



The validation of a triangulated boundary representation in 3D

Report

M.E. Hoefsloot

Faculty of Civil Engineering and Geosciences
Department of Geodesy, Department of OTB
Delft University of Technology
Jaffalaan 9, 2628 BX Delft, the Netherlands
Email: M.E.Hoefsloot@student.tudelft.nl

The validation of a triangulated boundary representation in 3D

Report

M.E. Hoefsloot

Faculty of Civil Engineering and Geosciences
Department of Geodesy, Department of OTB
Delft University of Technology
Jaffalaan 9, 2628 BX Delft, the Netherlands
Email: M.E.Hoefsloot@student.tudelft.nl

Preface

This report is part of the course “Individual Assignment” and this course is part of the master study “Geodetic Engineering - Geo-information and Land Management”. In the professional practice of an engineer, problem solving is core business. A central element in this problem solving is designing. In the “Individual Assignment”, this designing will be trained in an integrated way, using a real life problem. This assignment should be done individually and it has to be completed within 5 months. The course can be done in Dutch as well as English; this depends on the topic. The course contains three central elements:

- making a time-schedule for the assignment;
- making the specific design;
- presenting the results of the research project by writing a report and an optional oral presentation.

This report is the third element.

I would like to thank my supervisor ir. E. Verbree with whom I worked together on this topic. He has been a great help in all areas. Furthermore, I would like to thank ir. J.J.C.M. Hoefsloot, my father; especially for his help on thinking out problems, solving C++ difficulties, and reviewing this report.

Zoetermeer, Delft, Februari 2005
M.E. Hoefsloot

Content

Preface	i
1. Introduction	1
2. A tetrahedralization to represent the volume of an object	3
2.1 Definition of a triangulation and a tetrahedralization.....	3
2.2 Problems with triangulations and tetrahedralizations.....	4
2.3 Delaunay triangulation and tetrahedralization.....	4
2.4 Computation algorithms	5
2.5 Space subdivision	7
3. A triangulation with intersecting facets	9
3.1 Triangulation: A surface model	9
3.2 Topological relationships between triangles	10
3.3 Calculating whether triangles intersect.....	14
3.4 Calculating where triangles intersect.....	16
4. Present solutions	21
4.1 “Triangle”	21
4.2 “TetGen”	22
5. “Intersect”	25
5.1 The purpose of “Intersect”	25
5.2 The code itself	25
5.3 Assignment for the future	26
6. A worked out example	27
6.1 Two examples	27
6.2 Step 1: “TetGen”, a check	27
6.3 Step 2: “Intersect”	28
6.4 Step 3: Manual edits – part 1	29
6.5 Step 4: “Triangle”	30
6.6 Step 5: Manual edits – part 2	31
6.7 Step 6: “TetGen”	34
6.8 About the result	35
7. Conclusions and recommendations	37
7.1 Conclusions	37
7.2 Recommendations	37
References	39
Appendix A: Predicates_intersect.cxx	41
Appendix B: Intersect.cxx – part 1	45
Appendix C: Intersect.cxx – part 2	49
Appendix D: On the CD	51

1. Introduction

For many reasons we want to have a boundary representation of an object in 3D, which is valid and closed. One reason is that we want to construct a tetrahedralization; therefore, the boundary representation should be a valid and closed triangulation.

For constructing this tetrahedralization, the program “TetGen” can be used. The input of “TetGen” should be a valid and closed representation, if not “TetGen” detects the invalid polygons. The aim of this research is to conduct a solution to detect invalid triangles and to make them valid.

“TetGen” is a program, written by Hang Si, for generating quality tetrahedral meshes and three-dimensional Delauney triangulations. It currently computes exact Delauney tetrahedralizations, constrained Delauney tetrahedralizations, and quality tetrahedral meshes. “TetGen” incorporates a suit of geometrical and mesh generation algorithms. “TetGen” may be freely copied, modified, and redistributed under the copyright notices given in the licence file.

To come to a solution of the problem, the first step is to make an inventory of all topological relations existing in 3D. A topological relation is defined as the set of properties, which are invariant under homeomorphisms. When there is a kind of elastic transformation, metric properties are changing, topological properties not.

It is clear, the validation has to be done in small steps, because of the programming and because of the different topological relations. Therefore, a framework has to be designed and every step or possibility has to be added to it.

As said before, the outcome of this research should be a programming code, which can be used to change an invalid triangulated boundary representation into a valid one in 3D. This will be done in six steps.

- 1 Make an inventory of all topological relations that exist between two faces.
- 2 Find a way to test whether two triangles intersect and to determine their topological relation.
- 3 Find the way to derive the points of intersection.
- 4 Look to the solution of Hang Si (“TetGen”), and use this program to implement the new solution in C++.
- 5 Look to the solutions or parts of solutions made by other programmers.
- 6 Find possibilities, like indexing methods, which can make the algorithm more efficient.

The steps are needed to give an answer to the research question:

What is an efficient way to change an invalid 3D triangulated boundary representation into a valid one?

The use of the code of Hang Si and the use of C++ are the only preconditions that are defined in the assignment. This is because the code of Hang Si is good, and it is extensible by others because it is written in C++ and therefore well readable.

Chapter 2 presents theory behind triangulations and tetrahedralizations. More about problems with triangulations, and how to solve these problems is explained in Chapter 3. Chapter 4 continues with present solutions to create triangulations and tetrahedralizations. The

implementation of the new solution “Intersect” is presented in Chapter 5. Chapter 6 contains two worked out examples, and chapter 7 concludes this research and has recommendations for further research.

2. A tetrahedralization to represent the volume of an object

The chapter is pure theory, it contains some definitions (§ 2.1) and explanations of problems that can occur with triangulations and tetrahedralizations (§ 2.2), the properties of a Delauney triangulation and tetrahedralization (§ 2.3), some calculation algorithms (§ 2.4) and the use of space subdivision (§ 2.5).

2.1 Definition of a triangulation and a tetrahedralization

Many physical phenomena in science and engineering can be modelled by partial differential equations (PDEs). When these equations have complicated boundary conditions or when they are posed on irregularly shaped objects or domains, they usually do not admit closed-form solutions. Therefore, a numerical approximation of the solution is necessary.

Numerical methods for solving PDEs include the finite element method (FEM), the finite volume method (FVM, also known as the control volume method), and the boundary element method (BEM). They are used to model disparate phenomena such as mechanical deformation, heat transfer, fluid flow, electromagnetic wave propagation, and quantum mechanics. These methods numerically approximate the solution of a linear or nonlinear PDE by replacing the continuous system with a finite number of coupled linear or nonlinear algebraic equations. This process of discretization associates a variable with each of a finite number of points in the problem domain. For instance, to simulate heat conduction through an electrical component, the temperature is recorded at a number of points, called nodes, on the surface and in the interior of the component.

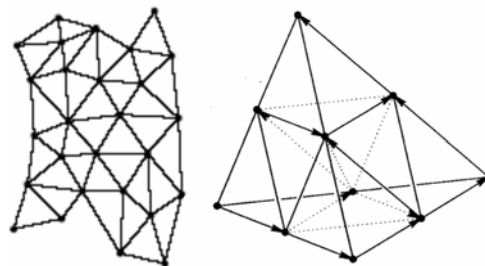


Figure 1: A triangulation (left) and a tetrahedralization (right).

It is not enough to choose a set of points to act as nodes; the problem domain (or in the BEM, the boundary of the problem domain) must be partitioned into small pieces of simple shape. In the FEM, these pieces are called elements, and are usually triangles (in two dimensions), or tetrahedra (in three dimensions). The FEM employs a node at every element vertex (and sometimes at other locations); each node is typically shared among several elements. The collection of nodes and elements is called a finite element mesh. Because elements have simple shapes, it is easy to approximate the behaviour of a PDE, such as the heat equation, on each element. By accumulating these effects over all the elements, one derives a system of equations whose solution approximates a set of physical quantities such as the temperature at each node.

The FVM and the BEM also use meshes, although with differences in terminology and differences in the meshes themselves. Finite volume meshes are composed of control volumes, which sometimes are clusters of triangles or tetrahedra, and sometimes are the cells

of a geometric structure known as the Voronoi diagram. In either case, an underlying simplicial mesh is typically used to interpolate the nodal values and to generate the control volumes. Boundary element meshes do not partition an object; only its boundaries are partitioned. Hence, a two-dimensional domain would have boundaries divided into straight-line elements, and a three-dimensional domain would have boundaries partitioned into polygonal (typically triangular) elements.

It may be stated that a *triangulation* is a boundary representation, where the problem domain is partitioned into triangles (in two dimensions) or where boundary element meshes are partitioned into triangular elements (in three dimensions). A *tetrahedralization* then is a volume representation of the problem domain (in three dimensions) that is partitioned into tetrahedrals. Figure 1 shows a triangulation (in two dimensions) and a tetrahedralization (in three dimensions).

2.2 Problems with triangulations and tetrahedralizations

Unfortunately, discretizing an object of simulation is a more difficult problem than it appears at first glance. A useful mesh satisfies constraints that sometimes seem almost contradictory. A mesh must conform to the object or domain being modelled, and ideally should meet constraints on both the size and shape of its elements.

Furthermore, the constraints of element size and element shape are difficult to reconcile because elements must meet squarely along the full extent of their shared edges or faces. Figure 2 illustrates illegal meetings between adjacent elements. For instance, at left, the edge of one triangular element is a portion of an edge of an adjoining element.

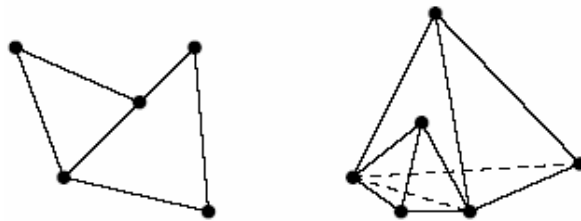


Figure 2: Illegal meetings between adjacent elements.

When the coordinates of the nodes of the elements are known it is possible to compute the points of intersection and create new and legal elements and legal meetings between adjacent elements. A renowned method of Aftosmis et al. (1997) for detecting intersecting triangles is described in Paragraph 3.3, because it is implemented in the code of Si (2004), O'Rourke (1998) developed that code.

2.3 Delaunay triangulation and tetrahedralization

In mathematics, and computational geometry, the **Delaunay triangulation** or **Delone triangularization** for a set \mathbf{P} of points in the plane is the triangulation $DT(\mathbf{P})$ of \mathbf{P} such that no point in \mathbf{P} is inside the circumcircle of any triangle in $DT(\mathbf{P})$. Delaunay triangulations maximize the minimum angle of all the angles of the triangles in the triangulation; they tend to avoid "sliver" triangles. Boris Delaunay (1934) invented the triangulation.

In the n -dimensional case, it is stated as follows. For a set \mathbf{P} of points in the (n -dimensional) Euclidean space, the **Delaunay triangulation** is the triangulation $DT(\mathbf{P})$ of \mathbf{P} such that no point in \mathbf{P} is inside the circum-hypersphere of any simplex in $DT(\mathbf{P})$. Equivalently, the Delaunay triangulation of a discrete point set \mathbf{P} is the geometric dual of the **Voronoi tessellation** for \mathbf{P} . It is known that the Delaunay triangulation exists and is unique for \mathbf{P} , if \mathbf{P} is a set of points in **general position**. That means for a two-dimensional set of points: no three points are on the same line and no four are on the same circle, or for an n -dimensional set of points: no $n + 1$ points are on the same hyperplane and no $n + 2$ points are on the same hypersphere. An elegant proof of this fact is outlined below. It is worth mentioning, because it reveals connections between the two constructs fundamental for computational and combinatorial geometry.

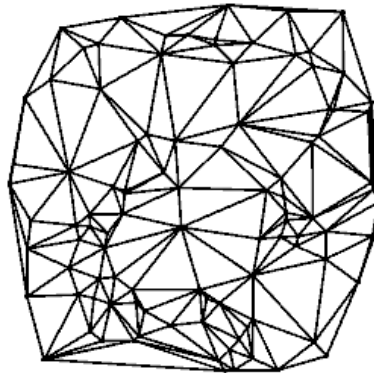


Figure 3: This is the Delaunay triangulation of a random set of points in the plane.

The problem of finding the Delaunay triangulation of a set of points in n -dimensional Euclidean space, can be converted to the problem of finding the convex hull of a set of points in $(n + 1)$ -dimensional space, by giving all points \mathbf{p} an extra coordinate equal to \mathbf{p}^2 , taking the bottom side of the convex hull, and mapping back to n -dimensional space by deleting the last coordinate. As the convex hull is unique, so is the triangulation, assuming all facets of the convex hull are simplexes. A facet not being a simplex implies that $n + 2$ of the original points lay on the same d -hypersphere, and the points were not in general position.

On the other hand, it is easily seen that for the set of three points on the same line there is no Delaunay triangulation (in fact, no triangulation at all). And, for four points on the same circle (e.g., the vertices of a rectangle) the Delaunay triangulation is not unique: clearly, the two possible triangulations that split the quadrangle into two triangles satisfy the Delaunay condition. Generalizations are possible to metrics other than Euclidean. However, in these cases the Delaunay triangulation is not guaranteed to exist or be unique. The text of this paragraph is found in “The Free Encyclopaedia WikipediA” (<http://en.wikipedia.org>).

2.4 Computation algorithms

The area of a triangle is one half the base times the altitude. However, this formula is not directly useful if you want the area of a triangle T , whose three vertices are arbitrary points a , b , c . Let denote this area $A(T)$. The base is easy: $|a - b|$, but the altitude is not so immediately available from the coordinates, unless the triangle happens to be oriented with one side parallel to one of the axes.

