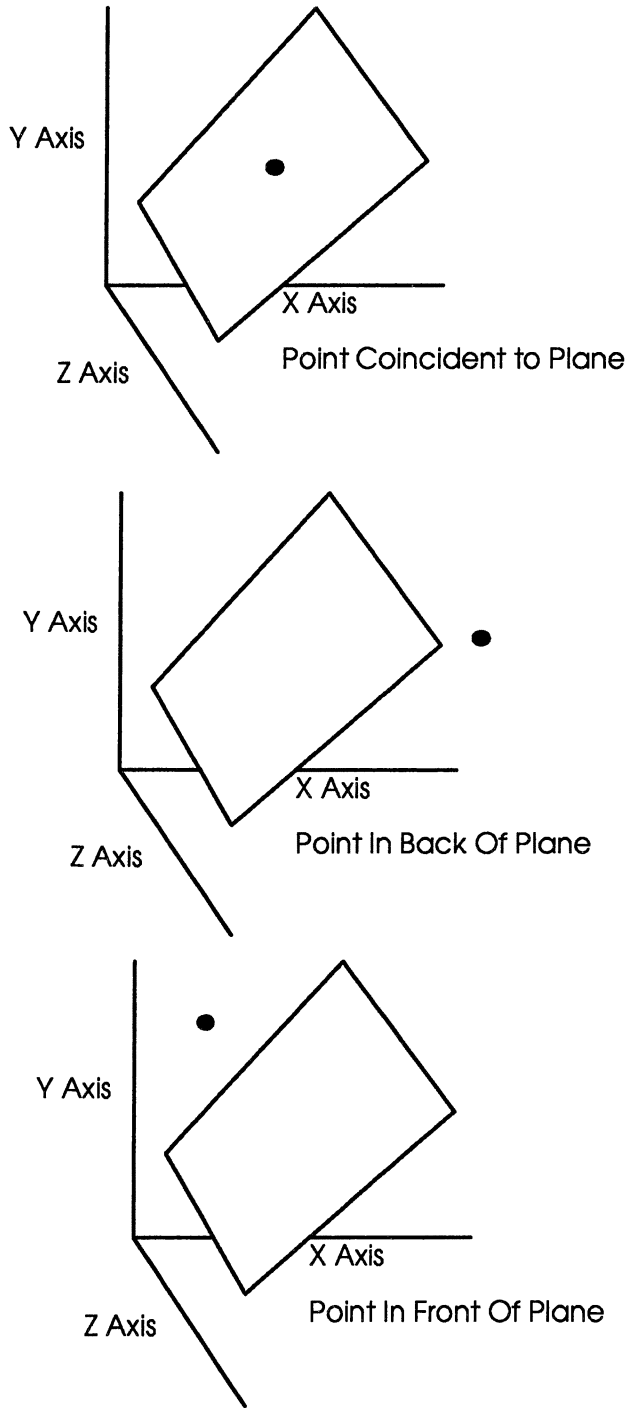


Figure 3-5 - Example of Point Coincident to, a Point In Back of, and a Point In Front of a Plane

The Angle service in this class takes the current plane and a plane that is passed in as a parameter and it returns the angle between the two spanning planes. The Normal service in this class returns the normal vector to the current plane. Shown in Figure 3-6 are examples of the normal vector to a plane and the angle between two spanning planes:

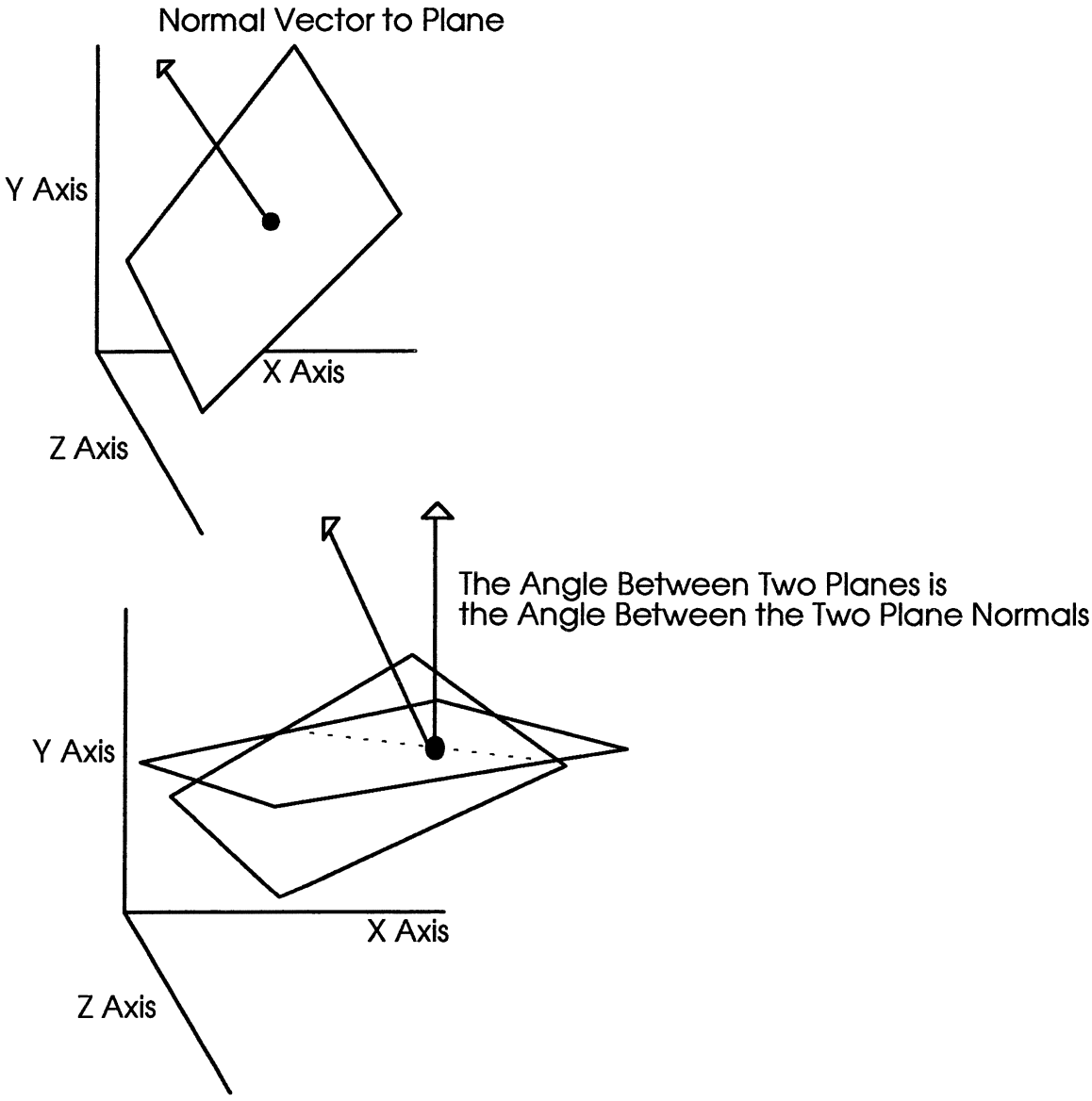


Figure 3-6 - Example of the Normal Vector to the Plane and the Angle Between Two Planes

3.1.4 Line_Segment2 Class

The Line_Segment2 class is a representation of a line segment in 2D space. Internally it is represented with 2 points labeled P1 and P2. Common constructors, destructors, and accessors are provided with the class to make the class as open and re-

usable as possible, but they are not shown using the Coad/Yourdon object model. Shown in Figure 3-7 is the object that this class represents and the Object-&-Class diagram.

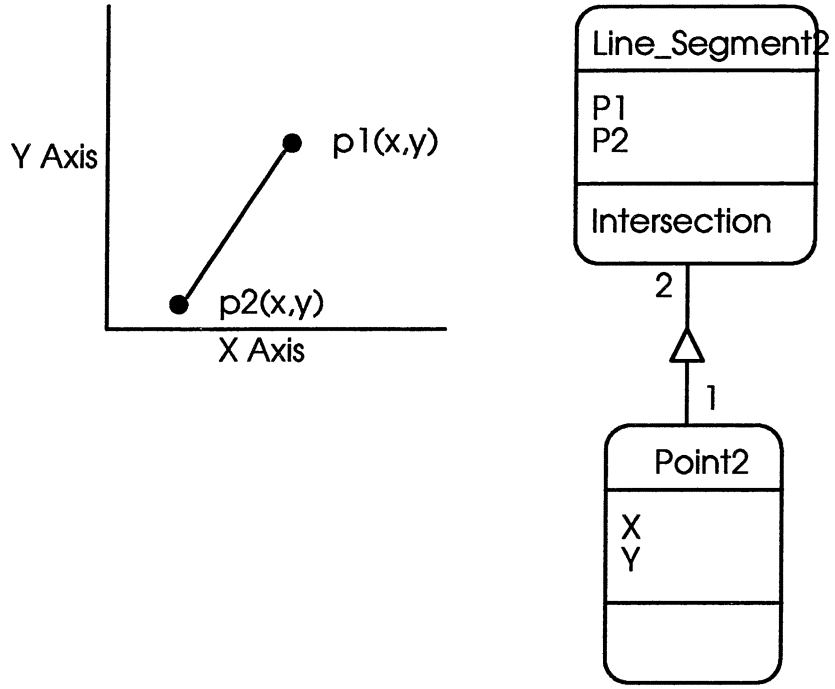


Figure 3-7 - Line_Segment2 Class Representation and Object-&-Class Diagram

The services that are shown in this class design were added during the OOD phase of the design. The Intersection service in this class takes the current line segment and a second line segment that is passed in as a parameter and it returns a point as the intersection point between the two line segments. If the line segments do not intersect in a single point then the origin (0,0) is returned. This error value should be sufficient for this project since all of the data resides in quadrant I. Adding in cases in the code to handle intersection error properly is left for future expansion of this thesis. Shown in Figure 3-8 are examples of a line segment intersecting, and not intersecting a second line segment:

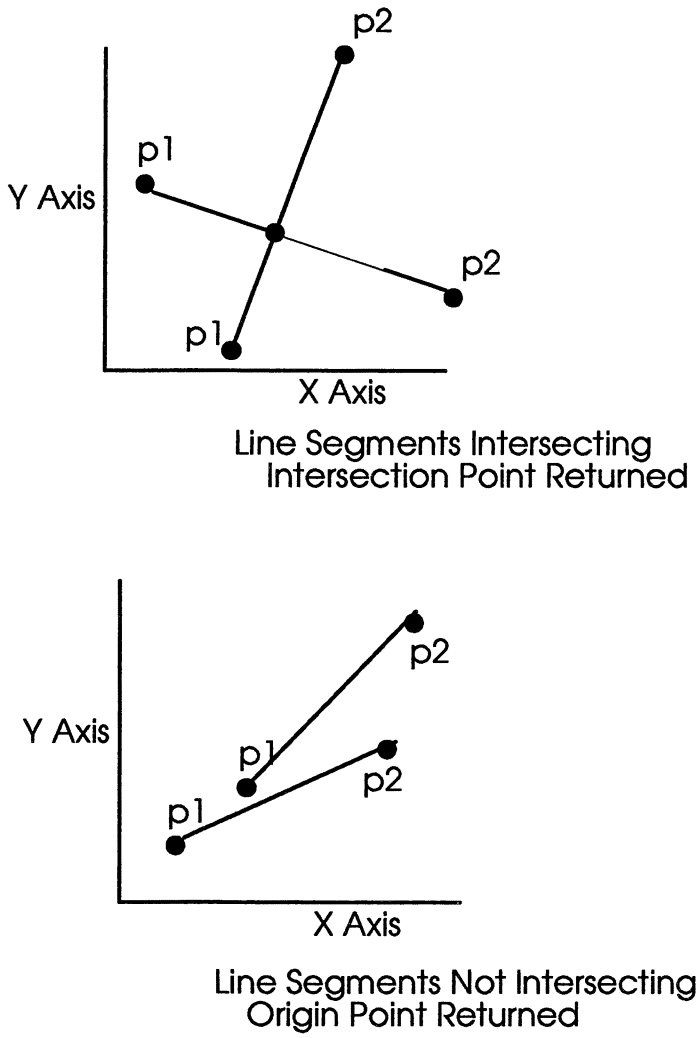


Figure 3-8 - Example of a Line Segment Intersecting, and not Intersecting a Second Line Segment

3.1.5 Line_Segment3 Class

The Line_Segment3 class is a representation of a line segment in 3D space. Internally it is represented with 2 points labeled P1 and P2. Common constructors, destructors, and accessors are provided with the class to make the class as open and re-usable as possible, but they are not shown using the Coad/Yourdon object model. Shown in Figure 3-9 is the object that this class represents and the Object-&-Class diagram:

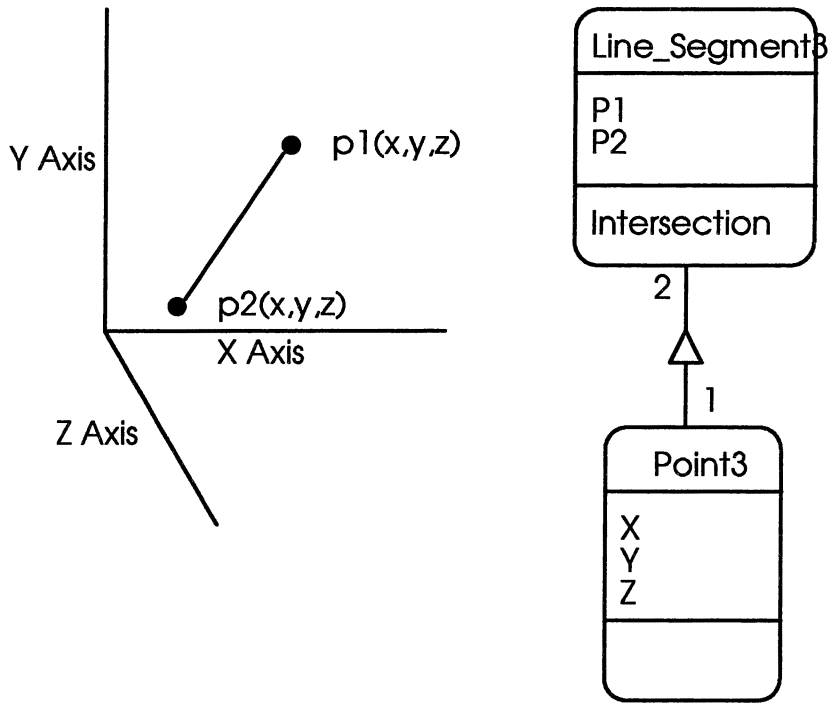


Figure 3-9 - Line_Segment3 Class Representation and Object-&-Class Diagram

The services that are shown in this class design were added during the OOD phase of the design. The Intersection service in this class takes the current line segment object and a plane that is passed in as a parameter and it returns a point as the intersection point between the line segment and the plane. If the line segment is coincident to the plane then the midpoint of the segment is returned. If the line segment is in back of or in front of the plane then the origin (0,0,0) is returned since there is no intersection to return. This error value should be sufficient for this project since all of the data resides in quadrant I. Adding in cases in the code to handle intersection error properly is left for future expansion of this thesis. Shown in Figure 3-10 are examples of a line segment spanning, coincident to, and in back of a plane:

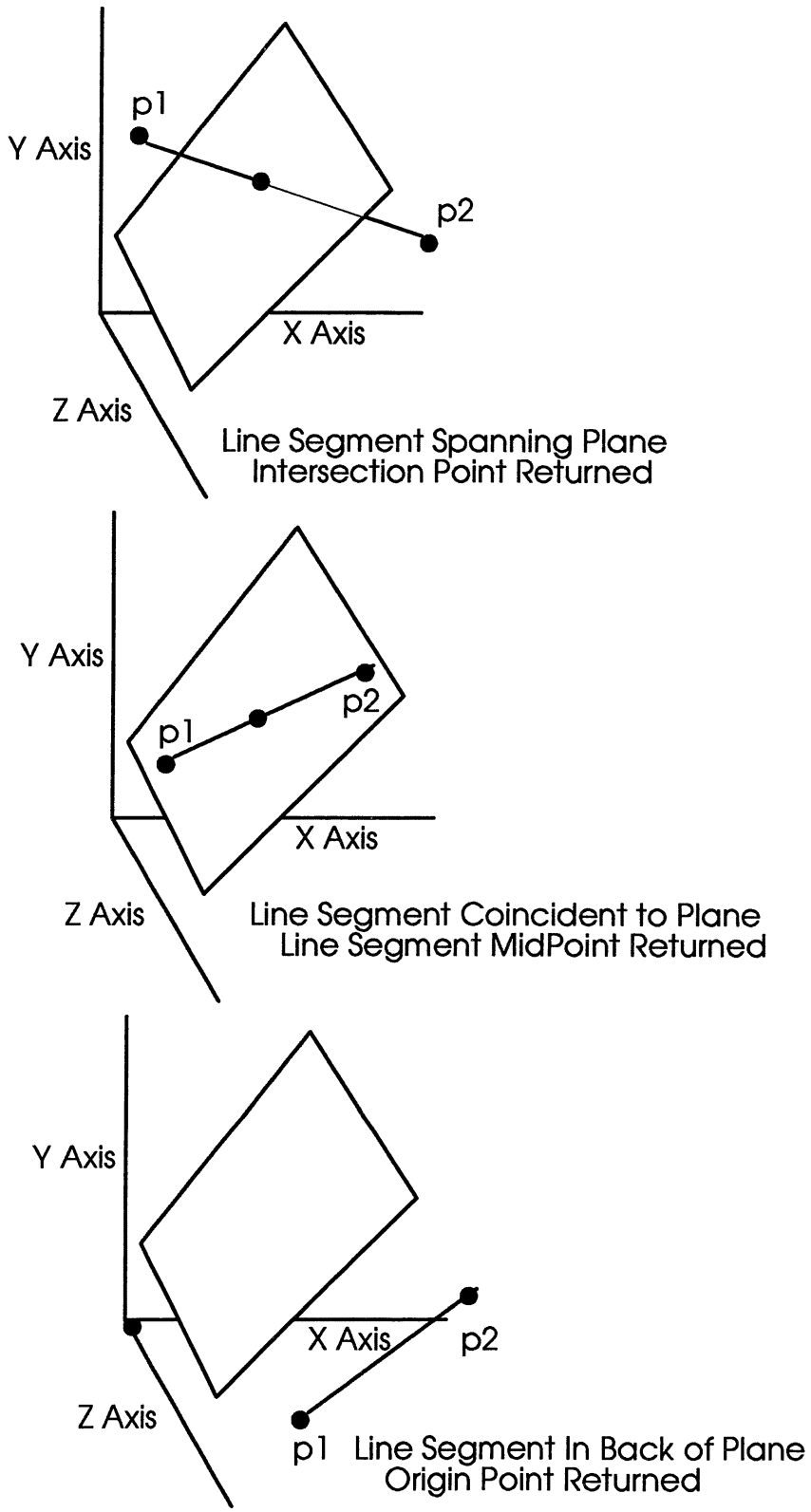


Figure 3-10 - Example of a Line Segment Spanning, Coincident to, and In Back Of a Plane

3.1.6 SubFacet2 Class

The SubFacet2 class is a representation of a subfacet of a 2D tetrahedron. In 2D, three points are needed to define a tetrahedron, two points are needed to define a facet of a tetrahedron, and one point is needed to define a subfacet of a tetrahedron. Internally this class is derived from the Point2 class. Common constructors, destructors, and accessors are provided with the class to make the class as open and re-usable as possible, but they are not shown using the Coad/Yourdon object model. The SubFacet2 class is derived from the Point2 class since it is very similar in structure. Shown in Figure 3-11 is the object that this class represents and the Object-&-Class diagram:

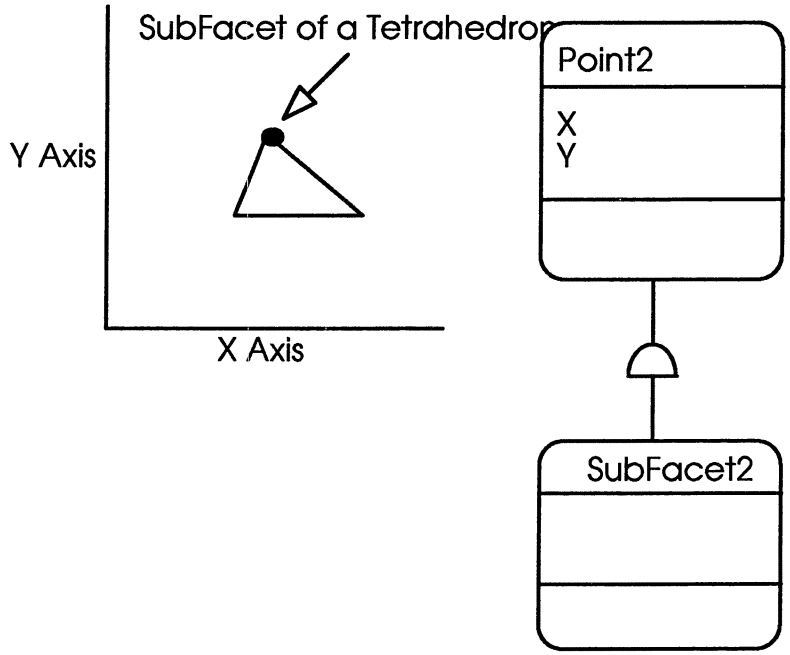


Figure 3-11 - SubFacet2 Class Representation and Object-&-Class Diagram

3.1.7 SubFacet3 Class

The SubFacet3 class is a representation of a subfacet of a 3D tetrahedron. In 3D, four points are needed to define a tetrahedron, three points are needed to define a facet of a tetrahedron, and two points are needed to define a subfacet of a tetrahedron. Internally this class is derived from the Line_Segment3 class. Common constructors, destructors, and accessors are provided with the class to make the class as open and re-usable as possible, but they are not shown using the Coad/Yourdon object model. The SubFacet3 class is derived from the Line_Segment3 class since it is very similar in structure. Shown in Figure 3-12 is the object that this class represents and the Object-&-Class diagram:

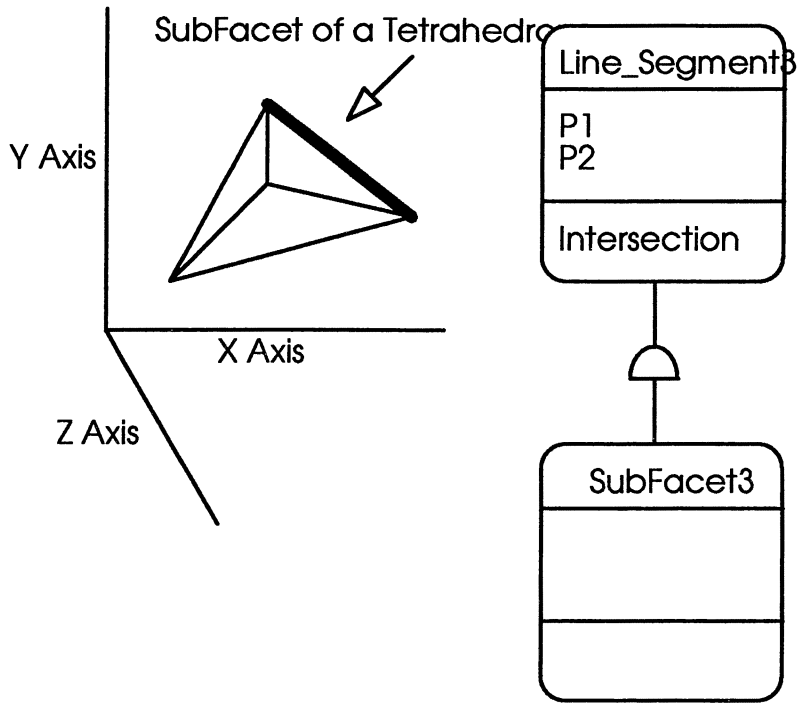


Figure 3-12 - SubFacet3 Class Representation and Object-&-Class Diagram

3.1.8 Facet2 Class

The Facet2 class is a representation of a facet of a 2D tetrahedron. In 2D, three points are needed to define a tetrahedron and two points are needed to define a facet of a tetrahedron. Internally this class is derived from the Line_Segment2 class. Common constructors, destructors, and accessors are provided with the class to make the class as open and re-usable as possible, but they are not shown using the Coad/Yourdon object model. The Facet2 class is derived from the Line_Segment2 class since it is very similar in structure. Shown in Figure 3-13 is the object that this class represents and the Object-&-Class diagram:

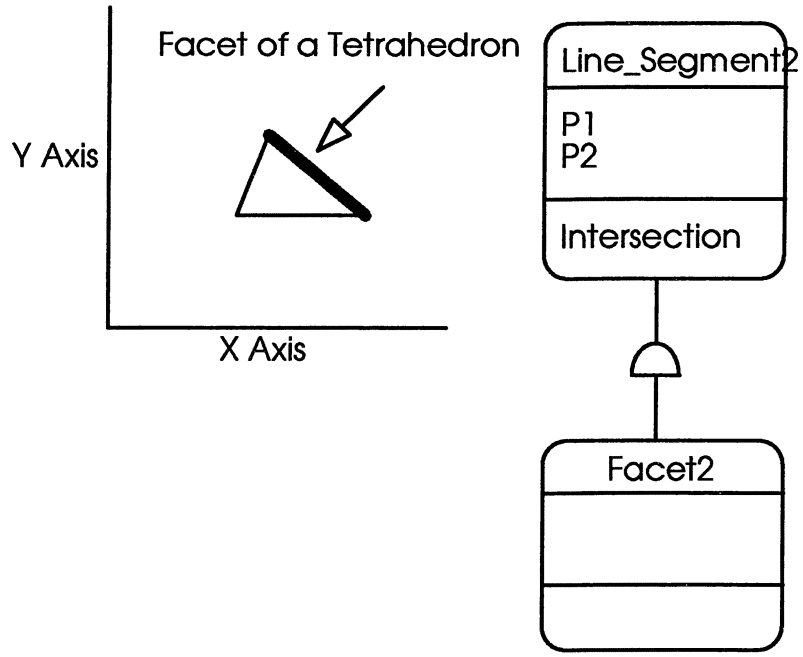


Figure 3-13 - Facet2 Class Representation and Object-&-Class Diagram

3.1.9 Facet3 Class

The Facet3 class is a representation of a facet of a 3D tetrahedron. In 3D, four points are needed to define a tetrahedron and three points are needed to define a facet of a tetrahedron. Internally this class is derived from the Plane class. Common constructors, destructors, and accessors are provided with the class to make the class as open and re-usable as possible, but they are not shown using the Coad/Yourdon object model. The Facet3 class is derived from the Plane class since it is very similar in structure. Shown in Figure 3-14 is the object that this class represents and the Object-&-Class diagram:

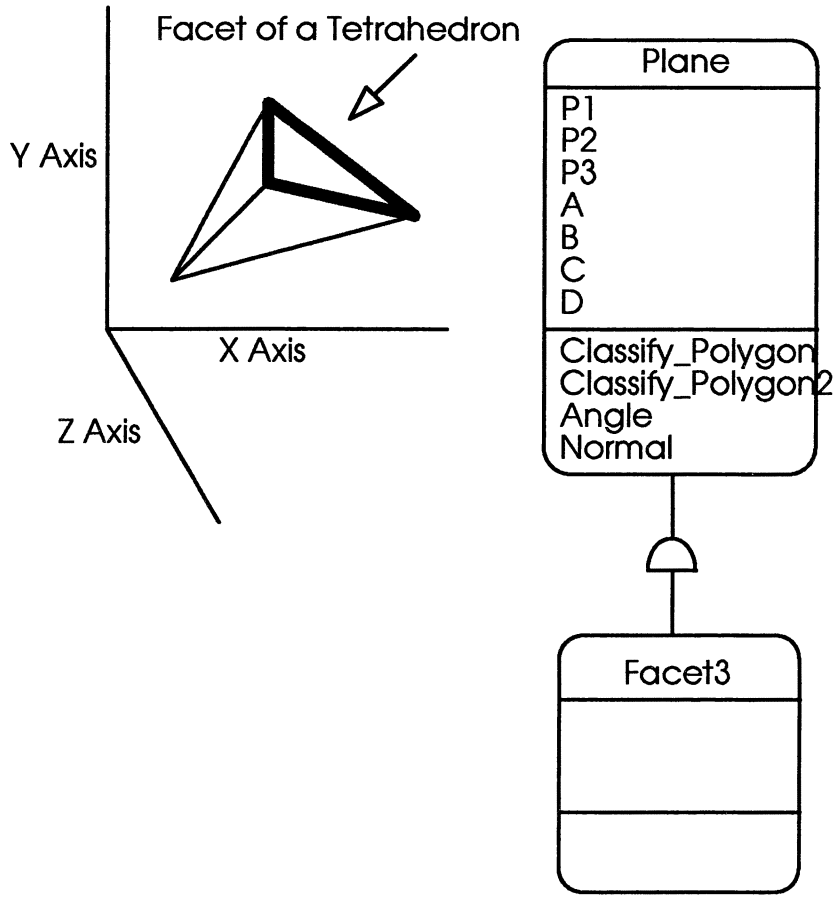


Figure 3-14 - Facet3 Class Representation and Object-&-Class Diagram

3.1.10 OOA Model Diagram

All of the Coad/Yourdon OOA Layers that were developed during the OOA phase of the project are shown together on one diagram which forms an object model diagram. The Coad/Yourdon OOA Object Model Diagram for the Constrained Tetrahedrization project follows:

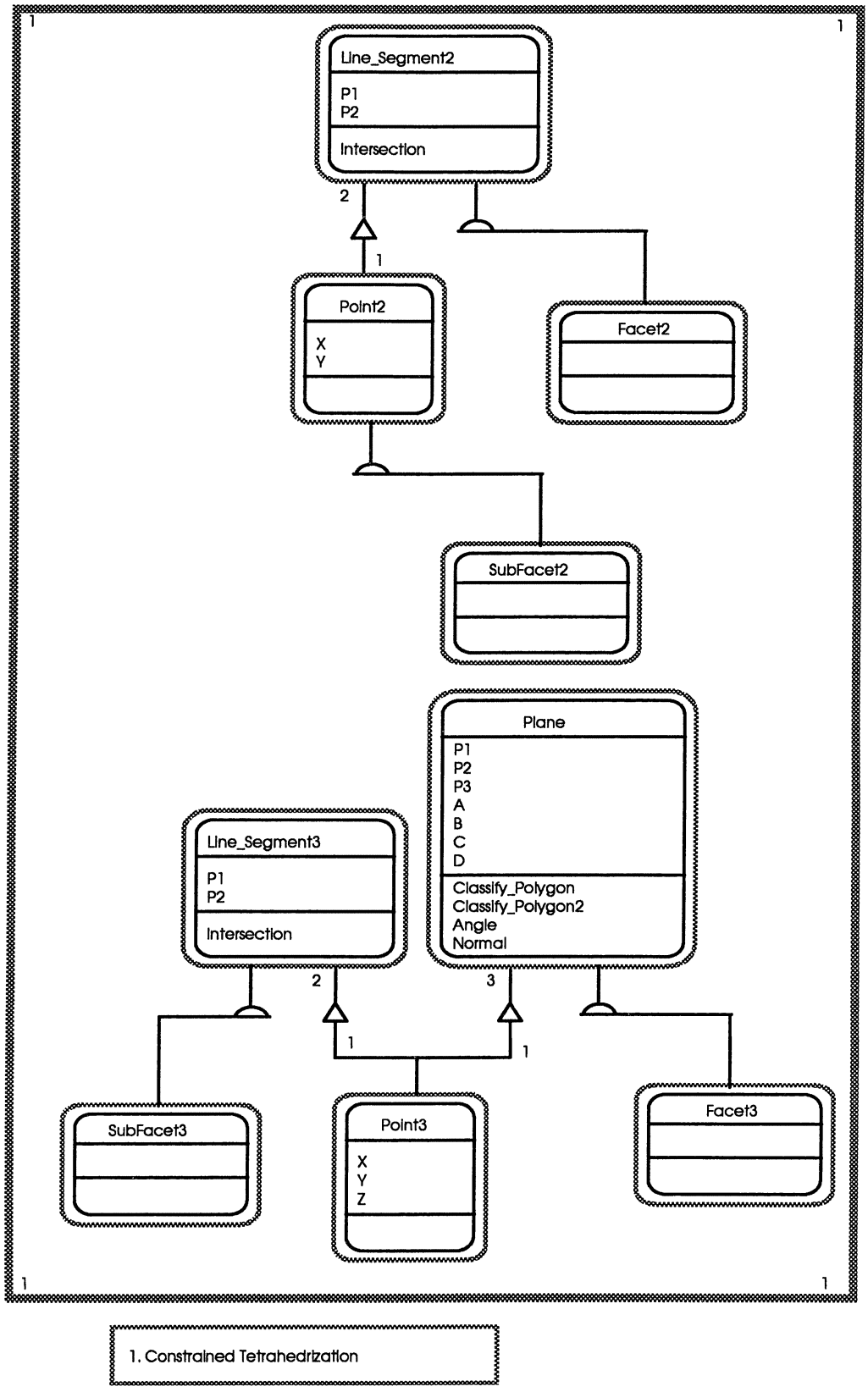


Figure 3-15 - Coad/Yourdon OOA Object Model Diagram

3.2 OOD Derived Classes

The derived classes were designed during the OOD phase of the project with the flushing out of the OOA design and the implementation of the Constrained Tetrahedrization algorithm in mind. The classes were designed to solve specific problems in the Constrained Tetrahedrization algorithm and were built using only the base classes that were designed during the OOA phase of the project.

The four major activities of OOD are:[8]

1. Problem Domain Component
2. Human Interaction Component
3. Task Management Component
4. Data Management Component

3.2.1 Problem Domain Component

The problem domain component of OOD is the activity where the design modifies the OOA results to resolve a design consideration. In this activity you can also show the other three OOD components in a collapsed form. In the Constrained Tetrahedrization project there were some classes that were not discussed in the OOA results that were needed to solve some problems in the Constrained Tetrahedrization algorithm. The main Constrained Tetrahedrization algorithm as presented below contains none of the objects, except for Plane, that were created in the OOA phase of the project. This is where the OOD phase of the project was needed to complete the overall project design. The first object added was the Constrained_Tetrahedrization object which represented the main project class.

```
constrain(P',F,T)
begin
      construct B;
      construct C, the convex hull of P';
      for j = 1 to k do
            K_j <- plane(f_j) ∩ C;
      for j = 1 to k do
            compute constraints for K_j;
      for j = 1 to k do
            triangulate K_j with constraints;
      triangulate facets of C with constraints;
      for i = 1 to l do
            T_i <- tetrahedrization of R_i;
      T <- ∪ T_i
end.
```

Figure 3-16 - Constrained Tetrahedrization Algorithm[17]

As parameters, P' is a set of points in E^3, F is a set of k triangles which have vertices in P' and which intersect in (possibly empty) mutual faces, and T is the output of the tetrahedrization.[17] The set of points P', since an unknown number of points could be entered, was represented by a generic List class. The set of k triangles F, since an unknown number of partitioning planes could be entered, was also represented by a generic List class. In the implementation, the three points that formed the "triangles" or partitioning planes did not come from the set of points in P', but they were randomly generated points. The implementation differed at this point from the paper[17] because the paper[17] stated that F is a set of k triangles which have vertices in P'. The theory of partitioning a polygon or a convex hull with a plane should not rely on the points of the plane also belonging to the points that form the polygon or convex hull. Any arbitrary plane should be able to partition a polygon or a convex hull. The next obvious classes that came from this level of the project were the Convex_Hull class and the BSP_Tree class which were explicitly called out in the algorithm. These four obvious classes formed the basis for a top-down OOD design.