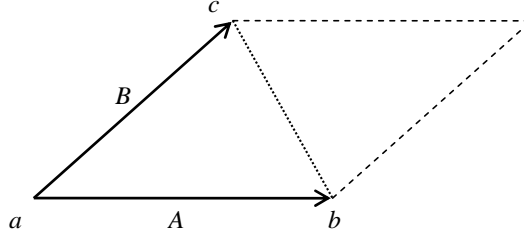


Figure 4: Cross product parallelogram

From linear algebra, we know that the magnitude of the cross product of two vectors is the area of the parallelogram they determine: If A and B are vectors, then $|A \times B|$ is the area of the parallelogram with sides A and B , as shown in Figure 4. Since any triangle can be viewed as half of a parallelogram, this gives an immediate method of computing the area from its coordinates. Just let $A = b - a$ and $B = c - a$. Then the area is half the length of $A \times B$. The cross product can be computed from the following determinant, where \hat{i} , \hat{j} , and \hat{k} are unit vectors in the x , y , and z directions respectively:

$$A(T) = \frac{1}{2} \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \end{vmatrix} = \frac{1}{2} \left((A_1 B_2 - A_2 B_1) \hat{i} + (A_2 B_0 - A_0 B_2) \hat{j} + (A_0 B_1 - A_1 B_0) \hat{k} \right) \quad (1)$$

For two-dimensional vectors, $A_2 = B_2 = 0$, so the above calculation reduces to $(A_0 B_1 - A_1 B_0) \hat{k}$: The cross product is a vector normal (perpendicular) to the plane of the triangle. Thus, the area is given by

$$A(T) = \frac{1}{2} (A_0 B_1 - A_1 B_0) \quad (2)$$

Substitution of $A = b - a$ and $B = c - a$ yields

$$\begin{aligned} 2A(T) &= a_0 b_1 - a_1 b_0 + a_1 c_0 - a_0 c_1 + b_0 c_1 - b_1 c_0 \\ &= (b_0 - a_0)(c_1 - a_1) - (c_0 - a_0)(b_1 - a_1) \end{aligned} \quad (3)$$

This achieves the immediate goal: an expression for the area of the triangle as a function of the coordinates of its vertices. There is however, another way to represent the calculation of the cross product that is formally identical but generalizes more easily to higher dimensions. The expression obtained above, is the value of the 3×3 determinant of the three point coordinates, with the third coordinate replaced by 1:

$$\begin{vmatrix} a_0 & a_1 & 1 \\ b_0 & b_1 & 1 \\ c_0 & c_1 & 1 \end{vmatrix} = (b_0 - a_0)(c_1 - a_1) - (c_0 - a_0)(b_1 - a_1) = 2A(T) \quad (4)$$

One of the benefits of the determinant formulation of the area of a triangle is that it extends directly into higher dimensions. In three dimensions, the volume of a tetrahedron T with vertices a, b, c, d is

$$\begin{aligned}
 6V(T_{abcd}) &= \begin{vmatrix} a_0 & a_1 & a_2 & 1 \\ b_0 & b_1 & b_2 & 1 \\ c_0 & c_1 & c_2 & 1 \\ d_0 & d_1 & d_2 & 1 \end{vmatrix} \\
 &= -(a_2 - d_2)(b_1 - d_1)(c_0 - d_0) + (a_1 - d_1)(b_2 - d_2)(c_0 - d_0) \\
 &\quad + (a_2 - d_2)(b_0 - d_0)(c_1 - d_1) - (a_0 - d_0)(b_2 - d_2)(c_1 - d_1) \\
 &\quad - (a_1 - d_1)(b_0 - d_0)(c_2 - d_2) + (a_0 - d_0)(b_1 - d_1)(c_2 - d_2)
 \end{aligned} \tag{5}$$

This volume is signed; it is positive if (a,b,c) form a counterclockwise circuit when viewed from the side away from d , so that the face normal determined by the right-hand rule points toward the outside. For example, let $a = (1, 0, 0)$, $b = (0, 1, 0)$, $c = (0, 0, 1)$, and $d = (0, 0, 0)$. Then (a,b,c) is counterclockwise from outside; see Figure 5. Substitution into Equation (5) yields a determinant of 1, so $V(T) = \frac{1}{6}$. This accords with the $\frac{1}{3}$ base area times high rule: $\frac{1}{3} \cdot \frac{1}{2} \cdot 1$. Remarkably, this theorem generalizes directly also: The volume of a polyhedron may be computed by summing the (signed) volumes of tetrahedra formed by an arbitrary point and each triangular face of the polyhedron. Here all the faces must be oriented counterclockwise from outside.

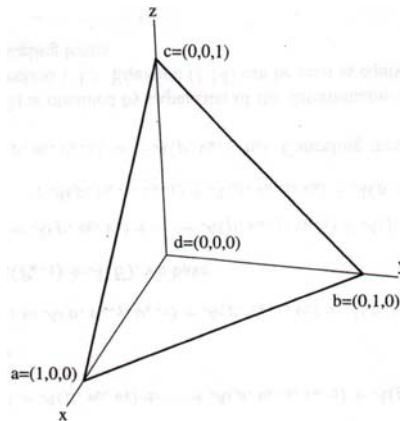


Figure 5: Tetrahedron at the origin

2.5 Space subdivision

Dealing with very complex scenes is one of the major challenges for current computer graphics research. To facilitate manipulation of such environments it is necessary to develop spatial subdivision structures which allow the implementation of efficient searching algorithms for such applications as rendering (with ray-tracing or radiosity-style methods), collision detection in animation and also for interactive environments (virtual reality etc.).

The performance of spatial subdivision data structures is notoriously difficult to analyse. Some previous work has been done however. In this section a look at some of the spatial subdivision structures is presented. The following classes of spatial subdivision structures are interesting: uniform grids, recursive grids and octrees (i.e. object-octrees).

Uniform grid

Uniform grids are built by subdividing the sides of the bounding box Δ of the scene, along the x, y, z -axes into n_x, n_y and n_z subdivisions. Each element of this subdivision is called a voxel. In each voxel, a pointer toward the items intersecting it is stored.

Recursive grid

Recursive grid is a method in which each voxel containing a number of pointers greater than MAXP (where MAXP stands for MAXimum number of Polygons), is recursively subdivided.

Octree

Octrees can be viewed as a special case of recursive grids for which subdivision into eight sub-voxels is performed at every step. Their advantage is a natural adaptation to the geometric complexity of a scene and the fact that special optimisation can be performed for ray-traversal. Their main drawbacks are that a hierarchy of large depth may be created, a penalty of traversing the hierarchy is often incurred and that duplication of objects in lists is frequent.

Hierarchy of Uniform Grids (HUG)

The method HUG first groups objects of the same size, and then creates clusters of neighbouring objects in the same size group. After that, an appropriately subdivided uniform grid is placed around each cluster and a Hierarchy of Uniform Grids is constructed.

***K*-dimensional (*k*-d) Tree (Spatial Hierarchy)**

The k -d tree, or k -dimensional binary search tree is a binary search tree that generalizes the 1-d tree or ordinary binary search tree to R^k . A partition of space into hyperrectangles is obtained by splitting alternating coordinate axes by hyperplanes through data points. Insertion and search are implemented as for the standard binary search tree algorithms. These trees are used for a variety of other operations, including orthogonal range searching (report all points within a given rectangle), partial match queries (report all points whose values match a given k -dimensional vector with possibly a number of wild cards, for example, we may search for all points with values $(a_1, *, *, a_4, a_5, *)$, where $*$ denotes a wild card). Additionally, nearest neighbour searching is greatly facilitated by k -d trees. Si (2004) implemented this method of space subdivision into "TetGen".

3. A triangulation with intersecting facets

More about problems with triangulations, intersecting triangles is explained in this chapter. It starts with a paragraph about the use of triangulations in practice (§ 3.1). To determine the points of intersection of two adjacent triangles it is useful to know something about topological relations between objects, especially about two triangles in three-dimensional space (§ 3.2), this paragraph is based on Pigot (1991). Aftosmis et al. (1997) developed a new method to detect whether two triangles are intersecting and O'Rourke (1998) about where two segments and where a segment and a face are intersecting, these two calculation methods are lied down in paragraph § 3.3 and § 3.4.

3.1 Triangulation: A surface model

In recent years, point primitives have received growing attention in computer graphics. There are two main reasons for this new interest in points: On one hand, we have witnessed a dramatic increase in the polygonal complexity of computer graphics models. The overhead of managing, processing and manipulating very large polygonal meshes has led many researchers to question the future utility of polygons as the fundamental graphics primitive. On the other hand, modern 3D digital photography and 3D scanning systems facilitate the ready acquisition of complex, real-world objects. These techniques generate huge volumes of point samples and they have created the need for advanced point processing. Conceptually, points constitute the atomic digital building blocks of object geometry and appearance - just as pixels form the digital elements of 2D images.

The research of CGL (2001) in point-based graphics was inspired by the desire to design an alternative pipeline for efficient 3D content creation. Such a pipeline is displayed below.

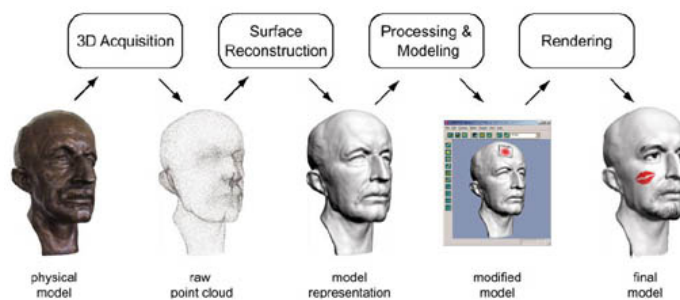


Figure 6: The pipeline from a physical model to a digital model

They start with a real world model, which will be digitized by some 3D acquisition process. As a result, they obtain a raw 3D point cloud. The first important component of their pipeline is a surface model (in this case a triangulation) – a powerful mathematical representation that interpolates the discrete point samples and allows for sophisticated surface processing including cleaning, hole-filling and the like. Next, users will want to edit the surface geometry or modify its appearance attributes. This includes editing operators, such as surface deformation and Boolean operations, as well as an editing metaphor. In addition, 3D painting, texturing carving and effect filtering is needed, in the same way as we know it from conventional 2D image editing. Other important tools encompass compression or digital watermarking. Finally, they have to provide efficient methods for the rendering of point

sampled geometry to display their models in high quality. This includes dedicated point pipelines, hardware acceleration and the removal of aliasing.

3.2 Topological relationships between triangles

In well-shaped triangulations, elements must meet squarely along the full extent of their shared points or edges. Starting point of this research is the case of non-well-shaped triangulations, therefore, in this paragraph an overview of all relations or meetings between two adjacent triangles in two and three dimensions based on the research of Pigot (1991).

Point-set topology (classical topology) provides an intuitive view of topological relationships. In the paper of Pigot (1991), point-set binary topological relationships between 1-simplexes in R^3 , 2-simplexes in R^3 and 3-simplexes in R^3 are based on the consideration of the fundamental boundary, interior and exterior point-sets of any n -simplex in R^n . Additional point-sets are formed generically by embedding the n -simplex and its fundamental point-sets for R^n , within R^{n+1} . Consideration of the possible intersections of these point-sets with the boundary point-set of a second n -simplex then gives the fundamental topological relationships. The relationships are point-set topological relationships because they are derived from the intersection of these fundamental point-sets only. In all of the following discussion, a 1-simplex is called an interval, a 2-simplex is called a face and a 3-simplex is called a volume.

Metric topological spaces are a subset of general topological spaces. An n -simplex in R^n divides R^n into three useful and intuitive point-sets, well known in point-set topology.

- Interior set $^\circ$ of an n -simplex C : a point x is an interior point of C provided there exist an open subset U such that x is an element of U and U is strictly contained within C . The union of all such points is the interior set.
- Boundary set ∂ of an n -simplex C : $\partial C = C - C^\circ$
- Exterior set of an n -simplex C : complement of C

A simple and complete method can be found for finding all topological relationships between closed, connected n -simplexes. In the paper of Pigot (1991), a powerful and fundamental method is used which is based on the set intersection of the boundary, interior and exterior point-sets of an n -simplex n_1 and the boundary, interior and exterior sets of another n -simplex n_2 in R^n . In the paper, the generic term set is used instead of point-set. The theory can now be summarized in five steps as follows:

- 1 Formulate the boundary, interior and exterior sets of an n -simplex n_1 in R^n .
- 2 Derive basic relationships based on all possible set intersections of the boundary set of a second n -simplex n_2 and the interior, boundary and exterior sets of the n -simplex n_1 from step 1.
- 3 Consider the union of the interior, exterior and boundary sets of any n -simplex in R^n as an n -manifold equivalent to R^n with the definition of the open/closed properties of these sets strictly relative to R^n .
- 4 Disconnect R^{n+1} into two new open sets by choosing an embedding of R^n (created in step 3) in R^{n+1} such that the n orthogonal basis vectors of R^n are coincident with n of the $n+1$ orthogonal basis vectors of R^{n+1} .
- 5 Derive additional relationships based on the possible set intersections of the boundary set of an n -simplex n_2 with the boundary, interior and exterior sets of a second n -simplex n_1 , with the boundary set of n_2 intersecting either or both of the two new sets predicted in step 4.

The boundary, interior and exterior sets of a face (or 2-simplex) all in R^2 are shown in Figure 7.

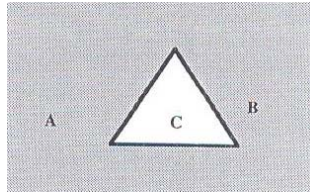


Figure 7: Exterior (A), Boundary (B) and Interior (C) point-sets of a face in R^2

The union of sets A, B and C is a 2-manifold equivalent to R^2 . All possible binary topological relationships between faces in R^2 can then be derived from the possible set relationships between the boundary, interior and exterior sets A, B and C of a_1 and the boundary set X of a_2 . For example, if the boundary set X of a_2 is contained within the interior set C, then the face a_2 will be contained within a_1 . The combinations matrix showing the possible relationships between the boundary of the face a_2 and the exterior, boundary and interior sets A, B and C of a_1 is shown in Table 1.

Table 1: Set intersection relationships between the boundary set of a_2 and the interior, exterior and boundary sets of a_1 in R^2 .

	1	2	3	4	5	6	7
Exterior A	X			X	X		X
Boundary B		X		X	X	X	
Interior C			X	X		X	X

Note that the seventh relationship in the last column of the table is not possible in R^2 because of the restriction to closed, connected faces. The six distinct relationships, their names and spatial interpretations are shown in Figure 8.

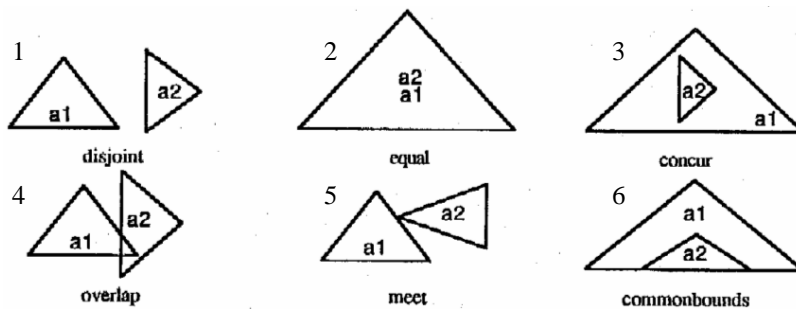


Figure 8: Six possible relationships between faces based on the intersection of the boundary set of face a_2 and the exterior, boundary and interior sets of face a_1 in R^2 .

If we define the open/closed properties of these sets strictly relative to R^2 then these properties and the set relationships in R^2 are preserved when the 2-manifold (equivalent to R^2) formed by their union is embedded in R^3 . If the embedding is chosen such that any two orthogonal basis vectors of R^2 are coincident to two of any three orthogonal basis vectors of R^3 then R^2

disconnects R^3 into two open sets with the third open set corresponding to R^2 itself. The situation is shown in Figure 9.