3.2.1.1 Constrained_Tetrahedrization Class

The Constrained_Tetrahedrization class is a representation of the main Constrained Tetrahedrization class that is the OOA subject layer and the main implementation of the algorithm.[17] Common constructors, destructors, and accessors are provided with the class to make the class as open and re-usable as possible, but they are not shown using the Coad/Yourdon object model. Shown in Figure 3-17 is the object that this class represents and the Object-&-Class diagram:

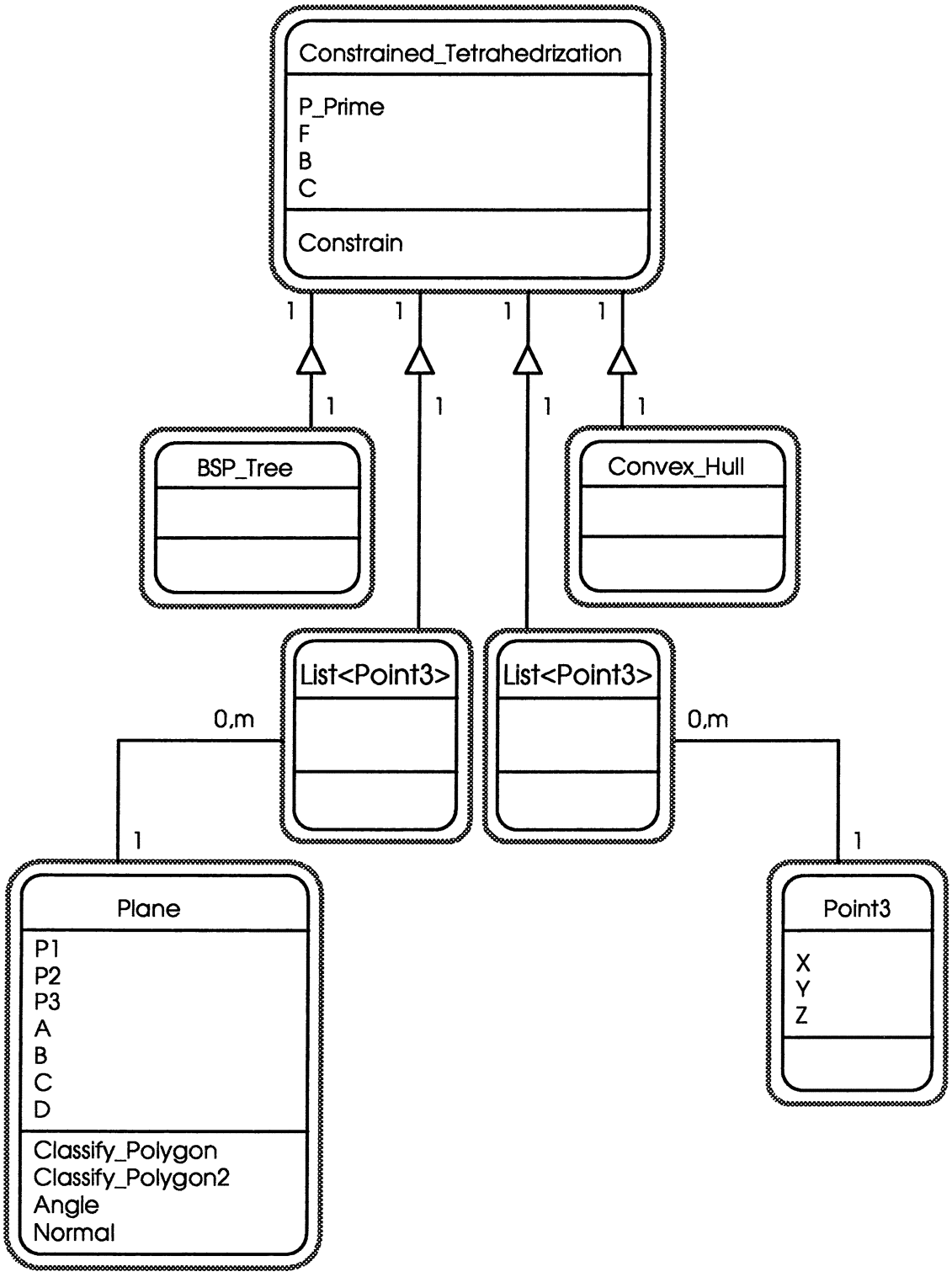


Figure 3-17 - Constrained_Tetrahedrization Class Representation and Object-&-Class Diagram

3.2.1.2 List Class

The List class is a representation of a double linked list of objects. Common constructors, destructors, accessors, and standard list services are provided with the class to make the class as open and re-usable as possible, but they are not shown using the Coad/Yourdon object model. The List class comes in several instances: List<Point3>, List<Point2>, List<Facet3>, List<Facet2>, List<SubFacet3>, List<SubFacet2>, and List<Plane>. All of theses instances were implemented in the OOP phase of the project through the use of C++ generic templates. Shown in Figure 3-18 is the object that this class represents and the Object-&-Class diagram:

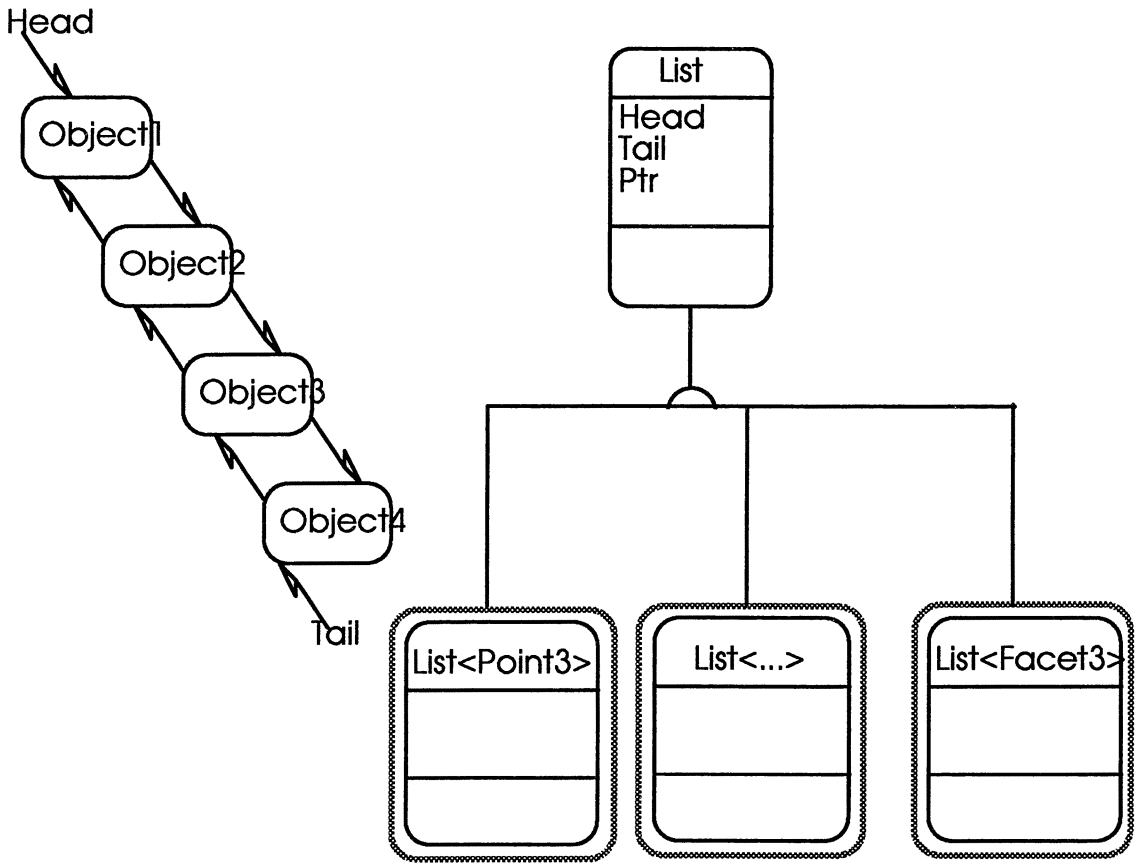


Figure 3-18 - List Class Representation and Object-&-Class Diagram

3.2.1.3 Convex_Hull Class

The Convex_Hull class is a representation of a geometric convex hull. A convex hull is represented geometrically by a list of facets which contain three vertices each. These facets are triangular and when they are all connected together they make up the facets or the outer surface of the convex hull. A more in depth explanation of convex hulls can be found in chapter 5. Common constructors, destructors, accessors, and

standard list services are provided with the class to make the class as open and re-usable as possible, but they are not shown using the Coad/Yourdon object model. In the OOP phase of the project, a single service called Giftwrapping[6,26] was developed that takes a List<Point3> class and an address to a List<Facet3> class as parameters and then fills in the List<Facet3> class with the facets of the convex hull. This was an improper design, and the Giftwrapping service has been kept, but the Convex_Hull class keeps the list of Facet3's internally. The Convex_Hull class is derived from the List<Facet3> class. Shown in Figure 3-19 is the object that this class represents and the Object-&-Class diagram:

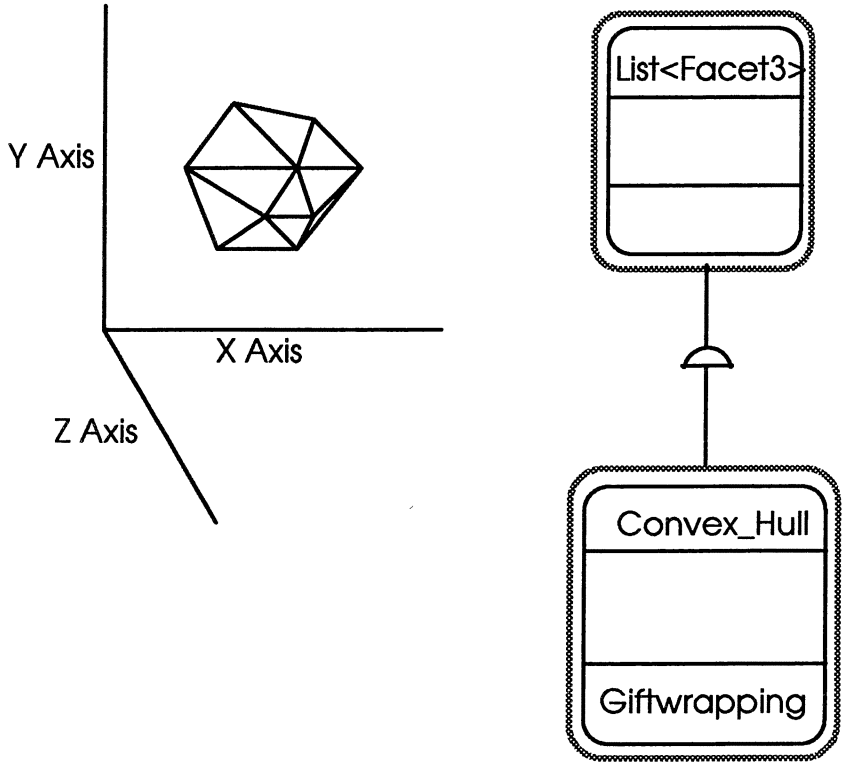


Figure 3-19 - Convex_Hull Class Representation and Object-&-Class Diagram

3.2.1.4 BSP_Tree Class

The BSP Tree class is a representation of a special binary tree called a binary space partition (BSP) tree. A BSP Tree is a data structure that represents a recursive, hierarchical subdivision of n-dimensional space into convex subspaces. BSP tree construction is a process which takes a subspace and partitions it by any hyperplane that intersects the interior of that subspace. The result is two new subspaces that can be further partitioned by recursive application of the method.[33] A more in depth explanation of BSP trees can be found in chapter 6. Common constructors, destructors, accessors, and standard list services are provided with the class to make the class as open and re-usable as possible, but they are not shown using the Coad/Yourdon object model.

Shown in Figure 3-20 and Figure 3-21 is an example of BSP tree construction in 2D and the Object-&-Class diagram:

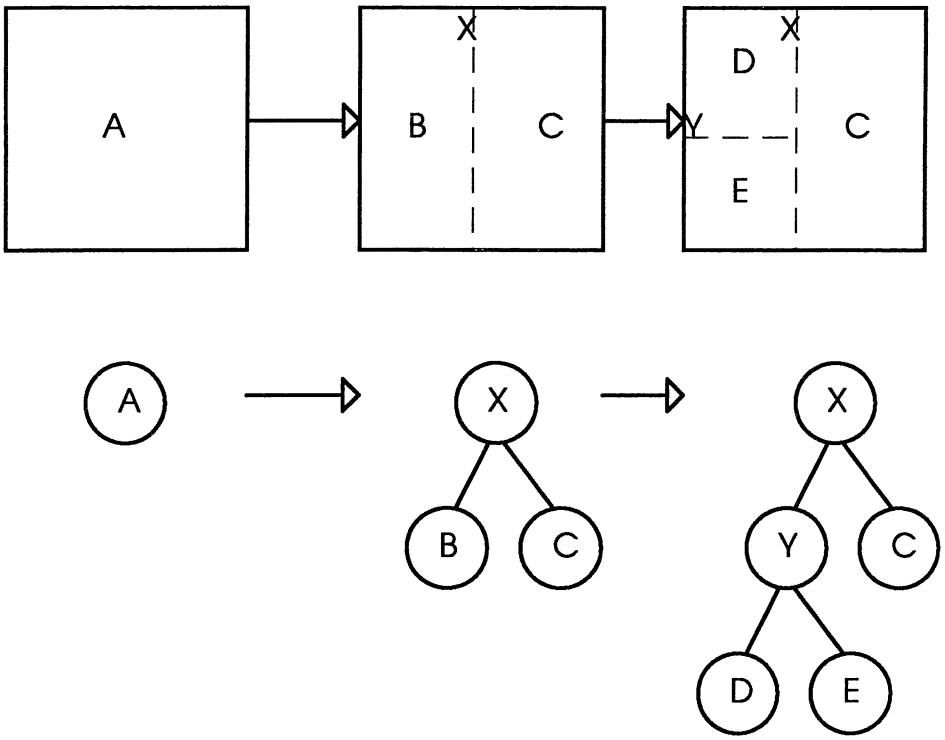


Figure 3-20 - Example of a BSP Tree Construction in 2D.[33]

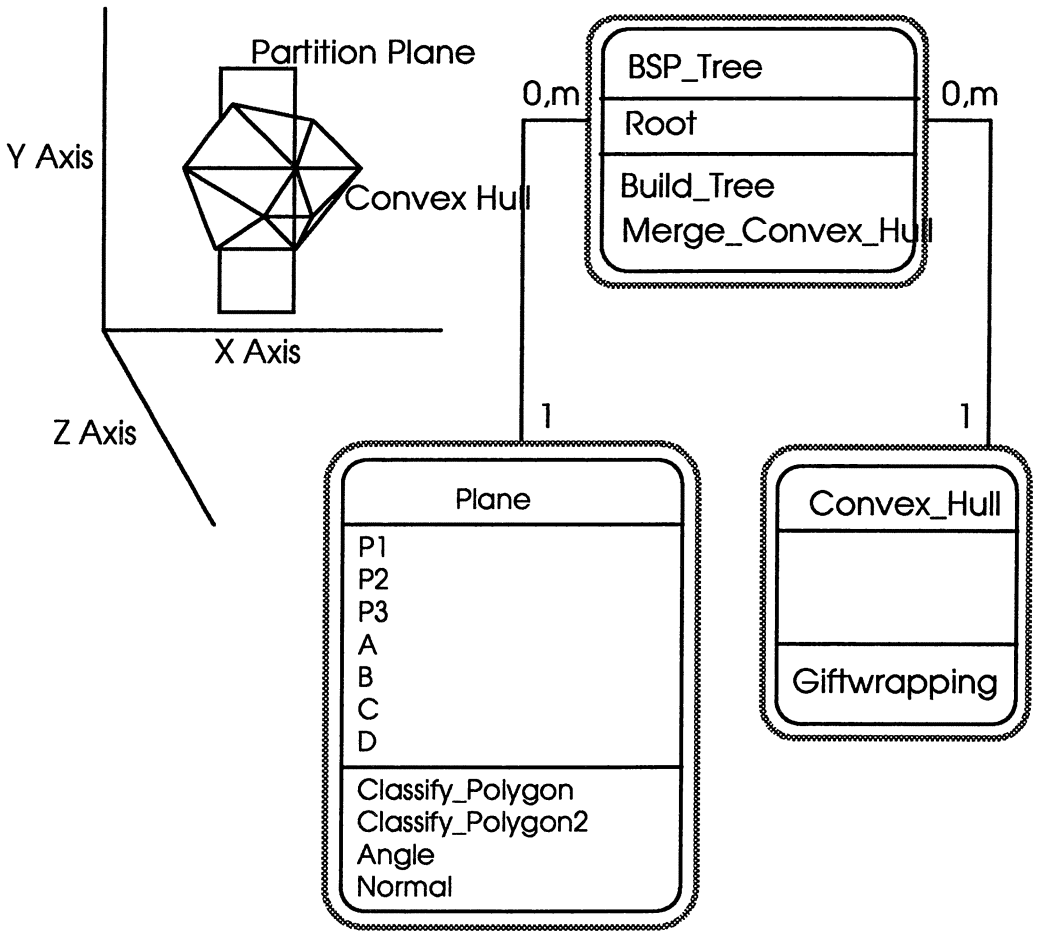


Figure 3-21 - BSP_Tree Class Representation and Object-&-Class Diagram

3.2.2 Human Interaction Component

The human interaction component of OOD is the activity where the design takes graphical user interface issues into design consideration. The Constrained Tetrahedrization project was designed to operate as a single thread of execution task using the standard input and standard output which is common to UNIX programs. There was only one class that was considered during the human interaction component of OOD which was a class that output geometric classes into to the standard OOGL output file format[23,32] so that they could later be displayed by the Geomview[15] X-Windows based 3D graphics viewing program.

3.2.2.1 OOGL Class

The OOGL class is an output class that creates OOGL files for later use with the Geomview package.[15,23,32] In the OOD phase of the project only the existence of this class was known about, because there was a requirement to view the results in a graphical manner, but the exact services that would be needed were not known. In the OOP phase

of the project, three services were developed to output OOGL files for three specific objects in the problem. The three objects that are able to be output in OOGL format are the Convex_Hull (i.e. List<Facet3>) class, the List<Plane> class, and the BSP_Tree class.

The first service, named Output_Convex_Hull, takes a List<Point3> class and a Convex_Hull (i.e. List<Facet3>) class as parameters and outputs an OOGL file named `chull.off` that will represent the Convex Hull graphically. This service could be updated in the future to only take a Convex_Hull (i.e. List<Facet3>) class as a parameter and generate the point list internally. The format[23,32] of this file is as follows:

```
OFF
n_p n_f n_e
p_1x p_1y p_1z
... {n_p points}
p_np x p_np y p_np z
3 f_1p_1 f_1p_2 f_1p_3
... {n_f facets}
3 f_nfp_1 f_nfp_2 f_nfp_3
```

Figure 3-22 - Sample chull.off OOGL File Output

Where n_p = number of points, n_f = number of facets, n_e = number of edges (set to 0 since it's not used in the current version of Geomview), the first set of real numbers are the points in the facets of the convex hull, and the second set of integer numbers are the facets. The only points that have to be listed in the OOGL file are the points actually in the facets, but for the convex hull object all of the points were listed in the OOGL file not only the external points, but also the internal points. The facets are then listed as integers which refer to the position of the points in the list.

The second service, named Output_Triangles, takes a List<Plane> class as an argument and outputs an OOGL file, named `triangs.off`, that will represent all of the partition planes graphically. The format[23,32] of this file is as follows:

```
LIST
{ = OFF
3 1 0
p_1x p_1y p_1z
p_2x p_2y p_2z
p_3x p_3y p_3z
3 0 1 2
}
{ = OFF
   ... {repeat for each object in List<Plane> class}
}
```

Figure 3-23 - Sample triangs.off OOGL File Output

The third service, named Output_BSP_Tree, takes a BSP_Tree class and a List<Point3> class as arguments and outputs an OOGL file, named bspchull.off, that will

represent the BSP_Tree partition planes and the multiple convex hulls that are in the BSP_Tree leaf nodes. The format[23,32] of this file is as follows:

```
LIST
{= OFF
  ... {partition plane information}
}
{ = OFF
  ... {repeat for each partition plane in the BSP_Tree class}
}
{ = OFF
   ... {convex hull information}
}
{ = OFF
  ... {repeat for each convex hull in the BSP_Tree class}
}
```

Figure 3-24 - Sample bspchull.off OOGL File Output

At the time of this writing, the third service has not been fully implemented because during the process in which the convex hull gets partitioned by the BSP tree, extra points, that are not in the original point set, are generated without point id's and are given a point id of 0 by default. The resulting output OOGL data, containing all of the extraneous points, is incorrect. A new service, much like the new service needed for the Output_Convex_Hull service, will have to be created which will generate the point lists internally and generate the correct OOGL output. Sample display output from the OOGL classes will be shown in chapter 4.

3.2.3 Task Management Component

The task management component of OOD is the activity where the design takes multiple tasks or concurrent behavior into design consideration. All services in this project are implemented in a single task. No task management component design is needed in this project.

3.2.4 Data Management Component

The data management component of OOD is the activity where the design takes the storage and retrieval of user objects from a data management system such as a database into design consideration. All services in this project are implemented to keep all attributes in local memory, no data persistence is needed. No data management component design is needed in this project.

Chapter IV

PROGRAM DESIGN

4. Program Design

The Object-Oriented Programming (OOP) implementation [6] of the program in the C++ language follows directly from the OOA and OOD design specifications as they were presented previously in chapter 3. The first step in the OOP process was to implement the OOA base classes in the C++ language, then to implement the rest of the OOA classes that were derived from the OOA base classes. After the OOP implementation of the OOA classes was completed to satisfaction, the OOP implementation of the OOD classes began. The OOP implementation of the OOD classes was a tedious task and took a long time, due to the algorithm complexities, to implement the code of the classes and to test each class as it was being developed. The BSP_Tree class and the Convex_Hull class were the most challenging to develop because of the complexity of the data structures and the complexity of the algorithms in those classes. The internal design of those two classes were complex enough and were crucial enough to the implementation of the Constrained Tetrahedrization algorithm[17] to be presented in chapter 5 and chapter 6. This chapter further explains the OOP implementation issues of the program.

4.1 OOA Base Class Program Implementation

The OOA base class design was easy to implement and the C++ code was very straight forward. The basic implementation of the Point2, Point3, Line_Segment2, Line_Segment3, SubFacet2, SubFacet3, Facet2, and Facet3 classes was produced in about two days worth of effort for all of the classes. Basic testing of the classes took about one day worth of effort for each class. Some of the implementation and testing of services in the Plane class took several weeks to work through due to the complexity of the Vector Calculus needed in some of the algorithms. Shown in Table 4-1 are the OOA base classes that were implemented and their corresponding filenames, these files are included in the appendix of this thesis:

Class	Implementation Filenames
Point2	point2.h, point2.cc
Point3	point3.h, point3.cc
Line_Segment2	lineseg2.h, lineseg2.cc
Line_Segment3	lineseg3.h, lineseg3.cc
SubFacet2	sfacet2.h, sfacet2.cc
SubFacet3	sfacet3.h, sfacet3.cc
Facet2	facet2.h, facet2.cc
Facet3	facet3.h, facet3.cc

Table 4-1 - Table of OOA Base Classes and Implementation Filenames

4.1.1 Standardized Implementation Style

For the base classes a standardized Object-Oriented C++ class implementation was followed in all of the classes. Each class would have all class attributes private and have public accessor services to all of the attributes. For example, shown below is a standard point2.h file that defines the Point2 class:

```
class Point2
{
   public:
      // Constructors & Destructors
      Point2();                  // Default Constructor
      Point2(double, double);    // Secondary Constructor
      Point2(Point2&);           // Copy Constructor
      ~Point2();                 // Default Destructor

      // Operators
      Point2& operator=(const Point2&);
      int operator==(const Point2&);
      int operator!=(const Point2&);
      friend ostream& operator<<(ostream&,Point2);

      // Accessor Services
      double GetX();
      void SetX(double);
      double GetY();
      void SetY(double);
      int GetID();
      void SetID(int);

      // Services
      double Distance(Point2);  // distance between 2 points

   private:
      // Attributes
      double X;
      double Y;
      int ID;
};
```

Figure 4-1 Point2 Base Class OOP Implementation Example

Notice the common elements of style in the class design. All private attributes are explicitly declared as private and not defaulted to private. Constructors are provided for standard allocation with no parameters, allocation with parameters, and copy constructors. The entire class is also indented and plenty of whitespace and comments are included to make the code more readable and maintainable. The Point3, Line_Segment2, Line_Segment3, SubFacet2, SubFacet3, Facet2, and Facet3 classes were all implemented in a similar style.

4.1.2 Plane Class Implementation

The Plane class implementation was a major undertaking considering the limited design and lack of information about Planes in Calculus and Geometry books. From the OOA design and a few Calculus books[3,5,13,25] all that is known about a plane is that we are given three points in E^3 to define the plane, and that the standard equation of a plane is Ax+By+Cz+D=0.

4.1.2.1 Plane Equations

The first problem that had to be overcome in the Plane class implementation was figuring out the standard equation of a plane given only 3 points that were in the plane. The first method that was implemented was Newell's Method, which was presented with implemented code in a Graphics Gems book[5,18]. For a first time quick implementation this seemed like it was the answer. It later turned out that the numbers produced by Newell's Method as implemented by the Graphics Gems book were not accurate enough for use by my project. Later on in the project a new method was developed that uses vectors to determine the plane equation.

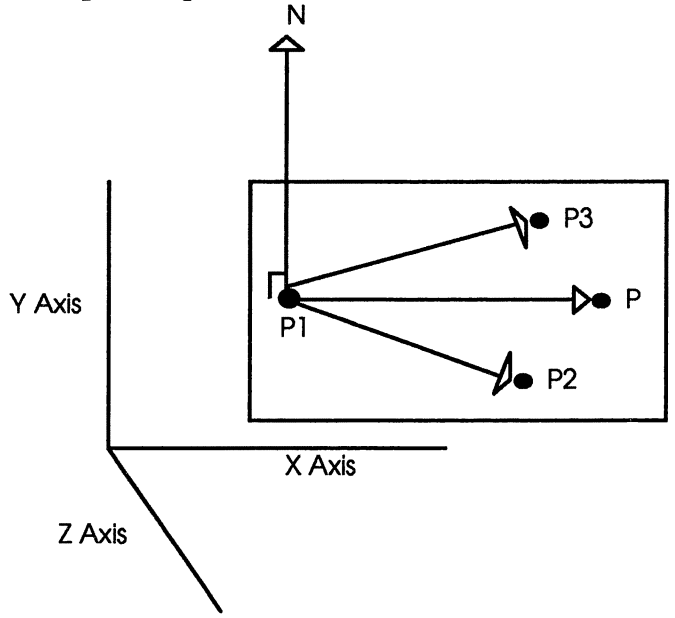


Figure 4-2 - Sample Plane with Vectors

The method of determining the plane equation using vectors is:

1. Form vectors P1P2 and P1P3.
2. Form the Normal vector N which is the cross product of vectors P1P2 and P1P3 (i.e. P1P2 X P1P3).
3. Given an arbitrary fourth point P in the plane, the dot product of vector P1P and N will be equal to 0 (i.e. P1P • (P1P2 X P1P3) = 0).
4. Solve for A, B, C, and D of the plane equation.

For example:

```
  Given the points {(0,1,1), (1,1,2), (-1,2,-2)}
P1P2 = (1-0)i + (1-1)j + (2-1)k
P1P2 = 1i + 0j + 1k
P1P3 = (-1-0)i + (2-1)j + (-2-1)k
P1P3 = -1i + 1j - 3k
N = (P1P2 X P1P3)
N = [(0)(-3)-(1)(1)]i - [(1)(-3)-(1)(-1)]j + [(1)(1)-(0)(-1)]k
N = -1i + 1j - 3k
P1P = (x-0)i + (y-1)j + (z-1)k
<(x-0)i+(y-1)j+(z-1)k>•<-1i+2j+1k> = 0
-1(x-0)+2(y-1)+1(z-1) = 0
-1x+0+2y-2+1z-1 = 0
-1x+2y+1z-3 = 0
The plane equation is: x-2y-1z+3 = 0
A = 1, B = -2, C = -1, and D = 3
```

Figure 4-3 - Example Plane Equation Calculation

This vector method of finding the plane equation translated very easily into a C++ implementation. The C++ implementation can be found in the file plane.cc which is included in the appendix as a reference.

4.1.2.2 Classify_Polygon Service of Plane Class

The Plane class contains two different types of Classify_Polygon services. These services take the current plane and either a second plane or a Point3 and determines if the second object is in front of, in back of, coincident to, or spanning the first plane object. These services have been described in some detail in section 3.1.3 and diagrams that graphically depict these two services are shown in Figure 3-4 and Figure 3-5.

The first service was first implemented by testing the signed distance of each point in the second plane to the first plane and then categorically determining if the other plane was spanning (differing signs), in front (all signs positive), in back (all signs negative), or coincident (all distances = 0 ± a tolerance). This would be the correct method to determine how a polygon interacted with a plane, but this method does not work to determine how two planes interact with each other. If all of the points of the second plane are on one side of the second plane then either IN_FRONT or IN_BACK would

have been returned, but that does not guarantee that the two planes will not intersect further out in space. This first implementation method of the first service was abandoned in favor of another method.

The first service determines if the second plane is parallel to the first plane. If the planes are parallel then the two planes are checked for coincidence. If the two planes are coincident then the value COINCIDENT is returned. If the two planes are parallel then the signed distance from one point in the second plane is tested from the first plane, and either IN_FRONT or IN_BACK is returned depending on the sign. Since planes are infinite in nature, if the planes are neither coincident nor parallel, then they must intersect somewhere so the value SPANNING is returned.

The second service determines the signed distance from a point to the plane. If the sign of the distance is positive and the distance is greater than 0 (± a tolerance) then the value IN_FRONT is returned. If the sign of the distance is negative and the distance is greater than 0 (± a tolerance) then the value IN_BACK is returned. If the point lies in the plane (± a tolerance) then the value COINCIDENT is returned. The value SPANNING is not returned by this service because geometrically speaking a point can not span a plane.

4.2 OOD Derived Class Program Implementation

The OOD derived class design implementation in C++ code was of average difficulty and pretty straight forward. The first basic implementation of the Constrained_Tetrahedrization, List, Convex_Hull, BSP_Tree, and OOGL classes was produced in about three days worth of effort for each class. However, the basic testing of the classes took several weeks worth of effort. A lot of the services in these classes had major bugs in some of their implementations that needed to be fixed. Shown in Table 4-2 are the OOD derived classes that were implemented and their corresponding filenames, these files are included in the appendix:

Class	Implementation Filenames
Constrained_Tetrahedrization	ctz.h, ctz.cc
List	list.h, list.cc
Convex_Hull	chull.h, chull.cc
BSP_Tree	bsptree.h, bsptree.cc
OOGL	oogl.h, oogl.cc

Table 4-2 - Table of OOD Derived Classes and Implementation Filenames

4.2.1 Constrained_Tetrahedrization Class Implementation

The Constrained_Tetrahedrization class (CTZ) is the main class that is in charge of implementing the Constrained Tetrahedrization algorithm.[17] The design of this algorithm was discussed previously in section 3.2.1 and section 3.2.1.1 of this thesis. The CTZ class has one service called Constrain, which takes a List<Point3> class and a List<Plane> class as inputs and it implements the Constrained Tetrahedrization algorithm

producing a BSP Tree with partition planes in the inner nodes and convex hulls in the leaf nodes as an output. Presented in Figure 4-4 is the C++ style pseudocode for the Constrain service of the CTZ class:

```
//constrain(P', F, T)
//   begin
//      1. construct B, the BSP Tree of F;
//      2. construct C, the convex hull of P';
//      3. for j = 1 to k do
//            Kj <- plane(fj) intersect C;
//      4. for j = 1 to k do
//            compute constraints for Kj;
//      5. for j = 1 to k do
//            triangulate Kj with constraints;
//            triangulate facets of C with constraints;
//      6. for i = 1 to l do
//            Ti <- tetrahedrization of Ri;
//            T <- T union Ti;
//   end.
```

Figure 4-4 - Constrain Service Pseudocode

Currently, the CTZ class only implements a few steps of the Constrained Tetrahedrization algorithm. Step 1 is implemented by creating a new BSP_Tree class and constructing the BSP tree partition plane structure using the List<Plane> class that is passed into the CTZ class. The details of BSP tree construction are presented in chapter 6. At this point in the algorithm the BSP tree leaf nodes are empty since the partitions are not partitioning any objects. Step 2 is implemented by creating a new Convex_Hull class and defining the facets of the convex hull using the List<Point3> class that is passed into the CTZ class. The details of convex hull construction are presented in chapter 5. Step 3 is implemented by merging the convex hull into the BSP tree. The details of merging objects into a BSP tree are presented in chapter 6. After the merging is complete, the BSP tree leaf nodes will have convex hull's inside of them representing the part of the convex hull that resides in that space partition. Step 4 and beyond of the Constrained Tetrahedrization algorithm has not been completed due to time constraints on this thesis. The completion of this algorithm is left for future expansion.

4.2.2 List Class Implementation

The List class is the major container class in this project. Every major class in the project uses this class to store a list of objects as an attribute. This class was mainly developed as a specialized convex hull class because a convex hull is defined as a list of facets. It is also useful to hold onto a list of points instead of having to work with arrays which are not flexible enough due to fixed limits on array size. This List class was implemented using C++ templates to give it a generic behavior and allow it to store objects of any type. On future projects it is recommended that a pre-developed and pre-tested list class be used instead of developing one from scratch. Possibly the list class that comes as part of the popular Standard Template Library (STL) tool kit would be enough.

4.2.3 Convex_Hull Class Implementation

The Convex_Hull class is a major part of implementing the Constrained Tetrahedrization algorithm. This class was modified from a FORTRAN algorithm and an ANSI C algorithm which were developed during previous course work. The convex hull wasn't an original class in this project, but it existed as a specialized service called Giftwrapping. This Giftwrapping service was an algorithm adapted from previous course work and from the text book[26] for that course. It was later determined during the OOD phase of the project that a convex hull really was its own class, and it should be derived from List<Facet3> or List<Facet2> depending on if the convex hull was in 3D or 2D space using the Giftwrapping routine as a service of the class. The details of convex hull construction are presented in chapter 5. This class was difficult to implement and test because of the plane equations discussed in section 4.1.2.1 and because of the method of determining the angle between two planes. Once the Plane class was fixed, the Convex_Hull class was easily debugged and tested. The convex hull that was generated by the Giftwrapping service was compared against the convex hull that the quickhull[28] program computed for the same point set. From 100 random point test data set, both programs generated the same 56 facet convex hull.

4.2.4 BSP_Tree Class Implementation

The BSP_Tree class is a major part of implementing the Constrained Tetrahedrization algorithm. The BSP tree is a data structure that was first developed for the computer graphics field to partition graphical objects for hidden surface removal and therefore faster object rendering. The initial information about BSP trees was obtained from an ACM SIGGRAPH paper[22] and from the Internet at a site that maintains the BSP Tree FAQ[33]. The details of BSP tree construction are presented in chapter 6. This class was difficult to implement and test because by the initial class design the storage for the partitioning planes and the partitioned objects resided in the internal nodes of the tree. This design made the algorithms for tree traversal difficult and harder to implement, also the algorithms for convex hull partitioning were difficult and harder to implement. After researching the BSP tree material further, it was found that the partitioned objects should only reside in the leaf nodes of the tree while only the partitioning planes reside in the internal nodes of the tree. After redesigning the class appropriately, the implementation and testing became easier for the tree traversal and the convex hull partitioning routines.

4.2.5 OOGL Class Implementation

The OOGL class is the only class that fell into the Human Interaction Component of the OOD design. It is not a true Graphical User Interface (GUI) class because its job is to take geometric objects and output their description in the OOGL description language[23,32] which will then later be displayed by the Geomview program[15] which acts like the GUI. The design of this class really is not Object-Oriented, but it is more functional in nature. The idea of a class that has no attributes, but just has services that

take parameters in and give output is a functional design. This class should be removed and the services should be distributed to the classes that need them. For example the service OOGL::Output_Convex_Hull should be removed and it should become the service Convex_Hull::OOGL_Output which is a more Object-Oriented approach to the problem. This redesign will be left as an opportunity for future expansion.

4.3 Other Implementation

Other things that were implemented that were not included in the class design were the functional design services that are in the files general.h, general.cc, chsplit.h, and chsplit.cc. The functions presented in general.h and general.cc that are interesting to the design are functions that test float and double numbers within a tolerance to avoid round off error in calculations involving floating point numbers. Also of interest are several functions that do different matrix operations which are used by several services in the Plane class and the Convex_Hull class to assist in vector calculations. The chsplit.h and chsplit.cc files provide a service to the BSP_Tree class that will take a Plane object and a Convex_Hull object as a parameter and it will split the Convex_Hull object into two halves. This particular function should be redesigned in an Object-Oriented manner and placed into either the Plane class or the BSP_Tree class. This redesign will be left as an opportunity for future expansion.

4.4 Test Driver Implementation

In general, the use of test drivers at the unit test phase of the project to test class implementations is extremely useful in an Object-Oriented system. In the classical functional systems a light unit test phase and a heavy integration testing phase is normal, but in the Object-Oriented world the heavy use of the unit test phase makes the integration testing phase much easier. Having worked on projects that have taken both approaches, the Object-Oriented approach to testing is greatly preferred. Those work experiences influenced the project setup to make heavy use of unit testing to simplify the integration testing phase of the project. This section describes some of the test drivers that were used to test some of the class implementations.

4.4.1 Test_Pla Driver

The Test_Pla driver tests the Plane class implementation by creating planes from three E^3 coordinates and then testing the generated plane equation against the one obtained by hand using Calculus. It was very important to test the Plane class thoroughly since it is so heavily used in all of the other classes of the project.

4.4.2 Test_Lis Driver

The Test_Lis driver tests the generic List class implementation by creating lists of integers and then testing the services of the class by inserting and deleting elements of the list at the head, tail, and middle of the list. It was very important to test the generic List class thoroughly since it is so heavily used in all of the other classes of the project for object storage needs.

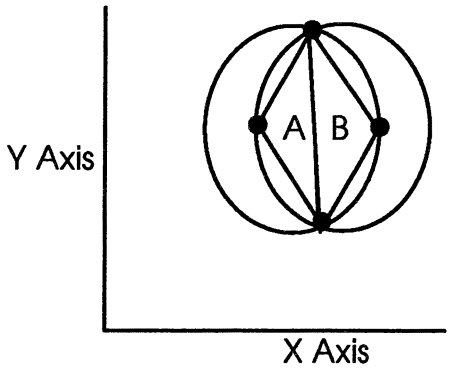
4.4.3 Test_Del Driver

The Test_Del driver tests the Delaunay Triangulation module of the code by running several different sets of triangles through the module and testing the output against the expected Delaunay output which was computed by hand. The Delaunay Triangulation module of the code is used when splitting one large facet into two smaller facets. This method is used to generate new facets from split facets when the convex hull is being split by a partitioning plane in the BSP_Tree class.

4.4.3.1 Delaunay Triangulation

Delaunay triangulation is a method of arranging a network of non-overlapping triangles, which have shared sides, where no points in the triangulation network are enclosed by circumscribing circles of any triangle.[2,26,29] A singular triangle only has one unique Delaunay triangulation. In E^2 a Delaunay triangulation is best represented by an illustration of two triangles that have one shared side. The joined triangles which share a long side which causes a circumscribed circle to enclose the last point of the other triangle is not a Delaunay triangulation. The joined triangles which share a short side which do not cause a circumscribed circle to enclose the last point of the other triangle is a Delaunay triangulation. Shown in Figure 4-5 is a graphical representation of a non-Delaunay triangulation and a Delaunay triangulation.

Non-Delaunay Triangulation
Circle around triangle A encloses third point from triangle B
Circle around triangle B encloses third point from triangle A



Delaunay Triangulation
Circle around triangle A does not enclose third point from triangle B
Circle around triangle B does not enclose third point from triangle A

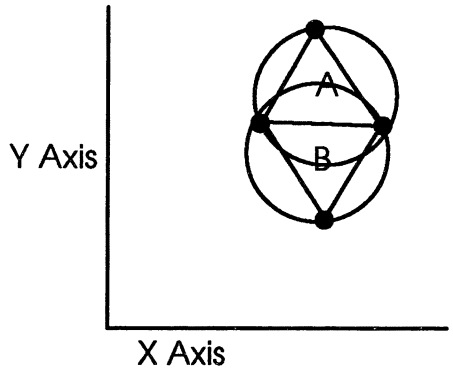


Figure 4-5 - Delaunay Triangulation

4.4.4 Test_Ctz Driver

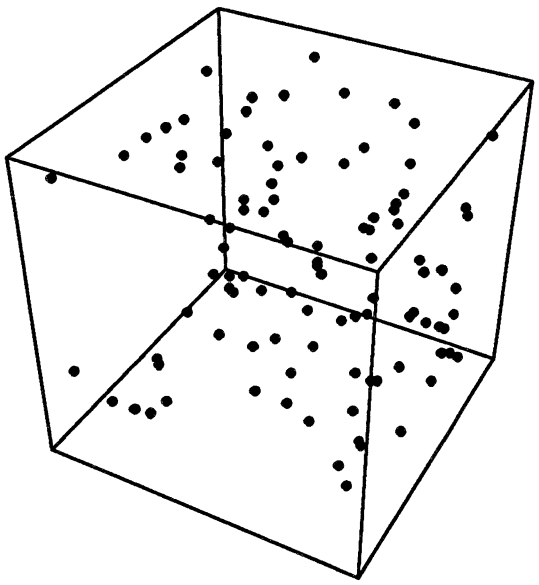
The Test_Ctz driver tests the Constrained_Tetrahedrization (CTZ) class implementation by running a list of 100 random points in 3D (x, y, and z range 0 to 1) and 5 random partition triangles (planes) (all points in range 0 to 1) through the Constrain service of the class. Since the CTZ module is the main module of the code that implements the algorithm[17], this driver became the main means of running and testing the project output. The Test_Ctz driver only requires two files as input, points.dat which contains a list of 3D points terminated by a origin point (0.0, 0.0, 0.0), and triangle.dat which contains a list of 3D triangles terminated by a zero triangle (three origin points). The Test_Ctz driver then creates a Constrained_Tetrahedrization class and calls the

Constrain service with the List<Point3> class and the List<Plane> class that were created from the input files as parameters.