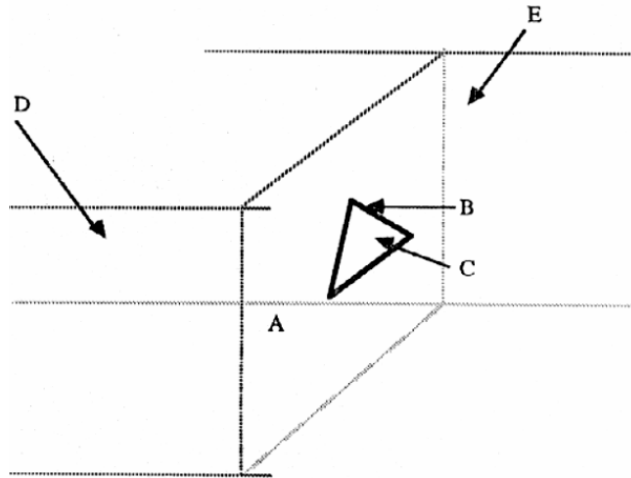


Figure 9: New point sets D & E obtained by embedding the union of the boundary (B), exterior (A) and interior (C) of a face a_1 in R^3 ($A \cup B \cup C = R^2$).

All possible binary topological relationships between faces in R^3 can be derived in the same way as for R^2 , by considering the possible set relationships between boundary set of a face a_2 and the boundary, interior and exterior sets of the face a_1 plus the two new sets D and E , which result from embedding R^2 in R^3 . Since all set relationships derived for R^2 are preserved in R^3 , only the combinations involving the new sets D and E will be considered.

By examination of Figure 9, the set relationships can be divided into two groups. The first group represents the situation where the boundary set X of a_2 is contained within the plane P formed from the union of the interior, exterior and boundary sets of a_1 . This situation corresponds to faces in R^2 and it is considered above. The second group corresponds to the situation where the boundary set X of a_2 intersects either D or E but not both. This corresponds to the spatial situation where a_2 is completely on one side of the plane P formed by the boundary, interior and exterior sets A , B and C of a_1 . In this situation, the boundary set X of a_2 may intersect the plane P and hence the boundary, interior and exterior sets A , B and C or not at all. All combinations are shown in Table 2 (1-8).

Since the topological relationships are the same no matter which set D or E on either side of the plane P the boundary set of a_2 intersects, the combinations are shown in the table with the marker offset between D and E . Note that relationship 7 is not possible between two closed connected simplexes. The other seven relationships are shown spatially in Figure 10 (left).

The third group of relationships occurs when the boundary set X of a_2 intersects both D and E and hence must intersect the sets A , B and C of a_1 at an interval whose boundaries correspond to two points from the boundary set X of a_2 and interior corresponds to the interior set Y of a_2 . The possible combinations between the boundary set X of a_2 and the boundary, interior and exterior sets A , B and C of a_1 when the boundary set X intersects both D and E as well, are shown in Table 2 (9-15).

**Table 2: 1-8: Set intersections between the boundary set X of a_2 and the exterior (A), boundary (B) and interior (C) sets of a_1 in R^3 when a_1 intersects only one of the sets D or E .
 9-15: Set Intersections between the boundary of set X of a_2 and the exterior (A), boundary (B) and interior (C) sets of a_1 in R^3 when the boundary of a_2 intersects both of the sets D and E .**

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Exterior A				X		X	X	X			X		X	X	X
Boundary B			X		X	X		X		X		X	X		X
Interior C		X			X		X	X	X			X		X	X
Above D									X	X	X	X	X	X	X
Below E	X	X	X	X	X	X	X	X		X	X	X	X	X	X

The spatial interpretations are shown in Figure 10 (right). Note that for relationship 10, the interior set of the face a_2 may be used to derive a second possibility. These relationships are marked 10a and 10b in the spatial interpretations of these relationships, shown in Figure 10 (right). In addition, relationship 15 is not possible between closed, connected faces. By examination of all relationships in Figure 8 and Figure 10, the number of unique relationships between faces in R^3 is fourteen since relationships 1, 4 and 11 are particular types of the disjoint relationship shown in Figure 8 and relationships 3, 6 and 13 are particular types of the meet relationship shown in Figure 8.

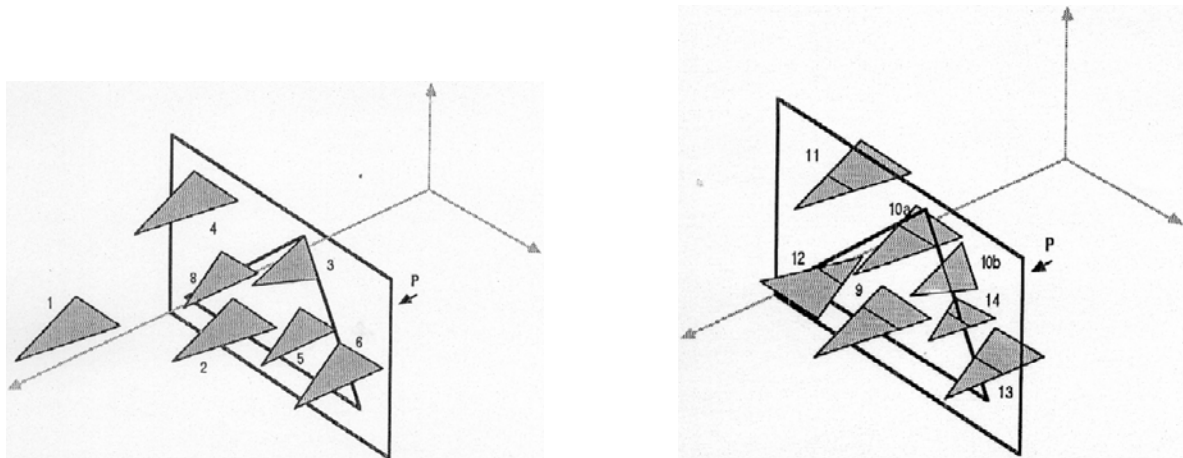


Figure 10: Relationships formed by the intersection of the boundary set X of a face a_2 with the boundary, interior and exterior sets (A , B and C) of a face a_1 when the boundary set of the face intersects the point-set D (or/and E). a_2 is shown shaded, however only the black outline is the boundary set of a_2 .

To reduce these fourteen relationships in detail, the union of the boundary and interior point-sets of a_1 and a_2 in each relationship is considered. Relationships, which are homeomorphic, can then be reduced to their homeomorphs. As a result, the complete two-layer hierarchy of binary topological relationships between faces (2-simplexes) in R^3 is shown in Figure 11. Note that if intersecting faces are not in the same plane, the number of points of intersection is two or less.

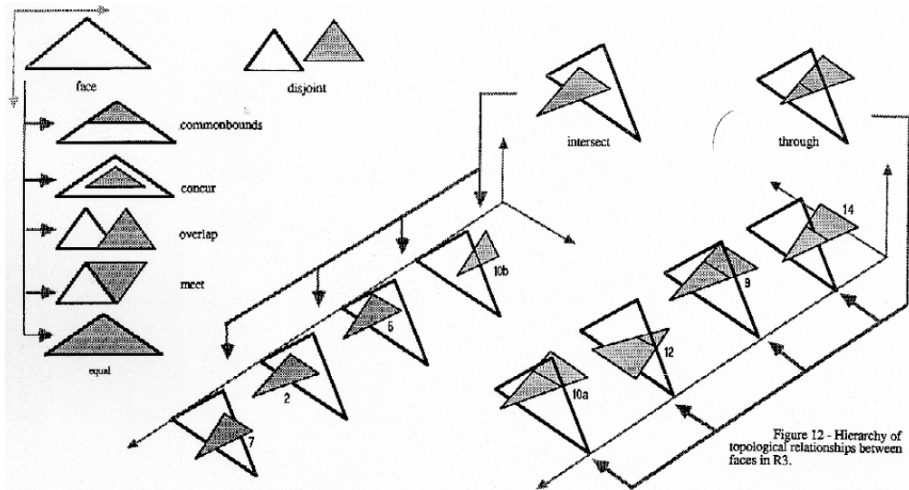


Figure 11: Hierarchy of topological relationships between faces in R^3 .

3.3 Calculating whether triangles intersect

Elements, with illegal meetings between adjacent elements, have to be replaced by elements with legal ones. Aftosmis et al. (1997) has developed a new method for rapid and robust Cartesian mesh generation for component-based geometry. The new algorithm adopts a novel strategy, which first intersects the components to extract the wetted surface before proceeding with volume mesh generation in a second phase. The intersection scheme is based on a robust geometry engine, that uses adaptive precision arithmetic, and which automatically and consistently handles geometric degeneracies with an algorithmic tie-breaking routine. The volume mesh generation takes the intersected surface triangulation as input and generates the mesh through cell division of an initially uniform coarse grid.

Intersection of Generally Positioned Triangles in R^3

With the task of intersecting a particular triangle reduced to an intersection test between that triangle and those on the list of candidates provided by a space subdivision method, the intersection problem is re-cast as a series of triangle-triangle intersection computations. Figure 12 shows a view of two intersecting triangles as a model for discussion. Each intersecting triangle-triangle pair will contribute one segment to the final polyhedra that will comprise the wetted surface of the configuration. The assumption of data in *general* (as opposed to *arbitrary*) position implies that the intersection is always non-degenerate. Triangles may not share vertices, and edges of triangle-triangle pairs do not intersect exactly. Thus, all intersections will be proper.

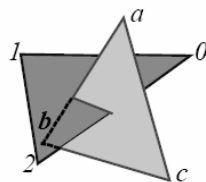


Figure 12: An intersecting pair of generally positioned triangles in three dimensions.

Several approaches exist to compute such intersections but a particularly attractive technique comes in the form of a Boolean test. This predicate can be performed robustly and quickly using only multiplication and addition, thus avoiding the inaccuracy and robustness pitfalls associated with division using fixed width representations of floating point numbers. It is useful to present a rather comprehensive treatment of this intersection primitive because subsequent sections on robustness will return to these relations.

For two triangles that properly intersect in 3D space, the following conditions must exist:

- 1 Two edges of one triangle must cross the plane of the other.
- 2 If condition (1) exists, there must be two edges (of the six available), which pierce within the boundaries of the triangles.

One approach for checking these conditions is to compute directly the pierce points of the edges of one triangle in the plane of the other. Pierce locations from one triangle's edges may then be tested for containment within the boundary of the other triangle. This approach, while conceptually simple, is error prone when implemented using finite precision mathematics. In addition to demanding special effort to trap out zeros, the floating-point division required by this approach may result in numbers not exactly representable by finite width words. This results in a loss of control over precision and may cause serious problems with robustness.

An alternative to this slope-pierce test is to consider a Boolean check based on computation of a triple product without division. A series of such logical checks have the attractive property that they permit to establish the existence and connectivity of the segments without relying on the problematic computation of the pierce locations. The final step of computing the locations of these points may then be relegated to post-processing where they may be grouped together and, since the connectivity is already established, floating point errors will not have fatal consequences.

The Boolean primitive for the 3D intersection of an edge and a triangle is based on the concept of the signed volume of a tetrahedron in R^3 . This signed volume is based on the well-established relationship for the computation of the volume of a simplex, T , in d dimensions in determinate form (see for ex. O'Rourke (1998)). The signed volume $V(T)$ of the simplex T with vertices $(v_0, v_1, v_2, \dots, v_d)$ in d dimensions is:

$$d!V(T_{v_0v_1v_2\dots v_d}) = \det \begin{bmatrix} v_{0_0} & v_{0_1} & \cdots & v_{0_{d-1}} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ v_{d_0} & v_{d_1} & \cdots & v_{d_{d-1}} & 1 \end{bmatrix} \quad (6)$$

where v_{k_j} denotes the j^{th} coordinate of the k^{th} vertex with $j, k \in \{0, 1, 2, \dots, d\}$. In 3 dimensions, Equation (7) gives six times the signed volume of the tetrahedron T_{abcd} .

$$6V(T_{abcd}) = \begin{vmatrix} a_0 & a_1 & a_2 & 1 \\ b_0 & b_1 & b_2 & 1 \\ c_0 & c_1 & c_2 & 1 \\ d_0 & d_1 & d_2 & 1 \end{vmatrix} = \begin{vmatrix} a_0 - d_0 & a_1 - d_1 & a_2 - d_2 \\ b_0 - d_0 & b_1 - d_1 & b_2 - d_2 \\ c_0 - d_0 & c_1 - d_1 & c_2 - d_2 \end{vmatrix} \quad (7)$$

This volume serves as the fundamental building block of the geometry routines. It is positive when (a, b, c) forms a counterclockwise circuit when viewed from an observation point located

on the side of the plane defined by (a,b,c) which is opposite from d . Positive and negative volumes define the two states of the Boolean test while zero indicates that the four vertices are exactly coplanar. If the vertices are indeed coplanar, then the situation constitutes a “tie”. In applying this logical test to edge ab and triangle $(0,1,2)$ in Figure 12, ab crosses the plane if and only if (iff) the signed volumes T_{012a} and T_{012b} have opposite signs. Figure 13 presents a graphical look at the application of this test.

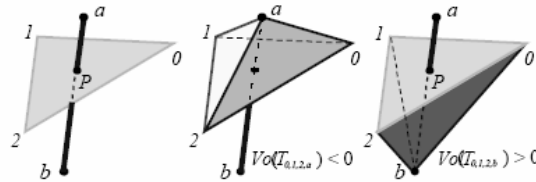


Figure 13: Boolean test to check if edge ab crosses the plane defined by triangle $(0,1,2)$ through computation of two signed volumes.

With a and b established on opposite sides of the plane $(0,1,2)$, all that remains is to determine if ab pierces within the boundary of the triangle. This will be the case only if the three tetrahedra formed by connecting the end points of ab with the three vertices of the triangle $(0,1,2)$ (taken two at a time) all have the same sign, that is:

$$\begin{aligned} V(T_{a12b}) < 0 \wedge V(T_{a01b}) < 0 \wedge V(T_{a20b}) < 0 \quad \text{or} \\ V(T_{a12b}) > 0 \wedge V(T_{a01b}) > 0 \wedge V(T_{a20b}) > 0 \end{aligned} \tag{8}$$

Figure 14 illustrates this test for the case where the three volumes are all positive.

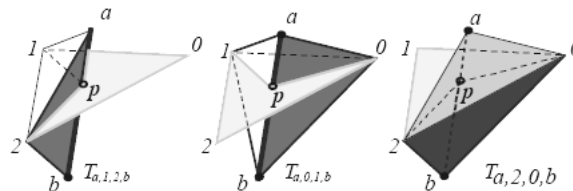


Figure 14: Boolean test for pierce of a line segment ab within the boundary of a triangle $(0,1,2)$.

After determining the existence of all the segments, which result from intersections between tri-tri pairs, and connecting a linked list of all such segments to the triangles that intersect to produce them, all that remains is to compute actually the locations of the pierce points. This is accomplished by using a parametric representation of each intersected triangle and the edge, which pierces it. The technique is a straightforward three dimensional generalization of the 2D method presented in “Computational Geometry in C” (O’Rourke, 1994).