The actual data file that contains the list of points and the list of triangles are shown in the appendix for reference. The Mathematica program by Wolfram Research Inc. is a system of doing Mathematics by computer.[36] Mathematica is a computer program that was used to prove some of the equations used in some of the algorithms or that was used to show quick graphical output for visual purposes. Shown in Figure 4-6 are the Mathematica commands and the Mathematica output that graphically show the 100 random points and the 5 random partition triangles (planes) that were used to test the project implementation through the Test_Ctz driver:

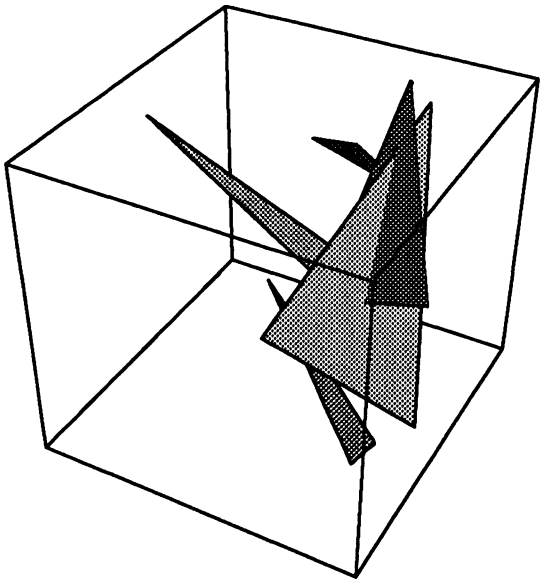
```
In[1]:=
   t1 = ReadList["points.dat", Number, RecordLists->True];
   pts = Map[Point,t1];
   Show[Graphics3D[pts]]

Out[1]=
```



```
In[2]:=
   t2 = ReadList["triangle.dat", Number];
   t2 = Partition[t2,3];
   t2 = Partition[t2,3];
   tris = Map[Polygon,t2];
   Show[Graphics3D[tris]]

Out[2]=
```



```
In[3]:=
   Show[Graphics3D[pts],Graphics3D[tris]]

Out[3]=
```

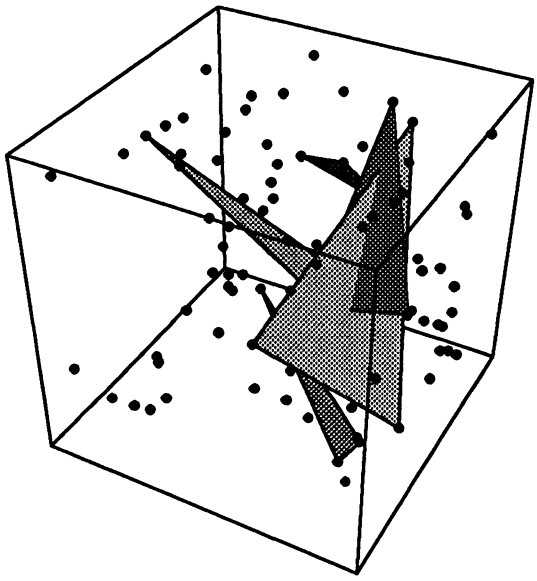


Figure 4-6 - Mathematica Graphical Output of Sample Project Input Data

4.5 Final Implementation Testing

All of the final implementation testing was done using the Geomview package[15] to view the OOGL files[23,32] that were produced by running the test_ctz test driver. Shown in Figure 4-7 is a sample screen of the Geomview program displaying a dodecahedron (12 sided polygon).

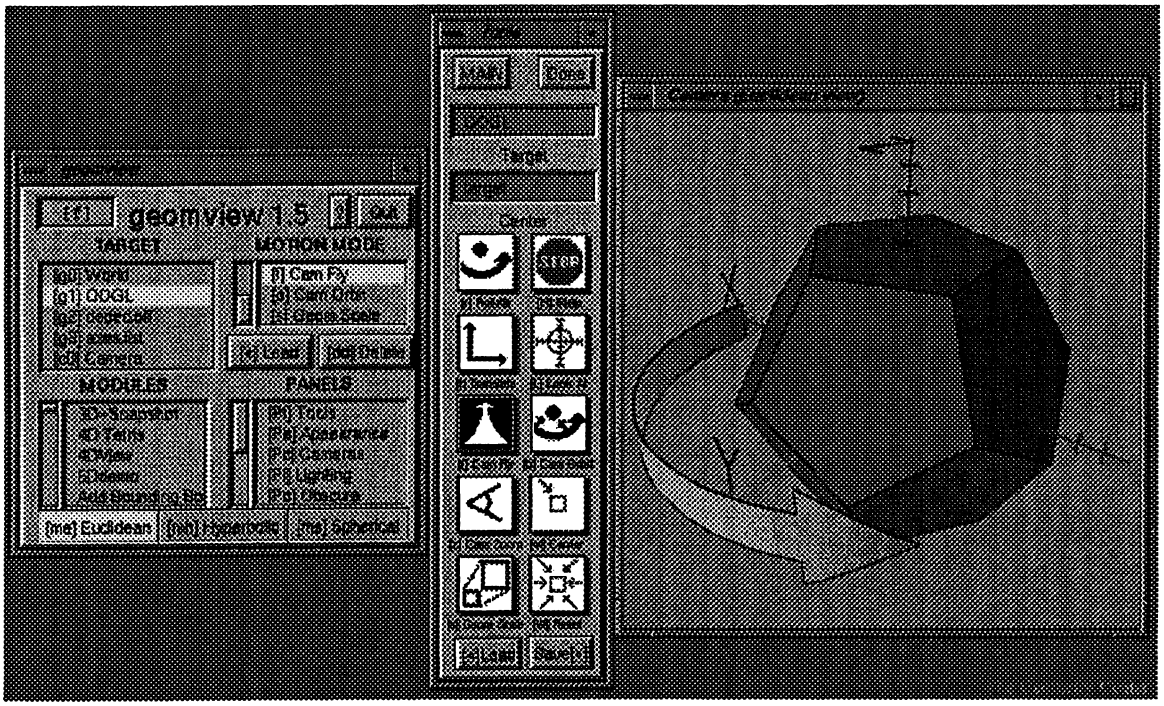


Figure 4-7 - Example Geomview screen [15]

Shown in Figure 4-8 is a Geomview Camera of the five partition planes that were used to construct the BSP Tree and partition the convex hull that is shown in Figure 4-9. Shown in Figure 4-10 is the convex hull after being partitioned by one partition plane during testing.

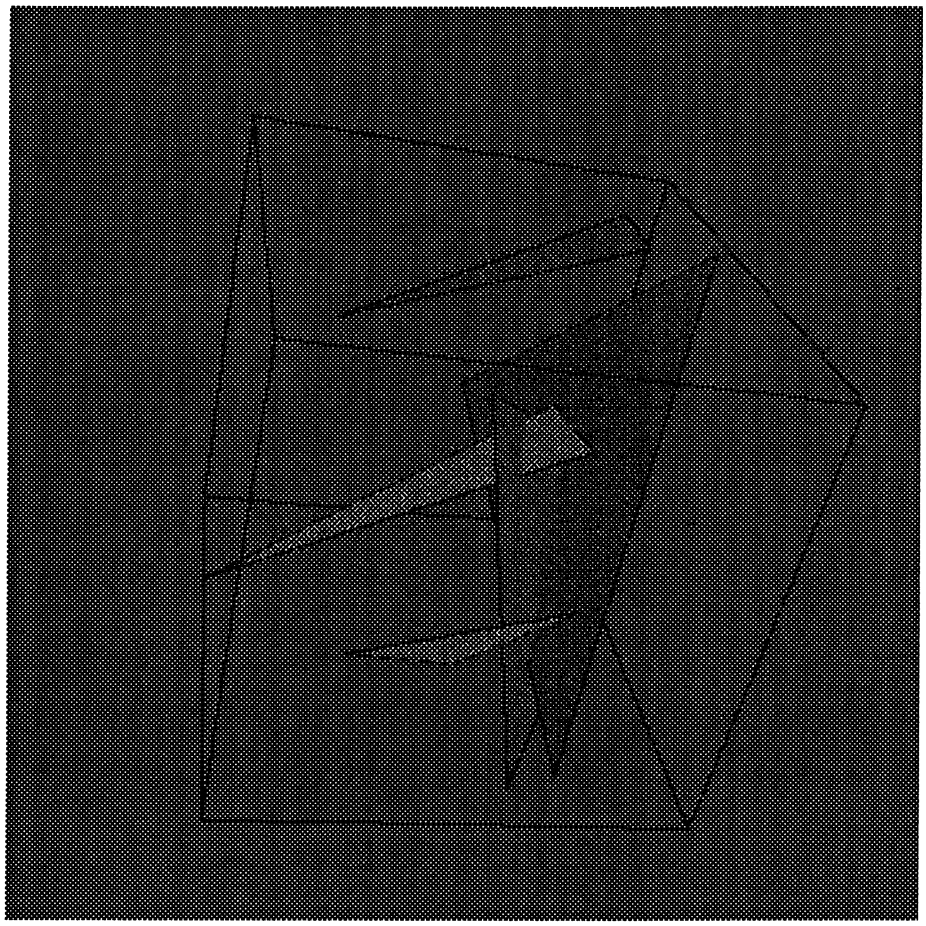


Figure 4-8 - Geomview Camera View of the Five Partition Planes

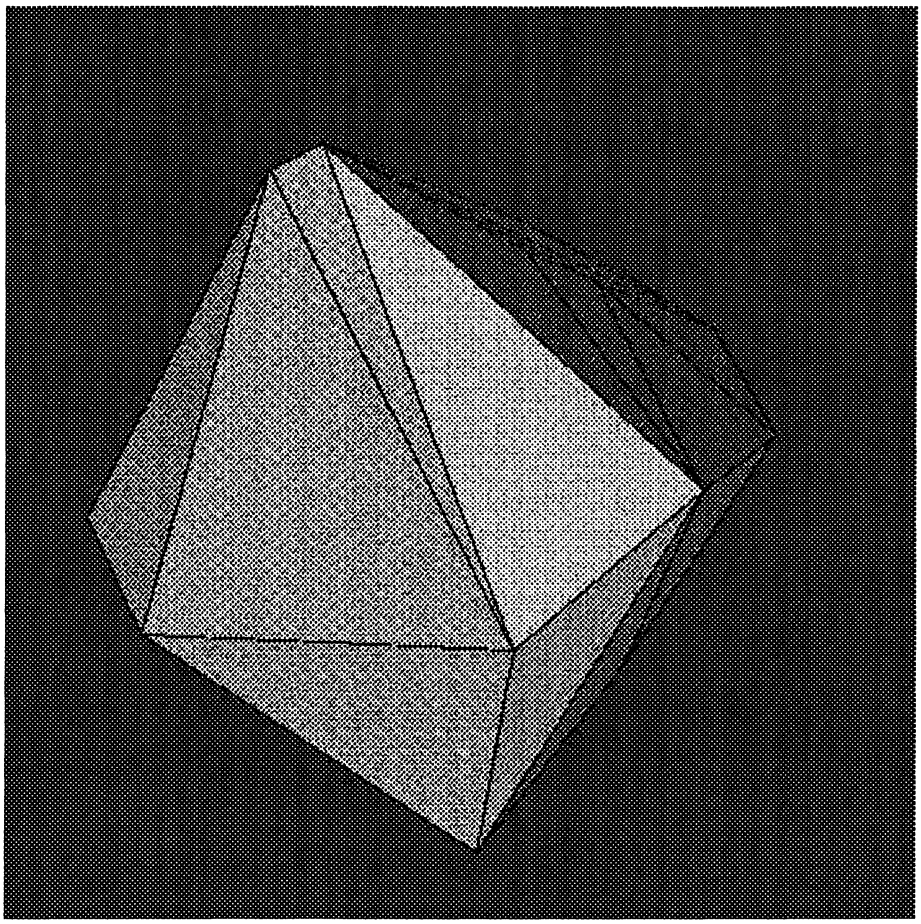


Figure 4-9 - Geomview Camera View of the Convex Hull

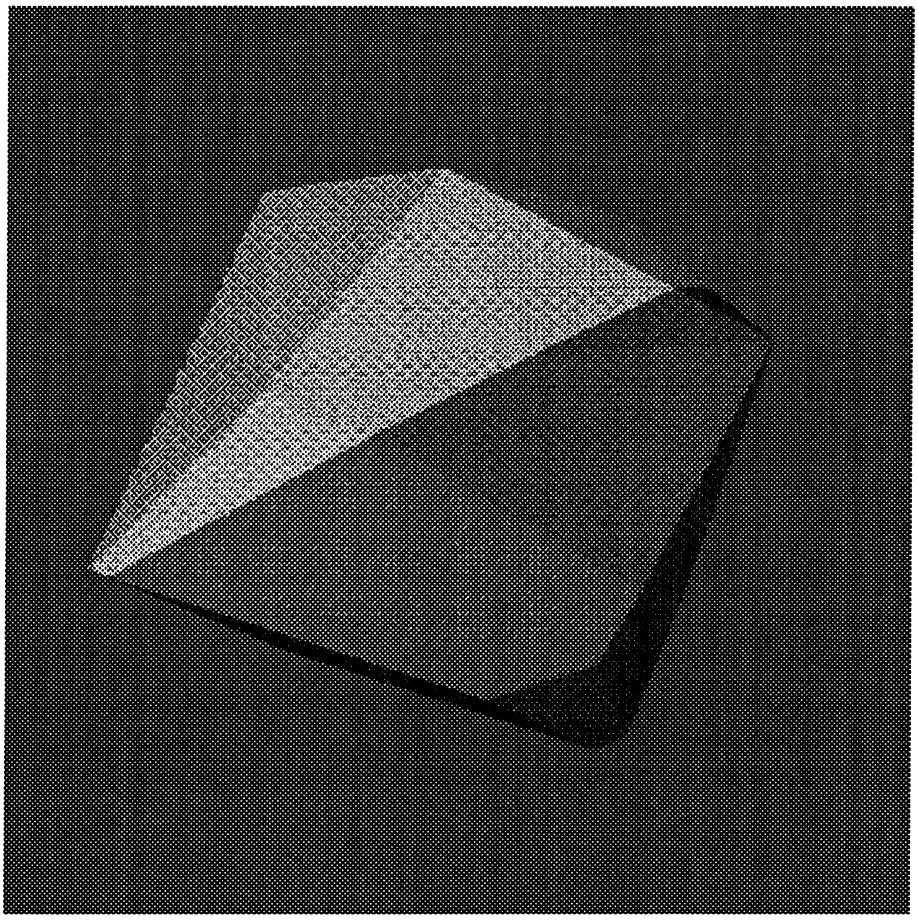


Figure 4-10 - Geomview Camera View of the Convex Hull Split by One Plane

Chapter V

Convex Hulls

5. Convex Hulls

The convex hull of a set of points P is the smallest convex set that contains P.[26,29] For example, if P = $\{v_1, v_2, v_3, v_4\}$ in E^2 where v_4 is an interior point, then the convex hull of P is a triangle containing the vertices v_1, v_2, and v_3 as shown in Figure 5-1.

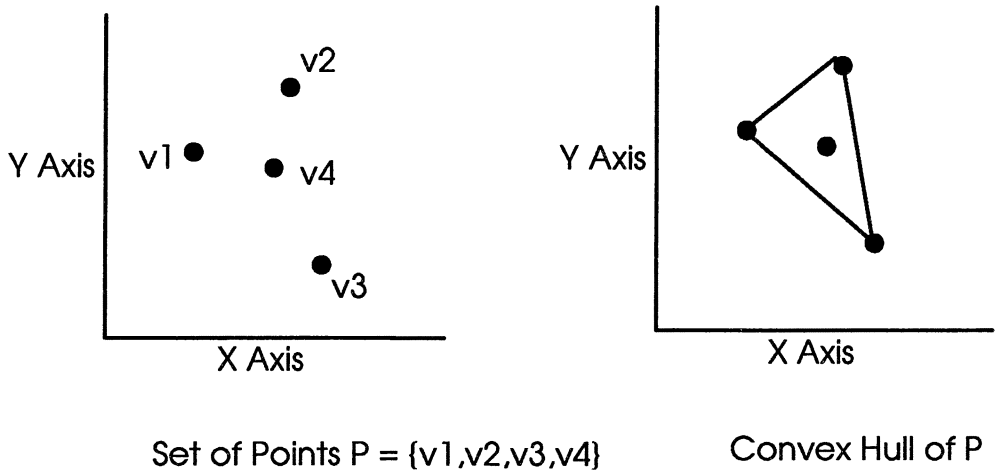


Figure 5-1 - Example of a Set of Points P in E^2 and the Convex Hull of P

By definition, the set of points P is a finite set. A good working concept of a convex hull in E^2 can be illustrated by stretching a rubber band around all of the points in P and then letting go of the rubber band and letting the rubber band form around the outside of the points.[26] In E^2, if the number of points n = 1, then the convex hull is the point itself. In E^2, if the number of points n = 2, then the convex hull is the line segment joining the two points. In E^2, if the number of points n = 3, then the convex hull is a triangle around the three points. In E^2, if the number of points n > 3, then the convex hull is a convex polygon with the number of sides >= 3. For the purposes of this project, the minimum convex hull could only be computed with the number of points n >= 3 in E^2. In

E^2, a facet is a line segment and a subfacet is a point. Shown in Figure 5-2 is an example of a convex hull with $n > 3$ in E^2.

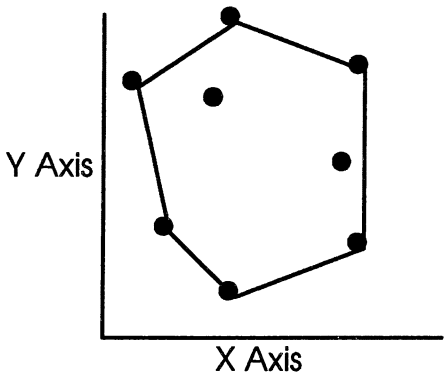


Figure 5-2 - Example of a Convex Hull with $n > 3$ in E^2

5.1 Convex Sets

Set A is convex if for every pair of points p and q in set A, the line segment pq is in set A.[29] For example in Figure 5-3, the set on the left is a convex set because the line segment pq resides entirely in set A. The set on the right is not a convex set because the line segment pq does not reside in set A.

Figure 5-3 - Example of a Convex Set and a Non-Convex Set

5.2 Convex Hulls in Three Dimensions

Convex hulls in three dimensions have the same basic definition as a convex hull in two dimensions. The convex hull of a set of points P is the smallest convex set that contains P.[26] By definition, the set of points P is a finite set. A good working concept of a convex hull in E^3 can be illustrated by stretching a piece of plastic wrap around all of the points in P and then shrink wrapping the plastic wrap around the outside of the points.

In E^3, if the number of points n = 1, then the convex hull is the point itself. In E^3, if the number of points n = 2, then the convex hull is the line segment joining the two points. In E^3, if the number of points n = 3, then the convex hull is a triangle around the three points. In E^3, if the number of points n > 3, then the convex hull is a convex polygon with the number of sides >= 3. For the purposes of this project, the minimum convex hull could only be computed with the number of points n >= 4 in E^3. In E^3, a facet is a triangle and a subfacet is a line segment. Shown in Figure 5-4 is an example of a convex hull with n > 4 in E^3.

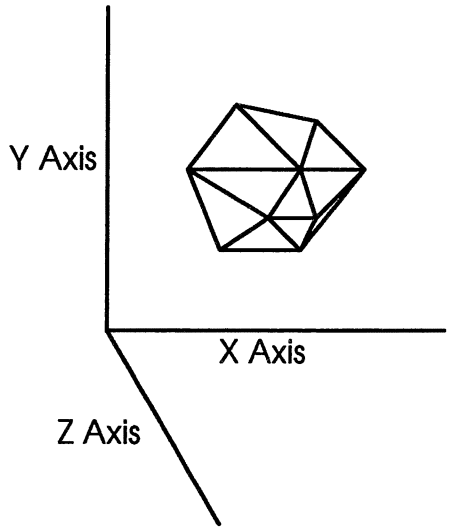


Figure 5-4 - Example Convex Hull with n > 4 in E^3

5.3 Convex Hull Algorithm

The algorithm to compute a convex hull given a set of points in E^3 was presented in Computational Geometry an Introduction and presented as the "gift-wrapping" method.[26] The algorithm is an extension of the standard beneath and beyond algorithm adapted to three dimensions by checking beneath and beyond on hyper-planes instead of individual points as in the two dimensional version of the algorithm. The basic idea is to proceed from a facet to an adjacent facet, in the guise in which one wraps a sheet around a plane-bounded object.[26]

The algorithm starts out by finding the initial facet in the convex hull. After finding this initial facet, it then builds half-planes to the other points in the set that are not in the initial facet in order to find the half-plane which forms the largest angle to the initial facet. It then builds another facet with this half-plane and continues on from there building facets until the convex hull is an enclosed faceted polygon.

The initial facet is built by first selecting the point of the least x, y, z value and calling it P1 of the initial facet. The hyperplane of maximum angle that is built from the first point P1 and orthogonal to vector n = <1,0,0> and vector a = <0,1,0> is used to find

the second point in the initial facet calling it P2. The P1 P2 line segment is called a subfacet of the initial convex hull facet. The hyperplane that is built from the first subfacet and is orthogonal to vector new n and also to vector new a, which is re-computed using the maximum angle and the new normal vector, is used to find the third point in the initial facet calling it P3. The P1 P2 P3 triangle is called the initial convex hull facet and each side of the facet are called subfacets of the initial convex hull facet. The algorithm time to find the initial facet in a convex hull is O(N log N) + C for finding P1, O(N) + C for finding P2 and O(N) + C for finding P3. This makes a total time for finding the initial facet of O(N log N) + O(2N) + C. According to an analysis shown in Preparata and Shamos[26] the time to build the initial facet should be $O(Nd^2) + O(Nd^3) = O(Nd^2)$ since N >= d.

After the initial convex hull subfacet has been found, the building of the rest of the convex hull facets comes from there. By the nature of how triangular facets work, each subfacet in a convex hull facet can only be part of two facets. In the rest of the algorithm if a subfacet already belongs to two facets then it is considered completed and can never be revisited again. If the subfacet only belongs to one facet, then hyperplanes may be formed in the same manner as was done for finding P3 of the initial facet and selecting the hyperplane that has the maximum angle. Once the new facet is formed, the subfacet that was used to join the new facet to the original facet is considered complete and the two new subfacets that are formed are used to keep the algorithm going. The algorithm completes when there are no more subfacets to process. At this point every facet will have been formed and the convex hull will be an enclosed faceted polygon. According to an analysis shown in Preparata and Shamos[26] the time to build the convex hull of a set of N points in d-dimensional space using the gift-wrapping technique is $O(N\varphi_{d-1}) + O(\varphi_{d-2}\log \varphi_{d-2})$ on average, where φ_{d-1} is the number of facets in the hull and φ_{d-2} is the number of subfacets in the hull, and $O(N^{\lfloor d/2 \rfloor + 1}) + O(N^{\lfloor d/2 \rfloor} \log N)$ on worst case. In E^3 this translates to $O(N^2) + O(N \log N)$, which simplifies to $O(N^2)$ on worst case.

5.4 Splitting Convex Hulls

One of the steps in the algorithm[17] involves merging a convex hull into a BSP tree. At the heart of this merging is to split the convex hull with the partition plane at each node of the BSP tree. The algorithm that recurses through the tree and does the splitting will be discussed in more detail in chapter 6. This section will discuss splitting a convex hull with a single plane. In order to more fully understand how to split a convex hull with a plane, we will start off with some simple two dimensional examples and move towards the more complex three dimensional examples.

5.4.1 Splitting in E^2

In E^2 the problem becomes how to split the convex hull with a line. The standard algorithm for splitting a convex hull is to test every facet against the partition and to lump the facets into one of four categories. These categories are: In Front, In Back, Spanning,

and Coincident. Both In Front and In Back mean that the facet lies entirely on one side or the other side of the partition. Spanning means that the facet is split by the partition and Coincident means that the facet lies inside of the partition. Once every facet is categorized, all of the In Front facets go into the front convex hull and all of the In Back facets go into the back convex hull. The facets that are Coincident go into both the front and the back convex hull. The facets that are Spanning need to be split by the partition and the part of the facet that is In Front goes into the front convex hull and the part of the facet that is In Back goes into the back convex hull. Once this entire categorization is completed any points that ended up inside of the partition because of splitting or Coincidence will become facets of both convex hulls. Shown in Figure 5-5 is an example of convex hulls in E^2 with facets that are In Front or In Back, Spanning, and Coincident.

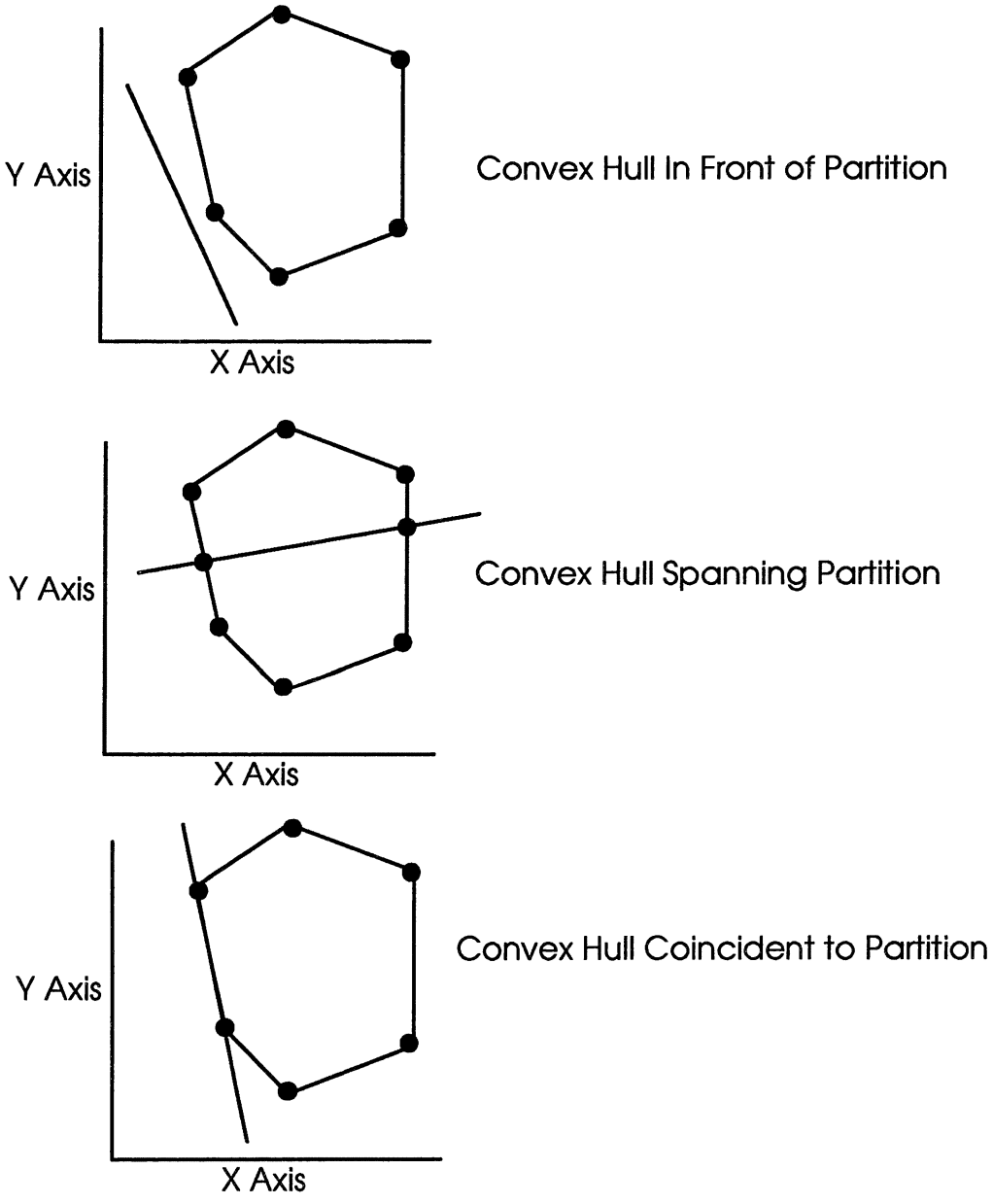


Figure 5-5 - Examples of Partitioning Convex Hulls in E2

The middle convex hull in Figure 5-5 depicts a convex hull that is Spanning the partition line. In this convex hull the categorization of the facets was that only two of the original convex hull facets are Spanning the partition line, two of the original convex hull facets were In Front of the partition, and two of the original convex hull facets were In Back of the partition. After the partitioning of this convex hull, the result is two new convex hulls, one in front of the partition and one in back of the partition. The front convex hull contains the two In Front facets of the original convex hull, plus three new facets to complete the front convex hull. The new facets are the two facets that are created by the partition plane splitting the facet in half and one facet that lies in the partition plane. In like manner, the back convex hull contains the two In Back facets of the original convex hull, plus three new facets to complete the back convex hull. The new facets are the two facets that are created by the partition plane splitting the facet in half and one facet that lies in the partition plane.

5.4.2 Splitting in E^3

In E^3 the problem is a little more difficult than in E^2 because the problem now becomes how to split the convex hull which looks like a faceted ball with a plane. Fortunately the standard algorithm for splitting a convex hull in E^3 is the same as it is in E^2. The standard algorithm for splitting a convex hull is to test every facet against the partition and to lump the facets into one of the four categories. The first thing that is different about convex hulls in E^3 is that the facets are triangles instead of line segments like they are in E^2. In E^2 it is almost trivial to calculate the intersection of a line and a line segment, but in E^3 the problem becomes how to compute the intersection of a plane with a triangle (almost like a plane) in E^3. Shown in Figure 5-6 is an example of a plane and a single facet in E^3 that is In Front or In Back, Spanning, and Coincident.

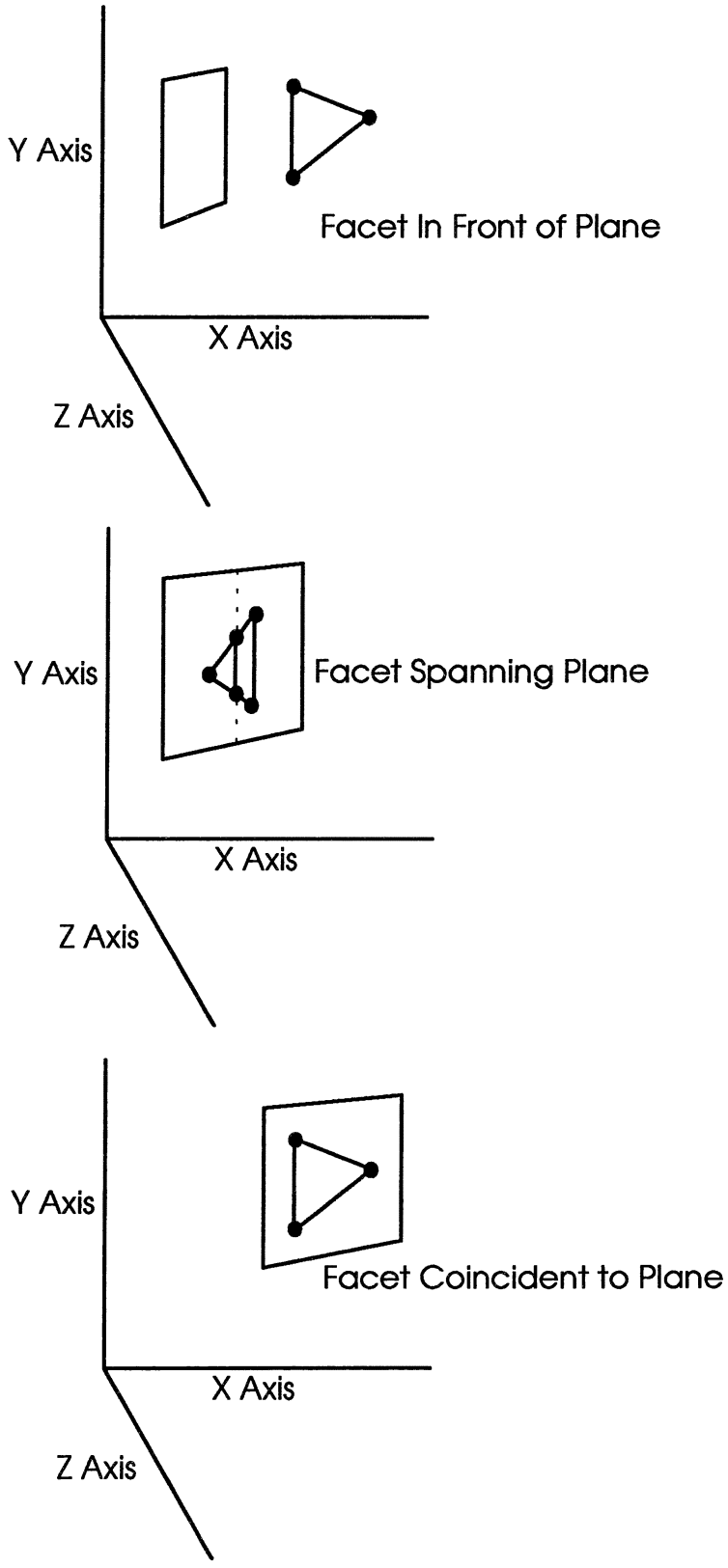


Figure 5-6 - Example of Partitioning a Facet in E^3

As shown in Figure 5-7, in the splitting there are several special cases where one or two of the points of a facet lie inside of the partitioning plane. In these special cases the coincident points are kept in a special list along with all of the new points that were generated from splitting the convex hull facets. After all of the facets in the convex hull have been visited, this coincident list is formed and it then becomes necessary to take all of these points that are in the plane and Delaunay Triangulate[2,21,29] them to form new facets. Those newly generated facets are then put into both the front convex hull and the back hull. This special method of dealing with coincident points was not implemented in the code, it is left for future expansion of the algorithm[17]. The omission of these calculations means that the resulting hull will not be a convex hull in the true sense of the definition because it will be an open ended shell where the plane partitions it. This will not stop the determination of success or failure of the algorithm[17] because by using the Geomview package we can see if the convex hull is being split properly by the partition planes.

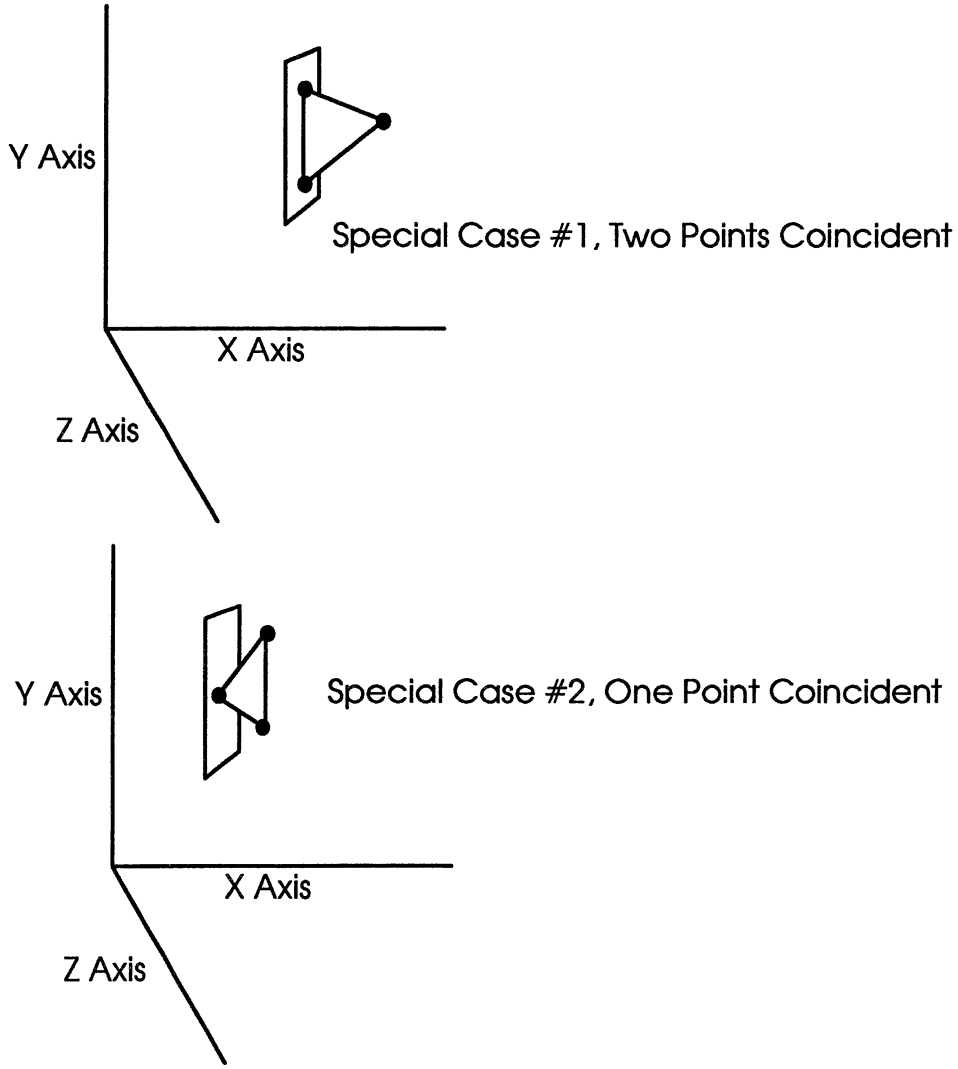


Figure 5-7 - Example of Special Case Coincident Facets in E^3

Shown in Figure 5-8 is an example of a convex hull in E^3 with facets that were Spanning the partition plane. The front convex hull is the one on top of the partition plane and the back convex hull is the one on the bottom of the partition plane. Also shown in Figure 5-9 is a top view of what the Delaunay Triangulation of the points Coincident to the plane would produce with the front and back convex hulls cut away for clarity.

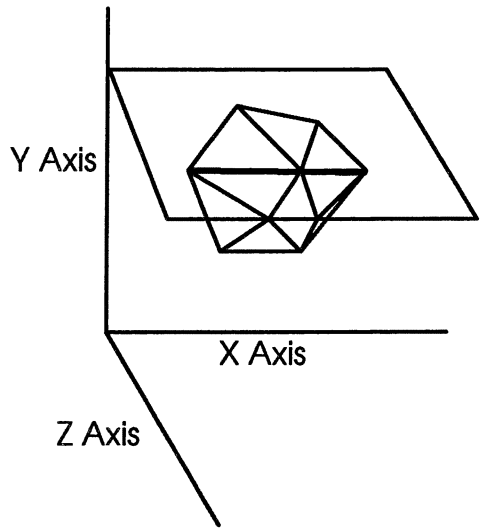


Figure 5-8 - Example of a Convex Hull Spanning a Plane in E^3

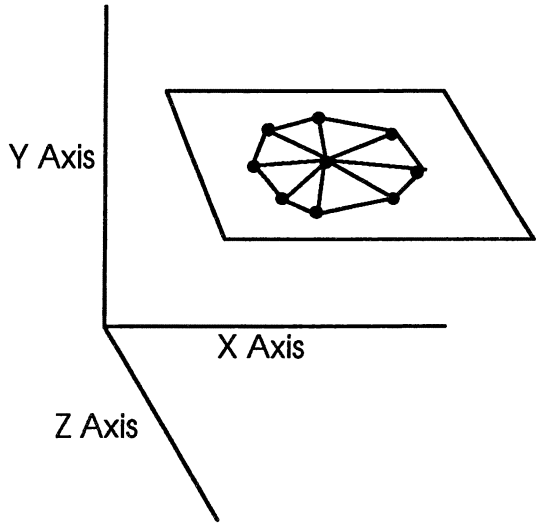


Figure 5-9 - Example of a Delaunay Triangulation of Coincident Points

Chapter VI

BSP Trees

6. BSP Trees

A Binary Space Partitioning (BSP) tree is a data structure that represents a recursive, hierarchical subdivision of n-dimensional space into convex subspaces.[33] BSP trees are built using hyperplanes (E^{d-1} objects) to subdivide E^d space. For example, in E^3 a hyperplane would be an E^2 object or a plane and in E^2 a hyperplane would be an E^1 object or a line. The generic BSP tree is a typical tree that closely models a normal binary tree data structure or a k-d tree data structure. The added feature is that a BSP tree node contains a list of all polygons in that partition of the tree. For example:

```
BSP_Tree:
   plane partition
   list polygons
   BSP_Tree *front, *back
   int leaf_node
```

Shown in Figure 6-1 is an example BSP tree in E^2 with two partitioning lines X and Y which subdivide the space into three subspaces D, E, and C [33]:

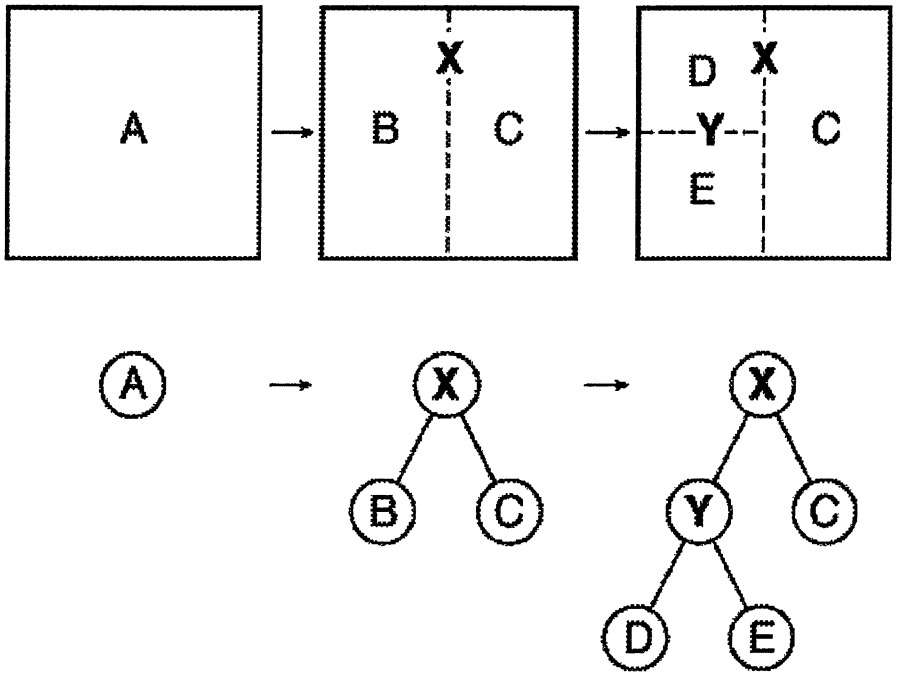


Figure 6-1 - Sample BSP Tree in E^2

Notice, that in a BSP tree, if any points resided in subspace D that they would be contained in the node that represented that subspace in the tree. In this example, the internal nodes of the tree represent the partitioning lines and the leaf nodes of the tree represent the subspaces.

6.1 BSP Tree Construction

The method for building a BSP tree is to first select a partition plane, partition the set of polygons with the plane, and then recurse with each of the two new sets. The input for the project was a set of planes in E^3 space that was going to be used as a partitioning set for the BSP tree. In the BSP_Tree class the construction process and the merging process were separated into two separate processes. The basic pseudocode algorithm for doing the recursive BSP tree construction is:

```
procedure BSP_Tree_Build(BSP_Tree *t, List<Plane> l)
   List<Plane> front_list, back_list
   if l is not empty then
      if t is null then
         t <- new BSP_Tree
         t.partition_plane <- l
         while l is not empty do
            case (t.partition_plane and l)
               parallel in front: add l to front_list
               parallel in back: add l to back_list
               coincident: ignore l
```

```
                spanning: add l to both front_list and back_list
            end case
        end while
        if front_list is not empty then
            BSP_Tree_Build(t.front, front_list)
        else
            t.front = new subspace
        end if
        if back_list is not empty then
            BSP_Tree_Build(t.back, back_list)
        else
            t.back = new subspace
        end if
    else
        ERROR t is not null
    end if
  end if
end procedure
```

Figure 6-2 - Pseudocode for BSP Tree Construction

6.2 BSP Tree Merging

The method for merging a convex hull into a BSP tree is to start at the root node of the tree with the convex hull, which is simply a list of facets, and to test each facet of the convex hull against the partition plane at that node. All of the facets will fall into one of four categories:

- In Front of the plane
- In Back of the plane
- Coincident inside of the plane
- Spanning across the plane

For every facet, if the facet is In Front of the plane then place the facet in a temporary front facet list, or if the facet is In Back of the plane then place the facet in a temporary back facet list. If the facet is coincident then place all of the points of the facet into a temporary point list. The last and most difficult case is if the facet is spanning the plane. If the facet is spanning the plane then the facet must be split into two separate facets and the new facets must be placed into their respective facet lists, the temporary front facet list and the temporary back facet list. The point(s) that are inside of the partition plane as the product of splitting the facet with the partition plane should also be added to the temporary point list. After all of the facets in the convex hull have been exhausted, the list of points that are inside of the partition plane need to be placed into new facets that form a Delaunay triangulation and then those new facets need to be added to both the temporary front facet list and the temporary back facet list. Then recurse down the front BSP sub-tree with the temporary front facet list and recurse down the back BSP sub-tree with the temporary back facet list. If the BSP tree node visited has a left or right sub-tree that is a leaf node, then the list of facets that belong in that subspace should be added to the facet list in that leaf node. The basic pseudocode algorithm for recursively merging a convex hull with the BSP tree is:

```
procedure Merge_Convex_Hull(BSP_Tree *t, List<Facet> ch)
   List<Plane> temp_front_list, temp_back_list
   List<Point3> temp_point_list
   if ch is not empty then
      if t is not null then
         while ch is not empty do
            case (t.partition_plane and ch facet)
               in front: add ch facet to temp_front_list
               in back: add ch facet to temp_back_list
               coincident: add all three points of ch facet
                  to temp_point_list
               spanning: split ch facet into two new facets
                  add front new facet to temp_front_list
                  add back new facet to temp_back_list
                  add ch facet and plane intersection points
                     to temp_point_list
            end case
         end while
         form temp_point_list into new facets
         put new facets into temp_front_list
         put new facets into temp_back_list
         if t.front is not null then
            t.front.front_list <- temp_front_list
         else
            Merge_Convex_Hull(t.front, temp_front_list)
         end if
         if t.back is not null then
            t.back.back_list <- temp_back_list
         else
            Merge_Convex_Hull(t.back, temp_back_list)
         end if
      else
         ERROR t is null
      end if
   end if
end procedure
```

Figure 6-3 - Pseudocode for Convex Hull Merge Into BSP Tree

Chapter VII

PROGRAM IMPLEMENTATION

7. Program Implementation

The OOP implementation of the program design, that was described previously in chapter 4, had some physical implementation issues that were important to a successful implementation of the project. This chapter further explains those physical implementation issues.

7.1 Compiler and Platform Choice

The compiler that was originally chosen for this project was Turbo C++ v1.5 for the MS-DOS platform. This compiler and platform were originally chosen because of the simplicity and speed of the compiler/platform combination. After the initial design of the project had already been started an additional requirement was added to the project that required the code to be compatible across a wide range of compilers and platforms. Due to this new requirement at this stage of the project, the development was being done on the PC platform, and also being tested on three different UNIX platforms running g++ (GNU C++) as the compiler. However, once the project got bigger the code no longer ran on the PC platform using the Turbo C++ compiler because the code segment was larger than 64K with the debugging information turned on. Under MS-DOS based PC architectures there is a limit of 64K for each code segment, data segment, and stack segment. In MS-DOS based PC architectures the way to compensate for this limitation is to change the memory model that the compiler compiles your code under. The tradeoff is that in your code the pointers must now be addressed as either "near" or "far" pointers depending on how they are used and what they reference. The resulting code would definitely not be portable to any platforms other than a MS-DOS based PC, which would violate the new requirement.

7.1.1 Compiler Choice Alternative

The alternative to moving to an entirely UNIX solution was to download the DJGPP package (GNU C++ ported to MS-DOS) [12] and to port the project entirely to the g++ compiler for both the UNIX and PC platforms. The DJGPP compiler supports a flat memory architecture like UNIX does through the use of a MS-DOS memory extender. This new compiler ultimately fulfilled the multiple platform requirement because the program could now be compiled under the standard g++ compiler for any platform. The programming implementation part of the project was carried to completion using the PC platform with the DJGPP g++ compiler for development and many different UNIX platforms with the standard g++ compiler for testing.

7.1.2 Output and Final Considerations

Towards the end of the project all of the testing had to be done on the DEC Alpha OSF/1 UNIX platform since that was the only platform that the Geomview package [15] supported at that time. Toward the end of the project, the network speed was just too slow to support remote X-Windows via modem or network. This meant that the ability to visually test my OOGL [23,32] output was lost, so the last platform that was added to the project was a Linux UNIX based PC platform also using the GNU g++ compiler. This new platform allowed all of the conveniences of a powerful UNIX workstation at the price of a PC in my own home which was extremely nice. It allowed compiling, testing, and viewing of the OOGL output in the Geomview package all on one machine. The Geometry Center [15] had just released the Linux version of the Geomview package at that time which made the whole development and testing system come together at the right time.

7.2 Platforms

In summary, the platforms that were mentioned in section 7.1 were:

1. IBM PC compatible, running MS-DOS v6.22
2. Sun 3/60 workstation, running SunOS v4.1.1 UNIX
3. Sun Sparc20 workstation, running Solaris v5.4 UNIX
4. DEC Alpha workstation, running OSF/1 UNIX
5. IBM PC compatible, running Linux UNIX

7.2.1 IBM PC Compatible (MS-DOS)

The IBM PC compatible machine running MS-DOS v6.22 was an excellent choice for a preliminary development platform. The physical machine was a 486-DX2/66 with 16 MB RAM, 256K cache, and a built-in Math Co-Processor. The code was developed using the wide range of editors, compilers, and development tools that are available on the PC platform. As the project got larger the downside of the platform was that the MS-DOS based compilers only supported a code segment, data segment, and stack segment of

64K. The code segment of the project was larger than 64K with the debugging flags turned on, so the project would not run on the platform. If the memory model was switched to the large or huge models to compensate for the small segment size, then the pointers didn't work because the PC uses "near" and "far" pointers which would not make the code portable to UNIX platforms. After switching compilers to DJGPP (GNU C++ for MS-DOS), which supports a flat memory model like UNIX platforms do, then the code could be further developed and the code remained compatible with UNIX platforms throughout the development.

7.2.2 Sun 3/60

The Sun 3/60 workstation running SunOS v4.1.1 UNIX was a good platform choice at first to test the code that was developed on the PC because it gave the closest results that the PC code produced, and the speed was just a little bit faster than the PC on running the project. As the project got larger, the downsides of the machine became more evident. The machine was slow at compiling, ran out of memory often, and broke down on a regular basis. In the middle of the project the machine was rendered useless. It would have been nice to get some timing trials from this machine as well as the newer UNIX machines, but now that is not possible. This workstation was state of the art about 10 years ago, but today it is really lacking in power.

7.2.3 Sun Sparc20

The Sun Sparc20 workstation Solaris v5.4 UNIX was an excellent platform choice to take over where the old Sun 3/60 machine had left off. The machine was fast and seemed to run with exactly the same results as the old Sun 3/60 machine. Once we got a working C/C++ compiler on the machine, the project compiled with a few warnings that were quickly resolved, and the project moved on. Once everything was configured properly, the Sun Sparc20 is a full featured machine that is extremely fast. A lot of delay in development time could have been avoided if this machine had been available for use at the start of the project and was used as the sole development machine. As of the writing of this thesis, the Sun Sparc20 workstation is new considered a low end workstation and the new Sun Sparc Ultra workstations are the top of the line.

7.2.4 DEC Alpha

The DEC Alpha workstation running OSF/1 UNIX had problems right from the start of choosing the platform. The machine was extremely fast; however it had a lot of floating point problems that the Sun platforms and the PC platform never encountered. It was a real chore to find all of the floating point problems that the DEC Alpha machine kept pointing out. As it turned out, the DEC Alpha would catch floating point programming errors and inconsistencies that the Sun and PC platforms just ignored, rounded off, or simply could not catch. Toward the end of the project, this machine was

used exclusively for testing and viewing the OOGL output files because this was the only platform that the Geomview package was supported on at that time.

7.2.5 IBM PC Compatible (Linux)

The IBM PC compatible machine running the Linux UNIX operating system was added to the project at the last stages of development and testing. The physical machine was a 386-DX40 with 8 MB RAM, 128K cache, and a 387 Math Co-Processor. The benefits of this machine were that it ran the UNIX operating system, ran the X-Windows GUI, had the g++ compiler, and a version of the Geomview package for Linux was just released as this machine was being moved into the development and testing of the project. This machine proved to be invaluable because it was like having a powerful UNIX workstation on the desk at the cost of a cheap PC. The downsides of this platform were that the 386-DX40 is getting antiquated and the machine takes over 40 minutes to compile and link all of the source code files, while its workstation counterparts took between 1-2 minutes to compile and link the same source code files.

BIBLIOGRAPHY

[1] Alpert, Sherman R., and Lam, Richard B. 1997. The Ultimately Publishable Computer Science Paper for the Latter '90s: A Tip for Authors. *Communications of the ACM*, Vol. 40, No. 1, January 1997: 94.

[2] Banks, Dave. 1995. Gridomatic - A Hybrid Structured/Unstructured Grid Generator Report, Delaunay Triangulation. http://www-mae.engr.ucdavis.edu/CFD/dbanks/Hybrid/report/node7.html http://www-mae.engr.ucdavis.edu/CFD/dbanks/Hybrid/report/node29.html November 10, 1995.

[3] Cartesian Coordinates in Space. 1995. Geometry Formulas and Facts, excerpt from *CRC Standard Mathematical Tables and Formulas*, 30th Ed. Boca Raton, Florida: CRC Press, LLC. http://www.geom.umn.edu/docs/reference/CRC-formulas/node39.html

[4] Chand, Donald R., and Kapur, Sham S. 1970. An Algorithm for Convex Polytopes. *Journal of the ACM*, Vol. 17, No. 1, January 1970: 78-86.

[5] Chin, Norman. 1992. "Partitioning a 3-D Convex Polygon with an Arbitrary Plane." *Graphics Gems III*. Academic Press, Inc. 219-222, 502-510.

[6] Coad, Peter, and Nicola, Jill. 1993. *Object-Oriented Programming*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

[7] Coad, Peter, and Yourdon, Edward. 1991. *Objected-Oriented Analysis*, 2nd Ed. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

[8] Coad, Peter, and Yourdon, Edward. 1991. *Objected-Oriented Design.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

[9] Cormen, Thomas H., Leiserson, Charles E., and Rivest, Ronald L. 1989. *Introduction to Algorithms*. New York: McGraw-Hill Book Company.

[10] Clarkson, Ken. 1996. Hull: A Program for Convex Hulls. Lucent Technologies, Bell Laboratories, Computing and Mathematical Sciences Research Division. http://netlib.bell-labs.com/netlib/voronoi/hull.html. November 14, 1996.

[11] Direction Angles and Direction Cosines. 1995. Geometry Formulas and Facts, excerpt from *CRC Standard Mathematical Tables and Formulas*, 30th Ed. Boca Raton, Florida: CRC Press, LLC. http://www.geom.umn.edu/docs/reference/CRC-formulas/node52.html

[12] DJGPP Gnu C++ Software Package for the IBM-PC Compatible Platform. ftp://ftp.coast.net//SimTel/vendors/djgpp

[13] Ellis, Robert, and Gulick, Denny. 1986. *Calculus with Analytic Geometry*, 3rd Ed. New York: Harcourt Brace Jovanovich, Publishers.

[14] Fuchs, Henry, Kedem, Zvi M., and Naylor, Bruce F. 1980. On Visible Surface Generation by a Priori Tree Structures. *ACM Publication 0-89791-021-4/80/0700-0124*.

[15] Geomview: 3D Visualization Software. 1997. The Geometry Center: Center for the Computation and Visualization of Geometric Structures, University of Minnesota. http://www.geom.umn.edu/software/geomview/. April 29, 1997.

[16] Hazlewood, Carol. 1993. Approximating Constrained Tetrahedrizations. Computer Aided Geometric Design 10. 67-87.

[17] Hazlewood, Carol. 1993. Using Binary Space Partitions to Approximate Constrained Tetrahedrizations. September 18, 1993, Southwest Texas State University.

[18] Kirk, David. 1992. *Graphics Gems III*. Academic Press, Inc. 393-407.

[19] Lines. 1995. Geometry Formulas and Facts, excerpt from *CRC Standard Mathematical Tables and Formulas*, 30th Ed. Boca Raton, Florida: CRC Press, LLC. http://www.geom.umn.edu/docs/reference/CRC-formulas/node54.html

[20] Midtbo, Terje. 1993. Spatial Modelling by Delaunay Networks of Two and Three Dimensions. February, 1993. http://www.iko.unit.no/tmp/term/node7.html

[21] Mücke, Ernst P. 1995. A Robust Implementation For Three-Dimensional Delaunay Triangulations. April 12, 1995, Los Alamos National Laboratory.

[22] Naylor, Bruce F. 1994. Binary Space Partitioning Trees A Tutorial. *ACM SIGGRAPH 1994*, Course Notes 23, Computational Representations of Geometry.

[23] OOGL File Formats. 1996. The Geometry Center: Center for the Computation and Visualization of Geometric Structures, University of Minnesota. http://www.geom.umn.edu/software/geomview/ooglman.html. October 14, 1996.

[24] Paterson, Michael S., and Yao, Frances F. 1989. Binary Partitions with Applications to Hidden-Surface Removal and Solid Modelling. *ACM Publication 0-89791-318-3/89/0006/0023*.

[25] Planes. 1995. Geometry Formulas and Facts, excerpt from *CRC Standard Mathematical Tables and Formulas*, 30th Ed. Boca Raton, Florida: CRC Press, LLC. http://www.geom.umn.edu/docs/reference/CRC-formulas/node53.html

[26] Preparata, Franco P., and Shamos, Michael Ian. 1985. *Computational Geometry An Introduction.* New York: Springer-Verlag, Inc.

[27] Protter, Murray H., and Morrey, Charles B. Jr. 1966. *Analytic Geometry.* New York: Addison-Wesley Publishing Company.

[28] Qhull Home Page. 1997. The Geometry Center: Center for the Computation and Visualization of Geometric Structures, University of Minnesota. http://www.geom.umn.edu:80/software/qhull/. April 14, 1997.

[29] Sun, Yichu E. 1994. Approximating Constrained Tetrahedrizations. M.S. thesis., Southwest Texas State University.

[30] Sung, Kelvin, and Shirley, Peter. 1992. "Ray Tracing with the BSP Tree." *Graphics Gems III.* Academic Press, Inc. 271-274, 538-546.

[31] Turabian, Kate L. 1996. *A Manual for Writers of Term Papers, Theses, and Dissertations*, 6th Ed. Chicago: The University of Chicago Press.

[32] Tutorial: The OOGL Geom File Formats. 1996. The Geometry Center: Center for the Computation and Visualization of Geometric Structures, University of Minnesota. http://www.geom.umn.edu/software/geomview/oogltour.html. October 14, 1996.

[33] Wade, Bretton. 1997. *BSP Tree Frequently Asked Questions (FAQ).* comp.graphics.algorithms, http://reality.sgi.com/bspfaq/. Silicon Graphics, Inc., June 8, 1997.

[34] Watson, D. F. 1993. *Computing the Delaunay Simplicial Complex.*

[35] Watt, Alan. 1993. *3D Computer Graphics*, 2nd Ed. New York: Addison-Wesley Publishing Company.

[36] Wolfram, Stephen. 1991. *Mathematica: A System for Doing Mathematics by Computer*, 2nd Ed. New York: Addison-Wesley Publishing Company.

VITA

Brian James Collins was born in Wichita, Kansas, on April 12, 1968, the son of Harold Edward Collins and Jeannette Almaira Collins. After completing his work at McCollum High School, San Antonio, Texas, in 1986, he entered Southwest Texas State University in San Marcos, Texas. He received the degree of Bachelor of Science with a major in Computer Science and minor in Mathematics from Southwest Texas State University in December, 1991. In September, 1992, he entered the Graduate School of Southwest Texas State University, San Marcos, Texas on a full-time basis. In May, 1993, he accepted a full-time position as a Software Engineer on the Combat Support Services Training Simulation System (CSSTSS) project for a year and then on the Advanced Tomahawk Weapons Control System (ATWCS) project for two years with the Lockheed Martin Corporation, in Austin, Texas. During that time he entered the Graduate School of Southwest Texas State University, San Marcos, Texas on a part-time basis while working at the Lockheed Martin Corporation, Austin, Texas on a full-time basis. In January, 1996, he left the Lockheed Martin Corporation to pursue his graduate studies at Southwest Texas State University, San Marcos, Texas on a full-time basis. In September, 1996, he accepted a full-time position as a Software Engineer with Rockwell Collins Incorporated, in Cedar Rapids, Iowa and finished working on his Masters thesis on a part-time basis.

Permanent address: 1029 26th St., SE
Cedar Rapids, Iowa 52403

This thesis was typed by Brian James Collins.

APPENDIX

Files Included in this Appendix:

1. bsp_tree.h
2. chsplit.h
3. chull2.h
4. chull3.h
5. ctz.h
6. facet2.h
7. facet3.h
8. faceti.h
9. general.h
10. lineseg2.h
11. lineseg3.h
12. list.h
13. oogl.h
14. plane.h
15. point2.h
16. point3.h
17. subfacet2.h
18. subfacet3.h
19. template.h
20. bsp_tree.cc
21. chsplit.cc
22. chull2.cc
23. chull3.cc
24. ctz.cc
25. facet2.cc
26. facet3.cc
27. faceti.cc
28. general.cc
29. lineseg2.cc
30. lineseg3.cc
31. list.cc
32. oogl.cc
33. plane.cc
34. point2.cc
35. point3.cc
36. subfacet2.cc
37. subfacet3.cc
38. test_ctz.cc
39. test_del.cc
40. test_lis.cc
41. test_p2.cc