3.4 Calculating where triangles intersect

For determining where triangles intersect, first some general and theoretical algorithms are needed. They are described in this paragraph. Later on, the corresponding code will be treated in Chapter 5.

Segment-segment intersection

Let two endpoints a and b form a segment on the line L_{ab} and let two endpoints c and d form a segment on the line L_{cd} . A common method of computing the point of intersections to solve slope-intercept equations for L_{ab} and L_{cd} simultaneously: two equations in two unknowns (the x and y coordinates of the point of intersection). Instead, a parametric representation of the two segments is used, as the meaning of the variables seems more intuitive.

Let $A = b - a$ and $C = d - c$; these vectors point along the segments. Any point on the line L_{ab} can be represented as the vector sum $p(s) = a + sA$, which takes us to a point a on L_{ab} , and then moves some distance along the line by scaling A by s . See Figure 15. The variable s is called the *parameter* of this equation. Consider the values obtained for $s = 0$, $s = 1$, and $s = \frac{1}{2}$; $p(0) = a$, $p(1) = a + A = a + b - a = b$, and $p(\frac{1}{2}) = (a + b)/2$. These examples demonstrate that $p(s)$ for $s \in [0, 1]$ represents all the points on the segment ab , with the value of s representing the fraction of the distance between the endpoints; in particular, the extremes of s yield the endpoints.

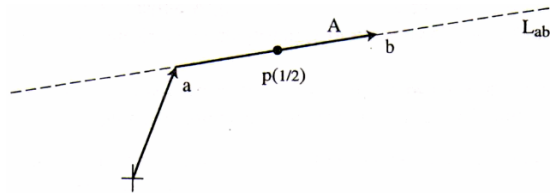


Figure 15: $p(s)$ is shown.

We can similarly represent the points on the second segment by $q(t) = c + tC$, $t \in [0, 1]$. A point of intersection between the segments is then specified by values of s and t that make $p(s)$ equal to $q(t)$: $a + sA = c + tC$. This vector equation also comprises two equations in two unknowns: the x and y equations, both with s and t as unknowns. With the usual convention of subscripts 0 and 1 indicating x and y coordinates, its solution is

$$\begin{aligned} s &= [a_0(d_1 - c_1) + c_0(a_1 - d_1) + d_0(c_1 - a_1)] / D \\ t &= [a_0(c_1 - b_1) + b_0(a_1 - c_1) + c_0(b_1 - a_1)] / D \\ D &= a_0(d_1 - c_1) + b_0(c_1 - d_1) + c_0(a_1 - b_1) + d_0(b_1 - a_1) \end{aligned} \quad (9)$$

Division by zero is a possibility in these equations. The denominator D happens to be zero if and only if the two lines are parallel. Some parallel segments involve intersection, and some do not.

Segment-triangle intersection

The computation of the point of intersection between a segment and a triangle in three dimensions is more difficult, but still ultimately straightforward. In fact, this is one of the most prevalent geometric computations performed today, because it is a key step in “ray tracing” used in computer graphics. A parametric representation will be used again. Let $T = \Delta abc$ be the triangle and qr the segment, where q is viewed as the originating (“query”) endpoint in case qr represents a ray and r is the “ray” endpoint. Throughout the calculation, it will be assumed that $r \neq q$, so the input segment has nonzero length.

The first step is to determine whether qr intersects the plane π containing T . This halfway goal will be pursued throughout this subsection, before turning to determining whether the point of intersection lies in the triangle. Recall that just as all points on a line

must satisfy a linear equation in x and y , so too must all the points on a plane satisfy an equation

$$\pi : Ax + By + Cz = D \quad (10)$$

The plane will be represented by these four coefficients. It will help to view the first three coefficients as a vector (A, B, C) , for then, the plane equation can be viewed as a dot product:

$$\pi : (x, y, z) \cdot (A, B, C) = D \quad (11)$$

The geometric meaning of this equation is that every point (x, y, z) on the plane projects to the same length onto (A, B, C) . From this and Figure 16 it should be clear that $N = (A, B, C)$ is a vector normal (i.e. perpendicular) to the plane. If this vector is unit length, then D is the distance from the origin to π .

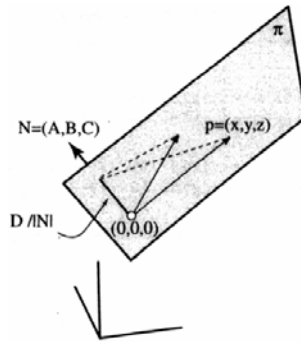


Figure 16: The dot product of a point on the plane with N is a constant, D when $|N|=1$.

Just as in two dimensions, any point $p(t)$ on the segment can be represented by moving out to the q endpoint, and then adding a scaled version of a vector along the segment: $p(t) = q + t(r - q)$. Let us temporarily assume that $q = (0, 0, 0)$ so that $p(t) = tr$; this will make the calculations more transparent. Now we are seeking a value of the parameter t that will stretch r out to the plane. Because every point on the plane must satisfy Equation (11), we must have

$$\begin{aligned} p(t) \cdot (A, B, C) &= D \\ tr \cdot N &= D \\ t(r \cdot N) &= D \\ t &= \frac{D}{r \cdot N} \end{aligned} \quad (12)$$

Generalizing Equation (12) for arbitrary q is not difficult:

$$\begin{aligned} [q + t(r - q)] \cdot N &= D \\ q \cdot N + t(r - q) \cdot N &= D \\ t(r - q) \cdot N &= D - q \cdot N \\ t &= \frac{D - q \cdot N}{(r - q) \cdot N} \end{aligned} \quad (13)$$

You may wonder why this more complex situation yields an equation in only one unknown, whereas the simpler segment-segment intersection led to two equations in two unknown parameters. The reason is that the intersection is not pinpointed with respect to the triangle yet; that will involve other unknowns. This same strategy could be followed in two dimensions, but the situation was simple enough to permit jumping right to simultaneous determination of the parameters.

One more step remains: obtaining the plane coefficients. In this case, we usually start with three points determining a triangle in space. The coefficients may be found using the observation above that three of them define a vector normal to the triangle. The cross product (see Equation (1)) can be used to determine N . With N in hand, the fourth coefficient D can be found by substituting any point on the plane into Equation (10). With $A = b - a$ and $B = c - a$ N will be:

$$N = \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \end{vmatrix} = (A_1B_2 - A_2B_1)\hat{i} + (A_2B_0 - A_0B_2)\hat{j} + (A_0B_1 - A_1B_0)\hat{k} \quad (14)$$

$$N = \begin{bmatrix} A_1B_2 - A_2B_1 \\ A_2B_0 - A_0B_2 \\ A_0B_1 - A_1B_0 \end{bmatrix} = \begin{bmatrix} (b_1 - a_1)(c_2 - a_2) - (b_2 - a_2)(c_1 - a_1) \\ (b_2 - a_2)(c_0 - a_0) - (b_0 - a_0)(c_2 - a_2) \\ (b_0 - a_0)(c_1 - a_1) - (b_1 - a_1)(c_0 - a_0) \end{bmatrix}$$

Projection to two dimensions

The situation is that there are points of intersection known and they lay on the plane π , containing triangle T , so the problem is fundamentally two-dimensional, not three-dimensional. However, it would take a bit of work to translate and rotate π so that it coincides with, say the xy -plane. One observation solves the problem: Projecting out the coordinate corresponding to the largest component of the vector N normal to π guarantees nondegeneracy. Thus, a nearly horizontal plane has a large z -component, and the projection to the xy -plane is called for (see Figure 17). From now on, it is possible to adapt two-dimensional solutions to solve three-dimensional plane problems.

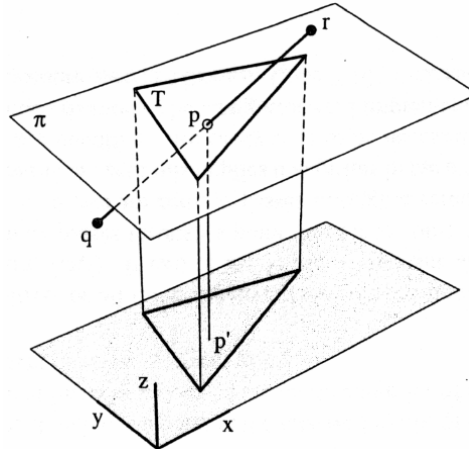


Figure 17: Projection to two dimensions.

4. Present solutions

In the line of creating triangulations and tetrahedralizations two applications are developed already. “Triangle” generates exact triangulations and “TetGen” generates exact tetrahedralizations. The use of these programs is explained in this chapter, starting with “Triangle” (§ 4.1) and after that “TetGen” (§ 4.2). Other solutions are not discussed in this report.

4.1 “Triangle”

“Triangle” generates exact Delaunay triangulations, constrained Delaunay triangulations, Voronoi diagrams, and quality conforming Delaunay triangulations. The latter can be generated with no small angles, and are thus suitable for finite element analysis.

Command Line Switches

Below is the command line syntax of “Triangle” with a list of short descriptions. Underscores indicate that numbers may optionally follow certain switches. If no command line switches are specified, your .node input file is read, and the Delaunay triangulation is returned in .node and .ele output files. Do not leave any space between a switch and its numeric parameter. If the `-p` switch is used ‘input_file’ must be a file with extension .poly. If `-r` is used, you must supply .node and .ele files, and possibly a .poly file and an .area file as well.

```
triangle [-prq__a__uAcjevngBPNEIOXzo_YS__LiFlsCQVh] input_file
```

- `-p` Reads a Planar Straight Line Graph (.poly file).
- `-r` Refines a previously generated mesh.
- `-q` Quality mesh generation by a variant of Jim Ruppert's Delaunay refinement algorithm.
- `-a` Imposes a maximum triangle area.
- `-u` Imposes a user-defined constraint on triangle size.
- `-A` Assigns an additional attribute to each triangle that identifies what segment-bounded region each triangle belongs to.
- `-c` Creates segments on the convex hull of the triangulation.
- `-j` Jettisons vertices that are not part of the final triangulation from the output .node file.
- `-e` Outputs (to an .edge file) a list of edges of the triangulation.
- `-v` Outputs the Voronoi diagram associated with the triangulation.
- `-n` Outputs (to a .neigh file) a list of triangles neighbouring each triangle.
- `-g` Outputs the mesh to an Object File Format (.off) file, suitable for viewing with the Geometry Center's Geomview package.
- `-B` No boundary markers in the output .node, .poly, and .edge output files.
- `-P` No output .poly file.
- `-N` No output .node file.
- `-E` No output .ele file.
- `-I` No iteration numbers.
- `-O` No holes, ignores the holes in the .poly file.
- `-X` No exact arithmetic.
- `-o2` Generates second-order sub parametric elements with six nodes each.
- `-Y` No new vertices on the boundary.
- `-S` Specifies the maximum number of Steiner points (vertices that are not in the input, but are added to meet the constraints on minimum angle and maximum area).

- L Do not use diametral lenses to determine whether sub segments are encroached; use diametral circles instead (as in Ruppert's algorithm).
- i Uses an incremental rather than divide-and-conquer algorithm to form a Delaunay triangulation.
- F Uses Steven Fortune's sweepline algorithm to form a Delaunay triangulation.
- l Uses only vertical cuts in the divide-and-conquer algorithm.
- s Specifies that segments should be forced into the triangulation by recursively splitting them at their midpoints, rather than by generating a constrained Delaunay triangulation.
- C Check the consistency of the final mesh.
- Q Quiet: Suppresses all explanation of what "Triangle" is doing, unless an error occurs.
- V Verbose: Gives detailed information about what "Triangle" is doing.
- h Help: Displays these instructions.

File Formats

All files may contain comments prefixed by the character '#'. Vertices, triangles, edges, holes, and maximum area constraints must be numbered consecutively, starting from either 1 or 0. Whichever you choose, all input files must be consistent; if the vertices are numbered from 1, so must be all other objects. "Triangle" automatically detects your choice while reading the .node (or .poly) file. When calling "Triangle" from another program, use the -z switch if you wish to number objects from zero.

Examples of How to Use "Triangle"

``triangle dots'` reads vertices from dots.node, and writes their Delaunay triangulation to dots.1.node and dots.1.ele. (dots.1.node is identical to dots.node.) ``triangle -I dots'` writes the triangulation to dots.ele instead. (No additional .node file is needed, so none is written.)

``triangle -pe object.1'` reads a PSLG from object.1.poly (and possibly object.1.node, if the vertices are omitted from object.1.poly) and writes its constrained Delaunay triangulation to object.2.node and object.2.ele. The segments are copied to object.2.poly, and all edges are written to object.2.edge.

``triangle -pq31.5a.1 object'` reads a PSLG from object.poly (and possibly object.node), generates a mesh whose angles are all 31.5 degrees or greater and whose triangles all have areas of 0.1 or less, and writes the mesh to object.1.node and object.1.ele. Each segment may be broken up into multiple sub segments; these are written to object.1.poly.

4.2 "TetGen"

"TetGen" generates exact Delaunay tetrahedralizations, exact constrained Delaunay tetrahedralizations, and quality tetrahedral meshes. The latter are nicely graded and whose tetrahedra have radius-edge ratio bounded, thus are suitable for finite element and finite volume analysis.

Command Line Syntax

Below is the command line syntax of "TetGen" with a list of short descriptions. Underscores indicate that numbers may optionally follow certain switches. Do not leave any space between a switch and its numeric parameter. 'input_file' contains input data depending on the switches you supplied which may be a piecewise linear complex or a list of nodes. File formats and detailed description of command line switches can be found in user's manual.

```
tetgen [-pq_a__AriMS__T__dzo_fengGOBNEFICQVvh] input_file
```

- p Tetrahedralizes a piecewise linear complex.
- q Quality mesh generation. A minimum radius-edge ratio may be specified (default 2.0).
- a Applies a maximum tetrahedron volume constraint.
- A Assigns attributes to identify tetrahedra in certain regions.
- r Reconstructs/Refines a previously generated mesh.
- i Inserts a list of additional points into mesh.
- M Does not merge coplanar facets.
- S Specifies maximum number of added Steiner points.
- T Set a tolerance for coplanar test (default 1e-8).
- d Detect intersections of PLC facets.
- z Numbers all output items starting from zero.
- j Jettison unused vertices from output .node file.
- o2 Generates second-order sub parametric elements.
- f Outputs faces (including non-boundary faces) to .face file.
- e Outputs sub segments to .edge file.
- n Outputs tetrahedra neighbours to .neigh file.
- g Outputs mesh to .mesh file for viewing by Medit.
- G Outputs mesh to .msh file for viewing by Gid.
- O Outputs mesh to .off file for viewing by Geomview.
- B Suppresses output of boundary information.
- N Suppresses output of .node file.
- E Suppresses output of .ele file.
- F Suppresses output of .face file.
- I Suppresses mesh iteration numbers.
- C Checks the consistency of the final mesh.
- Q Quiet: No terminal output except errors.
- V Verbose: Detailed information, more terminal output.
- v Prints the version information.
- h Help: A brief instruction for using “TetGen”.