42. test_pla.cc
43. proc_off.cc
44. makefile.djg
45. makefile.sol
46. makefile.linux
47. points.dat
48. points2.dat
49. triangle.dat
50. triangle2.dat

```
/* bsp_tree.h */

#ifndef BSP_TREE_H
#define BSP_TREE_H 1
#ifdef __cplusplus

#include "plane.h"
#include "point3.h"
#include "list.h"
#include "chull3.h"

class BSP_Tree
{
   public:
      BSP_Tree();
      ~BSP_Tree();
      void Build_BSP_Tree(List<Plane>);
      void Merge_Convex_Hull(Convex_Hull3);
      void Convex_Hull_Insert(List<Point3>&, Convex_Hull3);
      void OOGL_Output(List<Point3>&);
      void Show();

   private:
      void Show_Class(Poly_Class);
      void R_Build_BSP_Tree(BSP_Tree*, List<Plane>);
      void R_Merge_Convex_Hull(BSP_Tree*, Convex_Hull3);
      void R_Convex_Hull_Insert(List<Point3>&, BSP_Tree*,
Convex_Hull3);
      void R_OOGL_Output(List<Point3>&, BSP_Tree*);
      void R_Show(BSP_Tree*);

      Plane partition;
      Convex_Hull3 polygons;
      BSP_Tree *front;
      BSP_Tree *back;
      int leaf_node;
};

#endif
#endif
/* chsplit.h */

#ifndef CONVEX_HULL_SPLIT
#define CONVEX_HULL_SPLIT 1
#ifdef __cplusplus

#include "faceti.h"
#include "point2.h"
#include "point3.h"
#include "facet3.h"
#include "plane.h"
#include "list.h"
#include "chull3.h"


// Given 4 points in E3, return the Delaunay Triangulation
void Delaunay_Triangulate (Point3, Point3, Point3, Point3,
                           Facet3&, Facet3&);


// Given n points in E3 and a plane that the points are contained in,
```

```
// return a Triangulation
Convex_Hull3 Triangulate(List<Point3>&, Plane&);


// Given n points in E2, return a Triangulation
List<Facet_Index> Triangulate2(List<Point2>&);


// Given a Facet and a Plane, split the facet and put the
// Convex Hull facets in the Convex Hull.  Extra added points
// will be added to the Point List.
void Split_Facet_With_Plane (List<Point3>&,
                             List<Point3>&,
                             Facet3&,
                             Plane&,
                             Convex_Hull3&,
                             Convex_Hull3&);


// Given a Facet and a Plane, split the facet and put the
// Convex Hull facets in the Convex Hull.  Extra added points
// will be added to the new point List.  Caller must add new
// points back to p on their own.
void Split_Facet_With_Plane2 (List<Point3>& p,
                              List<Point3>& new_points,
                              Facet3 f,
                              Plane partition_plane,
                              Convex_Hull3& front_facets,
                              Convex_Hull3& back_facets);


// Given a Polygon represented as a Facet, split the polygon with
// the plane and put the Convex Hull facets in the Convex Hull.
void Split_Polygon_With_Plane (Facet3&,
                               Plane&,
                               Convex_Hull3&,
                               Convex_Hull3&);


// Given a Convex Hull and a Plane, split the Convex Hull with the
// Plane and put the new Convex Hull facets in each of the output
// Convex Hulls.  Extra added points will be added to the Point List.
void Split_Convex_Hull_With_Plane (List<Point3>&,
                                   List<Point3>&,
                                   Convex_Hull3&,
                                   Plane&,
                                   Convex_Hull3&,
                                   Convex_Hull3&);

#endif
#endif
/* chull2.h - Convex Hull Class */

#ifndef CONVEX_HULL2_H
#define CONVEX_HULL2_H 1
#ifdef __cplusplus

#include "list.h"
#include "point2.h"
#include "facet2.h"

class Convex_Hull2 : public List<Facet2>
{
```

```
   public:
      // Convex Hull - Gift Wrapping Method
      // Given n points p1 .. pn E3 produces a convex hull
      // Represented by a list of facets f1 .. fm.

      // Algorithm presented as pseudo-code in:
      // Computational Geometry an Introduction,
      // Franco P. Preparata and Michael Ian Shamos,
      // Springer-Verlag, 1985, pp 131-136.

      // This particular implementation was modified
      // from the original algorithm by Carol Hazlewood, PhD
      // and Brian Collins.

      void GiftWrapping (List<Point2>&);

   private:
      Facet2 Find_Initial_Facet (List<Point2>&);
};

#endif
#endif
/* chull.h - Convex Hull Class */

#ifndef CONVEX_HULL3_H
#define CONVEX_HULL3_H 1
#ifdef __cplusplus

#include "list.h"
#include "point3.h"
#ifdef DEBUG
#include "facet3.h"
#include "plane.h"
#endif

class Convex_Hull3 : public List<Facet3>
{
   public:
      // Convex Hull - Gift Wrapping Method
      // Given n points p1 .. pn E3 produces a convex hull
      // Represented by a list of facets f1 .. fm.

      // Algorithm presented as pseudo-code in:
      // Computational Geometry an Introduction,
      // Franco P. Preparata and Michael Ian Shamos,
      // Springer-Verlag, 1985, pp 131-136.

      // This particular implementation was modified
      // from the original algorithm by Carol Hazlewood, PhD
      // and Brian Collins.

      void GiftWrapping (List<Point3>&);
#ifdef DEBUG
      int Verify_Split_Convex_Hull (Plane&);
#endif

   private:
      Facet3 Find_Initial_Facet (List<Point3>&);
#ifdef DEBUG
      int Verify_Initial_Facet (List<Point3>&, Facet3&);
#endif
};
```

```
#endif
#endif
/* ctz.h */

#ifndef CTZ_H
#define CTZ_H 1
#ifdef __cplusplus

// Computing a Constrained Tetrahedrization

//Algorithm presented in paper:
//      Hazlewood, Carol.  Using Binary Space Paritions to
//      Approximate Constrained Tetrahedrizations.  September
//      18, 1993.

#include "list.h"
#include "point3.h"
#include "plane.h"

// USAGE:
// P' is a set of n points in E^3
// F is a set of k triangles which have vertices in P'
// and which intersect in (possibly empty) mutual faces

void Constrain (List<Point3>&, List<Plane>&);

#endif
#endif
/* facet2.h - Facet2 Class */

#ifndef FACET2_H
#define FACET2_H 1
#ifdef __cplusplus

#include <iostream.h>
#include "point2.h"
#include "lineseg2.h"

// NOTE:
//    In 2D a facet is a line segment and a subfacet is a point.

class Facet2 : public Line_Segment2
{
   public:
      // Operators
      Facet2& operator=(const Facet2&);
      int sf_equal(const Facet2&);
      friend ostream& operator<<(ostream&,Facet2);

      // Accessors
      void SetFacet(Point2, Point2);
      void SetID(int);
      int GetID();

      // Services
      void Show();
      void Show_Full();

   private:
      // Attributes
      int id;
};
```

```
#endif
#endif
/* facet3.h - Facet3 Class */

#ifndef FACET3_H
#define FACET3_H 1
#ifdef __cplusplus

#include <iostream.h>
#include "plane.h"
#include "point3.h"

class Facet3 : public Plane
{
   public:
      // Operators
      Facet3& operator=(const Facet3&);
      int sf_equal(const Facet3&);
      friend ostream& operator<<(ostream&,Facet3);

      // Accessors
      void SetFacet(Point3, Point3, Point3);
      void SetID(int);
      int GetID();

      // Services
      void Show();
      void Show_Full();

   private:
      // Attributes
      int id;
};

#endif
#endif
/* faceti.h - Facet Index Class */

#ifndef FACET_INDEX_H
#define FACET_INDEX_H 1
#ifdef __cplusplus

#include <iostream.h>

class Facet_Index
{
   public:
      Facet_Index();                      // Default Constructor
      Facet_Index(int,int,int);           // Secondary Constructor
      Facet_Index(const Facet_Index&);    // Copy Constructor
      ~Facet_Index();                     // Default Constructor

      void operator=(const Facet_Index&);
      int operator!=(const Facet_Index&);
      int operator==(const Facet_Index&);
      friend ostream& operator<<(ostream&,Facet_Index);

      int GetP1Index();
      int GetP2Index();
      int GetP3Index();
      void SetP1Index(int);
      void SetP2Index(int);
      void SetP3Index(int);
```

```
   private:
      int p1_index;
      int p2_index;
      int p3_index;
};

#endif
#endif
/* general.h */

#ifndef GENERAL_H
#define GENERAL_H 1
#ifdef __cplusplus

#define TOLER 0.000001
#define MIN_DOUBLE 1.0e-10
#define MIN_FLOAT 1.0e-07
#define MAX_DOUBLE 1.0e+10
#define MAX_FLOAT 1.0e+07

typedef enum {COINCIDENT, IN_BACK_OF, IN_FRONT_OF, SPANNING}
Poly_Class;

/* Test if a double value1 is "near" another double value2 */
int DEQ (double value1, double value2);


/* Test if a float value1 is "near" another float value2 */
int FEQ (float value1, float value2);


/* Test if a double value is "near" 0.0 */
int DEQ0 (double value);


/* Test if a float value is "near" 0.0 */
int FEQ0 (float value);


/* Sign of a value returns: -1 = negative, 1 = positive, 0 = "near"
zero */
int sgn (double value);


/* Find the determant of a 2x2 matrix */
double det2(double a11, double a12,
            double a21, double a22);


/* Find the determant of a 3x3 matrix */
double det3(double a11, double a12, double a13,
            double a21, double a22, double a23,
            double a31, double a32, double a33);


/* Solve a system of 3 equations and 3 unknowns */
void solve(double a11, double a12, double a13, double a14,
           double a21, double a22, double a23, double a24,
           double a31, double a32, double a33, double a34,
           double &a, double &b, double &c, double &d);
```

```
/* Standard Deviation of n doubles */
double standard_deviation(double X[], int n);

#endif
#endif
/* lineseg2.h - Line_Segment2 Class */

#ifndef LINE_SEGEMENT2_H
#define LINE_SEGEMENT2_H 1
#ifdef __cplusplus

#include "point2.h"

class Line_Segment2
{
   public:
      // Constructors & Destructors
      Line_Segment2();                          // Default Constructor
      Line_Segment2(Point2, Point2);            // Secondary Constructor
      Line_Segment2(const Line_Segment2&);      // Copy Constructor
      ~Line_Segment2();                         // Default Destructor

      // Operators
      Line_Segment2& operator=(const Line_Segment2&);

      // Accessors
      void SetP1(Point2);
      Point2 GetP1();
      void SetP2(Point2);
      Point2 GetP2();
      void SetSegment(Point2, Point2);

      // Services
      int Is_On_Line_Segment(Point2);
      void Show();

   protected:
      // Attributes
      Point2 p1, p2;
};


#endif
#endif
/* lineseg3.h - Line_Segment3 Class */

#ifndef LINE_SEGEMENT3_H
#define LINE_SEGEMENT3_H 1
#ifdef __cplusplus

#include "point3.h"
#include "plane.h"

class Line_Segment3
{
   public:
      // Constructors & Destructors
      Line_Segment3();                          // Default Constructor
      Line_Segment3(Point3, Point3);            // Secondary Constructor
      Line_Segment3(const Line_Segment3&);      // Copy Constructor
      ~Line_Segment3();                         // Default Destructor

      // Operators
```

```
      Line_Segment3& operator=(const Line_Segment3&);

      // Accessors
      void SetP1(Point3);
      Point3 GetP1();
      void SetP2(Point3);
      Point3 GetP2();
      void SetSegment(Point3, Point3);

      // Services
      Point3 Intersection(Plane&);
      int Is_On_Line_Segment(Point3);
      void Show();

   protected:
      // Attributes
      Point3 p1, p2;
};


#endif
#endif
/* list.h */

#ifndef LIST_H
#define LIST_H 1
#ifdef __cplusplus

#include "facet3.h"                        // Only needed for Find2
function
template<class T>
class List
{
   public:

      List();                              // Default Constructor
      List(const List&);                   // Copy Constructor
      ~List();                             // Default Destructor

      List& operator=(const List&);
      void Insert_Head(const T);
      void Insert_Tail(const T);
      void Insert_At_Pointer(const T);
      T Remove_Head();
      T Remove_Tail();
      T Remove_At_Pointer();
      T Peek_Head() const;
      T Peek_Tail() const;
      T Peek_At_Pointer() const;
      int Is_Empty();                      // returns: 1=yes, 0=no
      void Clear();
      void Reset_Pointer();
      int Increment_Pointer();             // returns: 1=success,
0=failure
      int Decrement_Pointer();             // returns: 1=success,
0=failure
      void Show();
      int Is_Member(const T);              // returns: 1=yes, 0=no
      int Find(const T);                   // returns: 1=found, 0=not
found
      int Find2(const Facet3);             // Special Find for
List<Facet3> Only!
      int Num_Members();
```

```
   private:

      struct list_item;
      struct list_item
      {
         T data;
         struct list_item *prev;
         struct list_item *next;
      };
      typedef struct list_item *lptr;

      lptr head;
      lptr tail;
      lptr cur_ptr;
      int num_items;

};

#endif
#endif
/* oogl.h */

#ifndef OOGL_H
#define OOGL_H 1
#ifdef __cplusplus

#include "list.h"
#include "point3.h"
#include "chull3.h"
#include "plane.h"
#include "bsp_tree.h"


// Creates Output OOGL File: chull.off in current directory
void Convex_Hull_2_OOGL (List<Point3>&, Convex_Hull3&);


// Creates Output OOGL File: triangs.off in current directory
void Triangles_2_OOGL (List<Plane>&);


// Creates Output OOGL File: bspchull.off in current directory
void BSP_Tree_w_Convex_Hulls_2_OOGL (List<Point3>&, BSP_Tree&);


#endif
#endif
/* plane.h - Plane Class */

#ifndef PLANE_H
#define PLANE_H 1
#ifdef __cplusplus

#include <iostream.h>
#include "general.h"
#include "point3.h"


class Plane
{
   public:
      // Constructors & Destructors
```

```
      Plane();                          // Default Constructor
      Plane(Point3, Point3, Point3);    // Secondary Constructor
      Plane(const Plane&);              // Copy Constructor
      ~Plane();                         // Default Destructor

      // Operators
      Plane& operator=(const Plane&);
      int operator==(const Plane&);
      int operator!=(const Plane&);
      friend ostream& operator<<(ostream&,Plane);

      // Accessors
      void SetP1(Point3);
      Point3 GetP1();
      void SetP2(Point3);
      Point3 GetP2();
      void SetP3(Point3);
      Point3 GetP3();
      void SetPlane(Point3, Point3, Point3);
      double GetA();
      double GetB();
      double GetC();
      double GetD();

      // Services
      int Is_Parallel(Plane&);
      int Is_Coincident(Plane&);
      double Distance(Point3);          // Distance between point and
plane
      int Is_Point_On_Plane(Point3);
      Poly_Class Classify_Polygon(Plane&);
      Poly_Class Classify_Polygon2(Point3);
      Poly_Class Classify_Polygon3(Plane&);
      double Angle(Plane&);              // Angle between 2 planes
      Point3 Normal();                  // Normal vector to a plane
      void Show();
      void Show_Full();

   private:
      // Private Services
      void Calculate_Equation();
      void Newells_Method();

   protected:
      // Attributes
      Point3 p1, p2, p3;   // planes are formed by three points
      double a, b, c, d;   // Ax+By+Cz+D=0 equation of a plane
};

#endif
#endif
/* point2.h - Point2 Class */

#ifndef POINT2_H
#define POINT2_H 1
#ifdef __cplusplus

#include <iostream.h>

class Point2
{
   public:
      // Constructors & Destructors
```

```
      Point2();                  // Default Constructor
      Point2(double, double);    // Secondary Constructor
      Point2(const Point2&);     // Copy Constructor
      ~Point2();                 // Default Destructor

      // Operators
      Point2& operator=(const Point2&);
      int operator==(const Point2&);
      int operator!=(const Point2&);
      friend ostream& operator<<(ostream&,Point2);

      // Accessors
      void SetX(double);
      double GetX();
      void SetY(double);
      double GetY();
      void SetID(int);
      int GetID();
      void SetPoint(double, double);

      // Services
      double Distance(const Point2&);
      void Show_Full();
      void Show();

   protected:
      // Attributes
      double x, y;
      int id;
};

#endif
#endif
/* point3.h - Point3 Class */

#ifndef POINT3_H
#define POINT3_H 1
#ifdef __cplusplus

#include <iostream.h>

class Point3
{
   public:
      // Constructors & Destructors
      Point3();                          // Default Constructor
      Point3(double, double, double);    // Secondary Constructor
      Point3(const Point3&);             // Copy Constructor
      ~Point3();                         // Default Destructor

      // Operators
      Point3& operator=(const Point3&);
      int operator==(const Point3&);
      int operator!=(const Point3&);
      friend Point3 operator-(const Point3&, const Point3&);
      friend Point3 operator+(const Point3&, const Point3&);
      friend Point3 operator*(const Point3&, const double);
      friend ostream& operator<<(ostream&,Point3);

      // Accessors
      void SetX(double);
      double GetX();
      void SetY(double);
```

```
      double GetY();
      void SetZ(double);
      double GetZ();
      void SetID(int);
      int GetID();
      void SetPoint(double, double, double);

      // Services
      double Distance(const Point3&);
      double Dot_Product(const Point3&);
      double Norm();
      double Magnitude(const Point3&);  // used when using points as
vectors
      Point3 Cross(Point3&);
      double Rho(Point3&, Point3&, const Point3&);
      Point3 Compute_New_N(const double, Point3&);
      Point3 Compute_New_A(const Point3&, const Point3&, const
Point3&);
      void Show_Full();
      void Show();

   private:
      // Attributes
      double x, y, z;
      int id;
};

#endif
#endif
/* subfacet2.h - SubFacet2 Class */

#ifndef SUBFACET2_H
#define SUBFACET2_H 1
#ifdef __cplusplus

#include "point2.h"

// NOTE:
//    In 2D a facet is a line segment and a subfacet is a point.

class SubFacet2 : public Point2
{
   public:
      // Operators
      int operator==(const SubFacet2&);

      // Accessors
      void SetSubFacet(Point2);

};

#endif
#endif
/* subfacet3.h - SubFacet3 Class */

#ifndef SUBFACET3_H
#define SUBFACET3_H 1
#ifdef __cplusplus

#include "lineseg3.h"
#include "point3.h"

class SubFacet3 : public Line_Segment3
```

```
{
   public:
      // Operators
      int operator==(const SubFacet3&);

      // Accessors
      void SetSubFacet(Point3, Point3);

};

#endif
#endif
// GCC and DJGPP Template File

#if defined(__DJGPP__) || defined(__GNUC__)

#include "list.h"
#include "list.cc"
#include "facet3.h"
#include "point3.h"
#include "point2.h"
#include "faceti.h"
#include "plane.h"

template class List<Point3>;
template class List<Point2>;
template class List<Facet3>;
template class List<Facet_Index>;
template class List<Plane>;
template class List<int>;

#endif

/* bsp_tree.cc */

#ifdef __cplusplus

#include <fstream.h>
#include <stddef.h>
#include <iostream.h>
#include <stdlib.h>
#include "bsp_tree.h"
#include "facet3.h"
#include "chsplit.h"


BSP_Tree::BSP_Tree()
{
   polygons.Clear();
   front = (BSP_Tree*)0;
   back = (BSP_Tree*)0;
   leaf_node = 0;
}


BSP_Tree::~BSP_Tree()
{
   if (!polygons.Is_Empty())
      polygons.Clear();
   if (front)
      delete(front);
   if (back)
      delete(back);
```

```
}


void BSP_Tree::Build_BSP_Tree(List<Plane> partition_list)
{
   List<Plane> front_list;              // List of Planes in front of
cur plane
   List<Plane> back_list;               // List of Planes in back of
cur plane

   if (!partition_list.Is_Empty())
   {
      partition = partition_list.Remove_Head();
      while (!partition_list.Is_Empty())
      {
         switch
(partition.Classify_Polygon3(partition_list.Peek_Head()))
         {
            case IN_FRONT_OF:
               front_list.Insert_Tail(partition_list.Remove_Head());
               break;
            case IN_BACK_OF:
               back_list.Insert_Tail(partition_list.Remove_Head());
               break;
            case COINCIDENT:
               // Don't add coincident planes to the tree
               break;
            case SPANNING:
               front_list.Insert_Tail(partition_list.Peek_Head());
               back_list.Insert_Tail(partition_list.Remove_Head());
               break;
         }
      }

      if (!front_list.Is_Empty())
      {
         front = new BSP_Tree;
         R_Build_BSP_Tree(front, front_list);
      }
      else
      {
         front = new BSP_Tree;
         front->leaf_node = 1;
      }

      if (!back_list.Is_Empty())
      {
         back = new BSP_Tree;
         R_Build_BSP_Tree(back, back_list);
      }
      else
      {
         back = new BSP_Tree;
         back->leaf_node = 1;
      }
   }
   else
   {
      cerr << "ERROR: Building BSP Tree with empty partition list" <<
endl;
      exit(1);
   }
}
```

```
void BSP_Tree::Merge_Convex_Hull(Convex_Hull3 convex_hull)
{
   Convex_Hull3 temp_front_list;
   Convex_Hull3 temp_back_list;
   List<Point3> temp_point_list;
   Facet3 temp_facet;

   if (!convex_hull.Is_Empty())
   {
      if (leaf_node == 1)
      {
         // We made it to a leaf node, just copy the facets
         while (!convex_hull.Is_Empty())
            polygons.Insert_Tail(convex_hull.Remove_Head());
      }
      else
      {
         // We made it to an internal node, test facets against
partition
         while (!convex_hull.Is_Empty())
         {
            switch
(partition.Classify_Polygon(convex_hull.Peek_Head()))
            {
               case IN_FRONT_OF:

temp_front_list.Insert_Tail(convex_hull.Remove_Head());
                  break;
               case IN_BACK_OF:
                  temp_back_list.Insert_Tail(convex_hull.Remove_Head());
                  break;
               case COINCIDENT:
                  temp_facet = convex_hull.Remove_Head();
                  temp_point_list.Insert_Tail(temp_facet.GetP1());
                  temp_point_list.Insert_Tail(temp_facet.GetP2());
                  temp_point_list.Insert_Tail(temp_facet.GetP3());
                  break;
               case SPANNING:
                  // TBD Add spanning split of facet here
                  // Where are we going to put newly formed coincident
points?
                  break;
            }
         }

         // Form points inside plane into new facets
         if (!temp_point_list.Is_Empty())
         {
            // Need to figure out how to make points inside plane into
facets
            // Add new facets to both lists
         }

         if (!temp_front_list.Is_Empty())
            R_Merge_Convex_Hull(front, temp_front_list);
         if (!temp_back_list.Is_Empty())
            R_Merge_Convex_Hull(back, temp_back_list);
      }
   }
   else
   {
```

```
      cerr << "WARNING: Merging an empty convex hull" << endl;
   }
}


void BSP_Tree::R_Merge_Convex_Hull(BSP_Tree *t, Convex_Hull3
convex_hull)
{
  // TBD, need to add recursive split code here
}


void BSP_Tree::Convex_Hull_Insert(List<Point3>& p, Convex_Hull3
convex_hull)
{
   Convex_Hull3 temp_front_list;
   Convex_Hull3 temp_back_list;
   List<Point3> temp_point_list;
   Facet3 temp_facet;

   if (!convex_hull.Is_Empty())
   {
      if (leaf_node == 1)
      {
         // We made it to a leaf node, just copy the facets
         while (!convex_hull.Is_Empty())
            polygons.Insert_Tail(convex_hull.Remove_Head());
      }
      else
      {
         // We made it to an internal node, test facets against
partition
         while (!convex_hull.Is_Empty())
         {
            switch
(partition.Classify_Polygon(convex_hull.Peek_Head()))
            {
               case IN_FRONT_OF:

temp_front_list.Insert_Tail(convex_hull.Remove_Head());
                  break;
               case IN_BACK_OF:
                  temp_back_list.Insert_Tail(convex_hull.Remove_Head());
                  break;
               case COINCIDENT:
                  temp_facet = convex_hull.Remove_Head();
                  temp_point_list.Insert_Tail(temp_facet.GetP1());
                  temp_point_list.Insert_Tail(temp_facet.GetP2());
                  temp_point_list.Insert_Tail(temp_facet.GetP3());
                  break;
               case SPANNING:
                  // This service will split the facet with the
partition plane
                  // it will add new points and facets to our lists
                  temp_facet = convex_hull.Remove_Head();
                  Split_Facet_With_Plane(p,
                                         temp_point_list,
                                         temp_facet,
                                         partition,
                                         temp_front_list,
                                         temp_back_list);
                  break;
            }
```

```
         }

         // TBD take points in temp_point_list, Delaunay Triangulate
them
         // to form new facets and put them into both front and back
lists

         if (!temp_front_list.Is_Empty())
            R_Convex_Hull_Insert(p, front, temp_front_list);
         if (!temp_back_list.Is_Empty())
            R_Convex_Hull_Insert(p, back, temp_back_list);
      }
   }
   else
   {
      cerr << "WARNING: Merging an empty convex hull" << endl;
   }
}


void BSP_Tree::OOGL_Output(List<Point3>& p)
{
   Point3 tmp_point;
   Facet3 tmp_facet;
   Point3 tmp_point1, tmp_point2, tmp_point3;
   Convex_Hull3 tmp_front;
   fstream fp;

   fp.open("bspchull.off", ios::app);

   if(fp.fail() | fp.bad())
   {
      cerr << "Error opening file bspchull.off for output." << endl;
      exit(1);
   }

   fp << "LIST" << endl;

   fp.close();

   if (!polygons.Is_Empty())
   {
      List<Point3> tmp_p(p);
      List<Point3> tmp_p2(p);

      fp.open("bspchull.off", ios::app);

      if(fp.fail() | fp.bad())
      {
         cerr << "Error opening file bspchull.off for output." << endl;
         exit(1);
      }

      fp << "{ = OFF" << endl;
      fp << "   " << p.Num_Members() << " ";
      fp << polygons.Num_Members() << " " << "0" << endl;

      // Output Points Part
      tmp_p.Reset_Pointer();
      while (!tmp_p.Is_Empty())
      {
         tmp_point = tmp_p.Remove_Head();
         fp << "   " << tmp_point.GetX() << " " << tmp_point.GetY();
```

```
            fp << " " << tmp_point.GetZ() << endl;
        }

        // Output Front Convex Hull Part
        tmp_front.Reset_Pointer();
        while (!tmp_front.Is_Empty())
        {
            tmp_facet = tmp_front.Remove_Head();
            tmp_point1 = tmp_facet.GetP1();
            tmp_point2 = tmp_facet.GetP2();
            tmp_point3 = tmp_facet.GetP3();
            fp << "   3 " << tmp_point1.GetID() - 1 << " ";
            fp << tmp_point2.GetID() - 1 << " ";
            fp << tmp_point3.GetID() - 1 << endl;
        }
        fp << "}" << endl << endl;

        /*
        fp << "{ = OFF" << endl;
        fp << "    " << p.Num_Members() << " ";
        fp << convex_hull_back_part->Num_Members() << " " << "0" << endl;

        // Output Points Part
        tmp_p2.Reset_Pointer();
        while (!tmp_p2.Is_Empty())
        {
            tmp_point = tmp_p2.Remove_Head();
            fp << "    " << tmp_point.GetX() << " " << tmp_point.GetY();
            fp << " " << tmp_point.GetZ() << endl;
        }

        // Output Back Convex Hull Part
        tmp_back.Reset_Pointer();
        while (!tmp_back.Is_Empty())
        {
            tmp_facet = tmp_back.Remove_Head();
            tmp_point1 = tmp_facet.GetP1();
            tmp_point2 = tmp_facet.GetP2();
            tmp_point3 = tmp_facet.GetP3();
            fp << "   3 " << tmp_point1.GetID() - 1 << " ";
            fp << tmp_point2.GetID() - 1 << " ";
            fp << tmp_point3.GetID() - 1 << endl;
        }
        fp << "}" << endl << endl;
      */

        fp.close();
    }
    if (front != NULL)
        R_OOGL_Output(p, front);
    if (back != NULL)
        R_OOGL_Output(p, back);
}


void BSP_Tree::R_Build_BSP_Tree(BSP_Tree *tree, List<Plane>
partition_list)
{
    List<Plane> front_list;              // List of Planes in front of
cur plane
    List<Plane> back_list;               // List of Planes in back of
cur plane
```

```
    if (!partition_list.Is_Empty())
    {
       tree->partition = partition_list.Remove_Head();
       while (!partition_list.Is_Empty())
       {
          switch (tree-
>partition.Classify_Polygon3(partition_list.Peek_Head()))
          {
             case IN_FRONT_OF:
                front_list.Insert_Tail(partition_list.Remove_Head());
                break;
             case IN_BACK_OF:
                back_list.Insert_Tail(partition_list.Remove_Head());
                break;
             case COINCIDENT:
                // Don't add coincident planes to the tree
                break;
             case SPANNING:
                front_list.Insert_Tail(partition_list.Peek_Head());
                back_list.Insert_Tail(partition_list.Remove_Head());
                break;
          }
       }
       if (!front_list.Is_Empty())
       {
          tree->front = new BSP_Tree;
          R_Build_BSP_Tree(tree->front, front_list);
       }
       else
       {
          tree->front = new BSP_Tree;
          tree->front->leaf_node = 1;
       }

       if (!back_list.Is_Empty())
       {
          tree->back = new BSP_Tree;
          R_Build_BSP_Tree(tree->back, back_list);
       }
       else
       {
          tree->back = new BSP_Tree;
          tree->back->leaf_node = 1;
       }
    }
    else
    {
       cerr << "ERROR: Building BSP Tree with empty partition list" <<
endl;
       exit(1);
    }
}


void BSP_Tree::R_Convex_Hull_Insert(List<Point3>& p,
                                    BSP_Tree* tree,
                                    Convex_Hull3 convex_hull)
{
   Convex_Hull3 temp_front_list;
   Convex_Hull3 temp_back_list;
   List<Point3> temp_point_list;
   Facet3 temp_facet;
```

```
   if (!convex_hull.Is_Empty())
   {
      if (tree->leaf_node == 1)
      {
         // We made it to a leaf node, just copy the facets
         while (!convex_hull.Is_Empty())
            tree->polygons.Insert_Tail(convex_hull.Remove_Head());
      }
      else
      {
         // We made it to an internal node, test facets against
partition
         while (!convex_hull.Is_Empty())
         {
            switch (tree-
>partition.Classify_Polygon(convex_hull.Peek_Head()))
            {
               case IN_FRONT_OF:

temp_front_list.Insert_Tail(convex_hull.Remove_Head());
                  break;
               case IN_BACK_OF:
                  temp_back_list.Insert_Tail(convex_hull.Remove_Head());
                  break;
               case COINCIDENT:
                  temp_facet = convex_hull.Remove_Head();
                  temp_point_list.Insert_Tail(temp_facet.GetP1());
                  temp_point_list.Insert_Tail(temp_facet.GetP2());
                  temp_point_list.Insert_Tail(temp_facet.GetP3());
                  break;
               case SPANNING:
                  // This service will split the facet with the
partition plane
                  // it will add new points and facets to our lists
                  temp_facet = convex_hull.Remove_Head();
                  Split_Facet_With_Plane(p,
                                         temp_point_list,
                                         temp_facet,
                                         tree->partition,
                                         temp_front_list,
                                         temp_back_list);
                  break;
            }
         }

         // TBD take points in temp_point_list, Delaunay Triangulate
them
         // to form new facets and put them into both front and back
lists

         if (!temp_front_list.Is_Empty())
            R_Convex_Hull_Insert(p, tree->front, temp_front_list);
         if (!temp_back_list.Is_Empty())
            R_Convex_Hull_Insert(p, tree->back, temp_back_list);
      }
   }
   else
   {
      cerr << "WARNING: Merging an empty convex hull" << endl;
   }
}
```

```
void BSP_Tree::R_OOGL_Output(List<Point3>& p, BSP_Tree* t)
{
   Point3 tmp_point;
   Facet3 tmp_facet;
   Point3 tmp_point1, tmp_point2, tmp_point3;
   fstream fp;

   if (!t->polygons.Is_Empty())
   {
      List<Point3> tmp_p(p);
      List<Point3> tmp_p2(p);
      Convex_Hull3 tmp_front(t->polygons);
      //Convex_Hull3 tmp_back(*t->convex_hull_back_part);

      fp.open("bspchull.off", ios::app);

      if(fp.fail() | fp.bad())
      {
         cerr << "Error opening file bspchull.off for output." << endl;
         exit(1);
      }

      fp << "{ = OFF" << endl;
      fp << "   " << p.Num_Members() << " ";
      fp << t->polygons.Num_Members() << " " << "0" << endl;

      // Output Points Part
      tmp_p.Reset_Pointer();
      while (!tmp_p.Is_Empty())
      {
         tmp_point = tmp_p.Remove_Head();
         fp << "   " << tmp_point.GetX() << " " << tmp_point.GetY();
         fp << " " << tmp_point.GetZ() << endl;
      }

      // Output Front Convex Hull Part
      tmp_front.Reset_Pointer();
      while (!tmp_front.Is_Empty())
      {
         tmp_facet = tmp_front.Remove_Head();
         tmp_point1 = tmp_facet.GetP1();
         tmp_point2 = tmp_facet.GetP2();
         tmp_point3 = tmp_facet.GetP3();
         fp << "   3 " << tmp_point1.GetID() - 1 << " ";
         fp << tmp_point2.GetID() - 1 << " ";
         fp << tmp_point3.GetID() - 1 << endl;
      }
      fp << "}" << endl << endl;

      /*
      fp << "{ = OFF" << endl;
      fp << "   " << p.Num_Members() << " ";
      fp << t->convex_hull_back_part->Num_Members() << " " << "0" <<
endl;

      // Output Points Part
      tmp_p2.Reset_Pointer();
      while (!tmp_p2.Is_Empty())
      {
         tmp_point = tmp_p2.Remove_Head();
         fp << "   " << tmp_point.GetX() << " " << tmp_point.GetY();
         fp << " " << tmp_point.GetZ() << endl;
      }
```

```
        // Output Back Convex Hull Part
        tmp_back.Reset_Pointer();
        while (!tmp_back.Is_Empty())
        {
           tmp_facet = tmp_back.Remove_Head();
           tmp_point1 = tmp_facet.GetP1();
           tmp_point2 = tmp_facet.GetP2();
           tmp_point3 = tmp_facet.GetP3();
           fp << "   3 " << tmp_point1.GetID() - 1 << " ";
           fp << tmp_point2.GetID() - 1 << " ";
           fp << tmp_point3.GetID() - 1 << endl;
        }
        fp << "}" << endl << endl;
        */

        fp.close();
     }
     if (t->front != NULL)
        R_OOGL_Output(p, t->front);
     if (t->back != NULL)
        R_OOGL_Output(p, t->back);
  }


  void BSP_Tree::Show()
  {
     cout << "---" << endl;
     cout << "Root Node" << endl;
     cout << "   Partition: ";
     partition.Show();
     if (front != NULL)
        cout << "   Has front." << endl;
     else
        cout << "   No front." << endl;
     if (back != NULL)
        cout << "   Has back." << endl;
     else
        cout << "   No back." << endl;
     if (leaf_node == 1)
     {
        cout << "   Convex Hull: ";
        polygons.Show();
     }
     else
     {
        if (front != NULL)
        {
           cout << "   Front: ";
           R_Show(front);
        }
        else
           cout << "   Front: NULL" << endl;
        if (back != NULL)
        {
           cout << "   Back: ";
           R_Show(back);
        }
        else
           cout << "   Back: NULL" << endl;
     }

  }
```

```
void BSP_Tree::R_Show(BSP_Tree *tree)
{
   cout << "---" << endl;
   if (tree->leaf_node == 0)
   {
      cout << "   Partition: ";
      tree->partition.Show();
      if (tree->front != NULL)
         cout << "   Has front." << endl;
      else
         cout << "   No front." << endl;
      if (tree->back != NULL)
         cout << "   Has back." << endl;
      else
         cout << "   No back." << endl;

      if (tree->front != NULL)
      {
         cout << "   Front: ";
         R_Show(tree->front);
      }
      else
         cout << "   Front: NULL" << endl;
      if (tree->back != NULL)
      {
         cout << "   Back: ";
         R_Show(tree->back);
      }
      else
         cout << " Back: NULL" << endl;

   }
   else
   {
      cout << "Leaf Node" << endl;
      cout << "   Convex Hull: ";
      tree->polygons.Show();
   }
}


void BSP_Tree::Show_Class(Poly_Class p)
{
   switch (p)
   {
      case COINCIDENT:
         cout << "COINCIDENT";
         break;
      case IN_BACK_OF:
         cout << "IN_BACK_OF";
         break;
      case IN_FRONT_OF:
         cout << "IN_FRONT_OF";
         break;
      case SPANNING:
         cout << "SPANNING";
         break;
   }
}
```

```
#endif
/* chsplit.cc */

#ifdef __cplusplus

#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>

#include "chsplit.h"
#include "lineseg3.h"
#include "general.h"

void Delaunay_Triangulate (Point3 p1, Point3 p2, Point3 p3, Point3 p4,
                           Facet3& f1, Facet3& f2)
{
   Point3 A, B;
   Point3 center;
   double radius;
   double denom, n, m;

   // translate points to origin
   A = p2 - p1;
   B = p3 - p1;

   // calculate n & m
   denom = 2.0 * (A.Dot_Product(B)*A.Dot_Product(B) -
                  B.Dot_Product(B)*A.Dot_Product(A));
   n = (B.Dot_Product(A - B)*A.Dot_Product(A)) / denom;
   m = (B.Dot_Product(A - B)*B.Dot_Product(B)) / denom;

   // find center of circle and translate back to original position
   center = A*m + B*n + p1;

   // find radius of circle
   radius = p1.Distance(center);

   // find correct Delaunay triangulation
   if (p4.Distance(center) >= radius)
   {
      f1.SetPlane(p1, p2, p3);
      f2.SetPlane(p1, p3, p4);
   }
   else
   {
      f1.SetPlane(p1, p2, p4);
      f2.SetPlane(p2, p3, p4);
   }
}


Convex_Hull3 Triangulate(List<Point3>& new_points, Plane&
container_plane)
{
   List<Point3> p(new_points);
   List<Point2> translated_points;
   List<Facet_Index> triangulated;
   double A, B, C, D;
   double d, s, cx, cy, sx, sy;
   Point2 tmp_point2;
   Point3 tmp_point3;
   Facet_Index tmp_facet_index;
```

```
    Facet3 tmp_facet;
    Convex_Hull3 output_convex_hull;
    int i;

    // Translate and Rotate container plane to be the XY Plane
    // this will make it easy to translate to E2 coordinates
    // and call the triangulation of the points in E2
    A = container_plane.GetA();
    B = container_plane.GetB();
    C = container_plane.GetC();
    D = container_plane.GetD();
    if (DEQ0(C))
    {
       // Plane is perpendicular to the Z axis
    }
    else
    {
       // Translate container plane to Origin

       // Translation Matrix:
       // [1, 0, 0, 0]
       // [0, 1, 0, 0]
       // [0, 0, 1, 0]
       // [0, 0, d, 1]

       d = D / C;
       s = sqrt(A*A + B*B + C*C);
       cx = B/(s*sqrt(((B*B)/(A*A+B*B+C*C)) + ((C*C)/(A*A+B*B+C*C))));
       cy = sqrt(((B*B)/(A*A+B*B+C*C)) + ((C*C)/(A*A+B*B+C*C)));
       sx = C/(s*sqrt(((B*B)/(A*A+B*B+C*C)) + ((C*C)/(A*A+B*B+C*C))));
       sy = A/s;

       p.Reset_Pointer();
       while (!p.Is_Empty())
       {
          tmp_point3 = p.Remove_Head();
          tmp_point2.SetX(cy*tmp_point3.GetX() +
                          (cx*(d+tmp_point3.GetZ()) +
                              tmp_point3.GetY()*sx)*sy);
          tmp_point2.SetY(cx*tmp_point3.GetY() -
(d+tmp_point3.GetZ())*sx);
          tmp_point2.SetID(tmp_point3.GetID());
          translated_points.Insert_Tail(tmp_point2);
       }

    }

    // Triangulate the points in E2
    triangulated = Triangulate2(translated_points);

    // Translate back to the points in E3
    triangulated.Reset_Pointer();
    while (!triangulated.Is_Empty())
    {
       tmp_facet_index = triangulated.Remove_Head();
       new_points.Reset_Pointer();
       for(i=0; i<tmp_facet_index.GetP1Index(); i++)
          new_points.Increment_Pointer();
       tmp_facet.SetP1(new_points.Peek_At_Pointer());
       new_points.Reset_Pointer();
       for(i=0; i<tmp_facet_index.GetP2Index(); i++)
          new_points.Increment_Pointer();
       tmp_facet.SetP2(new_points.Peek_At_Pointer());
```

```
      new_points.Reset_Pointer();
      for(i=0; i<tmp_facet_index.GetP3Index(); i++)
         new_points.Increment_Pointer();
      tmp_facet.SetP3(new_points.Peek_At_Pointer());
      output_convex_hull.Insert_Tail(tmp_facet);
   }

   return output_convex_hull;
}


List<Facet_Index> Triangulate2(List<Point2>& new_points)
{
   List<Point2> p(new_points);
   fstream fp;
   Point2 tmp_point;
   List<Facet_Index> triangulation;
   int p1_index, p2_index, p3_index;
   Facet_Index tmp_facet_index;

   fp.open("points.tmp", ios::out);
   if(fp.fail() | fp.bad())
   {
      cerr << "Error opening file points.tmp for output." << endl;
      exit(1);
   }

   p.Reset_Pointer();
   while (!p.Is_Empty())
   {
      tmp_point = p.Remove_Head();
      fp << tmp_point.GetX() << " " << tmp_point.GetY() << endl;
   }

   fp.close();

   system("./voronoi -t <points.tmp >points.tri");
   system("echo ""0 0 0"" >> points.tri");

   fp.open("points.tri", ios::in);
   if(fp.fail() | fp.bad())
   {
      cerr << "Error opening file points.tri for input." << endl;
      exit(1);
   }

   triangulation.Clear();
   while (!fp.eof())
   {
      fp >> p1_index >> p2_index >> p3_index;

      // there is an aparent bug in many C++ implementations that
      // does not catch eof properly when reading multiple things
      // on a line, this line is a work around for that bug
      if ((p1_index == 0) && (p2_index == 0) && (p3_index == 0))
         break;

      tmp_facet_index.SetP1Index(p1_index);
      tmp_facet_index.SetP2Index(p2_index);
      tmp_facet_index.SetP3Index(p3_index);

      triangulation.Insert_Tail(tmp_facet_index);
   }
```

```
   fp.close();

   return triangulation;
}


void Split_Facet_With_Plane2 (List<Point3>& p,
                              List<Point3>& new_points,
                              Facet3 f,
                              Plane partition_plane,
                              Convex_Hull3& front_facets,
                              Convex_Hull3& back_facets)
{
   Poly_Class tmp_class;
   Poly_Class p1_class, p2_class, p3_class;
   Line_Segment3 sf1, sf2, sf3;
   Point3 intersect_sf1, intersect_sf2, intersect_sf3;
   Facet3 new_facet1, new_facet2, new_facet3;
   Point3 tmp_point;

   tmp_class = partition_plane.Classify_Polygon(f);
   switch (tmp_class)
   {
      case IN_FRONT_OF:
         front_facets.Insert_Tail(f);

         // Check for special case where 1 or 2 points are coincident
         // but facet is still considered In Front of partition plane

         p1_class = partition_plane.Classify_Polygon2(f.GetP1());
         p2_class = partition_plane.Classify_Polygon2(f.GetP2());
         p3_class = partition_plane.Classify_Polygon2(f.GetP3());
         if (p1_class == COINCIDENT)
            new_points.Insert_Tail(f.GetP1());
         if (p2_class == COINCIDENT)
            new_points.Insert_Tail(f.GetP2());
         if (p3_class == COINCIDENT)
            new_points.Insert_Tail(f.GetP3());

         break;

      case IN_BACK_OF:
         back_facets.Insert_Tail(f);

         // Check for special case where 1 or 2 points are coincident
         // but facet is still considered In Back of partition plane

         p1_class = partition_plane.Classify_Polygon2(f.GetP1());
         p2_class = partition_plane.Classify_Polygon2(f.GetP2());
         p3_class = partition_plane.Classify_Polygon2(f.GetP3());
         if (p1_class == COINCIDENT)
            new_points.Insert_Tail(f.GetP1());
         if (p2_class == COINCIDENT)
            new_points.Insert_Tail(f.GetP2());
         if (p3_class == COINCIDENT)
            new_points.Insert_Tail(f.GetP3());

         break;

      case COINCIDENT:
         front_facets.Insert_Tail(f);
         back_facets.Insert_Tail(f);
```

```
      new_points.Insert_Tail(f.GetP1());
      new_points.Insert_Tail(f.GetP2());
      new_points.Insert_Tail(f.GetP3());

      break;

   case SPANNING:
      // What side of the plane are the points on?
      p1_class = partition_plane.Classify_Polygon2(f.GetP1());
      p2_class = partition_plane.Classify_Polygon2(f.GetP2());
      p3_class = partition_plane.Classify_Polygon2(f.GetP3());
      sf1.SetSegment(f.GetP1(), f.GetP2());
      sf2.SetSegment(f.GetP2(), f.GetP3());
      sf3.SetSegment(f.GetP3(), f.GetP1());

      if ((p1_class == p2_class) &&
          (p1_class != COINCIDENT) &&
          (p2_class != COINCIDENT) &&
          (p3_class != COINCIDENT))
      {
         // Case 1:
         // p1 and p2 are on one side & p3 is on the other side

         intersect_sf2 = sf2.Intersection(partition_plane);
         intersect_sf3 = sf3.Intersection(partition_plane);
         intersect_sf2.SetID(p.Num_Members()+1);
         p.Insert_Tail(intersect_sf2);
         new_points.Insert_Tail(intersect_sf2);
         intersect_sf3.SetID(p.Num_Members()+1);
         p.Insert_Tail(intersect_sf3);
         new_points.Insert_Tail(intersect_sf3);
         new_facet1.SetP1(f.GetP3());
         new_facet1.SetP2(intersect_sf2);
         new_facet1.SetP3(intersect_sf3);
         Delaunay_Triangulate(f.GetP1(),
                              f.GetP2(),
                              intersect_sf2,
                              intersect_sf3,
                              new_facet2,
                              new_facet3);
         if (p3_class == IN_FRONT_OF)
         {
            front_facets.Insert_Tail(new_facet1);
            back_facets.Insert_Tail(new_facet2);
            back_facets.Insert_Tail(new_facet3);
         }
         else
         {
            back_facets.Insert_Tail(new_facet1);
            front_facets.Insert_Tail(new_facet2);
            front_facets.Insert_Tail(new_facet3);
         }
      }
      else if ((p1_class == p3_class) &&
               (p1_class != COINCIDENT) &&
               (p2_class != COINCIDENT) &&
               (p3_class != COINCIDENT))
      {
         // Case 2:
         // p1 and p3 are on one side & p2 is on the other side

         intersect_sf1 = sf1.Intersection(partition_plane);
         intersect_sf2 = sf2.Intersection(partition_plane);
```

```
    intersect_sf1.SetID(p.Num_Members()+1);
    p.Insert_Tail(intersect_sf1);
    new_points.Insert_Tail(intersect_sf1);
    intersect_sf2.SetID(p.Num_Members()+1);
    p.Insert_Tail(intersect_sf2);
    new_points.Insert_Tail(intersect_sf2);
    new_facet1.SetP1(f.GetP2());
    new_facet1.SetP2(intersect_sf1);
    new_facet1.SetP3(intersect_sf2);
    Delaunay_Triangulate(f.GetP1(),
                         intersect_sf1,
                         intersect_sf2,
                         f.GetP3(),
                         new_facet2,
                         new_facet3);
    if (p2_class == IN_FRONT_OF)
    {
       front_facets.Insert_Tail(new_facet1);
       back_facets.Insert_Tail(new_facet2);
       back_facets.Insert_Tail(new_facet3);
    }
    else
    {
       back_facets.Insert_Tail(new_facet1);
       front_facets.Insert_Tail(new_facet2);
       front_facets.Insert_Tail(new_facet3);
    }
}
else if ((p2_class == p3_class) &&
         (p1_class != COINCIDENT) &&
         (p2_class != COINCIDENT) &&
         (p3_class != COINCIDENT))
{
    // Case 3:
    // p2 and p3 are on one side & p1 is on the other side

    intersect_sf1 = sf1.Intersection(partition_plane);
    intersect_sf3 = sf3.Intersection(partition_plane);
    intersect_sf1.SetID(p.Num_Members()+1);
    p.Insert_Tail(intersect_sf1);
    new_points.Insert_Tail(intersect_sf1);
    intersect_sf3.SetID(p.Num_Members()+1);
    p.Insert_Tail(intersect_sf3);
    new_points.Insert_Tail(intersect_sf3);
    new_facet1.SetP1(f.GetP1());
    new_facet1.SetP2(intersect_sf1);
    new_facet1.SetP3(intersect_sf3);
    Delaunay_Triangulate(f.GetP2(),
                         f.GetP3(),
                         intersect_sf3,
                         intersect_sf1,
                         new_facet2,
                         new_facet3);
    if (p1_class == IN_FRONT_OF)
    {
       front_facets.Insert_Tail(new_facet1);
       back_facets.Insert_Tail(new_facet2);
       back_facets.Insert_Tail(new_facet3);
    }
    else
    {
       back_facets.Insert_Tail(new_facet1);
       front_facets.Insert_Tail(new_facet2);
```

```
                front_facets.Insert_Tail(new_facet3);
            }
        }
        else
        {
            // Case 4-6:
            // 1 point is coincident and the other two points
            // are on opposite sides of the plane
            if (p1_class == COINCIDENT)
            {
                // Case 4:
                // p1 is coincident, p2 and p3 are on opposite sides
                intersect_sf2 = sf2.Intersection(partition_plane);
                intersect_sf2.SetID(p.Num_Members()+1);
                p.Insert_Tail(intersect_sf2);
                new_points.Insert_Tail(intersect_sf2);
                new_facet1.SetP1(f.GetP1());
                new_facet1.SetP2(f.GetP2());
                new_facet1.SetP3(intersect_sf2);
                new_facet2.SetP1(f.GetP1());
                new_facet2.SetP2(f.GetP3());
                new_facet2.SetP3(intersect_sf2);

                if ((p2_class == IN_FRONT_OF) && (p3_class ==
IN_BACK_OF))
                {
                    front_facets.Insert_Tail(new_facet1);
                    back_facets.Insert_Tail(new_facet2);
                }
                else if ((p2_class == IN_BACK_OF) && (p3_class ==
IN_FRONT_OF))
                {
                    back_facets.Insert_Tail(new_facet1);
                    front_facets.Insert_Tail(new_facet2);
                }
                else
                {
                    // Error we are messed up somewhere
                    cout << "ERROR Messed up spanning cases!" << endl;
                    exit(1);
                }
            }
            else if (p2_class == COINCIDENT)
            {
                // Case 5:
                // p2 is coincident, p1 and p3 are on opposite sides
                intersect_sf3 = sf3.Intersection(partition_plane);
                intersect_sf3.SetID(p.Num_Members()+1);
                p.Insert_Tail(intersect_sf3);
                new_points.Insert_Tail(intersect_sf3);
                new_facet1.SetP1(f.GetP2());
                new_facet1.SetP2(f.GetP3());
                new_facet1.SetP3(intersect_sf3);
                new_facet2.SetP1(f.GetP2());
                new_facet2.SetP2(f.GetP1());
                new_facet2.SetP3(intersect_sf3);

                if ((p1_class == IN_FRONT_OF) && (p3_class ==
IN_BACK_OF))
                {
                    back_facets.Insert_Tail(new_facet1);
                    front_facets.Insert_Tail(new_facet2);
                }
```

```
                else if ((p1_class == IN_BACK_OF) && (p3_class ==
IN_FRONT_OF))
                {
                    front_facets.Insert_Tail(new_facet1);
                    back_facets.Insert_Tail(new_facet2);
                }
                else
                {
                   // Error we are messed up somewhere
                   cerr << "ERROR Messed up spanning cases!" << endl;
                   exit(1);
                }
             }
             else if (p3_class == COINCIDENT)
             {
                // Case 6:
                // p3 is coincident, p2 and p1 are on opposite sides
                intersect_sf1 = sf1.Intersection(partition_plane);
                intersect_sf1.SetID(p.Num_Members()+1);
                p.Insert_Tail(intersect_sf1);
                new_points.Insert_Tail(intersect_sf1);
                new_facet1.SetP1(f.GetP3());
                new_facet1.SetP2(f.GetP2());
                new_facet1.SetP3(intersect_sf1);
                new_facet2.SetP1(f.GetP3());
                new_facet2.SetP2(f.GetP1());
                new_facet2.SetP3(intersect_sf1);

                if ((p1_class == IN_FRONT_OF) && (p2_class ==
IN_BACK_OF))
                {
                   back_facets.Insert_Tail(new_facet1);
                   front_facets.Insert_Tail(new_facet2);
                }
                else if ((p1_class == IN_BACK_OF) && (p2_class ==
IN_FRONT_OF))
                {
                   front_facets.Insert_Tail(new_facet1);
                   back_facets.Insert_Tail(new_facet2);
                }
                else
                {
                   // Error we are messed up somewhere
                   cerr << "ERROR Messed up spanning cases!" << endl;
                   exit(1);
                }
             }
             else
             {
                // Why in the heck did we get here, we are out
                // of spanning cases to check
                cerr << "ERROR Out of spanning cases!" << endl;
                exit(1);
             }
          }
          break;
   }
}


void Split_Facet_With_Plane (List<Point3>& p,
                             List<Point3>& new_points,
                             Facet3& f,
```

```
                                Plane& partition_plane,
                                Convex_Hull3& front_convex_hull,
                                Convex_Hull3& back_convex_hull)
{
   Poly_Class tmp_class;
   Poly_Class p1_class, p2_class, p3_class;
   Line_Segment3 sf1, sf2, sf3;
   Point3 intersect_sf1, intersect_sf2, intersect_sf3;
   Facet3 new_facet1, new_facet2, new_facet3;
   Point3 tmp_point;

   tmp_class = partition_plane.Classify_Polygon(f);
   switch (tmp_class)
   {
      case IN_FRONT_OF:
         front_convex_hull.Insert_Tail(f);

         // Check for special case where 1 or 2 points are coincident
         // but facet is still considered In Front of partition plane

         p1_class = partition_plane.Classify_Polygon2(f.GetP1());
         p2_class = partition_plane.Classify_Polygon2(f.GetP2());
         p3_class = partition_plane.Classify_Polygon2(f.GetP3());
         if (p1_class == COINCIDENT)
            new_points.Insert_Tail(f.GetP1());
         if (p2_class == COINCIDENT)
            new_points.Insert_Tail(f.GetP2());
         if (p3_class == COINCIDENT)
            new_points.Insert_Tail(f.GetP3());

         break;

      case IN_BACK_OF:
         back_convex_hull.Insert_Tail(f);

         // Check for special case where 1 or 2 points are coincident
         // but facet is still considered In Back of partition plane

         p1_class = partition_plane.Classify_Polygon2(f.GetP1());
         p2_class = partition_plane.Classify_Polygon2(f.GetP2());
         p3_class = partition_plane.Classify_Polygon2(f.GetP3());
         if (p1_class == COINCIDENT)
            new_points.Insert_Tail(f.GetP1());
         if (p2_class == COINCIDENT)
            new_points.Insert_Tail(f.GetP2());
         if (p3_class == COINCIDENT)
            new_points.Insert_Tail(f.GetP3());

         break;

      case COINCIDENT:
         front_convex_hull.Insert_Tail(f);
         back_convex_hull.Insert_Tail(f);
         new_points.Insert_Tail(f.GetP1());
         new_points.Insert_Tail(f.GetP2());
         new_points.Insert_Tail(f.GetP3());

         break;

      case SPANNING:
         // What side of the plane are the points on?
         p1_class = partition_plane.Classify_Polygon2(f.GetP1());
         p2_class = partition_plane.Classify_Polygon2(f.GetP2());
```

```
p3_class = partition_plane.Classify_Polygon2(f.GetP3());
sf1.SetSegment(f.GetP1(), f.GetP2());
sf2.SetSegment(f.GetP2(), f.GetP3());
sf3.SetSegment(f.GetP3(), f.GetP1());

if (p1_class == p2_class)
{
   // Case 1:
   // p1 and p2 are on one side & p3 is on the other side

   intersect_sf2 = sf2.Intersection(partition_plane);
   intersect_sf3 = sf3.Intersection(partition_plane);
   intersect_sf2.SetID(p.Num_Members()+1);
   p.Insert_Tail(intersect_sf2);
   new_points.Insert_Tail(intersect_sf2);
   intersect_sf3.SetID(p.Num_Members()+1);
   p.Insert_Tail(intersect_sf3);
   new_points.Insert_Tail(intersect_sf3);
   new_facet1.SetP1(f.GetP3());
   new_facet1.SetP2(intersect_sf2);
   new_facet1.SetP3(intersect_sf3);
   Delaunay_Triangulate(f.GetP1(),
                        f.GetP2(),
                        intersect_sf2,
                        intersect_sf3,
                        new_facet2,
                        new_facet3);
   if (p3_class == IN_FRONT_OF)
   {
      front_convex_hull.Insert_Tail(new_facet1);
      back_convex_hull.Insert_Tail(new_facet2);
      back_convex_hull.Insert_Tail(new_facet3);
   }
   else
   {
      back_convex_hull.Insert_Tail(new_facet1);
      front_convex_hull.Insert_Tail(new_facet2);
      front_convex_hull.Insert_Tail(new_facet3);
   }
}
else if (p1_class == p3_class)
{
   // Case 2:
   // p1 and p3 are on one side & p2 is on the other side

   intersect_sf1 = sf1.Intersection(partition_plane);
   intersect_sf2 = sf2.Intersection(partition_plane);
   intersect_sf1.SetID(p.Num_Members()+1);
   p.Insert_Tail(intersect_sf1);
   new_points.Insert_Tail(intersect_sf1);
   intersect_sf2.SetID(p.Num_Members()+1);
   p.Insert_Tail(intersect_sf2);
   new_points.Insert_Tail(intersect_sf2);
   new_facet1.SetP1(f.GetP2());
   new_facet1.SetP2(intersect_sf1);
   new_facet1.SetP3(intersect_sf2);
   Delaunay_Triangulate(f.GetP1(),
                        intersect_sf1,
                        intersect_sf2,
                        f.GetP3(),
                        new_facet2,
                        new_facet3);
   if (p2_class == IN_FRONT_OF)
```

```
    {
       front_convex_hull.Insert_Tail(new_facet1);
       back_convex_hull.Insert_Tail(new_facet2);
       back_convex_hull.Insert_Tail(new_facet3);
    }
    else
    {
       back_convex_hull.Insert_Tail(new_facet1);
       front_convex_hull.Insert_Tail(new_facet2);
       front_convex_hull.Insert_Tail(new_facet3);
    }
 }
 else if (p2_class == p3_class)
 {
    // Case 3:
    // p2 and p3 are on one side & p1 is on the other side

    intersect_sf1 = sf1.Intersection(partition_plane);
    intersect_sf3 = sf3.Intersection(partition_plane);
    intersect_sf1.SetID(p.Num_Members()+1);
    p.Insert_Tail(intersect_sf1);
    new_points.Insert_Tail(intersect_sf1);
    intersect_sf3.SetID(p.Num_Members()+1);
    p.Insert_Tail(intersect_sf3);
    new_points.Insert_Tail(intersect_sf3);
    new_facet1.SetP1(f.GetP1());
    new_facet1.SetP2(intersect_sf1);
    new_facet1.SetP3(intersect_sf3);
    Delaunay_Triangulate(f.GetP2(),
                         f.GetP3(),
                         intersect_sf3,
                         intersect_sf1,
                         new_facet2,
                         new_facet3);
    if (p1_class == IN_FRONT_OF)
    {
       front_convex_hull.Insert_Tail(new_facet1);
       back_convex_hull.Insert_Tail(new_facet2);
       back_convex_hull.Insert_Tail(new_facet3);
    }
    else
    {
       back_convex_hull.Insert_Tail(new_facet1);
       front_convex_hull.Insert_Tail(new_facet2);
       front_convex_hull.Insert_Tail(new_facet3);
    }
 }
 else
 {
    // Case 4-6:
    // 1 point is coincident and the other two points
    // are on opposite sides of the plane
    if (p1_class == COINCIDENT)
    {
       // Case 4:
       // p1 is coincident, p2 and p3 are on opposite sides
       intersect_sf2 = sf2.Intersection(partition_plane);
       intersect_sf2.SetID(p.Num_Members()+1);
       p.Insert_Tail(intersect_sf2);
       new_points.Insert_Tail(intersect_sf2);
       new_facet1.SetP1(f.GetP1());
       new_facet1.SetP2(f.GetP2());
       new_facet1.SetP3(intersect_sf2);
```

```
                new_facet2.SetP1(f.GetP1());
                new_facet2.SetP2(f.GetP3());
                new_facet2.SetP3(intersect_sf2);

                if ((p2_class == IN_FRONT_OF) && (p3_class ==
IN_BACK_OF))
                {
                   front_convex_hull.Insert_Tail(new_facet1);
                   back_convex_hull.Insert_Tail(new_facet2);
                }
                else if ((p2_class == IN_BACK_OF) && (p3_class ==
IN_FRONT_OF))
                {
                   back_convex_hull.Insert_Tail(new_facet1);
                   front_convex_hull.Insert_Tail(new_facet2);
                }
                else
                {
                   // Error we are messed up somewhere
                   cout << "ERROR Messed up spanning cases!" << endl;
                   exit(1);
                }
             }
             else if (p2_class == COINCIDENT)
             {
                // Case 5:
                // p2 is coincident, p1 and p3 are on opposite sides
                intersect_sf3 = sf3.Intersection(partition_plane);
                intersect_sf3.SetID(p.Num_Members()+1);
                p.Insert_Tail(intersect_sf3);
                new_points.Insert_Tail(intersect_sf3);
                new_facet1.SetP1(f.GetP2());
                new_facet1.SetP2(f.GetP3());
                new_facet1.SetP3(intersect_sf3);
                new_facet2.SetP1(f.GetP2());
                new_facet2.SetP2(f.GetP1());
                new_facet2.SetP3(intersect_sf3);

                if ((p1_class == IN_FRONT_OF) && (p3_class ==
IN_BACK_OF))
                {
                   back_convex_hull.Insert_Tail(new_facet1);
                   front_convex_hull.Insert_Tail(new_facet2);
                }
                else if ((p1_class == IN_BACK_OF) && (p3_class ==
IN_FRONT_OF))
                {
                    front_convex_hull.Insert_Tail(new_facet1);
                    back_convex_hull.Insert_Tail(new_facet2);
                }
                else
                {
                   // Error we are messed up somewhere
                   cerr << "ERROR Messed up spanning cases!" << endl;
                   exit(1);
                }
             }
             else if (p3_class == COINCIDENT)
             {
                // Case 6:
                // p3 is coincident, p2 and p1 are on opposite sides
                intersect_sf1 = sf1.Intersection(partition_plane);
                intersect_sf1.SetID(p.Num_Members()+1);
```

```
                p.Insert_Tail(intersect_sf1);
                new_points.Insert_Tail(intersect_sf1);
                new_facet1.SetP1(f.GetP3());
                new_facet1.SetP2(f.GetP2());
                new_facet1.SetP3(intersect_sf1);
                new_facet2.SetP1(f.GetP3());
                new_facet2.SetP2(f.GetP1());
                new_facet2.SetP3(intersect_sf1);

                if ((p1_class == IN_FRONT_OF) && (p2_class ==
IN_BACK_OF))
                {
                   back_convex_hull.Insert_Tail(new_facet1);
                   front_convex_hull.Insert_Tail(new_facet2);
                }
                else if ((p1_class == IN_BACK_OF) && (p2_class ==
IN_FRONT_OF))
                {
                   front_convex_hull.Insert_Tail(new_facet1);
                   back_convex_hull.Insert_Tail(new_facet2);
                }
                else
                {
                   // Error we are messed up somewhere
                   cerr << "ERROR Messed up spanning cases!" << endl;
                   exit(1);
                }
             }
             else
             {
                // Why in the heck did we get here, we are out
                // of spanning cases to check
                cerr << "ERROR Out of spanning cases!" << endl;
                exit(1);
             }
          }
          break;
   }
}


void Split_Polygon_With_Plane (Facet3& polygon,
                               Plane& partition_plane,
                               Convex_Hull3& front_piece,
                               Convex_Hull3& back_piece)
{
   List<Point3> trash_points;
   List<Point3> trash_points2;

   Split_Facet_With_Plane(trash_points,
                          trash_points2,
                          polygon,
                          partition_plane,
                          front_piece,
                          back_piece);
}


void Split_Convex_Hull_With_Plane (List<Point3>& p,
                                   List<Point3>& new_points,
                                   Convex_Hull3& convex_hull,
                                   Plane& partition_plane,
                                   Convex_Hull3& front_convex_hull,
```

```
                                        Convex_Hull3& back_convex_hull)
{
   Convex_Hull3 tmp_convex_hull(convex_hull);
   Facet3 tmp_facet;

   tmp_convex_hull.Reset_Pointer();
   while (!tmp_convex_hull.Is_Empty())
   {
      tmp_facet = tmp_convex_hull.Remove_Head();
      Split_Facet_With_Plane(p,
                             new_points,
                             tmp_facet,
                             partition_plane,
                             front_convex_hull,
                             back_convex_hull);
   }
}


#endif
/* chull.cc */

#ifdef __cplusplus

#ifdef DEBUG
#include <iostream.h>
#endif
#include <assert.h>

#include "chull2.h"
#include "subfacet2.h"
#include "general.h"

// Convex Hull - Giftwrapping
// 1.  T <- 0
// 2.  F <- find an initial convex hull facet;
// 3.  Output F
// 4.  T <- subfacets of F;
// 5.  while (T /= 0) do
// 6.     F <- T (* copy of front element from list *);
// 7.     F' <- facet sharing e with F; (* giftwrapping *)
// 8.     Output F'
// 9.     Insert into T all subfacets of F' not yet present
//           and delete all those already present.
// 10. end while

void Convex_Hull2::GiftWrapping (List<Point2>& p)
{
}


// Find initial convex hull facet
Facet2 Convex_Hull2::Find_Initial_Facet (List<Point2>& p)
{
   List<Point2> tmp_p(p);
   List<Point2> tmp_p2(p);
   Point2 p1, p2, n, a;
   Point2 temp_point;
   Facet2 temp_facet;
   double max_rho, r;
   int i, loc;

#ifdef DEBUG
```

```
   cout << "      Finding Initial Facet" << endl;
#endif
   // Must have 2 or more points to form a facet in E2
   assert(p.Num_Members() >= 2);

   // p1 <- lexicographically smallest point in p
   tmp_p.Reset_Pointer();
   p1 = tmp_p.Remove_Head();
   while (!tmp_p.Is_Empty())
   {
      temp_point = tmp_p.Remove_Head();
      if (temp_point.GetX() < p1.GetX())
         p1 = temp_point;
      else if (temp_point.GetX() == p1.GetX())
         if (temp_point.GetY() < p1.GetY())
            p1 = temp_point;
   }

   temp_facet.SetP1(p1);                  // add vert 1 to facet
   n.SetX((double)1.0);                   // n = (1,0,0)
   a.SetY((double)1.0);                   // a = normal to n (0,1,0)

#ifdef DEBUG
   cout << "         Initial facet p1=";
   p1.Show_Full();
   cout << endl;
   cout << "         n=";
   n.Show_Full();
   cout << endl;
   cout << "         a=";
   a.Show_Full();
   cout << endl;
#endif

   // p2 = point of max rho
   max_rho = -MAX_DOUBLE;
   tmp_p2.Reset_Pointer();
   while (!tmp_p2.Is_Empty())
   {
      temp_point = tmp_p2.Remove_Head();
      if (temp_point != p1)
      {
         //r = p1.Rho(a, n, temp_point);
         if (r > max_rho)
         {
            max_rho = r;
            p2 = temp_point;
         }
      }
   }

   temp_facet.SetP2(p2);                  // add vert 2 to facet

   if (DEQ0(max_rho))
   {
      temp_point = n;                     // swap n and a
      n = a;
      a = temp_point;
   }
   else
   {
      //n = n.Compute_New_N(max_rho, a);
      //a = a.Compute_New_A(n, p1, p2);
```

```
    }

#ifdef DEBUG
    cout << "          Initial facet p2=";
    p2.Show_Full();
    cout << endl;
    cout << "          max_rho=" << max_rho << endl;
    cout << "          n=";
    n.Show_Full();
    cout << endl;
    cout << "          a=";
    a.Show_Full();
    cout << endl;
#endif


    return temp_facet;
}



#endif
/* chull.cc */

#ifdef __cplusplus

#ifdef DEBUG
#include <iostream.h>
#endif
#include <assert.h>

#include "chull3.h"
#include "subfacet3.h"
#include "general.h"
#include "plane.h"

// Convex Hull - Giftwrapping
// 1.  T <- 0
// 2.  F <- find an initial convex hull facet;
// 3.  Output F
// 4.  T <- subfacets of F;
// 5.  while (T /= 0) do
// 6.     F <- T (* copy of front element from list *);
// 7.     F' <- facet sharing e with F; (* giftwrapping *)
// 8.     Output F'
// 9.     Insert into T all subfacets of F' not yet present
//           and delete all those already present.
// 10. end while

void Convex_Hull3::GiftWrapping (List<Point3>& p)
{
   List<Facet3> T;  // T really should be a SubFacet List, but the
extra point
                    // is needed to carry along the third point of the
Facet
                    // that the SubFacet belongs to for the angle
calculations
                    // to be done in the future using this SubFacet.

   Facet3 F;                // Current Facet
   Facet3 SF1, SF2, SF3;    // SubFacets of F

   Facet3 F_prime;          // Next Facet
```

```
   Facet3 SF4, SF5, SF6;  // SubFacets of F'

   SubFacet3 temp_sf;
   Point3 p_prime;
   double max_angle;
   double min_angle;
   Point3 temp_point;
   double temp_angle;
   Plane temp_plane;
   int i;

   // 1.  T <- 0
   T.Clear();

   // 2.  F <- find an initial convex hull facet;
   F = Find_Initial_Facet(p);
   // 3.  Output F
   i = 1;
   F.SetID(i);
   (*this).Insert_Tail(F);
#ifdef DEBUG
   cout << "      Found facet " << i << ": ";
   F.Show();
   cout << endl;
#endif

   // 4.  T <- subfacets of F;
   SF1.SetP1(F.GetP1());
   SF1.SetP2(F.GetP2());
   SF1.SetP3(F.GetP3());
   SF2.SetP1(F.GetP2());
   SF2.SetP2(F.GetP3());
   SF2.SetP3(F.GetP1());
   SF3.SetP1(F.GetP3());
   SF3.SetP2(F.GetP1());
   SF3.SetP3(F.GetP2());
   T.Insert_Tail(SF1);
   T.Insert_Tail(SF2);
   T.Insert_Tail(SF3);

   // 5.  while (T /= 0) do
   while (!T.Is_Empty() && (i< 60))
   {
      List<Point3> tmp_p(p);
      // 6.     F <- T (* extract front element from list *);
      F = T.Peek_Head();

      // SubFacet we are interested in
      SF1.SetP1(F.GetP1());
      SF1.SetP2(F.GetP2());
      SF1.SetP3(F.GetP3());

      // 7.     F' <- facet sharing e with F; (* giftwrapping *)

      // F' shares SF1 with F and p' is the point which forms
      // the greatest angle between the hyperplanes of F and F'
      min_angle = MAX_DOUBLE;
      tmp_p.Reset_Pointer();
      while (!tmp_p.Is_Empty())
      {
         temp_point = tmp_p.Remove_Head();
         if ((temp_point != F.GetP1()) &&
             (temp_point != F.GetP2()) &&
```

```
            (temp_point != F.GetP3()))
        {
           temp_plane.SetPlane(SF1.GetP1(),
                               SF1.GetP2(),
                               temp_point);
           temp_angle = F.Angle(temp_plane);
           if (temp_angle < min_angle)
           {
              min_angle = temp_angle;
              p_prime = temp_point;
           }
        }
     }
     // Subfacets made from F'
     SF4.SetP1(SF1.GetP1());
     SF4.SetP2(SF1.GetP2());
     SF4.SetP3(p_prime);
     SF5.SetP1(SF1.GetP2());
     SF5.SetP2(p_prime);
     SF5.SetP3(SF1.GetP1());
     SF6.SetP1(p_prime);
     SF6.SetP2(SF1.GetP1());
     SF6.SetP3(SF1.GetP2());
     // F' is facet sharing e with F
     F_prime.SetP1(SF1.GetP1());
     F_prime.SetP2(SF1.GetP2());
     F_prime.SetP3(p_prime);
     // 8.     Output F'
     i++;
     F_prime.SetID(i);
     (*this).Insert_Tail(F_prime);
#ifdef DEBUG
     cout << "      Found facet " << i << ": ";
     F_prime.Show();
     cout << endl;
#endif

     // 9.     Insert into T all subfacets of F' not yet present
     //           and delete all those already present.
     if (T.Find2(SF4))
        T.Remove_At_Pointer();
     else
        T.Insert_Tail(SF4);
     if (T.Find2(SF5))
        T.Remove_At_Pointer();
     else
        T.Insert_Tail(SF5);
     if (T.Find2(SF6))
        T.Remove_At_Pointer();
     else
        T.Insert_Tail(SF6);
  }
}


// Find initial convex hull facet
Facet3 Convex_Hull3::Find_Initial_Facet (List<Point3>& p)
{
   List<Point3> tmp_p(p);
   List<Point3> tmp_p2(p);
   List<Point3> tmp_p3(p);
   Point3 p1, p2, p3, n, a;
   Point3 temp_point;
```

```
   Facet3 temp_facet;
   double max_rho, r;
   int i, loc;

#ifdef DEBUG
   cout << "      Finding Initial Facet" << endl;
#endif
   // Must have 3 or more points to form a facet in E3
   assert(p.Num_Members() >= 3);

   // p1 <- lexicographically smallest point in p
   tmp_p.Reset_Pointer();
   p1 = tmp_p.Remove_Head();
   while (!tmp_p.Is_Empty())
   {
      temp_point = tmp_p.Remove_Head();
      if (temp_point.GetX() < p1.GetX())
         p1 = temp_point;
      else if (temp_point.GetX() == p1.GetX())
         if (temp_point.GetY() < p1.GetY())
            p1 = temp_point;
         else if (temp_point.GetY() == p1.GetY())
            if (temp_point.GetZ() < p1.GetZ())
               p1 = temp_point;
   }

   temp_facet.SetP1(p1);                  // add vert 1 to facet
   n.SetX((double)1.0);                   // n = (1,0,0)
   a.SetY((double)1.0);                   // a = normal to n (0,1,0)

#ifdef DEBUG
   cout << "         Initial facet p1=";
   p1.Show_Full();
   cout << endl;
   cout << "         n=";
   n.Show_Full();
   cout << endl;
   cout << "         a=";
   a.Show_Full();
   cout << endl;
#endif

   // p2 = point of max rho
   max_rho = -MAX_DOUBLE;
   tmp_p2.Reset_Pointer();
   while (!tmp_p2.Is_Empty())
   {
      temp_point = tmp_p2.Remove_Head();
      if (temp_point != p1)
      {
         r = p1.Rho(a, n, temp_point);
         if (r > max_rho)
         {
            max_rho = r;
            p2 = temp_point;
         }
      }
   }

   temp_facet.SetP2(p2);                  // add vert 2 to facet

   if (DEQ0(max_rho))
   {
```

```
      temp_point = n;                    // swap n and a
      n = a;
      a = temp_point;
   }
   else
   {
      n = n.Compute_New_N(max_rho, a);
      a = a.Compute_New_A(n, p1, p2);
   }

#ifdef DEBUG
   cout << "          Initial facet p2=";
   p2.Show_Full();
   cout << endl;
   cout << "          max_rho=" << max_rho << endl;
   cout << "          n=";
   n.Show_Full();
   cout << endl;
   cout << "          a=";
   a.Show_Full();
   cout << endl;
#endif

   // p3 = point of max rho
   max_rho = -MAX_DOUBLE;
   tmp_p3.Reset_Pointer();
   while (!tmp_p3.Is_Empty())
   {
      temp_point = tmp_p3.Remove_Head();
      if ((temp_point != p1) && (temp_point != p2))
      {
         r = p1.Rho(a, n, temp_point);
         if (r > max_rho)
         {
            max_rho = r;
            p3 = temp_point;
         }
      }
   }

   temp_facet.SetP3(p3);                 // add vert 3 to facet

#ifdef DEBUG
   cout << "          Initial facet p3=";
   p3.Show_Full();
   cout << endl;
   cout << "          max_rho=" << max_rho << endl;
   cout << "       Initial Facet f=";
   temp_facet.Show();
   cout << endl;
   if (Verify_Initial_Facet(p,temp_facet))
      cout << "          Initial Facet Verified" << endl;
   else
      cout << "          Initial Facet NOT Verified" << endl;
#endif

   return temp_facet;
}


#ifdef DEBUG
// Verify Initial Facet Correctness
int Convex_Hull3::Verify_Initial_Facet (List<Point3>& p, Facet3& f)
```

```
{
   List<Point3> tmp_p(p);
   int status;
   Point3 temp_point;
   int front_count, back_count;
   Poly_Class temp_side;

   front_count = 0;
   back_count = 0;
   tmp_p.Reset_Pointer();
   while (!tmp_p.Is_Empty())
   {
      temp_point = tmp_p.Remove_Head();
      if ((temp_point != f.GetP1()) &&
          (temp_point != f.GetP2()) &&
          (temp_point != f.GetP3()))
      {
         temp_side = f.Classify_Polygon2(temp_point);
         assert((temp_side == IN_FRONT_OF) || (temp_side ==
IN_BACK_OF));
         if (temp_side == IN_FRONT_OF)
            front_count++;
         else
            back_count++;
      }
   }

   if (((front_count != 0) && (back_count == 0)) ||
       ((front_count == 0) && (back_count != 0)))
      status = 1;
   else
      status = 0;

   return status;
}

int Convex_Hull3::Verify_Split_Convex_Hull (Plane& partition_plane)
{
   int status;
   Convex_Hull3 tmp_convex_hull(*this);
   Facet3 tmp_facet;

   status = 1;
   tmp_convex_hull.Reset_Pointer();
   while(!tmp_convex_hull.Is_Empty())
   {
      tmp_facet = tmp_convex_hull.Remove_Head();
      if (partition_plane.Classify_Polygon(tmp_facet) == SPANNING)
      {
         status = 0;
         cout << "Failed Facet: " << tmp_facet << endl;
         cout << "Failed Plane: ";
         partition_plane.Show();
         cout << endl;
      }
   }
   return status;
}

#endif


#endif
```

```
/* ctz.cc */

#ifdef __cplusplus

// Computing a Constrained Tetrahedrization

//Algorithm presented in paper:
//      Hazlewood, Carol.  Using Binary Space Paritions to
//      Approximate Constrained Tetrahedrizations.  September
//      18, 1993.

//constrain(P', F, T)
//   begin
//      construct B, the BSP Tree of F;
//      construct C, the convex hull of P';
//      for j = 1 to k do
//         Kj <- plane(fj) intersect C;
//      for j = 1 to k do
//         compute constraints for Kj;
//      for j = 1 to k do
//         triangulate Kj with constraints;
//         triangulate facets of C with constraints;
//      for i = 1 to l do
//         Ti <- tetrahedrization of Ri;
//         T <- T union Ti;
//   end.

#include <iostream.h>
#include "ctz.h"
#include "bsp_tree.h"
#include "chull3.h"
#include "oogl.h"

void Constrain (List<Point3>& P_prime, List<Plane>& F)
{
   BSP_Tree B;                          // BSP Tree B
   Convex_Hull3 C;                       // Convex Hull C

   // Construct BSP Tree B from F
   cout << "   Building BSP Tree..." << endl;
   B.Build_BSP_Tree(F);
#ifdef DEBUG
   B.Show();
#endif

   // Construct Convex Hull C from P'
   cout << "   Building Convex Hull..." << endl;
   C.GiftWrapping(P_prime);
#ifdef DEBUG
   C.Show();
#endif

   // for j = 1 to k do
   //    Kj <- plane(fj) intersect C;
   // This has been changed from the original paper from a FOR loop
   // into a recursive tree algorithm
   cout << "   Merging Convex Hull Into BSP Tree..." << endl;
   B.Convex_Hull_Insert(P_prime, C);
   //B.Merge_Convex_Hull(C);
#ifdef DEBUG
   B.Show();
#endif
```

```
    // FUTURE ALGORITHM EXPANSION NEEDED
    // for (j=0; j<k; j++)
    //    compute constraints for Kj;

    // for (j=0; j<k; j++)
    //    Triangulate Kj with constraints;
    //    Triangulate facets of C with constraints;

    // for (i=0; i<k; i++)
    //    Ti <- tetrahedrization of Ri;
    //    T <- T union Ti;


    // Output OOGL File For Triangles
    cout << "   Outputing triangs.off file..." << endl;
    Triangles_2_OOGL(F);
    // Output OOGL File For Convex Hull
    cout << "   Outputing chull.off file..." << endl;
    Convex_Hull_2_OOGL(P_prime, C);
    // Output OOGL File For BSP Tree with Convex Hulls
    cout << "   Outputing bspchull.off file..." << endl;
    BSP_Tree_w_Convex_Hulls_2_OOGL(P_prime, B);
}


#endif
/* facet2.cc - Facet2 Class Implementation */

#ifdef __cplusplus

#include "facet2.h"


// Operators
Facet2& Facet2::operator=(const Facet2 &rhs)
{
   if (this == &rhs) return *this;
   p1 = rhs.p1;
   p2 = rhs.p2;
   id = rhs.id;
   return *this;
}

int Facet2::sf_equal(const Facet2& f2)
{
   return (((p1 == f2.p1) && (p2 == f2.p2)) ||
           ((p1 == f2.p2) && (p2 == f2.p1)));
}


// '<<' I/O Stream Operator
ostream& operator<<(ostream& s, Facet2 f)
{
   s << "{" << f.p1 << "," << f.p2 << "}";
   return s;
}


// Accessors

void Facet2::SetFacet(Point2 P1, Point2 P2)
{
```

```
    SetSegment(P1, P2);
}


void Facet2::SetID(int ID)
{
    id = ID;
}


int Facet2::GetID()
{
    return id;
}


// Services

void Facet2::Show()
{
    cout << "{";
    p1.Show();
    cout << ", ";
    p2.Show();
    cout << "}";
}


void Facet2::Show_Full()
{
    cout << "{";
    p1.Show_Full();
    cout << ", ";
    p2.Show_Full();
    cout << "}" << endl;
}


#endif
/* facet3.cc - Facet3 Class Implementation */

#ifdef __cplusplus

#include "facet3.h"


// Operators

Facet3& Facet3::operator=(const Facet3 &rhs)
{
    if (this == &rhs) return *this;
    p1 = rhs.p1;
    p2 = rhs.p2;
    p3 = rhs.p3;
    a = rhs.a;
    b = rhs.b;
    c = rhs.c;
    d = rhs.d;
    id = rhs.id;
    return *this;
}
```

```
int Facet3::sf_equal(const Facet3& f2)
{
   return (((p1 == f2.p1) && (p2 == f2.p2)) ||
           ((p1 == f2.p2) && (p2 == f2.p1)));
}


// '<<' I/O Stream Operator
ostream& operator<<(ostream& s, Facet3 f)
{
   s << "{" << f.p1 << "," << f.p2 << "," << f.p3 << "}";
   return s;
}


// Accessors

void Facet3::SetFacet(Point3 P1, Point3 P2, Point3 P3)
{
   SetPlane(P1, P2, P3);
}


void Facet3::SetID(int ID)
{
   id = ID;
}


int Facet3::GetID()
{
   return id;
}


// Services

void Facet3::Show()
{
   cout << "{";
   p1.Show();
   cout << ", ";
   p2.Show();
   cout << ", ";
   p3.Show();
   cout << "}";
}


void Facet3::Show_Full()
{
   cout << "{";
   p1.Show_Full();
   cout << ", ";
   p2.Show_Full();
   cout << ", ";
   p3.Show_Full();
   cout << "}" << endl;
}


#endif
/* faceti.cc - Facet Index Class */
```

```
#ifdef __cplusplus

#include "faceti.h"


// Default Constructor
Facet_Index::Facet_Index()
{
   p1_index = 0;
   p2_index = 0;
   p3_index = 0;
}


// Secondary Constructor
Facet_Index::Facet_Index(int P1Index, int P2Index, int P3Index)
{
   p1_index = P1Index;
   p2_index = P2Index;
   p3_index = P3Index;
}


// Copy Constructor
Facet_Index::Facet_Index(const Facet_Index& fi)
{
   p1_index = fi.p1_index;
   p2_index = fi.p2_index;
   p3_index = fi.p3_index;
}


// Default Destructor
Facet_Index::~Facet_Index()
{
   p1_index = 0;
   p2_index = 0;
   p3_index = 0;
}


// '=' Operator
void Facet_Index::operator=(const Facet_Index& rhs)
{
   p1_index = rhs.p1_index;
   p2_index = rhs.p2_index;
   p3_index = rhs.p3_index;
}


// '!=' Operator
int Facet_Index::operator!=(const Facet_Index& rhs)
{
   return (!((p1_index == rhs.p1_index) &&
             (p2_index == rhs.p2_index) &&
             (p3_index == rhs.p3_index)));
}


// '==' Operator
int Facet_Index::operator==(const Facet_Index& rhs)
{
```

```
    return ((p1_index == rhs.p1_index) &&
            (p2_index == rhs.p2_index) &&
            (p3_index == rhs.p3_index));
}


// '<<' I/O Stream Operator
ostream& operator<<(ostream& s, Facet_Index f)
{
   s << "{" << f.p1_index << "," << f.p2_index << "," << f.p3_index <<
"}";
   return s;
}


// Accessors


void Facet_Index::SetP1Index(int P1Index)
{
   p1_index = P1Index;
}


int Facet_Index::GetP1Index()
{
   return p1_index;
}


void Facet_Index::SetP2Index(int P2Index)
{
   p2_index = P2Index;
}

int Facet_Index::GetP2Index()
{
   return p2_index;
}


void Facet_Index::SetP3Index(int P3Index)
{
   p3_index = P3Index;
}


int Facet_Index::GetP3Index()
{
   return p3_index;
}


#endif
/* general.c */

#include <assert.h>
#include <math.h>

#include "general.h"


/* Test if a double value1 is "near" another double value2  */
```

```
int DEQ (double value1, double value2)
{
   int status;

   if (fabs(value1 - value2) >= (double)TOLER)
      status = 0;
   else
      status = 1;

   return status;
}


/* Test if a float value1 is "near" another float value2  */
int FEQ (float value1, float value2)
{
   int status;

   if (fabs((double)(value1 - value2)) >= (double)TOLER)
      status = 0;
   else
      status = 1;

   return status;
}


/* Test if a double value is "near" 0.0 */
int DEQ0 (double value)
{
   int status;

   if (fabs(value) >= (double)TOLER)
      status = 0;
   else
      status = 1;

   return status;
}


/* Test if a float value is "near" 0.0 */
int FEQ0 (float value)
{
   int status;

   if (fabs((double)value) >= (double)TOLER)
      status = 0;
   else
      status = 1;

   return status;
}


/* Sign of a value returns: -1 = negative, 1 = positive, 0 = "near"
zero */
int sgn (double value)
{
   int status;

   if (value <= -TOLER)
      status = -1;
```

```
   else if (value >= TOLER)
      status = 1;
   else
   {
      assert(DEQ0(value));
      status = 0;
   }

   return status;
}


/* Find the determant of a 2x2 matrix */
double det2(double a11, double a12,
            double a21, double a22)
{
  double det;
  det = a11*a22 - a21*a12;
  return det;
}


/* Find the determant of a 3x3 matrix */
double det3(double a11, double a12, double a13,
            double a21, double a22, double a23,
            double a31, double a32, double a33)
{
  double det;
  det = a11*a22*a33 + a12*a23*a31 + a13*a21*a32 -
        a13*a22*a31 - a11*a23*a32 - a12*a21*a33;
  return det;
}


void solve(double a11, double a12, double a13, double a14,
           double a21, double a22, double a23, double a24,
           double a31, double a32, double a33, double a34,
           double &a, double &b, double &c, double &d)
{
   double old;

   // Step 1 - Pivot 1 at A11
   old = a11;
   a11 = 1.0;
   a12 = a12/old;
   a13 = a13/old;
   a14 = a14/old;

   // Step 2 - 0 at A21 & A31
   old = -a21;
   a21 = 0.0;
   a22 = (old*a12)+a22;
   a23 = (old*a13)+a23;
   a24 = (old*a14)+a24;
   old = -a31;
   a31 = 0.0;
   a32 = (old*a12)+a32;
   a33 = (old*a13)+a33;
   a34 = (old*a14)+a34;

   // Step 3 - Pivot 1 at A22
   old = a22;
   a22 = 1.0;
```

```
    a23 = a23/old;
    a24 = a24/old;

    // Step 4 - 0 at A12 & A32
    old = -a12;
    a12 = 0.0;
    a13 = (old*a23)+a13;
    a14 = (old*a24)+a14;
    old = -a32;
    a32 = 0.0;
    a33 = (old*a23)+a33;
    a34 = (old*a24)+a34;

    // Step 5 - Pivot 1 at A33
    old = a33;
    a33 = 1.0;
    a34 = a34/old;

    // Step 6 - 0 at A13 & A23
    old = -a13;
    a13 = 0.0;
    a14 = (old*a34)+a14;
    old = -a23;
    a23 = 0.0;
    a24 = (old*a34)+a24;

    d = 1.0;  // ???
    a = a14*d;
    b = a24*d;
    c = a34*d;
}


/* Standard Deviation of n doubles */
double standard_deviation(double X[], int n)
{
    int i;
    double sum;
    double Xa;
    double sd;

    sum = 0.0;
    for (i=0; i<n; i++)
        sum = sum + X[i];
    Xa = sum / (double) n;
    sum = 0.0;
    for (i=0; i<n; i++)
        sum = sum + (X[i] - Xa)*(X[i] - Xa);
    sd = sqrt(sum/(double)(n-1));

    return sd;
}

/* lineseg2.cc - Line_Segment2 Class Implementation */

#ifdef __cplusplus

#include <iostream.h>
#include "lineseg2.h"


// Constructors & Destructors
```

```
Line_Segment2::Line_Segment2()    // Default Constructor
{
   p1.SetPoint(0.0, 0.0);
   p2.SetPoint(0.0, 0.0);
}


Line_Segment2::Line_Segment2(Point2 P1, Point2 P2)    // Secondary
Constructor
{
   p1 = P1;
   p2 = P2;
}


Line_Segment2::Line_Segment2(const Line_Segment2& ls)    // Copy
Constructor
{
   p1 = ls.p1;
   p2 = ls.p2;
}


Line_Segment2::~Line_Segment2()    // Default Destructor
{
   p1.SetPoint(0.0, 0.0);
   p2.SetPoint(0.0, 0.0);
}


// Operators

Line_Segment2& Line_Segment2::operator=(const Line_Segment2 &rhs)
{
   if (this == &rhs) return *this;
   p1 = rhs.p1;
   p2 = rhs.p2;
   return *this;
}


// Accessors

void Line_Segment2::SetP1(Point2 P1)
{
   p1 = P1;
}


Point2 Line_Segment2::GetP1()
{
   return p1;
}


void Line_Segment2::SetP2(Point2 P2)
{
   p2 = P2;
}


Point2 Line_Segment2::GetP2()
{
```

```
    return p2;
}


void Line_Segment2::SetSegment(Point2 P1, Point2 P2)
{
   p1 = P1;
   p2 = P2;
}


// Services

int Line_Segment2::Is_On_Line_Segment(Point2 q)
{
   // *** TBD don't forget about tolerances in this routine ***

   int status;

   // Form boundry box around end points to narrow down possibilities

   // Check X boundry
   if (((q.GetX() > p1.GetX()) && (q.GetX() > p2.GetX())) ||
       ((q.GetX() < p1.GetX()) && (q.GetX() < p1.GetX())))
   {
      status = 0;
   }
   else
   {
      // Check Y boundry
      if (((q.GetY() > p1.GetY()) && (q.GetY() > p2.GetY())) ||
          ((q.GetY() < p1.GetY()) && (q.GetY() < p2.GetY())))
      {
         status = 0;
      }
      else
      {
         // It is inside of the boundry box
         // *** TBD decide how to find out if it is on the line segment
         status = 1;
      }
   }

   return status;
}


void Line_Segment2::Show()
{
   cout << "{";
   p1.Show_Full();
   cout << ", ";
   p2.Show_Full();
   cout << "}";
}


#endif
/* lineseg3.cc - Line_Segment3 Class Implementation */

#ifdef __cplusplus

#include <iostream.h>
```

```
#include "lineseg3.h"


// Constructors & Destructors

Line_Segment3::Line_Segment3()   // Default Constructor
{
   p1.SetPoint(0.0, 0.0, 0.0);
   p2.SetPoint(0.0, 0.0, 0.0);
}


Line_Segment3::Line_Segment3(Point3 P1, Point3 P2)   // Secondary
Constructor
{
   p1 = P1;
   p2 = P2;
}


Line_Segment3::Line_Segment3(const Line_Segment3& ls)   // Copy
Constructor
{
   p1 = ls.p1;
   p2 = ls.p2;
}


Line_Segment3::~Line_Segment3()   // Default Destructor
{
   p1.SetPoint(0.0, 0.0, 0.0);
   p2.SetPoint(0.0, 0.0, 0.0);
}


// Operators
Line_Segment3& Line_Segment3::operator=(const Line_Segment3 &rhs)
{
   if (this == &rhs) return *this;
   p1 = rhs.p1;
   p2 = rhs.p2;
   return *this;
}


// Accessors

void Line_Segment3::SetP1(Point3 P1)
{
   p1 = P1;
}


Point3 Line_Segment3::GetP1()
{
   return p1;
}


void Line_Segment3::SetP2(Point3 P2)
{
   p2 = P2;
}
```

```
Point3 Line_Segment3::GetP2()
{
   return p2;
}


void Line_Segment3::SetSegment(Point3 P1, Point3 P2)
{
   p1 = P1;
   p2 = P2;
}


// Services

void Line_Segment3::Show()
{
   cout << "{";
   p1.Show_Full();
   cout << ", ";
   p2.Show_Full();
   cout << "}";
}


Point3 Line_Segment3::Intersection(Plane& P)
{
   Point3 tmp_point;
   double t;
   Poly_Class p1_side, p2_side;

   p1_side = P.Classify_Polygon2(p1);
   p2_side = P.Classify_Polygon2(p2);

   if ((p1_side == COINCIDENT) && ( p2_side == COINCIDENT))
   {
      // Line Segment lies inside of plane
      // return midpoint of linesegment
      tmp_point.SetX((p1.GetX()+p2.GetX())/2.0);
      tmp_point.SetY((p1.GetY()+p2.GetY())/2.0);
      tmp_point.SetZ((p1.GetZ()+p2.GetZ())/2.0);
      cout << "**********" << endl;
      cout << "WARNING LineSegment lies in Plane" << endl;
      (*this).Show();
      cout << endl;
      P.Show_Full();
      cout << "**********" << endl;
   }
   else if (p1_side == p2_side)
   {
      // Line Segment lies outside of plane
      // return (0,0,0)
      tmp_point.SetPoint(0.0, 0.0, 0.0);
      cout << "WARNING LineSegment does not intersect Plane" << endl;
   }
   else
   {
      t = - (P.GetA()*p2.GetX() +
             P.GetB()*p2.GetY() +
             P.GetC()*p2.GetZ() +
             P.GetD()) /
```

```
              (P.GetA()*(p1.GetX() - p2.GetX()) +
               P.GetB()*(p1.GetY() - p2.GetY()) +
               P.GetC()*(p1.GetZ() - p2.GetZ()));

      tmp_point.SetX(t*p1.GetX() + (1-t)*p2.GetX());
      tmp_point.SetY(t*p1.GetY() + (1-t)*p2.GetY());
      tmp_point.SetZ(t*p1.GetZ() + (1-t)*p2.GetZ());
#ifdef DEBUG
   if (!P.Is_Point_On_Plane(tmp_point))
      cout << " ERROR Calculated Point is not in Plane!" << endl;
#endif
   }

   return tmp_point;
}

int Line_Segment3::Is_On_Line_Segment(Point3 q)
{
   int status;
   Point3 V1, V2;    // 3 space vectors
   double m1, m2;    // vector magnitude
   double n;         // vector normal

   // Form vectors p1p2 and p1q
   V1.SetX(p2.GetX() - p1.GetX());
   V1.SetY(p2.GetY() - p1.GetX());
   V1.SetZ(p2.GetZ() - p1.GetZ());
   V2.SetX(q.GetX() - p1.GetX());
   V2.SetY(q.GetY() - p1.GetY());
   V2.SetZ(q.GetZ() - p1.GetZ());

   // Get Magnitudes of Vectors
   m1 = p1.Magnitude(p2);
   m2 = p1.Magnitude(q);

   // Normalize vectors
   n = V1.Norm();
   V1.SetX(V1.GetX()/n);
   V1.SetY(V1.GetY()/n);
   V1.SetZ(V1.GetZ()/n);
   n = V2.Norm();
   V2.SetX(V2.GetX()/n);
   V2.SetY(V2.GetY()/n);
   V2.SetZ(V2.GetZ()/n);

   // Test if q is on linesegment p1p2
   if ((V1.GetX()==V2.GetX())&&
       (V1.GetY()==V2.GetY())&&
       (V1.GetZ()==V2.GetZ()))
     if (m2<=m1)
       status=1;
     else
       status = 0;

/*
   int status;
   double t;
   double testx, testy, testz;

   if (!DEQ0(p2.GetX() - p1.GetX()))
   {
      // Linesegment is not in the X hyperplane
```

```
    // Find a parametric t value
    t = (q.GetX() - p1.GetX()) / (p2.GetX() - p1.GetX());

    if ((t > 1.0) || (t < 0.0))
    {
       // Point is too far left or right of the Linesegment
       status = 0;
    }
    else
    {
       testy = p1.GetY() + t*(p2.GetY() - p1.GetY());
       testz = p1.GetZ() + t*(p2.GetZ() - p1.GetZ());
       if (DEQ(q.GetY(), testy) && DEQ(q.GetZ(), testz))
       {
          // Point is on the Linesegment
          status = 1;
       }
       else
       {
          // Point is not on the Linesegment
          status = 0;
       }
    }
}
else if (!DEQ0(p2.GetY() - p1.GetY()))
{
    // Linesegment is not in the Y hyperplane

    // Find a parametric t value
    t = (q.GetY() - p1.GetY()) / (p2.GetY() - p1.GetY());

    if ((t > 1.0) || (t < 0.0))
    {
       // Point is too far left or right of the Linesegment
       status = 0;
    }
    else
    {
       testx = p1.GetX() + t*(p2.GetX() - p1.GetX());
       testz = p1.GetZ() + t*(p2.GetZ() - p1.GetZ());
       if (DEQ(q.GetX(), testx) && DEQ(q.GetZ(), testz))
       {
          // Point is on the Linesegment
          status = 1;
       }
       else
       {
          // Point is not on the Linesegment
          status = 0;
       }
    }
}
else if (!DEQ0(p2.GetZ() - p1.GetZ()))
{
    // Linesegment is not in the Z hyperplane

    // Find a parametric t value
    t = (q.GetZ() - p1.GetZ()) / (p2.GetZ() - p1.GetZ());

    if ((t > 1.0) || (t < 0.0))
    {
       // Point is too far left or right of the Linesegment
       status = 0;
```

```
      }
      else
      {
         testy = p1.GetY() + t*(p2.GetY() - p1.GetY());
         testx = p1.GetX() + t*(p2.GetX() - p1.GetX());
         if (DEQ(q.GetY(), testy) && DEQ(q.GetX(), testx))
         {
            // Point is on the Linesegment
            status = 1;
         }
         else
         {
            // Point is not on the Linesegment
            status = 0;
         }
      }
   }
   else
   {
      // Point is in all three hyperplanes!  Must not be a proper
Linesegment!
      cerr << "ERROR! Undefined Line Segment!" << endl;
      exit(1);
   }

*/
   return status;
}

#endif
/* list.cc */

#ifdef __cplusplus

#include <iostream.h>
#include <stdio.h>
//#define NDEBUG   // turn off assertions
#include <assert.h>
#include "list.h"


// Default Constructor
template<class T>
List<T>::List()
{
   head      = (lptr)0;
   tail      = (lptr)0;
   cur_ptr   = (lptr)0;
   num_items = 0;
}


// Copy Constructor
template<class T>
List<T>::List(const List& l)
{
   lptr cur;
   lptr lcur;

   head      = (lptr)0;
   tail      = (lptr)0;
   cur_ptr   = (lptr)0;
   num_items = 0;
```

```
   // Copy Contents of List l to our List
   lcur = l.head;
   while (lcur)
   {
      cur = new list_item;
      assert(cur);                          // Error Out of Memory!
      cur->data = lcur->data;               // '=' Operator for class T is
needed
      if (lcur == l.head)
      {
         cur->prev = (lptr)0;
         head = cur;                        // Set head to first item in
list
         cur_ptr = head;
      }
      else
      {
         cur->prev = cur_ptr;               // Place new item in list
         cur_ptr->next = cur;
         cur_ptr = cur_ptr->next;
      }
      if (lcur == l.tail)
      {
         cur->next = (lptr)0;
         tail = cur;                        // Set tail to last item in
list
      }

      lcur = lcur->next;
   }
   cur_ptr = head;
   num_items = l.num_items;
}


// Default Destructor
template<class T>
List<T>::~List()
{
   lptr cur;
   lptr next;

   if (head)
   {
      cur = head;
      while (cur)
      {
         next = cur->next;
         delete cur;
         cur = next;
      }
      head    = (lptr)0;
      tail    = (lptr)0;
      cur_ptr = (lptr)0;
      num_items = 0;
   }
}


// Overloaded '=' Operator
template<class T>
List<T>& List<T>::operator=(const List &l)
```

```
{
   lptr cur;
   lptr lcur;
   lptr next;

   if (this == &l) return *this;

   // Clear out Current List
   if (head)
   {
      cur = head;
      while (cur)
      {
         next = cur->next;
         delete cur;
         cur = next;
      }
      head    = (lptr)0;
      tail    = (lptr)0;
      cur_ptr = (lptr)0;
      num_items = 0;
   }

   // Copy Contents of List l to our list
   lcur = l.head;
   while (lcur)
   {
      cur = new list_item;
      assert(cur);                    // Error Out of Memory!
      cur->data = lcur->data;         // '=' Operator for class T is
needed
      if (lcur == l.head)
      {
         cur->prev = (lptr)0;
         head = cur;                  // Set head to first item in list
         cur_ptr = head;
      }
      else
      {
         cur->prev = cur_ptr;         // Place new item in list
         cur_ptr->next = cur;
         cur_ptr = cur_ptr->next;
      }
      if (lcur == l.tail)
      {
         cur->next = (lptr)0;
         tail = cur;                  // Set tail to last item in list
      }
      lcur = lcur->next;
   }
   cur_ptr = head;
   num_items = l.num_items;

   return *this;
}


// Insert Head
template<class T>
void List<T>::Insert_Head(const T item)
{
   lptr cur;
```

```
   cur = new list_item;
   assert(cur);                          // Error Out of Memory!
   cur->data = item;                     // '=' Operator for class T is
needed
   cur->prev = (lptr)0;
   cur->next = (lptr)0;

   if (!head)
   {
      head    = cur;
      tail    = cur;
      cur_ptr = head;
   }
   else
   {
      cur->next = head;
      head->prev = cur;
      head = cur;
   }
   ++num_items;
}


// Insert Tail
template<class T>
void List<T>::Insert_Tail(const T item)
{
   lptr cur;

   cur = new list_item;
   assert(cur);                          // Error Out of Memory!
   cur->data = item;                     // '=' Operator for class T is
needed
   cur->prev = (lptr)0;
   cur->next = (lptr)0;

   if (!head)
   {
      head    = cur;
      tail    = cur;
      cur_ptr = head;
   }
   else
   {
      cur->prev = tail;
      tail->next = cur;
      tail = cur;
   }
   ++num_items;
}


// Insert at pointer
template<class T>
void List<T>::Insert_At_Pointer(const T item)
{
   lptr cur;

   cur = new list_item;
   assert(cur);                          // Error Out of Memory!
   cur->data = item;                     // '=' Operator for class T is
needed
   cur->prev = (lptr)0;
```