Examples of How to Use “TetGen”

'tetgen object' reads vertices from object.node, and writes their Delaunay tetrahedralization to object.1.node and object.1.ele.

'tetgen -p object' reads a PLC from object.poly or object.smesh (and possibly object.node) and writes its constrained Delaunay tetrahedralization to object.1.node, object.1.ele and object.1.face.

'tetgen -pq1.414a.1 object' reads a PLC from object.poly or object.smesh (and possibly object.node), generates a mesh whose tetrahedra have radius-edge ratio smaller than 1.414 and have volume of 0.1 or less, and writes the mesh to object.1.node, object.1.ele and object.1.face.

5. “Intersect”

In this chapter, the functions of the new developed application “Intersect” are explained. The chapter is divided in three parts. The first paragraph is about the purpose of “Intersect”, the second about the code it selves and the third about what could be done in the future.

5.1 The purpose of “Intersect”

The aim of this research is to conduct a solution to detect invalid triangles, to make them valid. When a triangulation is not valid, with the command 'tetgen -d object' “TetGen” will write the intersecting and duplicated triangles to separate files. The nodes go to 'object.1.node', the faces to 'object.1.face' and the input file to 'object.1.smesh'. After that, “TetGen” is finished, but no tetrahedralization is formed. This function has the objective to open 'object.ply', to derive the points of intersection, to add them to the nodes of the intersecting faces and to get a new set of faces, which are not intersecting anymore. After that, the new faces will replace the old intersecting faces. These last steps are not in the code yet, that is an assignment for the future (see Paragraph 5.3).

Command Line Syntax

Below is the command line syntax of “Intersect” with a list of short descriptions. Do not leave any space between a switch and its numeric parameter. ‘input_file’ contains input data depending on the switches you supplied which may be a piecewise linear complex. File formats and detailed description of command line switches are found in user's manual of “TetGen”.

```
intersect [-dvh?] input_file
-d Detect intersections of PLC facets and put the points of intersection into
  'object_intersect.node'.
-v Print the version information.
-h Help: A brief instruction for using “Intersect”.
```

5.2 The code itself

The new code can be divided in three parts. The task of the first part is to detect the points of intersection of two adjacent triangles with illegal meetings. The second part is to get an output-file with the coordinates of the points of intersection and some more information. The last and most important part defines the functions to derive the points of intersection. This part is programmed like the functions explained in Paragraph 3.4. Its code is displayed in Appendix A.

The code to determine the points of intersection is constructed from several steps. Starting with the question whether two triangles intersect or not and ending with the coordinates of the points of intersection (see for details Appendix B).

- Detect whether triangle ABC and OPQ are intersecting.
 - If true, for line OP, PQ and OQ, detect whether the line intersects triangle ABC.

- If true, derive the point of intersection, if the line lies in the plane containing ABC, the coordinates are (99, 99, 99).
- If found, locate the point with respect to both triangles.
- If true, for line AB, BC and CA, detect whether the line intersects triangle ABC.
 - If true, derive the point of intersection, if the line lies in the plane containing OPQ, the coordinates are (99, 99, 99).
 - If found, locate the point with respect to both triangles.
- Make an overview of all points of intersection found.
- Print all data to a file.
- Go to the next triangle, and repeat the steps above.

The points of intersection have to be reproduced, a .node-file is created by the code as part of the function `detectinterfaces()` (see for details Appendix C). Not only the points of intersection, but also some extra information is saved. Every point of intersection contains six attributes:

- the number of the first triangle (ABC);
- a pointer to the biggest component of the normal vector of that triangle (0 = x-axis, 1 = y-axis, 2 = z-axis);
- an indication for the location of the point of intersection with respect to the face (1 = in A, 2 = in B, 3 = in C, 12 = on AB, 23 = on BC, 31 = on CA, 0 = in ABC);
- the number of the second triangle (OPQ);
- a pointer to the biggest component of the normal vector of that triangle (0 = x-axis, 1 = y-axis, 2 = z-axis);
- an indication for the location of the point of intersection with respect to the face (1 = in O, 2 = in P, 3 = in Q, 12 = on OP, 23 = on PQ, 31 = on QO, 0 = in OPQ);

5.3 Assignment for the future

As said before, there is more work to do. At this moment, it is possible to derive the points of intersection, but the next step to come to an object without illegal meetings of adjacent triangles is missing and the algorithm is not very efficient. With the use of “Triangle” it is possible to derive new triangles, but to do so some edits are needed, also the results of “Triangle” have to be implemented. Thus, the assignment for the future can be split up in four parts:

- 1 Create for every face that is intersected, a .poly file, which contains the original coordinates, the points of intersection, and a list of constraint-lines. That constraint-lines include the lines of intersection of two adjacent triangles and the convex hull of the intersected triangle, which includes the points of intersection that are on the edges of the original triangle.
- 2 Write an algorithm to get the new triangulations from “Triangle”, for every face that is intersected.
- 3 Translate the numbers of the points of intersection to numbers of points that fit in the original .ply file, do the same for all triangles, replace the intersected ones by the new.
- 4 Refine the algorithm of “Intersect”.

Furthermore, it is possible to implement the code of “Intersect” into the code of “TetGen”, this is also part of the refinement process.

6. A worked out example

In order to explain the use and the working of “Intersect”, two examples are developed. Every paragraph in this chapter describes just one stage in the process to come from an illegal triangulation to a legal tetrahedralization, the last paragraph (§ 6.8) is about the results.

6.1 Two examples

In Figure 18 and Figure 19 the two examples are shown. As can be seen, both examples are contain illegal meetings between faces.

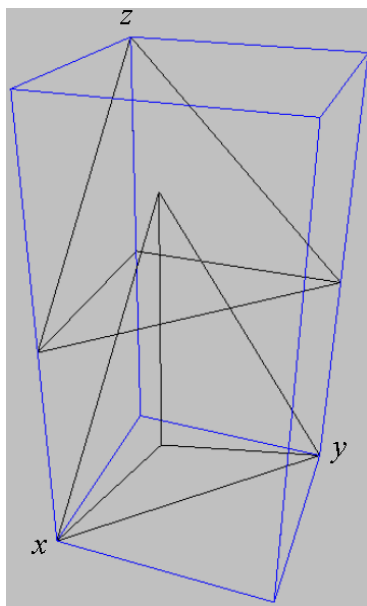


Figure 18: Example 1

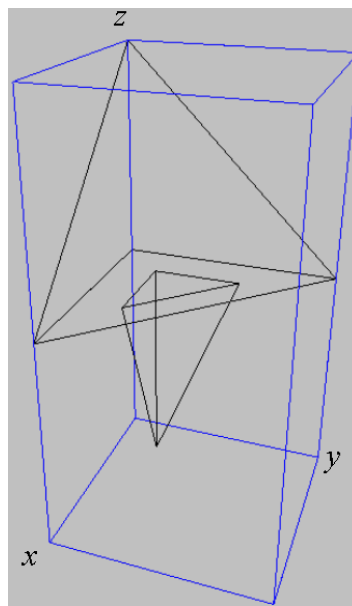


Figure 19: Example 2

Example 1

nodes

```
0 5 0 5
1 0 5 5
2 0 0 5
3 0 0 10
4 5 0 0
5 0 5 0
6 1 1 0
7 1 1 7
```

Example 1

faces

```
0 1 2
0 1 3
0 2 3
1 2 3
4 5 6
4 5 7
4 6 7
5 6 7
```

Example 2

nodes

```
0 5 0 5
1 0 5 5
2 0 0 5
3 0 0 10
4 3 1 5
5 1 3 5
6 1 1 5
7 1 1 0
```

Example 2

faces

```
0 1 2
0 1 3
0 2 3
1 2 3
4 5 7
4 6 7
5 6 7
```

6.2 Step 1: “TetGen”, a check

Because, it can be seen there are intersecting triangles, “TetGen” has a problem with the files. With the command: ‘tetgen example*.ply’, the examples are tested with “TetGen”. With the command ‘tetgen -d example*.ply’ more details can be printed on the screen. The screendumps are displayed in Table 3.

Table 3: Screendumps of ‘tetgen -d example*.ply’.

<pre>Opening example1.ply. Constructing Delaunay tetrahedrization. Creating surface mesh. Detecting intersecting facets. Facet #1 intersects facet #6 at triangles: (0, 1, 2) and (4, 5, 7) Facet #1 intersects facet #7 at triangles: (0, 1, 2) and (4, 6, 7) Facet #1 intersects facet #8 at triangles: (0, 1, 2) and (5, 6, 7)</pre>	<pre>Opening example2.ply. Constructing Delaunay tetrahedrization. Creating surface mesh. Detecting intersecting facets. Facet #1 intersects facet #5 at triangles: (0, 1, 2) and (4, 5, 7) Facet #1 intersects facet #6 at triangles: (0, 1, 2) and (4, 6, 7) Facet #1 intersects facet #7 at triangles: (0, 1, 2) and (5, 6, 7)</pre>
---	---

<pre>!! Found 3 pairs of faces are intersecting. Intersection seconds: 0 Writing example1.1.node. Writing example1.1.face. Writing example1.1.smesh. Output seconds: 0 Total running seconds: 0 Statistics: Input points: 8 Input facets: 8 Input holes: 0 Input regions: 0 Mesh points: 8 Mesh tetrahedra: 9 Mesh faces: 23 Mesh subfaces: 4 Mesh subsegments: 12</pre>	<pre>!! Found 3 pairs of faces are intersecting. Intersection seconds: 0 Writing example2.1.node. Writing example2.1.face. Writing example2.1.smesh. Output seconds: 0 Total running seconds: 0 Statistics: Input points: 8 Input facets: 7 Input holes: 0 Input regions: 0 Mesh points: 8 Mesh tetrahedra: 14 Mesh faces: 31 Mesh subfaces: 4 Mesh subsegments: 12</pre>
--	---

Now it is for sure known some meetings between two adjacent triangles are illegal.

6.3 Step 2: “Intersect”

“Intersect” has to be used to derive the coordinates of the points of intersecting, “Intersect” is based on “TetGen”, therefore, with the command: ‘intersect -d example*.ply’, the points of intersection are determined. The screendumps are displayed in Table 4.

Table 4: Screendumps of 'intersect -d example*.ply'.

<pre>Opening example1.ply. Constructing Delaunay tetrahedrization. Creating surface mesh. Detecting intersecting facets. Facet #1 intersects facet #6 at triangles: (0, 1, 2) and (4, 5, 7) 2 Point of Intersection is: (0.714286, 2.142857, 5.000000). 3 Point of Intersection is: (2.142857, 0.714286, 5.000000). 0 0.714286 2.142857 5.000000 1 2 0 6 0 23 1 2.142857 0.714286 5.000000 1 2 0 6 0 31 Facet #1 intersects facet #7 at triangles: (0, 1, 2) and (4, 6, 7) 2 Point of Intersection is: (1.000000, 1.000000, 5.000000). 3 Point of Intersection is: (2.142857, 0.714286, 5.000000). 2 1.000000 1.000000 5.000000 1 2 0 7 1 23 3 2.142857 0.714286 5.000000 1 2 0 7 1 31 Facet #1 intersects facet #8 at triangles: (0, 1, 2) and (5, 6, 7) 2 Point of Intersection is: (1.000000, 1.000000, 5.000000). 3 Point of Intersection is: (0.714286, 2.142857, 5.000000). 4 1.000000 1.000000 5.000000 1 2 0 8 0 23 5 0.714286 2.142857 5.000000 1 2 0 8 0 31 !! Found 3 pairs of faces are intersecting. !! Found 6 points of intersection.</pre>	<pre>Opening example2.ply. Constructing Delaunay tetrahedrization. Creating surface mesh. Detecting intersecting facets. Facet #1 intersects facet #5 at triangles: (0, 1, 2) and (4, 5, 7) 1 Point of Intersection is: (99.000000, 99.000000, 99.000000). 2 Point of Intersection is: (1.000000, 3.000000, 5.000000). 3 Point of Intersection is: (3.000000, 1.000000, 5.000000). 0 1.000000 3.000000 5.000000 1 2 0 5 0 2 1 3.000000 1.000000 5.000000 1 2 0 5 0 1 Facet #1 intersects facet #6 at triangles: (0, 1, 2) and (4, 6, 7) 1 Point of Intersection is: (99.000000, 99.000000, 99.000000). 2 Point of Intersection is: (1.000000, 1.000000, 5.000000). 3 Point of Intersection is: (3.000000, 1.000000, 5.000000). 2 1.000000 1.000000 5.000000 1 2 0 6 1 2 3 3.000000 1.000000 5.000000 1 2 0 6 1 1 Facet #1 intersects facet #7 at triangles: (0, 1, 2) and (5, 6, 7) 1 Point of Intersection is: (99.000000, 99.000000, 99.000000). 2 Point of Intersection is: (1.000000, 1.000000, 5.000000). 3 Point of Intersection is: (1.000000, 3.000000, 5.000000).</pre>
--	--

<pre> Intersection seconds: 0 Writing example1.1.node. Writing example1.1.face. Writing example1.1.smesh. Output seconds: 0.015 Total running seconds: 0.015 Statistics: Input points: 8 Input facets: 8 Input holes: 0 Input regions: 0 Mesh points: 8 Mesh tetrahedra: 9 Mesh faces: 23 Mesh subfaces: 4 Mesh subsegments: 12 </pre>	<pre> 4 1.000000 1.000000 5.000000 1 2 0 7 0 2 5 1.000000 3.000000 5.000000 1 2 0 7 0 1 !! Found 3 pairs of faces are intersecting. !! Found 6 points of intersection. Intersection seconds: 0 Writing example2.1.node. Writing example2.1.face. Writing example2.1.smesh. Output seconds: 0 Total running seconds: 0 Statistics: Input points: 8 Input facets: 7 Input holes: 0 Input regions: 0 Mesh points: 8 Mesh tetrahedra: 14 Mesh faces: 31 Mesh subfaces: 4 Mesh subsegments: 12 </pre>
---	---

The bold lines in Table 4 are printed to the output files. The output files ‘example*_intersect.node’ are displayed in Table 5.

Table 5: Content of output-files ‘example*_intersect.node’.

<pre> 6 3 6 0 0 0.714286 2.142857 5.000000 1 2 0 6 0 23 1 2.142857 0.714286 5.000000 1 2 0 6 0 31 2 1.000000 1.000000 5.000000 1 2 0 7 1 23 3 2.142857 0.714286 5.000000 1 2 0 7 1 31 4 1.000000 1.000000 5.000000 1 2 0 8 0 23 5 0.714286 2.142857 5.000000 1 2 0 8 0 31 # Generated by intersect -d example1.ply </pre>	<pre> 6 3 6 0 0 1.000000 3.000000 5.000000 1 2 0 5 0 2 1 3.000000 1.000000 5.000000 1 2 0 5 0 1 2 1.000000 1.000000 5.000000 1 2 0 6 1 2 3 3.000000 1.000000 5.000000 1 2 0 6 1 1 4 1.000000 1.000000 5.000000 1 2 0 7 0 2 5 1.000000 3.000000 5.000000 1 2 0 7 0 1 # Generated by intersect -d example2.ply </pre>
---	---

As can be noticed by taking the last attributes of the points of intersection of example 2, all points of intersection are on the nodes of the original object, the third attribute shows only face 1 is really intersected.

6.4 Step 3: Manual edits – part 1

The intersected triangles have to be triangulated again, this time with the points and segments of intersection. In order to derive that new triangulation, for every triangle a separate .poly file has to be created. So, to derive the triangulations of the examples in the chapter, for every example four .poly files are needed, note that you have to remove duplicated points. Because “Triangle” deals with only two-dimensional problems, the faces have to be projected to two dimensions (see Paragraph 3.4, last item). When all points of intersection are on the nodes of the triangle, the triangle is not really intersected; therefore, it is not needed to triangulate them. The resulting files are shown in Table 6.

Table 6: Content of 'example*_*.poly'.

<pre> example1_1.poly 6 2 0 0 # points of intersection (without z) 0 0.714286 2.142857 1 2.142857 0.714286 2 1.000000 1.000000 # coordinates of original triangle 3 5.0 0.0 4 0.0 5.0 5 0.0 0.0 # segments of intersection 6 0 0 0 1 1 2 1 2 2 0 3 3 4 4 4 5 5 5 3 0 0 </pre>	<pre> example2_1.poly 6 2 0 0 # points of intersection (without z) 0 1.000000 3.000000 1 3.000000 1.000000 2 1.000000 1.000000 # coordinates of original triangle 3 5.0 0.0 4 0.0 5.0 5 0.0 0.0 # segments of intersection 6 0 0 0 1 1 2 1 2 2 0 3 3 4 4 4 5 5 5 3 0 0 </pre>
<pre> example1_6.poly 5 2 0 0 # points of intersection (without x) 0 2.142857 5.000000 1 0.714286 5.000000 # coordinates of original triangle 2 0.0 0.0 3 5.0 0.0 4 1.0 7.0 # segments of intersection 6 0 0 0 1 1 2 3 2 3 0 3 0 4 4 4 1 5 1 2 0 0 </pre>	<pre> example1_7.poly 5 2 0 0 # points of intersection (without y) 0 1.000000 5.000000 1 2.142857 5.000000 # coordinates of original triangle 2 5.0 0.0 3 1.0 0.0 4 1.0 7.0 # segments of intersection 6 0 0 0 1 1 2 3 2 3 0 3 0 4 4 4 1 5 1 2 0 0 </pre>
<pre> example1_8.poly 5 2 0 0 # points of intersection (without x) 0 1.000000 5.000000 1 2.142857 5.000000 # coordinates of original triangle 2 5.0 0.0 3 1.0 0.0 4 1.0 7.0 # segments of intersection 6 0 0 0 1 1 2 3 2 3 0 3 0 4 4 4 1 5 1 2 0 0 </pre>	<pre> example2_5.poly -> not needed example2_6.poly -> not needed example2_7.poly -> not needed </pre>

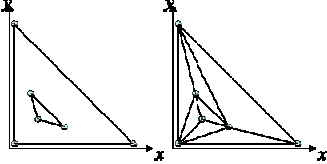
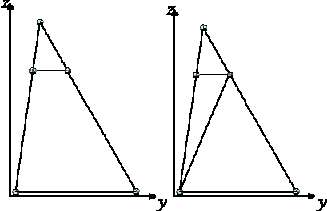
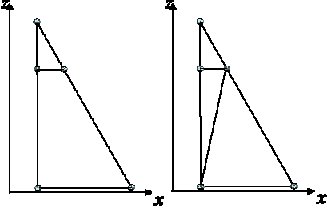
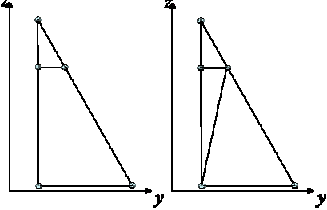
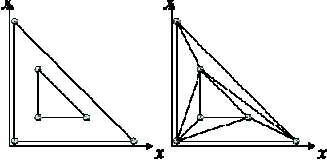
6.5 Step 4: “Triangle”

The program “Triangle” can be used to triangulate the intersected triangles and its points of intersection face by face, this is done with the command ‘triangle -c example*_*.poly’. The results can be interpreted graphically too. The output-files and the graphical interpretation are displayed in Table 7.

The use of this application results in three different files:

- example*_*.*.node: list of nodes used in the triangulation.
- example*_*.*.ele: list of faces (triangles) produced by “Triangle”.
- example*_*.*.poly: not needed in this research

Table 7: Results of “Triangle”. '*.*.node' (left), '*.*.ele' (middle), and graphical interpretation (right).

<pre>6 2 0 1 0 0.71428599999999998 2.1428569999999998 0 1 2.14285699999999998 0.71428599999999998 0 2 1 1 0 3 5 0 1 4 0 5 1 5 0 0 1 # Generated by triangle -p example1_1</pre>	<pre>7 3 0 0 5 0 4 1 0 5 2 2 4 0 1 3 5 1 2 4 1 5 3 5 1 3 4 6 2 1 0</pre>	
<pre>5 2 0 1 0 2.14285699999999998 5 1 1 0.71428599999999998 5 1 2 0 0 1 3 5 0 1 4 1 7 1 # Generated by triangle -p example1_6</pre>	<pre>3 3 0 0 1 2 0 1 0 4 1 2 3 0 2</pre>	
<pre>5 2 0 1 0 1 5 1 1 2.14285699999999998 5 1 2 5 0 1 3 1 0 1 4 1 7 1 # Generated by triangle -p example1_7</pre>	<pre>3 3 0 0 0 3 1 1 1 4 0 2 2 1 3</pre>	
<pre>5 2 0 1 0 1 5 1 1 2.14285699999999998 5 1 2 5 0 1 3 1 0 1 4 1 7 1 # Generated by triangle -p example1_8</pre>	<pre>3 3 0 0 0 3 1 1 1 4 0 2 2 1 3</pre>	
<pre>6 2 0 1 0 1 3 0 1 3 1 0 2 1 1 0 3 5 0 1 4 0 5 1 5 0 0 1 # Generated by triangle -p example2_1</pre>	<pre>7 3 0 0 4 5 0 1 2 5 1 2 0 5 2 3 0 1 3 4 1 0 2 5 3 1 5 6 0 3 4</pre>	

After we know the new triangles, they have to be implemented in a copy of the original ‘example*.ply’-file. This is done in the next paragraph.

6.6 Step 5: Manual edits – part 2

Start point of this step is the original .ply file (see Table 8) and the found points of intersection. Because these manual edits are confusing, we tried to explain the edits one by one.

Table 8: Original .ply files.

<pre> Ply format ascii 1.0 element vertex 8 property float32 x property float32 y property float32 z element face 8 property list uint8 int32 vertex_indices end_header 5.0 0.0 5.0 0.0 5.0 5.0 0.0 0.0 5.0 0.0 0.0 10.0 5.0 0.0 0.0 0.0 5.0 0.0 1.0 1.0 0.0 1.0 1.0 7.0 3 0 1 2 3 0 1 3 3 0 2 3 3 1 2 3 3 4 5 6 3 4 5 7 3 4 6 7 3 5 6 7 </pre>	<pre> Ply format ascii 1.0 element vertex 8 property float32 x property float32 y property float32 z element face 7 property list uint8 int32 vertex_indices end_header 5.0 0.0 5.0 0.0 5.0 5.0 0.0 0.0 5.0 0.0 0.0 10.0 3.0 1.0 5.0 1.0 3.0 5.0 1.0 1.0 5.0 1.0 1.0 0.0 3 0 1 2 3 0 1 3 3 0 2 3 3 1 2 3 3 4 5 7 3 4 6 7 3 5 6 7 </pre>
---	---

Thus, the original .ply files are the ones that are created in the first paragraph of this chapter. The whole process can be divided in several stages, and every stage ends with one or more files.

- A example1.ply
- B example1_intersect.*
- C example1_1.1.*
- D example1_6.1.*
- E example1_7.1.*
- F example1_8.1.*
- G example1_final.ply

To get an overview of all nodes in the final object, a scheme (see Table 9) is developed with the original nodes, the points of intersection, and the numbers of the nodes in every file. Now it is possible to translate the node-numbers from one file to another.

Table 9: Nodes of example 1.

<#>	<x><y><z>	A	B	C	D	E	F	G
0	5.0 0.0 5.0	0	-	3	-	-	-	0
1	0.0 5.0 5.0	1	-	4	-	-	-	1
2	0.0 0.0 5.0	2	-	5	-	-	-	2
3	0.0 0.0 10.0	3	-	-	-	-	-	3
4	5.0 0.0 0.0	4	-	-	2	2	-	4
5	0.0 5.0 0.0	5	-	-	3	-	2	5
6	1.0 1.0 0.0	6	-	-	-	3	3	6
7	1.0 1.0 7.0	7	-	-	4	4	4	7
8	0.714286 2.142857 5.0	-	0 and 5	0	0	-	1	8
9	2.142857 0.714286 5.0	-	1 and 3	1	1	1	-	9
10	1.0 1.0 5.0	-	2 and 4	2	-	0	0	10

The same is done for the faces (see Table 10). The last column is created with help of Table 9.

Table 10: Faces of example 1 (i = intersected, r = replaced).

<#>	A		B	C		D		E		F		G	
1	0	3 0 1 2	i	r	-	-	-	-	-	-	-	-	-
2	1	3 0 1 3	-	-	-	-	-	-	-	-	-	0	3 0 1 3
3	2	3 0 2 3	-	-	-	-	-	-	-	-	-	1	3 0 2 3
4	3	3 1 2 3	-	-	-	-	-	-	-	-	-	2	3 1 2 3
5	4	3 4 5 6	-	-	-	-	-	-	-	-	-	3	3 4 5 6
6	5	3 4 5 7	i	-	-	r	-	-	-	-	-	-	-
7	6	3 4 6 7	i	-	-	-	-	r	-	-	-	-	-
8	7	3 5 6 7	i	-	-	-	-	-	-	r	-	-	-
9				0	3 5 0 4	-	-	-	-	-	-	4	3 2 8 1
10				1	3 0 5 2	-	-	-	-	-	-	5	3 8 2 10
11				2	3 4 0 1	-	-	-	-	-	-	6	3 1 8 9
12				3	3 5 1 2	-	-	-	-	-	-	7	3 2 9 10
13				4	3 1 5 3	-	-	-	-	-	-	8	3 9 2 0
14				5	3 1 3 4	-	-	-	-	-	-	9	3 9 0 1
15				6	3 2 1 0	-	-	-	-	-	-	10	3 10 9 8
16						0	3 1 2 0	-	-	-	-	11	3 9 4 8
17						1	3 0 4 1	-	-	-	-	12	3 8 7 9
18						2	3 3 0 2	-	-	-	-	13	3 5 8 4
19								0	3 0 3 1	-	-	14	3 10 6 9
20								1	3 1 4 0	-	-	15	3 9 7 10
21								2	3 2 1 3	-	-	16	3 4 9 6
22										0	3 0 3 1	17	3 10 6 8
23										1	3 1 4 0	18	3 8 7 10
24										2	3 2 1 3	19	3 5 8 6

Now it is possible to create the new .ply file for example1. The same schemes (see Table 11 and Table 12) are made to solve the problems of example 2.

Table 11: Nodes of example 2.

<#>	<x><y><z>	A	B	C	G
0	5.0 0.0 5.0	0	-	3	0
1	0.0 5.0 5.0	1	-	4	1
2	0.0 0.0 5.0	2	-	5	2
3	0.0 0.0 10.0	3	-	-	3
4	3.0 1.0 5.0	4	1 and 3	1	4
5	1.0 3.0 5.0	5	0 and 5	0	5
6	1.0 1.0 5.0	6	2 and 4	2	6
7	1.0 1.0 0.0	7	-	-	7

Table 12: Faces of example 2 (i = intersected, r = replaced).

<#>	A	3 0 1 2	B	C	3 0 1 2	G	3 0 1 2
1	0	3 0 1 2	i	r	-	-	-
2	1	3 0 1 3	-	-	-	0	3 0 1 3
3	2	3 0 2 3	-	-	-	1	3 0 2 3
4	3	3 1 2 3	-	-	-	2	3 1 2 3
5	4	3 4 5 7	i	-	-	3	3 4 5 7
6	5	3 4 6 7	i	-	-	4	3 4 6 7
7	6	3 5 6 7	i	-	-	5	3 5 6 7
8				0	3 4 5 0	6	3 1 2 5
9				1	3 2 5 1	7	3 6 2 4
10				2	3 0 5 2	8	3 5 2 6
11				3	3 0 1 3	9	3 5 4 0
12				4	3 1 0 2	10	3 4 5 6
13				5	3 3 1 5	11	3 0 4 2
14				6	3 0 3 4	12	3 5 0 1

Last but not least, the results from the final stage can be copied to the original .ply file, if everything went ok, the problems should be solved. The final .ply files are displayed in Table 13.

Table 13: Final .ply files.

<pre>ply format ascii 1.0 element vertex 11 property float32 x property float32 y property float32 z element face 20 property list uint8 int32 vertex_indices end_header 5.0 0.0 5.0 0.0 5.0 5.0</pre>	<pre>Ply format ascii 1.0 element vertex 8 property float32 x property float32 y property float32 z element face 13 property list uint8 int32 vertex_indices end_header 5.0 0.0 5.0 0.0 5.0 5.0</pre>
--	---

<pre> 0.0 0.0 5.0 0.0 0.0 10.0 5.0 0.0 0.0 0.0 5.0 0.0 1.0 1.0 0.0 1.0 1.0 7.0 0.714286 2.142857 5.0 2.142857 0.714286 5.0 1.0 1.0 5.0 3 0 1 3 3 0 2 3 3 1 2 3 3 4 5 6 3 2 8 1 3 8 2 10 3 1 8 9 3 2 9 10 3 9 2 0 3 9 0 1 3 10 9 8 3 9 4 8 3 8 7 9 3 5 8 4 3 10 6 9 3 9 7 10 3 4 9 6 3 10 6 8 3 8 7 10 3 5 8 6 </pre>	<pre> 0.0 0.0 5.0 0.0 0.0 10.0 3.0 1.0 5.0 1.0 3.0 5.0 1.0 1.0 5.0 1.0 1.0 0.0 3 0 1 3 3 0 2 3 3 1 2 3 3 4 5 7 3 4 6 7 3 5 6 7 3 1 2 5 3 6 2 4 3 5 2 6 3 5 4 0 3 4 5 6 3 0 4 2 3 5 0 1 </pre>
--	---

These .ply files can also be viewed graphically (see Figure 20 and Figure 21). As you can notice, they have some smaller triangles than before and no illegal meetings between adjacent triangles.