```
   cur->next = (lptr)0;

   if (!head)
   {
      head    = cur;
      tail    = cur;
   }
   else if (cur_ptr == head)            // cur_ptr is at head
   {
      cur->next  = head;
      head->prev = cur;
      head       = cur;
   }
   else
   {
      cur->prev     = cur_ptr->prev;
      cur->next     = cur_ptr;
      cur_ptr->prev = cur;
   }
   cur_ptr = cur;
   ++num_items;
}


// Remove Head
template<class T>
T List<T>::Remove_Head()
{
   lptr cur;
   T item;

   assert(head);                        // Error can't Remove from
Empty List!
   cur = head;
   item = cur->data;                    // '=' Operator for Class T is
needed
   if (!cur->next)                      // head is last item in list
   {
      assert((head == cur) && (tail == cur)); // Error Messed up
List!?!?!
      head    = (lptr)0;
      tail    = (lptr)0;
      cur_ptr = (lptr)0;
   }
   else
   {
      if (cur_ptr == head)
         cur_ptr = cur->next;
      head       = cur->next;
      head->prev = (lptr)0;
      cur->next  = (lptr)0;
   }
   --num_items;
   delete cur;
   return item;
}


// Remove Tail
template<class T>
T List<T>::Remove_Tail()
{
   lptr cur;
```

```
   T item;

   assert(tail);                          // Error can't Remove from
Empty List!
   cur = tail;
   item = cur->data;                      // '=' Operator for Class T is
needed
   if (!cur->prev)                        // tail is last item in list
   {
      assert((head == cur) && (tail == cur)); // Error Messed up
List!?!?!
      head    = (lptr)0;
      tail    = (lptr)0;
      cur_ptr = (lptr)0;
   }
   else
   {
      if (cur_ptr == tail)
         cur_ptr = cur->prev;
      tail       = cur->prev;
      tail->next = (lptr)0;
      cur->prev  = (lptr)0;
   }
   --num_items;
   delete cur;
   return item;
}


// Remove At Pointer
template<class T>
T List<T>::Remove_At_Pointer()
{
   lptr cur;
   T item;

   assert(head && tail && cur_ptr);       // Error can't Remove from
Empty List!
   cur = cur_ptr;
   item = cur->data;                      // '=' Operator for Class T is
needed
   if ((!cur->prev) && (!cur->next))      // cur_ptr is last item in list
   {
      assert((head == cur) && (tail == cur)); // Error Messed up
List!?!?!
      head    = (lptr)0;
      tail    = (lptr)0;
      cur_ptr = (lptr)0;
   }
   else if (!cur->prev)                   // cur_ptr = head
   {
      assert(head == cur);                // Error Messed up List!?!?!
      head       = cur->next;
      head->prev = (lptr)0;
      cur->next  = (lptr)0;
      cur_ptr    = head;
   }
   else if (!cur->next)                   // cur_ptr = tail
   {
      assert(tail == cur);                // Error Messed up List!?!?!
      tail       = cur->prev;
      tail->next = (lptr)0;
      cur->prev  = (lptr)0;
```

```
      cur_ptr    = tail;
   }
   else
   {
      cur->prev->next = cur->next;
      cur->next->prev = cur->prev;
      cur_ptr         = cur->prev;
      cur->prev       = (lptr)0;
      cur->next       = (lptr)0;
   }
   --num_items;
   delete cur;
   return item;
}


// Peek Head
template<class T>
T List<T>::Peek_Head() const
{
   assert(head);                          // Error can't Peek at Empty
List!
   return head->data;
}


// Peek Tail
template<class T>
T List<T>::Peek_Tail() const
{
   assert(tail);                          // Error can't Peek at Empty
List!
   return tail->data;
}


// Peek at Pointer
template<class T>
T List<T>::Peek_At_Pointer() const
{
   assert(cur_ptr);                       // Error can't Peek at Empty
List!
   return cur_ptr->data;
}


// Is the List Empty?
template<class T>
int List<T>::Is_Empty()
{
   int status;

   if (!head)
      status = 1;
   else
      status = 0;
   return status;
}


// Nuke the List
template<class T>
void List<T>::Clear()
```

```
{
   lptr cur;
   lptr next;

   if (head)
   {
      cur = head;
      while (cur)
      {
         next = cur->next;
         delete cur;
         cur = next;
      }
      head    = (lptr)0;
      tail    = (lptr)0;
      cur_ptr = (lptr)0;
      num_items = 0;
   }
}


// Reset pointer to head of List
template<class T>
void List<T>::Reset_Pointer()
{
   if (head)
      cur_ptr = head;
}


// Increment Pointer, Returns 1=success 0=failure
template<class T>
int List<T>::Increment_Pointer()
{
   int status;

   if (cur_ptr)
   {
      if (cur_ptr->next)
      {
         cur_ptr = cur_ptr->next;
         status = 1;
      }
      else
         status = 0;
   }
   else
      status = 0;

   return status;
}


// Decrement Pointer, Returns 1=success 0=failure
template<class T>
int List<T>::Decrement_Pointer()
{
   int status;

   if (cur_ptr)
   {
      if (cur_ptr->prev)
      {
```

```
            cur_ptr = cur_ptr->prev;
            status = 1;
         }
         else
            status = 0;
      }
      else
         status = 0;

      return status;
}


// Show what is in the List
template<class T>
void List<T>::Show()
{
   lptr cur;

   cout << "L -> ";
   if (head)
   {
      cur = head;
      while (cur)
      {
         if (cur_ptr == cur)
            cout << "*";
         cout << cur->data << " ";       // ios::<< for Class T is
needed
         cur = cur->next;
      }
      cout << endl;
   }
   else
      cout << "EMPTY" << endl;
}


// Is this item a member of the List?
template<class T>
int List<T>::Is_Member (const T item)
{
   lptr cur;
   int found;

   found = 0;
   if (head)
   {
      cur = head;
      while (cur && (!found))
      {
         if (cur->data == item)          // '==' Operator for Class T is
needed
            found = 1;
         cur = cur->next;
      }
   }
   return found;
}


// Find item in the List, status=1 is found, status=0 is not found
// Pointer is set to found item
```

```
template<class T>
int List<T>::Find (const T item)
{
   lptr cur;
   int found;

   found = 0;
   if (head)
   {
      cur = head;
      while (cur && (!found))
      {
         if (cur->data == item)          // '==' Operator for Class T is
needed
         {
            found = 1;
            cur_ptr = cur;               // Set cur_ptr to found item
         }
         cur = cur->next;
      }
   }
   return found;
}


// Find item in the List, status=1 is found, status=0 is not found
// Pointer is set to found item
template<class T>
int List<T>::Find2 (const Facet3 item)
{
   return 0;
}

int List<Facet3>::Find2 (const Facet3 item)
{
   lptr cur;
   int found;

   found = 0;
   if (head)
   {
      cur = head;
      while (cur && (!found))
      {
         if (cur->data.sf_equal(item))
         {
            found = 1;
            cur_ptr = cur;
         }
         cur = cur->next;
      }
   }
   return found;
}


template<class T>
int List<T>::Num_Members()
{
   return num_items;
}

#endif
```

```
/* oogl.cc */

#ifdef __cplusplus

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include "oogl.h"
#include "facet3.h"


void Convex_Hull_2_OOGL (List<Point3>& p, Convex_Hull3& f)
{
   fstream fp;
   List<Point3> tmp_p(p);
   Convex_Hull3 tmp_f(f);
   Point3 tmp_point;
   Point3 tmp_point1, tmp_point2, tmp_point3;
   Facet3 tmp_facet;

   fp.open("chull.off", ios::out);

   if(fp.fail() | fp.bad())
   {
      cerr << "Error opening file chull.off for output." << endl;
      exit(1);
   }

   fp << "OFF" << endl;
   fp << p.Num_Members() << " " << f.Num_Members() << " 0" << endl;

   tmp_p.Reset_Pointer();
   while (!tmp_p.Is_Empty())
   {
      tmp_point = tmp_p.Remove_Head();
      fp << tmp_point.GetX() << " " << tmp_point.GetY() << " ";
      fp << tmp_point.GetZ() << endl;
   }

   tmp_f.Reset_Pointer();
   while (!tmp_f.Is_Empty())
   {
      tmp_facet = tmp_f.Remove_Head();
      tmp_point1 = tmp_facet.GetP1();
      tmp_point2 = tmp_facet.GetP2();
      tmp_point3 = tmp_facet.GetP3();
      fp << "3 " << tmp_point1.GetID() - 1 << " ";
      fp << tmp_point2.GetID() - 1 << " ";
      fp << tmp_point3.GetID() - 1 << endl;
   }

   fp.close();
}


void Triangles_2_OOGL (List<Plane>& lp)
{
   List<Plane> tmp_lp(lp);
   fstream fp;
   Plane tmp_plane;
   Point3 tmp_point1, tmp_point2, tmp_point3;

   fp.open("triangs.off", ios::out);
```

```
   if(fp.fail() | fp.bad())
   {
      cerr << "Error opening file triangs.off for output." << endl;
      exit(1);
   }

   fp << "LIST" << endl;

   tmp_lp.Reset_Pointer();
   while (!tmp_lp.Is_Empty())
   {
      tmp_plane = tmp_lp.Remove_Head();
      tmp_point1 = tmp_plane.GetP1();
      tmp_point2 = tmp_plane.GetP2();
      tmp_point3 = tmp_plane.GetP3();
      fp << "{ = OFF" << endl;
      fp << "   3 1 0" << endl;
      fp << "   " << tmp_point1.GetX() << " " << tmp_point1.GetY() << "
";
      fp << tmp_point1.GetZ() << endl;
      fp << "   " << tmp_point2.GetX() << " " << tmp_point2.GetY() << "
";
      fp << tmp_point2.GetZ() << endl;
      fp << "   " << tmp_point3.GetX() << " " << tmp_point3.GetY() << "
";
      fp << tmp_point3.GetZ() << endl;
      fp << "   3 0 1 2" << endl;
      fp << "}" << endl;
   }

   fp.close();
}


void BSP_Tree_w_Convex_Hulls_2_OOGL (List<Point3>& p, BSP_Tree& t)
{
   fstream fp;

   fp.open("bspchull.off", ios::out);

   if(fp.fail() | fp.bad())
   {
      cerr << "Error opening file bspchull.off for output." << endl;
      exit(1);
   }

   fp.close();

   t.OOGL_Output(p);
}


#endif
/* plane.cc */

#ifdef __cplusplus

#include <math.h>
#include <stdlib.h>
#include "general.h"
#include "plane.h"
```

```
Plane::Plane()
{
   p1.SetPoint(0.0, 0.0, 0.0);
   p1.SetID(-1);
   p2.SetPoint(0.0, 0.0, 0.0);
   p2.SetID(-1);
   p3.SetPoint(0.0, 0.0, 0.0);
   p3.SetID(-1);
   a = 0.0;
   b = 0.0;
   c = 0.0;
   d = 0.0;
}


Plane::Plane(Point3 P1, Point3 P2, Point3 P3)
{
   p1 = P1;
   p2 = P2;
   p3 = P3;
   Calculate_Equation();
}


Plane::Plane(const Plane &P2)
{
   p1 = P2.p1;
   p2 = P2.p2;
   p3 = P2.p3;
   Calculate_Equation();
}


Plane::~Plane()
{
   p1.SetPoint(0.0, 0.0, 0.0);
   p1.SetID(-1);
   p2.SetPoint(0.0, 0.0, 0.0);
   p2.SetID(-1);
   p3.SetPoint(0.0, 0.0, 0.0);
   p3.SetID(-1);
   a = 0.0;
   b = 0.0;
   c = 0.0;
   d = 0.0;
}


Plane& Plane::operator=(const Plane &rhs)
{
   if (this == &rhs) return *this;
   p1 = rhs.p1;
   p2 = rhs.p2;
   p3 = rhs.p3;
   a = rhs.a;
   b = rhs.b;
   c = rhs.c;
   d = rhs.d;
   return *this;
}
```

```
int Plane::operator==(const Plane& rhs)
{
   return (((p1 == rhs.p1) && (p2 == rhs.p2) && (p3 == rhs.p3)) ||
           ((p1 == rhs.p1) && (p2 == rhs.p3) && (p3 == rhs.p2)) ||
           ((p1 == rhs.p2) && (p2 == rhs.p1) && (p3 == rhs.p3)) ||
           ((p1 == rhs.p2) && (p2 == rhs.p3) && (p3 == rhs.p1)) ||
           ((p1 == rhs.p3) && (p2 == rhs.p1) && (p3 == rhs.p2)) ||
           ((p1 == rhs.p3) && (p2 == rhs.p2) && (p3 == rhs.p1)));
}


int Plane::operator!=(const Plane& rhs)
{
   return (!((*this) == rhs));
}


ostream& operator<<(ostream& s, Plane p)
{
   s << "{" << p.p1 << "," << p.p2 << "," << p.p3 << "}";
   return s;
}


void Plane::SetP1(Point3 P1)
{
   p1 = P1;
   Calculate_Equation();
}


Point3 Plane::GetP1()
{
   return p1;
}


void Plane::SetP2(Point3 P2)
{
   p2 = P2;
   Calculate_Equation();
}


Point3 Plane::GetP2()
{
   return p2;
}


void Plane::SetP3(Point3 P3)
{
   p3 = P3;
   Calculate_Equation();
}


Point3 Plane::GetP3()
{
   return p3;
}
```

```
void Plane::SetPlane(Point3 P1, Point3 P2, Point3 P3)
{
   p1 = P1;
   p2 = P2;
   p3 = P3;
   Calculate_Equation();
}


double Plane::GetA()
{
   return a;
}


double Plane::GetB()
{
   return b;
}


double Plane::GetC()
{
   return c;
}


double Plane::GetD()
{
   return d;
}


// Are two planes parallel
int Plane::Is_Parallel(Plane &P2)
{
   return ((a==P2.a)&&(b==P2.b)&&(c==P2.c));
}


// Are two planes coincident
int Plane::Is_Coincident(Plane &P2)
{
   return ((a==P2.a)&&(b=P2.b)&&(c==P2.c)&&(d==P2.d));
}


// Distance between Point and Plane?
// pg 832-833, Calculus - One and Several Variables 7th ed, Salas &
Hille
double Plane::Distance(Point3 P)
{
   double distance;
   double numer;
   double denom;

   numer = fabs(a*P.GetX() + b*P.GetY() + c*P.GetZ() + d);
   denom = sqrt(a*a + b*b + c*c);
   distance = numer/denom;

   return distance;
}
```

```
// Does the Point lie on the Plane?
// pg 824, Calculus - One and Several Variables 7th ed, Salas & Hille
int Plane::Is_Point_On_Plane(Point3 P)
{
   int status;

   if (P != p1)
   {
      if (DEQ0(a*(P.GetX()-p1.GetX()) +
               b*(P.GetY()-p1.GetY()) +
               c*(P.GetZ()-p1.GetZ())))
         status = 1;
      else
         status = 0;
   }
   else
   {
      if (DEQ0(a*(P.GetX()-p2.GetX()) +
               b*(P.GetY()-p2.GetY()) +
               c*(P.GetZ()-p2.GetZ())))
         status = 1;
      else
         status = 0;
   }
   return status;
}


Poly_Class Plane::Classify_Polygon(Plane& P2)
{
   Poly_Class return_value;
   Poly_Class s1, s2, s3;

   s1 = (*this).Classify_Polygon2(P2.GetP1());
   s2 = (*this).Classify_Polygon2(P2.GetP2());
   s3 = (*this).Classify_Polygon2(P2.GetP3());

   if ((s1 == COINCIDENT) && (s2 == COINCIDENT) && (s3 == COINCIDENT))
      return_value = COINCIDENT;
   else if ((s1 == IN_FRONT_OF) && (s2 == IN_FRONT_OF) && (s3 ==
IN_FRONT_OF))
      return_value = IN_FRONT_OF;
   else if ((s1 == IN_BACK_OF) && (s2 == IN_BACK_OF) && (s3 ==
IN_BACK_OF))
      return_value = IN_BACK_OF;
   else if ((s1 == COINCIDENT) && (s2 == IN_FRONT_OF) && (s3 ==
IN_FRONT_OF))
      return_value = IN_FRONT_OF;
   else if ((s1 == COINCIDENT) && (s2 == IN_BACK_OF) && (s3 ==
IN_BACK_OF))
      return_value = IN_BACK_OF;
   else if ((s2 == COINCIDENT) && (s1 == IN_FRONT_OF) && (s3 ==
IN_FRONT_OF))
      return_value = IN_FRONT_OF;
   else if ((s2 == COINCIDENT) && (s1 == IN_BACK_OF) && (s3 ==
IN_BACK_OF))
      return_value = IN_BACK_OF;
   else if ((s3 == COINCIDENT) && (s2 == IN_FRONT_OF) && (s1 ==
IN_FRONT_OF))
      return_value = IN_FRONT_OF;
   else if ((s3 == COINCIDENT) && (s2 == IN_BACK_OF) && (s1 ==
IN_BACK_OF))
```

```
      return_value = IN_BACK_OF;
   else if ((s1 == COINCIDENT) && (s2 == COINCIDENT) && (s3 ==
IN_FRONT_OF))
      return_value = IN_FRONT_OF;
   else if ((s1 == COINCIDENT) && (s2 == COINCIDENT) && (s3 ==
IN_BACK_OF))
      return_value = IN_BACK_OF;
   else if ((s2 == COINCIDENT) && (s3 == COINCIDENT) && (s1 ==
IN_FRONT_OF))
      return_value = IN_FRONT_OF;
   else if ((s2 == COINCIDENT) && (s3 == COINCIDENT) && (s1 ==
IN_BACK_OF))
      return_value = IN_BACK_OF;
   else if ((s1 == COINCIDENT) && (s3 == COINCIDENT) && (s2 ==
IN_FRONT_OF))
      return_value = IN_FRONT_OF;
   else if ((s1 == COINCIDENT) && (s3 == COINCIDENT) && (s2 ==
IN_BACK_OF))
      return_value = IN_BACK_OF;
   else
      return_value = SPANNING;

   return return_value;
}


Poly_Class Plane::Classify_Polygon2(Point3 P)
{
   Poly_Class return_value;
   double distance;
   double s1;

   distance = (*this).Distance(P);
   if (DEQ0(distance))
      return_value = COINCIDENT;
   else
   {
      s1 = a*P.GetX() + b*P.GetY() + c*P.GetZ() + d;
      if (s1 > 0.0)
         return_value = IN_FRONT_OF;
      else if (s1 < 0.0)
         return_value = IN_BACK_OF;
      else
         cout << "ERROR in Classify Polygon 2";
   }

   return return_value;
}


Poly_Class Plane::Classify_Polygon3(Plane &P2)
{
   Poly_Class return_value;

   // Since planes are infinite, if the two planes are not
   // parallel then they must be spanning.  If the two planes
   // are parallel, then they might be coincident.
   if ((*this).Is_Parallel(P2))
      if ((*this).Is_Coincident(P2))
         return_value = COINCIDENT;
      else
      {
         // Check IN_FRONT_OF or IN_BACK_OF
```

```
            return_value = (*this).Classify_Polygon2(P2.p1);
        }
    else
        return_value = SPANNING;

    return return_value;
}


// Angle between 2 planes
double Plane::Angle(Plane &P2)
{
    double cosangle;
    Point3 q1, q2, q3, q4, v1, v2, v3, c1, c2;

    // find common edge and assign to q1 and q2.
    // q3 is remaining point from calling plane
    // q4 is remaining point from plane P2

    if ((p1 != P2.GetP1()) &&
        (p1 != P2.GetP2()) &&
        (p1 != P2.GetP3()))
    {
        // p2,p3 common
        q1 = p2;
        q2 = p3;
        q3 = p1;
    }
    else if ((p2 != P2.GetP1()) &&
             (p2 != P2.GetP2()) &&
             (p2 != P2.GetP3()))
    {
        // p1,p3 common
        q1 = p1;
        q2 = p3;
        q3 = p2;
    }
    else
    {
        // p1,p2 common
        q1 = p1;
        q2 = p2;
        q3 = p3;
    }

    // find remainin point in P2

    if ((P2.GetP1() != q1) &&
        (P2.GetP1() != q2))
    {
        // p1 not common
        q4 = P2.GetP1();
    }
    else if ((P2.GetP2() != q1) &&
             (P2.GetP2() != q2))
    {
        // p2 not common
        q4 = P2.GetP2();
    }
    else
    {
        // p3 not common
        q4 = P2.GetP3();
```

```
    }

    // find angle between planes
    v2 = q2 - q1;
    v1 = q3 - q1;
    v3 = q4 - q1;
    c1 = v1.Cross(v2);
    c2 = v3.Cross(v2);
    cosangle = c1.Dot_Product(c2)/(c1.Norm()*c2.Norm());

    return cosangle;

//    double norm1, norm2;
//    Point3 un1, un2;
//    double theta;
//
//    norm1 = 1.0/sqrt(a*a + b*b + c*c);
//    norm2 = 1.0/sqrt(P2.a*P2.a + P2.b*P2.b + P2.c*P2.c);
//    un1 = (*this).Normal();
//    un2 = P2.Normal();
//    un1 = un1 * norm1;
//    un2 = un2 * norm2;
//    theta = acos(fabs(un1.Dot_Product(un2)));
//    removed fabs() because we want both acute and obtuse angles
//    theta = acos(un1.Dot_Product(un2));
//
//    return theta;
}


Point3 Plane::Normal()
{
    Point3 n;

    n.SetPoint(a,b,c);

    return n;
}


void Plane::Show()
{
    cout << "{";
    p1.Show_Full();
    cout << ", ";
    p2.Show_Full();
    cout << ", ";
    p3.Show_Full();
    cout << "}" << endl;
}


void Plane::Show_Full()
{
    cout << "{";
    p1.Show_Full();
    cout << ", ";
    p2.Show_Full();
    cout << ", ";
    p3.Show_Full();
    cout << "}" << endl;
    cout << "Equation (Ax+By+Cz+D=0): (";
    cout << a << ", " << b << ", " << c << ", " << d << ")" << endl;
```

```
}


void Plane::Calculate_Equation()
{
   double p12i, p12j, p12k;              // Vector P1P2
   double p13i, p13j, p13k;              // Vector P1P3
   double ni, nj, nk;                    // Vector N

   // Vector P1P2 = <P2x-P1x, P2y-P1y, P2z-P1z>
   p12i = p2.GetX() - p1.GetX();
   p12j = p2.GetY() - p1.GetY();
   p12k = p2.GetZ() - p1.GetZ();
   // Vector P1P3 = <P3x-P1x, P3y-P1y, P3z-P1z>
   p13i = p3.GetX() - p1.GetX();
   p13j = p3.GetY() - p1.GetY();
   p13k = p3.GetZ() - p1.GetZ();
   // Vector N = (P1P2 X P1P3)
   ni = p12j*p13k - p12k*p13j;
   nj = -(p12i*p13k - p12k*p13i);
   nk = p12i*p13j - p12j*p13i;
   // P1P . (P1P2 X P1P3) = 0
   a = ni;
   b = nj;
   c = nk;
   d = (p1.GetX()*ni) + (p1.GetY()*nj) + (p1.GetZ()*nk);
   // if a is negative, factor out the negative
   if (a < 0.0)
   {
      a = -a;
      b = -b;
      c = -c;
      d = -d;
   }

   // There are other ways of finding the equation of a plane
   // using a co-factor-minors method or using a marticies method.
   // Most of the methods leave the D coeficient = 1.0.
   // These methods are not accurate enough for this program,
   // so I had to resort to using Newell's Method.
   // Newell's Method is described in the book Graphics Gems III,
   // David Kirk, Academic Press, 1991, pp 231-232 & 517-518.

   //Newells_Method();
}


void Plane::Newells_Method()
{
   Point3 normal;
   Point3 refpt;
   Point3 u;
   Point3 v;
   double len;

   // compute the polygon normal and a reference point on the plane
   // unrolled for loop because this program is class based, not array
based
   u = p1;
   v = p2;
   normal.SetX((u.GetY() - v.GetY()) * (u.GetZ() + v.GetZ()));
   normal.SetY((u.GetZ() - v.GetZ()) * (u.GetX() + v.GetX()));
   normal.SetZ((u.GetX() - v.GetX()) * (u.GetY() + v.GetY()));
```

```
   refpt.SetX(u.GetX());
   refpt.SetY(u.GetY());
   refpt.SetZ(u.GetZ());
   u = p2;
   v = p3;
   normal.SetX(normal.GetX() + (u.GetY() - v.GetY()) * (u.GetZ() +
v.GetZ()));
   normal.SetY(normal.GetY() + (u.GetZ() - v.GetZ()) * (u.GetX() +
v.GetX()));
   normal.SetZ(normal.GetZ() + (u.GetX() - v.GetX()) * (u.GetY() +
v.GetY()));
   refpt.SetX(refpt.GetX() + u.GetX());
   refpt.SetY(refpt.GetY() + u.GetY());
   refpt.SetZ(refpt.GetZ() + u.GetZ());
   u = p3;
   v = p1;
   normal.SetX(normal.GetX() + (u.GetY() - v.GetY()) * (u.GetZ() +
v.GetZ()));
   normal.SetY(normal.GetY() + (u.GetZ() - v.GetZ()) * (u.GetX() +
v.GetX()));
   normal.SetZ(normal.GetZ() + (u.GetX() - v.GetX()) * (u.GetY() +
v.GetY()));
   refpt.SetX(refpt.GetX() + u.GetX());
   refpt.SetY(refpt.GetY() + u.GetY());
   refpt.SetZ(refpt.GetZ() + u.GetZ());

   // normalize the polygon normal to obtain the first 3 plane
coefficients
   len = normal.Norm();
   if (!DEQ0(len))
   {
      a = normal.GetX() / len;
      b = normal.GetY() / len;
      c = normal.GetZ() / len;
   }
   else
   {
      a = 0.0;
      b = 0.0;
      c = 0.0;
   }

   // compute the last coefficient of the plane equation
   len = len * 3.0;
   if (!DEQ0(len))
      d = -refpt.Dot_Product(normal) / len;
   else
      d = 0.0;
}

#endif
/* point2.cc - Point2 Class Implementation */

#ifdef __cplusplus

#include <math.h>
#include "point2.h"


// Constructors & Destructors

Point2::Point2()   // Default Constructor
{
```

```
    x  = 0.0;
    y  = 0.0;
    id = -1;
}


Point2::Point2(double X, double Y)    // Secondary Constructor
{
    x  = X;
    y  = Y;
    id = -1;
}


Point2::Point2(const Point2& p)    // Copy Constructor
{
    x  = p.x;
    y  = p.y;
    id = p.id;
}


Point2::~Point2()    // Default Destructor
{
    x  = 0.0;
    y  = 0.0;
    id = -1;
}


// Operators

Point2& Point2::operator=(const Point2& rhs)
{
    if (this==&rhs) return *this;
    x  = rhs.x;
    y  = rhs.y;
    id = rhs.id;
    return *this;
}


int Point2::operator==(const Point2 &rhs)
{
    return ((x == rhs.x) &&
            (y == rhs.y));
}


int Point2::operator!=(const Point2 &rhs)
{
    return (!((x == rhs.x) &&
              (y == rhs.y)));
}


ostream& operator<<(ostream& s, Point2 p)
{
    s << p.id;
    return s;
}
```

```
// Accessors

void Point2::SetX(double X)
{
   x = X;
}


double Point2::GetX()
{
   return x;
}


void Point2::SetY(double Y)
{
   y = Y;
}


double Point2::GetY()
{
   return y;
}


void Point2::SetID(int ID)
{
   id = ID;
}


int Point2::GetID()
{
   return id;
}


void Point2::SetPoint(double X, double Y)
{
   x = X;
   y = Y;
   id = -1;
}


// Services

// Distance Between Two Points
double Point2::Distance(const Point2& p2)
{
   return sqrt((p2.x-x)*(p2.x-x) +
               (p2.y-y)*(p2.y-y));
}


void Point2::Show_Full()
{
   cout << "(" << x << "," << y << ")";
}


void Point2::Show()
```

```
{
   cout << id;
}

#endif
/* point3.cc - Point3 Class Implementation */

#ifdef __cplusplus

#include <math.h>

#include "point3.h"
#include "general.h"


// Constructors & Destructors

Point3::Point3()   // Default Constructor
{
   x  = 0.0;
   y  = 0.0;
   z  = 0.0;
   id = -1;
}


Point3::Point3(double X, double Y, double Z)   // Secondary Constructor
{
   x  = X;
   y  = Y;
   z  = Z;
   id = -1;
}


Point3::Point3(const Point3& p)   // Copy Constructor
{
   x  = p.x;
   y  = p.y;
   z  = p.z;
   id = p.id;
}


Point3::~Point3()   // Default Destructor
{
   x  = 0.0;
   y  = 0.0;
   z  = 0.0;
   id = -1;
}


// Operators

Point3& Point3::operator=(const Point3& rhs)
{
   if (this == &rhs) return *this;
   x  = rhs.x;
   y  = rhs.y;
   z  = rhs.z;
   id = rhs.id;
   return *this;
```

```
}


int Point3::operator==(const Point3 &rhs)
{
   return ((x == rhs.x) &&
           (y == rhs.y) &&
           (z == rhs.z));
}


int Point3::operator!=(const Point3 &rhs)
{
   return (!((x == rhs.x) &&
             (y == rhs.y) &&
             (z == rhs.z)));
}


Point3 operator-(const Point3& lhs, const Point3& rhs)
{
   Point3 temp_point;

   temp_point.x  = lhs.x - rhs.x;
   temp_point.y  = lhs.y - rhs.y;
   temp_point.z  = lhs.z - rhs.z;
   temp_point.id = -1;

   return temp_point;
}


Point3 operator+(const Point3& lhs, const Point3& rhs)
{
   Point3 temp_point;

   temp_point.x  = lhs.x + rhs.x;
   temp_point.y  = lhs.y + rhs.y;
   temp_point.z  = lhs.z + rhs.z;
   temp_point.id = -1;

   return temp_point;
}


Point3 operator*(const Point3& lhs, const double rhs)
{
   Point3 temp_point;

   temp_point.x  = lhs.x * rhs;
   temp_point.y  = lhs.y * rhs;
   temp_point.z  = lhs.z * rhs;
   temp_point.id = -1;

   return temp_point;
}


ostream& operator<<(ostream& s, Point3 p)
{
   s << p.id;
   return s;
}
```

```
// Accessors

void Point3::SetX(double X)
{
   x = X;
}


double Point3::GetX()
{
   return x;
}


void Point3::SetY(double Y)
{
   y = Y;
}


double Point3::GetY()
{
   return y;
}


void Point3::SetZ(double Z)
{
   z = Z;
}


double Point3::GetZ()
{
   return z;
}


void Point3::SetID(int ID)
{
   id = ID;
}


int Point3::GetID()
{
   return id;
}


void Point3::SetPoint(double X, double Y, double Z)
{
   x = X;
   y = Y;
   z = Z;
   id = -1;
}


// Services
```

```
// Distance Between Two Points
double Point3::Distance(const Point3& p2)
{
   return sqrt((p2.x-x)*(p2.x-x) + (p2.y-y)*(p2.y-y) + (p2.z-z)*(p2.z-
z));
}


// Dot Product Between Two Points
double Point3::Dot_Product(const Point3& p2)
{
   return x*p2.x + y*p2.y + z*p2.z;
}


// Normalized Point ||p||
double Point3::Norm()
{
   return sqrt(x*x + y*y + z*z);
}


// Vector Magnitude
double Point3::Magnitude(const Point3& p2)
{
   return sqrt((p2.x-x)*(p2.x-x) + (p2.y-y)*(p2.y-y) + (p2.z-z)*(p2.z-
z));
}


// Cross Product Between Two Points
Point3 Point3::Cross(Point3& b)
{
   Point3 c;

   c.SetX(y*b.GetZ() - z*b.GetY());
   c.SetY(z*b.GetX() - x*b.GetZ());
   c.SetZ(x*b.GetY() - y*b.GetX());

   return c;
}


// Rho Calculation used in Giftwrapping
double Point3::Rho(Point3& a, Point3& n, const Point3& p2)
{
   double numer, denom;
   Point3 diff;

   diff = p2 - (*this);
   numer = -a.Dot_Product(diff);
   denom = n.Dot_Product(diff);
   if (DEQ0(denom))
   {
      if (numer < 0.0)
         return -MAX_DOUBLE;
      else
         return MAX_DOUBLE;
   }
   else
      return (numer/denom);
}
```

```
// Compute New N Calculation used in Giftwrapping
Point3 Point3::Compute_New_N(const double max_rho, Point3& a)
{
   double S;
   Point3 temp_point1, temp_point2;

   temp_point1 = ((*this) * max_rho) + a;
   S = (double)1.0 / temp_point1.Norm();
   if (a.Dot_Product(temp_point1) < 0.0)
      S = -S;
   temp_point2 = temp_point1 * S;
   return temp_point2;
}


Point3 Point3::Compute_New_A(const Point3& n, const Point3& p1, const
Point3& p2)
{
   Point3 temp_point1, temp_point2;
   double S;
   double a2, a3;

   a3 = -(p2.x - p1.x) / (p2.y - p1.y);
   a2 = -n.x / n.y;
   temp_point1.SetPoint(1.0, a2, a3);

   S = (double)1.0 / temp_point1.Norm();
   if ((*this).Dot_Product(temp_point1) < 0.0)
      S = -S;
   temp_point2 = temp_point1 * S;
   return temp_point2;
}


void Point3::Show_Full()
{
   cout << "(" << x << "," << y << "," << z << ")";
}


void Point3::Show()
{
   cout << id;
}

#endif
/* subfacet2.cc - SubFacet2 Class Implementation */

#ifdef __cplusplus

#include "subfacet2.h"


// Operators

int SubFacet2::operator==(const SubFacet2 &rhs)
{
   return ((x == rhs.x)&&(y == rhs.y));
}


// Accessors
```

```
void SubFacet2::SetSubFacet(Point2 P1)
{
   x = P1.GetX();
   y = P1.GetY();
   id = P1.GetID();
}


#endif
/* subfacet3.cc - SubFacet3 Class Implementation */

#ifdef __cplusplus

#include "subfacet3.h"


// Operators

int SubFacet3::operator==(const SubFacet3 &rhs)
{
   return (((p1 == rhs.p1) && (p2 == rhs.p2)) ||
           ((p1 == rhs.p2) && (p2 == rhs.p1)));
}


// Accessors

void SubFacet3::SetSubFacet(Point3 P1, Point3 P2)
{
   p1 = P1;
   p2 = P2;
}


#endif
/* test_ctz.cc */

#ifdef __cplusplus

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

#include "point3.h"
#include "list.h"
#include "plane.h"
#include "ctz.h"
#include "template.h"

main()
{
   fstream points_file;                  // file handle
   fstream triangles_file;               // file handle
   List<Point3> point_list;              // List of points
   List<Plane> triangle_list;            // List of triangles
   int n, m;
   Point3 a_point;
   Plane a_plane;
   Point3 plane_points[3];
   float p[3];
   float d[9];
```

```
   cout << "Reading Data Files..." << endl;

   cout << "   reading points" << endl;
   points_file.open("points.dat", ios::in);
   if (points_file.fail()|points_file.bad())
   {
      cout << "   ERROR Reading points file" << endl;
      exit(1);
   }
   n = 0;
   while (!points_file.eof())
   {
      points_file >> p[0] >> p[1] >> p[2];

      // there is an aparent bug in many C++ implementations that
      // does not catch eof properly when reading multiple things
      // on a line, this line is a work around for that bug
      if ((p[0] == 0.0) && (p[1] == 0.0) && (p[2] == 0.0))
         break;

      n++;
      a_point.SetPoint((double)p[0], (double)p[1], (double)p[2]);
      a_point.SetID(n);
      point_list.Insert_Tail(a_point);
#ifdef DEBUG
      cout << "      " << n << ": ";
      a_point.Show_Full();
      cout << endl;
#endif
   }
   points_file.close();
#ifdef DEBUG
   cout << "   " << n << " points read" << endl;
#endif

   cout << "   reading triangles" << endl;
   triangles_file.open("triangle.dat", ios::in);
   if (points_file.fail()|triangles_file.bad())
   {
      cout << "   ERROR reading triangles file" << endl;
      exit(1);
   }
   m=0;
   while (!triangles_file.eof())
   {
      triangles_file >> d[0] >> d[1] >> d[2] >> d[3] >> d[4] >> d[5] >>
d[6] >> d[7] >> d[8];

      // there is an aparent bug in many C++ implementations that
      // does not catch eof properly when reading multiple things
      // on a line, this line is a work around for that bug
      if ((d[0] == 0.0) && (d[1] == 0.0) && (d[2] == 0.0) &&
          (d[3] == 0.0) && (d[4] == 0.0) && (d[5] == 0.0) &&
          (d[6] == 0.0) && (d[7] == 0.0) && (d[8] == 0.0))
         break;

      plane_points[0].SetPoint((double)d[0], (double)d[1],
(double)d[2]);
      plane_points[1].SetPoint((double)d[3], (double)d[4],
(double)d[5]);
      plane_points[2].SetPoint((double)d[6], (double)d[7],
(double)d[8]);
```

```
      a_plane.SetPlane(plane_points[0], plane_points[1],
plane_points[2]);
      triangle_list.Insert_Tail(a_plane);
      m++;
#ifdef DEBUG
      cout << "      " << m << ": ";
      a_plane.Show_Full();
      cout << endl;
#endif
   }
   triangles_file.close();
#ifdef DEBUG
   cout << "   " << m << " triangles read" << endl;
#endif

   cout << "Running Constrained Tetrahedrizations Code" << endl;
   Constrain(point_list, triangle_list);
   cout << "done." << endl;
}

#endif
/* test_del.cc - Test Delaunay Triangulation */

#ifdef __cplusplus

#include <iostream.h>
#include "point3.h"
#include "facet3.h"
#include "chsplit.h"
#include "template.h"

main()
{
   Point3 p1, p2, p3, p4;
   Facet3 f1, f2;
   Facet3 test_f1, test_f2;
   int passed_test1, passed_test2;

   p1.SetPoint(2.0, 0.0, 1.0);
   p2.SetPoint(1.0, 1.0, 2.0);
   p3.SetPoint(3.0, 3.0, 1.0);
   p4.SetPoint(1.0, 20.0, 1.0);
   cout << "p1 = ";
   p1.Show_Full();
   cout << endl;
   cout << "p2 = ";
   p2.Show_Full();
   cout << endl;
   cout << "p3 = ";
   p3.Show_Full();
   cout << endl;
   cout << "p4 = ";
   p4.Show_Full();
   cout << endl;

   cout << "Testing Delaunay Triangulation Code" << endl;
   Delaunay_Triangulate (p1, p2, p3, p4, f1, f2);
   cout << "Facet f1 = ";
   f1.Show_Full();
   cout << "Facet f2 = ";
   f2.Show_Full();

   test_f1.SetFacet(p1, p2, p3);
```

```
    if (f1 == test_f1)
       passed_test1 = 1;
    else
       passed_test1 = 0;

    p3.SetPoint(20.0, 3.0, 1.0);
    p4.SetPoint(1.0, 4.0, 1.0);
    cout << endl;
    cout << "p1 = ";
    p1.Show_Full();
    cout << endl;
    cout << "p2 = ";
    p2.Show_Full();
    cout << endl;
    cout << "p3 = ";
    p3.Show_Full();
    cout << endl;
    cout << "p4 = ";
    p4.Show_Full();
    cout << endl;

    cout << "Testing Delaunay Triangulation Code" << endl;
    Delaunay_Triangulate (p1, p2, p3, p4, f1, f2);
    cout << "Facet f1 = ";
    f1.Show_Full();
    cout << "Facet f2 = ";
    f2.Show_Full();

    test_f2.SetFacet(p1, p2, p4);
    if (f1 == test_f2)
       passed_test2 = 1;
    else
       passed_test2 = 0;

    cout << endl;
    if (passed_test1 && passed_test2)
       cout << "Passed Delaunay Triangulation Test" << endl;
    else
       cout << "Failed Delaunay Triangulation Test" << endl;
}

#endif
/* test_lis.cc - Test List Class */

#include "list.h"
#include "template.h"

int main()
{
   List<int> L1;
   List<int> L2;
   int a;

   L1.Insert_At_Pointer(4);
   L1.Insert_Tail(5);
   L1.Insert_Head(3);
   L2 = L1;
   L1.Insert_Tail(6);
   L2.Remove_Tail();
   L1.Show();
   L2.Show();

   return 0;
```

```
}
#include <iostream.h>
#include "point2.h"
#include "template.h"

int main()
{
   Point2 a, b, c;
   Point2 temp;

   a.SetX(5.6);
   b.SetX(6.5);
   c.SetX(7.5);
   a.Show_Full();
   b.Show_Full();
   c.Show_Full();
   cout << endl;
   temp = a;
   a = b;
   b = temp;
   a.Show_Full();
   b.Show_Full();
   c.Show_Full();
   cout << endl;

   return 0;
}
#include <iostream.h>
#include "general.h"
#include "point3.h"
#include "plane.h"
#include "template.h"

int main()
{
   Point3 p1, p2, p3;
   Plane P1;
   Plane P2;
   double angle;

   // Set-Up Plane
   p1.SetPoint(0.0, 1.0, 1.0);
   p2.SetPoint(1.0, 1.0, 2.0);
   p3.SetPoint(-1.0, 2.0, -2.0);
   P1.SetPlane(p1,p2,p3);

   // See What Final Plane Looks Like
   P1.Show_Full();

   // Check Final Outputs
   if (!DEQ(P1.GetA(),1.0))
      cout << "Error A should be 1.0 but it is " << P1.GetA() << endl;
   else
      cout << "A is correct" << endl;
   if (!DEQ(P1.GetB(),-2.0))
      cout << "Error B should be -2.0 but it is " << P1.GetB() << endl;
   else
      cout << "B is correct" << endl;
   if (!DEQ(P1.GetC(),-1.0))
      cout << "Error C should be -1.0 but it is " << P1.GetC() << endl;
   else
      cout << "C is correct" << endl;
   if (!DEQ(P1.GetD(),3.0))
```

```
      cout << "Error D should be 3.0 but it is " << P1.GetD() << endl;
   else
      cout << "D is correct" << endl;

   // Set-Up Plane
   p1.SetPoint(0.0, 1.0, 1.0);
   p2.SetPoint(1.0, 0.0, 1.0);
   p3.SetPoint(1.0, 1.0, 0.0);
   P2.SetPlane(p1,p2,p3);

   // See What Final Plane Looks Like
   P2.Show_Full();

   // Check Final Outputs
   if (!DEQ(P2.GetA(),1.0))
      cout << "Error A should be 1.0 but it is " << P2.GetA() << endl;
   else
      cout << "A is correct" << endl;
   if (!DEQ(P2.GetB(),1.0))
      cout << "Error B should be 1.0 but it is " << P2.GetB() << endl;
   else
      cout << "B is correct" << endl;
   if (!DEQ(P2.GetC(),1.0))
      cout << "Error C should be 1.0 but it is " << P2.GetC() << endl;
   else
      cout << "C is correct" << endl;
   if (!DEQ(P2.GetD(),-2.0))
      cout << "Error D should be -2.0 but it is " << P2.GetD() << endl;
   else
      cout << "D is correct" << endl;

   // Test Angle Between Planes
   angle = P1.Angle(P2);
   cout << "Angle between P1 & P2: " << angle << endl;
   if (!DEQ(angle,1.079913648))
      cout << "Error angle should be 1.079913648 but it is " << angle
<< endl;
   else
      cout << "Angle is correct" << endl;

   p1.SetPoint(1.0, 0.0, 2.0);
   p2.SetPoint(-1.0, 3.0, 4.0);
   p3.SetPoint(3.0, 5.0, 7.0);
   P1.SetPlane(p1,p2,p3);
   P1.Show_Full();

   p1.SetPoint(2.0, 1.0, 3.0);
   p2.SetPoint(1.0, 3.0, 2.0);
   p3.SetPoint(-1.0, 2.0, 4.0);
   P1.SetPlane(p1,p2,p3);
   P1.Show_Full();

   cout << endl << endl << endl;

   p1.SetPoint(0.689413, 0.511246, 0.373577);
   p2.SetPoint(0.742424, 0.623402, 0.398663);
   p3.SetPoint(0.127354, 0.371654, 0.908200);
   P1.SetPlane(p1,p2,p3);
   P1.Show_Full();

   p1.SetPoint(0.691275, 0.866268, 0.560930);
   p2.SetPoint(0.305307, 0.982208, 0.574786);
   p3.SetPoint(0.494217, 0.909940, 0.644063);
```

```
   P2.SetPlane(p1,p2,p3);
   P2.Show_Full();

   return 0;
}
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <stdlib.h>

int main()
{
   fstream fp_in;
   fstream fp_out;
   int set_count = 0;
   char line[80 + 1];

   fp_in.open("bspchull.off", ios::in);
   fp_in.getline(line,80);  // burn off "LIST" line from .OFF file
   while(fp_in.getline(line,80))
   {
      cout << line << endl;
      if (line[0] == '{')
      {
         char fname[80];
         set_count++;
         strcpy(fname, "hull_##.off");
         switch(set_count)
         {
            case 1: fname[5] = '0'; fname[6] = '1'; break;
            case 2: fname[5] = '0'; fname[6] = '2'; break;
            case 3: fname[5] = '0'; fname[6] = '3'; break;
            case 4: fname[5] = '0'; fname[6] = '4'; break;
            case 5: fname[5] = '0'; fname[6] = '5'; break;
            case 6: fname[5] = '0'; fname[6] = '6'; break;
            case 7: fname[5] = '0'; fname[6] = '7'; break;
            case 8: fname[5] = '0'; fname[6] = '8'; break;
            case 9: fname[5] = '0'; fname[6] = '9'; break;
            default:
               cout << "ERROR, need more cases in switch()!" << endl;
               exit(-1);
         }
         fp_out.open(fname, ios::out);
         fp_out << "LIST" << endl;
      }
      fp_out << line << endl;
      if (line[0] == '}')
      {
         fp_out.close();
      }
   }
   fp_in.close();

   return 0;
}


#makefile.djg
CC      = gcc
CFLAGS  = -fno-implicit-templates -c -g -DDEBUG
#CFLAGS  = -fno-implicit-templates -c
ARFLAGS = -r
CLFLAGS = -f-no-implicit-templates -g -L .
```

```
#CLFLAGS = -f-no-implicit-templates -L .

OBJS = general.o point2.o point3.o plane.o lineseg.o\
 subfacet.o facet.o list.o

OBJS2 = chull.o bsp_tree.o chsplit.o oogl.o ctz.o faceti.o

all: libctz.a test_ctz test_del test_lis test_pla

test_pla: test_pla.o libctz.a
      gxx $(CLFLAGS) -o test_pla test_pla.o -lctz -lm

test_pla.o: template.h point3.h plane.h general.h test_pla.cc
      $(CC) $(CFLAGS) test_pla.cc

test_lis: test_lis.o libctz.a
      gxx $(CLFLAGS) -o test_lis test_lis.o -lctz -lm

test_lis.o: template.h list.h test_lis.cc
      $(CC) $(CFLAGS) test_lis.cc

test_ctz: test_ctz.o libctz.a
      gxx $(CLFLAGS) -o test_ctz test_ctz.o -lctz -lm

test_ctz.o: template.h point3.h plane.h list.h ctz.h test_ctz.cc
      $(CC) $(CFLAGS) test_ctz.cc

test_del: test_del.o libctz.a
      gxx $(CLFLAGS) -o test_del test_del.o -lctz -lm

test_del.o: template.h point3.h facet.h chsplit.h test_del.cc
      $(CC) $(CFLAGS) test_del.cc

libctz.a: $(OBJS) $(OBJS2)
      ar $(ARFLAGS) libctz.a $(OBJS)
      ar $(ARFLAGS) libctz.a $(OBJS2)
      ranlib libctz.a
      del libctz.lis
      nm --print-armap libctz.a > libctz.lis

ctz.o: list.h point3.h plane.h ctz.h bsp_tree.h chull.h oogl.h ctz.cc
      $(CC) $(CFLAGS) ctz.cc

faceti.o: faceti.h faceti.cc
      $(CC) $(CFLAGS) faceti.cc

oogl.o: list.h point3.h chull.h plane.h bsp_tree.h oogl.h\
 facet.h oogl.cc
      $(CC) $(CFLAGS) oogl.cc

chsplit.o: point3.h facet.h plane.h list.h chull.h chsplit.h\
 lineseg.h faceti.h chsplit.cc
      $(CC) $(CFLAGS) chsplit.cc

bsp_tree.o: plane.h point3.h list.h chull.h bsp_tree.h\
 facet.h chsplit.h bsp_tree.cc
      $(CC) $(CFLAGS) bsp_tree.cc

chull.o: list.h facet.h point3.h chull.h subfacet.h general.h\
 plane.h chull.cc
      $(CC) $(CFLAGS) chull.cc

list.o: facet.h list.h list.cc
```

```
	$(CC) $(CFLAGS) list.cc

subfacet.o: lineseg.h point3.h subfacet.h subfacet.cc
	$(CC) $(CFLAGS) subfacet.cc

facet.o: plane.h point3.h facet.h facet.cc
	$(CC) $(CFLAGS) facet.cc

lineseg.o: point3.h plane.h lineseg.h general.h
	$(CC) $(CFLAGS) lineseg.cc

plane.o: point3.h general.h plane.h plane.cc
	$(CC) $(CFLAGS) plane.cc

point2.o: point2.h point2.cc
	$(CC) $(CFLAGS) point2.cc

point3.o: point3.h general.h point3.cc
	$(CC) $(CFLAGS) point3.cc

general.o: general.h general.cc
	$(CC) $(CFLAGS) general.cc

clean:
	del test_ctz
	del test_ctz.exe
	del test_del
	del test_del.exe
	del test_lis
	del test_lis.exe
	del libctz.a
	del libctz.lis
	del *.o


#makefile.sol
CC      = CC
CFLAGS  = -c -g -DDEBUG
#CFLAGS  = -c
ARFLAGS = -xar
CLFLAGS = -g -L .
#CLFLAGS = -L .

OBJS = general.o point2.o point3.o plane.o lineseg.o\
 subfacet.o facet.o list.o

OBJS2 = chull.o bsp_tree.o chsplit.o oogl.o ctz.o faceti.o

all: libctz.a test_ctz test_del test_lis test_pla

test_pla: test_pla.o libctz.a
	$(CC) $(CLFLAGS) -o test_pla test_pla.o -lctz -lm

test_pla.o: template.h point3.h plane.h general.h test_pla.cc
	$(CC) $(CFLAGS) test_pla.cc

test_lis: test_lis.o libctz.a
	$(CC) $(CLFLAGS) -o test_lis test_lis.o -lctz -lm

test_lis.o: template.h list.h test_lis.cc
	$(CC) $(CFLAGS) test_lis.cc

test_ctz: test_ctz.o libctz.a
```

```
	$(CC) $(CLFLAGS) -o test_ctz test_ctz.o -lctz -lm

test_ctz.o: template.h point3.h plane.h list.h ctz.h test_ctz.cc
	$(CC) $(CFLAGS) test_ctz.cc

test_del: test_del.o libctz.a
	$(CC) $(CLFLAGS) -o test_del test_del.o -lctz -lm

test_del.o: template.h point3.h facet.h chsplit.h test_del.cc
	$(CC) $(CFLAGS) test_del.cc

libctz.a: $(OBJS) $(OBJS2)
	$(CC) $(ARFLAGS) -o libctz.a $(OBJS) $(OBJS2)
	rm -rf libctz.lis
	nm libctz.a > libctz.lis

ctz.o: list.h point3.h plane.h ctz.h bsp_tree.h chull.h oogl.h ctz.cc
	$(CC) $(CFLAGS) ctz.cc

faceti.o: faceti.h faceti.cc
	$(CC) $(CFLAGS) faceti.cc

oogl.o: list.h point3.h chull.h plane.h bsp_tree.h oogl.h\
 facet.h oogl.cc
	$(CC) $(CFLAGS) oogl.cc

chsplit.o: point3.h facet.h plane.h list.h chull.h chsplit.h\
 lineseg.h faceti.h chsplit.cc
	$(CC) $(CFLAGS) chsplit.cc

bsp_tree.o: plane.h point3.h list.h chull.h bsp_tree.h\
 facet.h chsplit.h bsp_tree.cc
	$(CC) $(CFLAGS) bsp_tree.cc

chull.o: list.h facet.h point3.h chull.h subfacet.h general.h\
 plane.h chull.cc
	$(CC) $(CFLAGS) chull.cc

list.o: facet.h list.h list.cc
	$(CC) $(CFLAGS) list.cc

subfacet.o: lineseg.h point3.h subfacet.h subfacet.cc
	$(CC) $(CFLAGS) subfacet.cc

facet.o: plane.h point3.h facet.h facet.cc
	$(CC) $(CFLAGS) facet.cc

lineseg.o: point3.h plane.h lineseg.h general.h lineseg.cc
	$(CC) $(CFLAGS) lineseg.cc

plane.o: point3.h general.h plane.h plane.cc
	$(CC) $(CFLAGS) plane.cc

point2.o: point2.h point2.cc
	$(CC) $(CFLAGS) point2.cc

point3.o: point3.h general.h point3.cc
	$(CC) $(CFLAGS) point3.cc

general.o: general.h general.cc
	$(CC) $(CFLAGS) general.cc

clean:
```

```
	rm -rf test_ctz test_del test_lis test_pla
	rm -rf libctz.a libctz.lis *.o Templates.DB core


# makefile.linux
CC      = g++
CFLAGS  = -fno-implicit-templates -c -g -DDEBUG
#CFLAGS  = -fno-implicit-templates -c
ARFLAGS = -r
CLFLAGS = -L .
#CLFLAGS = -f-no-implicit-templates -L .

OBJS = general.o plane.o list.o bsp_tree.o\
chsplit.o oogl.o ctz.o faceti.o

OBJS3 = point3.o lineseg3.o subfacet3.o facet3.o chull3.o

OBJS2 = point2.o lineseg2.o subfacet2.o facet2.o chull2.o

all: libctz.a test_ctz test_del test_lis test_pla test_p2

test_p2: test_p2.o libctz.a
	$(CC) $(CLFLAGS) -o test_p2 test_p2.cc -lctz -lm

test_p2.o: point2.h test_p2.cc
	$(CC) $(CFLAGS) test_p2.cc

test_pla: test_pla.o libctz.a
	$(CC) $(CLFLAGS) -o test_pla test_pla.cc -lctz -lm

test_pla.o: template.h point3.h plane.h general.h test_pla.cc
	$(CC) $(CFLAGS) test_pla.cc

test_lis: test_lis.o libctz.a
	$(CC) $(CLFLAGS) -o test_lis test_lis.cc -lctz -lm

test_lis.o: template.h list.h test_lis.cc
	$(CC) $(CFLAGS) test_lis.cc

test_ctz: test_ctz.o libctz.a
	$(CC) $(CLFLAGS) -o test_ctz test_ctz.cc -lctz -lm

test_ctz.o: template.h point3.h plane.h list.h ctz.h test_ctz.cc
	$(CC) $(CFLAGS) test_ctz.cc

test_del: test_del.o libctz.a
	$(CC) $(CLFLAGS) -o test_del test_del.cc -lctz -lm

test_del.o: template.h point3.h facet3.h chsplit.h test_del.cc
	$(CC) $(CFLAGS) test_del.cc

libctz.a: $(OBJS) $(OBJS2) $(OBJS3)
	ar $(ARFLAGS) libctz.a $(OBJS)
	ar $(ARFLAGS) libctz.a $(OBJS2)
	ar $(ARFLAGS) libctz.a $(OBJS3)
	ranlib libctz.a
	rm -rf libctz.lis
	nm --print-armap libctz.a > libctz.lis

ctz.o: list.h point3.h plane.h ctz.h bsp_tree.h chull3.h oogl.h ctz.cc
	$(CC) $(CFLAGS) ctz.cc

faceti.o: faceti.h faceti.cc
```

```
	$(CC) $(CFLAGS) faceti.cc

oogl.o: list.h point3.h chull3.h plane.h bsp_tree.h oogl.h\
 facet3.h oogl.cc
	$(CC) $(CFLAGS) oogl.cc

chsplit.o: point3.h point2.h facet3.h plane.h list.h chull3.h
chsplit.h\
 lineseg3.h faceti.h chsplit.cc general.h
	$(CC) $(CFLAGS) chsplit.cc

bsp_tree.o: plane.h point3.h list.h chull3.h bsp_tree.h\
 facet3.h chsplit.h bsp_tree.cc
	$(CC) $(CFLAGS) bsp_tree.cc

chull3.o: list.h facet3.h point3.h chull3.h subfacet3.h general.h\
 plane.h chull3.cc
	$(CC) $(CFLAGS) chull3.cc

chull2.o: list.h facet2.h point2.h chull2.h subfacet2.h general.h\
 chull2.cc
	$(CC) $(CFLAGS) chull2.cc

list.o: facet3.h list.h list.cc
	$(CC) $(CFLAGS) list.cc

subfacet3.o: lineseg3.h point3.h subfacet3.h subfacet3.cc
	$(CC) $(CFLAGS) subfacet3.cc

subfacet2.o: point2.h subfacet2.h subfacet2.cc
	$(CC) $(CFLAGS) subfacet2.cc

facet3.o: plane.h point3.h facet3.h facet3.cc
	$(CC) $(CFLAGS) facet3.cc

facet2.o: lineseg2.h point2.h facet2.h facet2.cc
	$(CC) $(CFLAGS) facet2.cc

lineseg3.o: point3.h plane.h lineseg3.h lineseg3.cc
	$(CC) $(CFLAGS) lineseg3.cc

lineseg2.o: point2.h lineseg2.h lineseg2.cc
	$(CC) $(CFLAGS) lineseg2.cc

plane.o: point3.h general.h plane.h plane.cc subfacet2.h point2.h
	$(CC) $(CFLAGS) plane.cc

point2.o: point2.h point2.cc
	$(CC) $(CFLAGS) point2.cc

point3.o: point3.h general.h point3.cc
	$(CC) $(CFLAGS) point3.cc

general.o: general.h general.cc
	$(CC) $(CFLAGS) general.cc

clean:
	rm -rf test_ctz
	rm -rf test_del
	rm -rf test_lis
	rm -rf test_pla
	rm -rf test_p2
	rm -rf libctz.a
```

```
rm -rf libctz.lis
rm -rf *.o
```

```
0.928495 0.926298 0.020661
0.802545 0.684286 0.855251
0.902036 0.744560 0.479446
0.158940 0.434187 0.919797
0.901578 0.132847 0.135655
0.691275 0.866268 0.560930
0.000580 0.891781 0.059420
0.217658 0.434126 0.963897
0.170415 0.294626 0.150792
0.133213 0.885189 0.093905
0.254402 0.208655 0.107730
0.833247 0.218543 0.103061
0.379101 0.906125 0.985839
0.668722 0.962065 0.440870
0.905484 0.940886 0.560717
0.502091 0.727226 0.383129
0.143773 0.778466 0.370769
0.228767 0.839808 0.100803
0.689413 0.511246 0.373577
0.633869 0.946226 0.878170
0.012421 0.085482 0.222449
0.781884 0.490768 0.777581
0.475814 0.141270 0.882168
0.921506 0.468123 0.067507
0.887143 0.943052 0.280831
0.748161 0.506088 0.199255
0.637104 0.083468 0.565386
0.932218 0.749474 0.168096
0.065950 0.802332 0.911405
0.212317 0.822810 0.846004
0.551866 0.996582 0.364238
0.200354 0.971648 0.452834
0.559771 0.439711 0.479781
0.466781 0.945158 0.866970
0.149480 0.494369 0.213446
0.603595 0.911618 0.086550
0.840968 0.822779 0.204962
0.916135 0.316385 0.968200
0.902066 0.399945 0.942808
0.303415 0.728782 0.511155
0.679983 0.634907 0.897153
0.873989 0.997009 0.552446
0.169622 0.735893 0.203131
0.031373 0.073885 0.910367
0.226051 0.739311 0.486648
0.284036 0.306009 0.892117
0.305307 0.982208 0.574786
0.002258 0.426618 0.786035
0.988342 0.363720 0.750175
0.895779 0.581347 0.616565
0.426466 0.633137 0.513169
0.235664 0.067751 0.177129
0.873379 0.287088 0.140446
0.926725 0.715598 0.295419
0.257942 0.584063 0.289071
0.566637 0.592639 0.552446
0.936003 0.809259 0.125828
0.742424 0.623402 0.398663
0.954192 0.379498 0.578997
0.127354 0.371654 0.908200
```

```
0.998962 0.785424 0.945616
0.584826 0.296823 0.414686
0.305673 0.854091 0.862239
0.580767 0.864345 0.102237
0.797998 0.506088 0.187933
0.881222 0.604694 0.408246
0.236793 0.196844 0.280923
0.887020 0.193182 0.955870
0.808802 0.704062 0.095981
0.164098 0.492447 0.798883
0.000671 0.971191 0.119175
0.884945 0.883663 0.154027
0.411481 0.469527 0.009033
0.133976 0.738731 0.641224
0.977752 0.120579 0.557573
0.468734 0.176092 0.476363
0.549547 0.354747 0.844630
0.895718 0.924558 0.188635
0.483718 0.492386 0.859279
0.641865 0.816645 0.388562
0.539018 0.449324 0.027619
0.899960 0.771935 0.021332
0.494217 0.909940 0.644063
0.229072 0.732383 0.527421
0.184271 0.707511 0.773370
0.621204 0.507767 0.520035
0.691000 0.325327 0.406659
0.527940 0.497940 0.108707
0.856868 0.308634 0.129978
0.503128 0.409284 0.966704
0.737022 0.082949 0.935240
0.680990 0.999145 0.790887
0.693228 0.332102 0.687246
0.202795 0.187628 0.053407
0.186285 0.826075 0.786523
0.049867 0.200201 0.024689
0.568224 0.527940 0.357311
0.665181 0.365246 0.080905
0.934751 0.076846 0.474044
0.372692 0.420820 0.420454
0.0 0.0 0.0


3
100
0.928495 0.926298 0.020661
0.802545 0.684286 0.855251
0.902036 0.744560 0.479446
0.158940 0.434187 0.919797
0.901578 0.132847 0.135655
0.691275 0.866268 0.560930
0.000580 0.891781 0.059420
0.217658 0.434126 0.963897
0.170415 0.294626 0.150792
0.133213 0.885189 0.093905
0.254402 0.208655 0.107730
0.833247 0.218543 0.103061
0.379101 0.906125 0.985839
0.668722 0.962065 0.440870
0.905484 0.940886 0.560717
0.502091 0.727226 0.383129
0.143773 0.778466 0.370769
0.228767 0.839808 0.100803
```

0.689413 0.511246 0.373577
0.633869 0.946226 0.878170
0.012421 0.085482 0.222449
0.781884 0.490768 0.777581
0.475814 0.141270 0.882168
0.921506 0.468123 0.067507
0.887143 0.943052 0.280831
0.748161 0.506088 0.199255
0.637104 0.083468 0.565386
0.932218 0.749474 0.168096
0.065950 0.802332 0.911405
0.212317 0.822810 0.846004
0.551866 0.996582 0.364238
0.200354 0.971648 0.452834
0.559771 0.439711 0.479781
0.466781 0.945158 0.866970
0.149480 0.494369 0.213446
0.603595 0.911618 0.086550
0.840968 0.822779 0.204962
0.916135 0.316385 0.968200
0.902066 0.399945 0.942808
0.303415 0.728782 0.511155
0.679983 0.634907 0.897153
0.873989 0.997009 0.552446
0.169622 0.735893 0.203131
0.031373 0.073885 0.910367
0.226051 0.739311 0.486648
0.284036 0.306009 0.892117
0.305307 0.982208 0.574786
0.002258 0.426618 0.786035
0.988342 0.363720 0.750175
0.895779 0.581347 0.616565
0.426466 0.633137 0.513169
0.235664 0.067751 0.177129
0.873379 0.287088 0.140446
0.926725 0.715598 0.295419
0.257942 0.584063 0.289071
0.566637 0.592639 0.552446
0.936003 0.809259 0.125828
0.742424 0.623402 0.398663
0.954192 0.379498 0.578997
0.127354 0.371654 0.908200
0.998962 0.785424 0.945616
0.584826 0.296823 0.414686
0.305673 0.854091 0.862239
0.580767 0.864345 0.102237
0.797998 0.506088 0.187933
0.881222 0.604694 0.408246
0.236793 0.196844 0.280923
0.887020 0.193182 0.955870
0.808802 0.704062 0.095981
0.164098 0.492447 0.798883
0.000671 0.971191 0.119175
0.884945 0.883663 0.154027
0.411481 0.469527 0.009033
0.133976 0.738731 0.641224
0.977752 0.120579 0.557573
0.468734 0.176092 0.476363
0.549547 0.354747 0.844630
0.895718 0.924558 0.188635
0.483718 0.492386 0.859279
0.641865 0.816645 0.388562
0.539018 0.449324 0.027619

```
0.899960 0.771935 0.021332
0.494217 0.909940 0.644063
0.229072 0.732383 0.527421
0.184271 0.707511 0.773370
0.621204 0.507767 0.520035
0.691000 0.325327 0.406659
0.527940 0.497940 0.108707
0.856868 0.308634 0.129978
0.503128 0.409284 0.966704
0.737022 0.082949 0.935240
0.680990 0.999145 0.790887
0.693228 0.332102 0.687246
0.202795 0.187628 0.053407
0.186285 0.826075 0.786523
0.049867 0.200201 0.024689
0.568224 0.527940 0.357311
0.665181 0.365246 0.080905
0.934751 0.076846 0.474044
0.372692 0.420820 0.420454


0.954192 0.379498 0.578997 0.633869 0.946226 0.878170 0.568224 0.527940
0.357311
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0


0.954192 0.379498 0.578997 0.633869 0.946226 0.878170 0.568224 0.527940
0.357311
0.689413 0.511246 0.373577 0.742424 0.623402 0.398663 0.127354 0.371654
0.908200
0.873379 0.287088 0.140466 0.833247 0.218543 0.103061 0.228767 0.839808
0.100803
0.691275 0.866268 0.560930 0.305307 0.982208 0.574786 0.494217 0.909940
0.644063
0.921506 0.468123 0.067507 0.637104 0.083468 0.565386 0.680990 0.999145
0.790887
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```