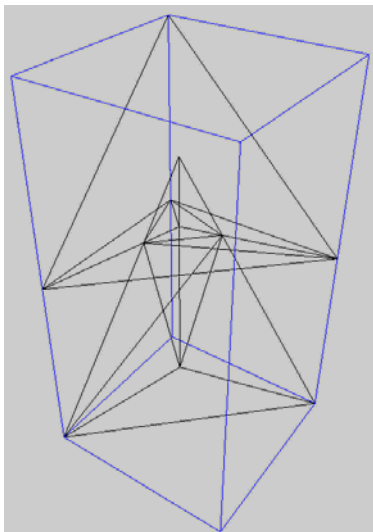


Figure 20: Example 1 - final.

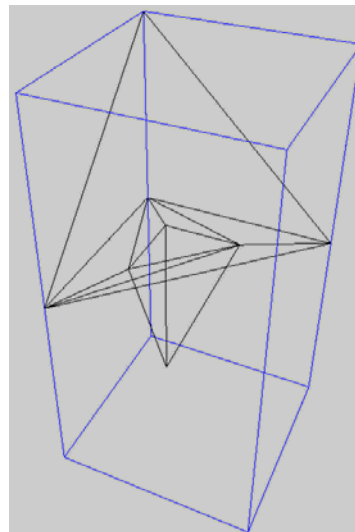


Figure 21: Example 2 - final.

6.7 Step 6: “TetGen”

The purpose of above steps was to create a new .ply files without intersecting triangles. When this is achieved, it is possible to make a tetrahedralization out of the triangulation. The resulting output is a screndump and four files per object. The screndump is displayed in Table 14.

Table 14: Screenshot of 'tetgen example*_final.ply'.

<pre> Opening example1_final.ply. Constructing Delaunay tetrahedrization. Delaunay seconds: 0.016 Creating surface mesh. Delaunizing segments. Constraining facets. Segment and facet seconds: 0 Removing unwanted tetrahedra. Hole seconds: 0 Removing illegal tetrahedra. Repair seconds: 0 Writing example1_final.1.node. Writing example1_final.1.ele. Writing example1_final.1.face. Writing example1_final.1.smesh. Output seconds: 0 Total running seconds: 0.031 Statistics: Input points: 11 Input facets: 20 Input holes: 0 Input regions: 0 Mesh points: 11 Mesh tetrahedra: 13 Mesh faces: 34 Mesh subfaces: 20 Mesh subsegments: 20 </pre>	<pre> Opening example2_final.ply. Constructing Delaunay tetrahedrization. Delaunay seconds: 0 Creating surface mesh. Delaunizing segments. Constraining facets. Segment and facet seconds: 0 Removing unwanted tetrahedra. Hole seconds: 0 Removing illegal tetrahedra. Repair seconds: 0 Writing example2_final.1.node. Writing example2_final.1.ele. Writing example2_final.1.face. Writing example2_final.1.smesh. Output seconds: 0 Total running seconds: 0 Statistics: Input points: 8 Input facets: 13 Input holes: 0 Input regions: 0 Mesh points: 8 Mesh tetrahedra: 8 Mesh faces: 22 Mesh subfaces: 13 Mesh subsegments: 12 </pre>
--	---

6.8 About the result

As you can see everything went ok, but it was not easy to execute the manual edits, especially not when the objects are much bigger than these.

7. Conclusions and recommendations

The chapter presents the conclusions and recommendations with respect to this individual assignment. In this research the attention was focussed on the question: *What is an efficient way to change an invalid 3D triangulated boundary representation into a valid one?* To answer this question, it was subdivided into six steps, these are answered in the main text.

7.1 Conclusions

The conclusions with respect to the research will be given by taking the six steps.

Make an inventory of all topological relations that exist between two faces.

An inventory is made based on the research of Pigot (1991). In general it can be stated that: If two intersecting triangles are not in the same plane they have one point or one line in common. (§ 3.2)

Find a way to test whether two triangles intersect and to determine their topological relation.

Aftosmis et al. (1997) developed a new method to detect if two triangles are intersecting and O'Rourke (1998) wrote a code that uses that same method. (§ 3.3)

Find the way to derive the intersection points.

With the theory of O'Rourke (1998) it is possible to detect whether two adjacent triangles are intersecting, and also where they are intersecting. (§ 3.4)

Look to the solution of Hang Si ("TetGen"), and use this program to implement the new solution in C++.

The implementation is done, and it is explained in Chapter 5. The full code is added in the Appendix A, B, and C, and on the CD which is part of the report.

Look to the solutions or parts of solutions made by other programmers.

"Triangle" and "TetGen" are solutions, which have the possibilities to solve parts of the problem. (§ 4.1 and § 4.2)

Find possibilities, like indexing methods, which can make the algorithm more efficient.

"TetGen" and also "Intersect" make use of an indexing method, which is efficient, but the code of "Intersect" can be more efficient. (§ 2.5)

7.2 Recommendations

As said in the conclusions and in § 5.3, "Intersect" can be more efficient and some manual edits have to be automated. Thus, there is an assignment for the future:

- 1 Create for every face that is intersected, a .poly file, which contains the original coordinates, the points of intersection, and a list of constraint-lines. That constraint-lines include the lines of intersection of two adjacent triangles and the convex hull of the intersected triangle, which includes the points of intersection that are on the edges of the original triangle.

- 2 Write an algorithm to get the new triangulations from “Triangle”, for every face that is intersected.
- 3 Translate the numbers of the points of intersection to numbers of points that fit in the original .ply file, do the same for all triangles, replace the intersected ones by the new.

Furthermore, the whole process can be batched and implemented in “TetGen” to make it much more easy to execute it in the future.

References

Aftosmis M.J., Berger M.J., Melton J.E. (1997) Robust and efficient Cartesian mesh generation for component-based geometry, Technical Report AIAA-97-0196, US Air Force Wright Laboratory, pp. 2-4.

Computer Graphics Laboratory, ETH Zurich (2001) Research on point-based graphics, <http://graphics.ethz.ch/>, last visited on 2005-02-17, last updated on 2004-07-19.

Delaunay B. (1934) Sur la sphère vide, *Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk*, **7**:793-800, 1934.

George P. and Borouchaki H. (1998) Delaunay Triangulation and Meshing, Application to Finite Elements, Hermes, Paris, pp. 5-11, 13-14.

O'Rourke J. (1994) Computational Geometry in C, Cambridge University Press, Cambridge.

O'Rourke J. (1998) Computational Geometry in C (Second Edition), Cambridge University Press, Cambridge, pp. 220-238.

Pigot S. (1991) Topological models for 3D spatial information systems, In: D. M. Mark and D. White (Eds.), Auto Carto 10, Baltimore, MD, pp. 374-377, 381-387.

Shewchuk J.R. (1996) Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator, First Workshop on Applied Computational Geometry (Philadelphia, Pennsylvania), ACM, pages 124-133.

Si H. (2004) TetGen: A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator, User's Manual (version 1.3), <http://tetgen.berlios.de>, last visited on 2005-02-28, last modified on 2004-12-13, pp. 17-18, 22.

The Free Encyclopaedia Wikipedia, http://en.wikipedia.org/wiki/Delaunay_triangulation, last visited on 2005-02-28, last modified on 2004-12-07.

Appendix A: Predicates_intersect.cxx

```

/*****
/*
/* NormalVec() Returns the normalvector of a plane defined by A, B and C */
/* DotDot() Returns the dot product of the two input vectors. */
/* PlaneCoeff() Returns the index of the largest component of N and D */
/* SubVec() Returns the vector from a to b */
/* SegPlaneIntNew() Returns a character for the type of intersection */
/* Abs() Returns the absolute value of a number */
/* SquareRoot() Returns the squareroot of a number */
/* Area3D() Returns the size of face a, b, c. */
/* LocatePoint() Returns a number for the location of a point within a T */
/* SegPlaneInt() Returns intersecting point p of a segment and a plane. */
/* SegSegInt() Returns intersecting point p of two segments. */
/*
/*
/* Functions inserted by Martine Hoefsloot */
/*
/*****

void NormalVec( REAL* a, REAL* b, REAL* c, REAL* N )
{
    N[0] = ( c[2] - a[2] ) * ( b[1] - a[1] ) -
           ( b[2] - a[2] ) * ( c[1] - a[1] );
    N[1] = ( b[2] - a[2] ) * ( c[0] - a[0] ) -
           ( b[0] - a[0] ) * ( c[2] - a[2] );
    N[2] = ( b[0] - a[0] ) * ( c[1] - a[1] ) -
           ( b[1] - a[1] ) * ( c[0] - a[0] );
}

REAL DotDot( REAL* a, REAL* b )
{
    REAL sum = 0.0;
    sum = a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
    return sum;
}

int PlaneCoeff( REAL* A, REAL* B, REAL* C, REAL* N, double* D )
{
    int i;
    double t;
    double biggest = 0.0;
    int m = 0;

    NormalVec( A, B, C, N);
    *D = DotDot( A, N);

    for (i = 0; i < 3; i++) {
        t = fabs(N[i]);
        if (t > biggest) {
            biggest = t;
            m = i;
        }
    }
    return m;
}

void SubVec( REAL* A, REAL* B, REAL* C )
{
    int i;
    for (i = 0; i < 3; i++)
        C[i] = A[i] - B[i];
}

double Abs(double Nbr)
{
    if( Nbr >= 0 )
        return Nbr;
    else
        return -Nbr;
}

double SquareRoot(double Nbr)

```

```
{
  double Number = Nbr / 2;
  const double Tolerance = 1.0e-8;

  do Number = (Number + Nbr / Number) / 2;
  while( Abs(Number * Number - Nbr) > Tolerance);

  return Number;
}

int Area3D(REAL* a, REAL* b, REAL* c)
{
  REAL sqarea;
  REAL area;
  int area_return;
  REAL ab[3];
  REAL bc[3];
  REAL ca[3];
  REAL sqdistab;
  REAL sqdistbc;
  REAL sqdistca;
  REAL distab;
  REAL distbc;
  REAL distca;
  REAL s;
  int i;

  for (i = 0; i < 3; i++ )
  {
    ab[i] = a[i] - b[i];
    bc[i] = b[i] - c[i];
    ca[i] = c[i] - a[i];
  }

  sqdistab = DotDot(ab, ab);
  sqdistbc = DotDot(bc, bc);
  sqdistca = DotDot(ca, ca);
  distab = SquareRoot(sqdistab);
  distbc = SquareRoot(sqdistbc);
  distca = SquareRoot(sqdistca);

  s = (distab + distbc + distca)/2;
  sqarea = s * (s - distab) * (s - distbc) * (s - distca);

  area = SquareRoot(sqarea);
  //printf("area = %f.\n", area);
  if (area > 0.000100) {
    area_return = 1;
  }
  else {
    area_return = 0;
  }

  return area_return;
}

int LocatePoint(REAL* a, REAL* b, REAL* c, REAL* p)
{
  int area0;
  int area1;
  int area2;

  area0 = Area3D(a, b, p);
  area1 = Area3D(b, c, p);
  area2 = Area3D(c, a, p);

  if (area2 == 0 && area0 == 0) {
    return 1; } // in A
  else if (area0 == 0 && area1 == 0) {
    return 2; } // in B
  else if (area1 == 0 && area2 == 0) {
    return 3; } // in C
  else if (area0 == 0 && area1 != 0 && area2 != 0) {
    return 12; } // on AB
  else if (area1 == 0 && area0 != 0 && area2 != 0) {
    return 23; } // on BC
}
```

```

else if (area2 == 0 && area0 != 0 && areal != 0) {
    return 31; } // on CA
else {
    return 0; } // inside or outside ABC
}

int PlaneCoeffm( REAL* a, REAL* b, REAL* c)
{
    REAL N[3];
    REAL D;
    int m;

    m = PlaneCoeff(a, b, c, N, &D);
    return m;
}

REAL SegPlaneIntX( REAL* a, REAL* b, REAL* c, REAL* q, REAL* r)
{
    REAL N[3];
    REAL D;
    REAL rq[3];
    REAL num, denom, t;
    REAL p[3];
    int i, m;

    m = PlaneCoeff(a, b, c, N, &D);
    num = D - DotDot( q, N );
    SubVec(r, q, rq);
    denom = DotDot( rq, N );

    if ((denom == 0.0) && (num == 0.0)) {
        p[0] = 99;
        return p[0];
    }
    else {
        t = num / denom;
        for( i = 0; i < 3; i++ ) {
            p[i] = q[i] + t * ( r[i] - q[i] );
        }
        if ((p[0] >= -0.000000) && (p[0] <= 0.000000))
            p[0] = 0.000000;
        return p[0];
    }
}

REAL SegPlaneIntY( REAL* a, REAL* b, REAL* c, REAL* q, REAL* r)
{
    REAL N[3];
    REAL D;
    REAL rq[3];
    REAL num, denom, t;
    REAL p[3];
    int i, m;

    m = PlaneCoeff(a, b, c, N, &D);
    num = D - DotDot( q, N );
    SubVec(r, q, rq);
    denom = DotDot( rq, N );

    if ((denom == 0.0) && (num == 0.0)) {
        p[1] = 99;
        return p[1];
    }
    else {
        t = num / denom;
        for( i = 0; i < 3; i++ ) {
            p[i] = q[i] + t * ( r[i] - q[i] );
        }
        if ((p[1] >= -0.000000) && (p[1] <= 0.000000))
            p[1] = 0.000000;
        return p[1];
    }
}

REAL SegPlaneIntZ( REAL* a, REAL* b, REAL* c, REAL* q, REAL* r)

```

```

{
  REAL N[3];
  REAL D;
  REAL rq[3];
  REAL num, denom, t;
  REAL p[3];
  int i, m;

  m = PlaneCoeff(a, b, c, N, &D);
  num = D - DotDot( q, N );
  SubVec(r, q, rq);
  denom = DotDot( rq, N );

  if ((denom == 0.0) && (num == 0.0)) {
    p[2] = 99;
    return p[2];
  }
  else {
    t = num / denom;
    for( i = 0; i < 3; i++ ) {
      p[i] = q[i] + t * ( r[i] - q[i] );
    }
    if ((p[2] >= -0.000000) && (p[2] <= 0.000000))
      p[2] = 0.000000;
    return p[2];
  }
}

REAL SegSegInt( REAL* a, REAL* b, REAL* q, REAL* r )
{
  /* calculates point of intersection p of two edges */
  /* if: p = q + t * (r - q) and p = a + s * (b - a) */
  /* */
  /* q + t * (r - q) = a + s * (b - a) */
  /* (r - q) * t - (b - a) * s = a - q */
  /* */
  /* */
  REAL nums, s;
  REAL numt, t;
  REAL AA, BB, DD, EE, FF;
  REAL denom;
  REAL aq[3], ab[3], rq[3];
  REAL p[3];
  int i;
  int j;

  for( j = 0; j < 3; j++ )
  {
    aq[j] = a[j] - q[j];
    ab[j] = a[j] - b[j];
    rq[j] = r[j] - q[j];
  }

  AA = DotDot(rq, rq);
  DD = DotDot(ab, ab);
  BB = DotDot(rq, ab);
  EE = DotDot(rq, aq);
  FF = DotDot(ab, aq);

  denom = AA * DD - BB * BB;
  nums = - BB * EE + AA * FF;
  numt = DD * EE - BB * FF;

  t = numt / denom;
  s = nums / denom;
  for( i = 0; i < 3; i++ )
  {
    p[i] = q[i] + t * ( r[i] - q[i] );
    // and p[i] = a[i] + s * ( b[i] - a[i] )
  }
  return p[3];
}

```

Appendix B: Intersect.cxx – part 1

```

if (b->verbose > 1) {
    printf("  Checking intersecting faces.\n");
}
/* Edited by Martine */
REAL pp[9][10];
REAL poi[6];
REAL ppoi[3];

// Perform a brute-force compare on the set.
for (i = 0; i < arraysize; i++) {
    sfacel.sh = subfacearray[i];
    A = (point) sfacel.sh[3];
    B = (point) sfacel.sh[4];
    C = (point) sfacel.sh[5];
    for (j = i + 1; j < arraysize; j++) {
        sface2.sh = subfacearray[j];
        O = (point) sface2.sh[3];
        P = (point) sface2.sh[4];
        Q = (point) sface2.sh[5];
        intersect = triangle_triangle_inter(A, B, C, O, P, Q);
        if (intersect == INTERSECT || intersect == SHAREFACE) {
            if (!b->quiet) {
                if (intersect == INTERSECT) {
                    printf("  Facet #%d intersects facet #%d at triangles:\n",
                        shellmark(sfacel), shellmark(sface2));
                    printf("    (%4d, %4d, %4d) and (%4d, %4d, %4d)\n",
                        pointmark(A), pointmark(B), pointmark(C),
                        pointmark(O), pointmark(P), pointmark(Q));

                    REAL abcop, abcpq, abcqo, opqab, opqbc, opqca;
                    REAL s_o, s_p, s_q, s_a, s_b, s_c;

                    s_o = orient3d(A, B, C, O);
                    s_p = orient3d(A, B, C, P);
                    s_q = orient3d(A, B, C, Q);
                    s_a = orient3d(O, P, Q, A);
                    s_b = orient3d(O, P, Q, B);
                    s_c = orient3d(O, P, Q, C);

                    int m;
                    int pointofintersect = 0;

                    abcop = triangle_edge_inter_tail(A, B, C, O, P, s_o, s_p);
                    if (abcop == INTERSECT || abcop == SHAREVERTEX) {
                        poi[0] = ppoi[0] = SegPlaneIntX(A, B, C, O, P);
                        poi[1] = ppoi[1] = SegPlaneIntY(A, B, C, O, P);
                        poi[2] = ppoi[2] = SegPlaneIntZ(A, B, C, O, P);
                        poi[3] = 1;
                        printf("    1 Point of Intersection is: (%f, %f, %f).\n", poi[0],
                            poi[1], poi[2]);
                        if ((poi[3] == 1) && (poi[0] != 99.000000)) {
                            pp[pointofintersect][0] = poi[0];
                            pp[pointofintersect][1] = poi[1];
                            pp[pointofintersect][2] = poi[2];
                            pp[pointofintersect][3] = shellmark(sfacel);
                            pp[pointofintersect][4] = PlaneCoeffm(A, B, C);
                            pp[pointofintersect][5] = LocatePoint(A, B, C, ppoi);
                            pp[pointofintersect][6] = shellmark(sface2);
                            pp[pointofintersect][7] = PlaneCoeffm(O, P, Q);
                            pp[pointofintersect][8] = LocatePoint(O, P, Q, ppoi);
                            (pointofintersect)++;
                        }
                    }
                    abcpq = triangle_edge_inter_tail(A, B, C, P, Q, s_p, s_q);
                    if (abcpq == INTERSECT || abcpq == SHAREVERTEX) {
                        poi[0] = ppoi[0] = SegPlaneIntX(A, B, C, P, Q);
                        poi[1] = ppoi[1] = SegPlaneIntY(A, B, C, P, Q);
                        poi[2] = ppoi[2] = SegPlaneIntZ(A, B, C, P, Q);
                        poi[3] = 1;
                        printf("    2 Point of Intersection is: (%f, %f, %f).\n", poi[0],
                            poi[1], poi[2]);
                        for (m = 0; m < pointofintersect; m++) {
                            REAL distancepp;

```

```

distancepp = (pp[m][0] - poi[0])*(pp[m][0] - poi[0]) +
              (pp[m][1] - poi[1])*(pp[m][1] - poi[1]) +
              (pp[m][2] - poi[2])*(pp[m][2] - poi[2]);
if (distancepp <= 0.00000005) {
    poi[3] = 0;
    break;
}
else
    poi[3] = 1;
}

if ((poi[3] == 1) && (poi[0] != 99.000000)) {
    pp[pointofintersect][0] = poi[0];
    pp[pointofintersect][1] = poi[1];
    pp[pointofintersect][2] = poi[2];
    pp[pointofintersect][3] = shellmark(sfacel);
    pp[pointofintersect][4] = PlaneCoeffm(A, B, C);
    pp[pointofintersect][5] = LocatePoint(A, B, C, ppoi);
    pp[pointofintersect][6] = shellmark(sface2);
    pp[pointofintersect][7] = PlaneCoeffm(O, P, Q);
    pp[pointofintersect][8] = LocatePoint(O, P, Q, ppoi);
    (pointofintersect)++;
}
}
}
abcqo = triangle_edge_inter_tail(A, B, C, Q, O, s_q, s_o);
if (abcqo == INTERSECT || abcqo == SHAREVERTEX) {
    poi[0] = ppoi[0] = SegPlaneIntX(A, B, C, Q, O);
    poi[1] = ppoi[1] = SegPlaneIntY(A, B, C, Q, O);
    poi[2] = ppoi[2] = SegPlaneIntZ(A, B, C, Q, O);
    poi[3] = 1;
    printf("    3 Point of Intersection is: (%f, %f, %f).\n", poi[0],
           poi[1], poi[2]);
    for (m = 0; m < pointofintersect; m++) {
        REAL distancepp;
        distancepp = (pp[m][0] - poi[0])*(pp[m][0] - poi[0]) +
                    (pp[m][1] - poi[1])*(pp[m][1] - poi[1]) +
                    (pp[m][2] - poi[2])*(pp[m][2] - poi[2]);
        if (distancepp <= 0.00000005) {
            poi[3] = 0;
            break;
        }
        else
            poi[3] = 1;
    }
    if ((poi[3] == 1) && (poi[0] != 99.000000)) {
        pp[pointofintersect][0] = poi[0];
        pp[pointofintersect][1] = poi[1];
        pp[pointofintersect][2] = poi[2];
        pp[pointofintersect][3] = shellmark(sfacel);
        pp[pointofintersect][4] = PlaneCoeffm(A, B, C);
        pp[pointofintersect][5] = LocatePoint(A, B, C, ppoi);
        pp[pointofintersect][6] = shellmark(sface2);
        pp[pointofintersect][7] = PlaneCoeffm(O, P, Q);
        pp[pointofintersect][8] = LocatePoint(O, P, Q, ppoi);
        (pointofintersect)++;
    }
}
}
opqab = triangle_edge_inter_tail(O, P, Q, A, B, s_a, s_b);
if (opqab == INTERSECT || opqab == SHAREVERTEX) {
    poi[0] = ppoi[0] = SegPlaneIntX(O, P, Q, A, B);
    poi[1] = ppoi[1] = SegPlaneIntY(O, P, Q, A, B);
    poi[2] = ppoi[2] = SegPlaneIntZ(O, P, Q, A, B);
    poi[3] = 1;
    printf("    4 Point of Intersection is: (%f, %f, %f).\n", poi[0],
           poi[1], poi[2]);
    for (m = 0; m < pointofintersect; m++) {
        REAL distancepp;
        distancepp = (pp[m][0] - poi[0])*(pp[m][0] - poi[0]) +
                    (pp[m][1] - poi[1])*(pp[m][1] - poi[1]) +
                    (pp[m][2] - poi[2])*(pp[m][2] - poi[2]);
        if (distancepp <= 0.00000005) {
            poi[3] = 0;
            break;
        }
        else
            poi[3] = 1;
    }
}
}
}

```

```

    }
    if ((poi[3] == 1) && (poi[0] != 99.000000)) {
        pp[pointofintersect][0] = poi[0];
        pp[pointofintersect][1] = poi[1];
        pp[pointofintersect][2] = poi[2];
        pp[pointofintersect][3] = shellmark(sface1);
        pp[pointofintersect][4] = PlaneCoeffm(A, B, C);
        pp[pointofintersect][5] = LocatePoint(A, B, C, ppoi);
        pp[pointofintersect][6] = shellmark(sface2);
        pp[pointofintersect][7] = PlaneCoeffm(O, P, Q);
        pp[pointofintersect][8] = LocatePoint(O, P, Q, ppoi);
        (pointofintersect)++;
    }
}
opqbc = triangle_edge_inter_tail(O, P, Q, B, C, s_b, s_c);
if (opqbc == INTERSECT || opqbc == SHAREVERTEX) {
    poi[0] = ppoi[0] = SegPlaneIntX(O, P, Q, B, C);
    poi[1] = ppoi[1] = SegPlaneIntY(O, P, Q, B, C);
    poi[2] = ppoi[2] = SegPlaneIntZ(O, P, Q, B, C);
    poi[3] = 1;
    printf("    5 Point of Intersection is: (%f, %f, %f).\n", poi[0],
        poi[1], poi[2]);
    for (m = 0; m < pointofintersect; m++) {
        REAL distancepp;
        distancepp = (pp[m][0] - poi[0])*(pp[m][0] - poi[0]) +
            (pp[m][1] - poi[1])*(pp[m][1] - poi[1]) +
            (pp[m][2] - poi[2])*(pp[m][2] - poi[2]);
        if (distancepp <= 0.00000005) {
            poi[3] = 0;
            break;
        }
        else
            poi[3] = 1;
    }
    if ((poi[3] == 1) && (poi[0] != 99.000000)) {
        pp[pointofintersect][0] = poi[0];
        pp[pointofintersect][1] = poi[1];
        pp[pointofintersect][2] = poi[2];
        pp[pointofintersect][3] = shellmark(sface1);
        pp[pointofintersect][4] = PlaneCoeffm(A, B, C);
        pp[pointofintersect][5] = LocatePoint(A, B, C, ppoi);
        pp[pointofintersect][6] = shellmark(sface2);
        pp[pointofintersect][7] = PlaneCoeffm(O, P, Q);
        pp[pointofintersect][8] = LocatePoint(O, P, Q, ppoi);
        pointofintersect++;
    }
}
opqca = triangle_edge_inter_tail(O, P, Q, C, A, s_c, s_a);
if (opqca == INTERSECT || opqca == SHAREVERTEX) {
    poi[0] = ppoi[0] = SegPlaneIntX(O, P, Q, C, A);
    poi[1] = ppoi[1] = SegPlaneIntY(O, P, Q, C, A);
    poi[2] = ppoi[2] = SegPlaneIntZ(O, P, Q, C, A);
    poi[3] = 1;
    printf("    6 Point of Intersection is: (%f, %f, %f).\n", poi[0],
        poi[1], poi[2]);
    for (m = 0; m < pointofintersect; m++) {
        REAL distancepp;
        distancepp = (pp[m][0] - poi[0])*(pp[m][0] - poi[0]) +
            (pp[m][1] - poi[1])*(pp[m][1] - poi[1]) +
            (pp[m][2] - poi[2])*(pp[m][2] - poi[2]);
        if (distancepp <= 0.00000005) {
            poi[3] = 0;
            break;
        }
        else
            poi[3] = 1;
    }
    if ((poi[3] == 1) && (poi[0] != 99.000000)) {
        pp[pointofintersect][0] = poi[0];
        pp[pointofintersect][1] = poi[1];
        pp[pointofintersect][2] = poi[2];
        pp[pointofintersect][3] = shellmark(sface1);
        pp[pointofintersect][4] = PlaneCoeffm(A, B, C);
        pp[pointofintersect][5] = LocatePoint(A, B, C, ppoi);
        pp[pointofintersect][6] = shellmark(sface2);
        pp[pointofintersect][7] = PlaneCoeffm(O, P, Q);
    }
}

```

```
        pp[pointofintersect][8] = LocatePoint(O, P, Q, ppoi);
        (pointofintersect)++;
    }
}

int k;
for (k = 0; k < pointofintersect; k++) {
    fprintf(fout, "%5d %10f %10f %10f %5d %2d %2d %5d %2d %2d\n",
        *numberofintersectpoint, pp[k][0], pp[k][1], pp[k][2],
        shellmark(sface1), int(pp[k][4]), int(pp[k][5]),
        shellmark(sface2), int(pp[k][7]), int(pp[k][8]));
    printf("%5d %10f %10f %10f %5d %2d %2d %5d %2d %2d\n",
        *numberofintersectpoint, pp[k][0], pp[k][1], pp[k][2],
        shellmark(sface1), int(pp[k][4]), int(pp[k][5]),
        shellmark(sface2), int(pp[k][7]), int(pp[k][8]));
    (*numberofintersectpoint)++;
}
/* End edited by Martine */
}
```


Appendix C: Intersect.cxx – part 2

```

/* Edited by Martine */
int numberofintersectpoint = 0;

FILE* fout;
char outnodefilename[FILENAME_SIZE];
sprintf(outnodefilename, "%s_intersect.node", b->infile);
fout = fopen(outnodefilename, "w");
fprintf(fout, "
                                \n");
/* End edited by Martine */

// Recursively split the set of triangles into two sets using a cut plane
// parallel to x-, or, y-, or z-axes. Stop splitting when the number
// of subfaces is not decreasing anymore. Do tests on the current set.
interecursive(subfacearray, subfaces->items, 0, xmin, xmax, ymin, ymax,
              zmin, zmax, &internum, &numberofintersectpoint, fout);

/* Edited by Martine */
fprintf(fout, "# Generated by %s\n", b->commandline);
rewind(fout);
fprintf(fout, "%5d %d %d %d", numberofintersectpoint, in->mesh_dim, 6,
        in->pointmarkerlist != NULL ? 1 : 0);
fclose(fout);
/* End edited by Martine */

if (!b->quiet) {
    if (internum > 0) {
/* Edited by Martine */
        printf("\n!! Found %d pairs of faces are intersecting.\n", internum);
        printf("!! Found %d points of intersection.\n\n", numberofintersectpoint);
/* End edited by Martine */
    }
}

```


Appendix D: On the CD

On the attached CD you can find six folders and a .pdf document:

- Data Examples and some shapefiles.
- Example 1 The files used in Chapter 6.
- Example 2 The files used in Chapter 6.
- Intersect The compiled application “Intersect”.
- TetGen The compiled application “TetGen”.
- Triangle The compiled application “Triangle”.
- Intersect.pdf This report.