University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Delaunay Triangulation in 3D

State of the Art and Concept of Doctoral Thesis

Pavel Maur

Technical Report No. DCSE/TR-2002-02
January, 2002

Distribution: public

Technical Report No. DCSE/TR-2002-02
January 2002

Delaunay Triangulation in 3D

Pavel Maur

Abstract

The Delaunay triangulation is one of the most popular and most often used methods in problems related to the generation of meshes. A lot of the optimal properties of Delaunay triangulation are known in 2D, where it has been intensively studied during the last twenty years, although the fundamentals were formulated early in the twentieth century (Voronoi, 1908 and Delaunay, 1934). This thesis presents Delaunay triangulation without addition or displacement of points in 3D space. It focuses on its properties and on a summarization of existing sequential algorithms. Also our experience with the implementation of the incremental insertion algorithm is presented and observed features are discussed.

The properties of Delaunay triangulation in 3D (or generally in higher dimensions) are not as good as in 2D and different kinds of methods are used mainly to remove the tetrahedra of undesirable shape. Although this area of research was not within our main scope, we present an existing simple method for tetrahedra shape improvement. We have implemented this method and our results are presented and discussed.

In the implementation of algorithms, which have to deal with inprecise floating-point arithmetic on real computers, the question of numerical stability becomes very important for the proper function of the implementation. We introduce several existing approaches for increasing the numerical stability of algorithms, two of them for an exact evaluation of geometric predicates are presented in more details. We made a comparison of them and we mention the results of incorporating one of them in our implementation.

This work was supported by the Ministry of Education of Czech Republic – project MSM 235200005.

Copies of this report are available on
`http://www.kiv.zcu.cz/publications/`
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Copyright © 2002 University of West Bohemia in Pilsen, Czech Republic

Table of Contents

TABLE OF CONTENTS 1

1 INTRODUCTION 2

2 DELAUNAY TRIANGULATION 3

3 ALGORITHMS FOR 3D DELAUNAY TRIANGULATION 5

- 3.1 Local improvement 5
- 3.2 Incremental construction 6
- 3.3 Incremental insertion 12
- 3.4 Higher dimension embedding 15
- 3.5 Divide and conquer 15

4 OUR IMPLEMENTATION OF DT TRIANGULATOR 18

- 4.1 Explanation of the algorithm 18
- 4.2 Results of implementation 20

5 THE QUALITY OF THE MESH 23

- 5.1 Measuring tetrahedra properties 23
- 5.2 Properties of Delaunay tetrahedra 23

6 POST-OPTIMIZATION OF DT 26

- 6.1 Post-optimization by geometrical criteria 27
- 6.2 Changing the number of tetrahedra 27
- 6.3 Removing of slivers 28
- 6.4 Results of geometrical criteria use 29
- 6.5 Results of criteria changing the number of tetrahedra 32
- 6.6 Results of removing slivers 32
- 6.7 Summary 33

7 NUMERICAL STABILITY AND ROBUSTNESS 35

- 7.1 Geometric predicates 36
- 7.2 LN generator 37
- 7.3 Adaptive floating-point geometric predicates 39
- 7.4 Practical experience 42

8 OUR GOALS AND FUTURE WORK 48

9 REFERENCES 49

APPENDIX A: OTHER DT IMPLEMENTATIONS 52

APPENDIX B: ACTIVITIES 53

1 Introduction

There are practical problems, which are described by nonlinear equations and which cannot be solved analytically. Thus various approximate methods are used, which are based on numerical computing of discretized original task (e.g. finite element methods) [TSW99]. The discretization of the domain is provided by the meshing – the **mesh** is a network of discrete cells over the domain. Meshes can be divided into *structured* (points are distributed regularly, basic elements of discretization are quadrilaterals and hexahedra in 2D and 3D, respectively) and *unstructured* (irregularly placed points, such meshes consist of triangles in 2D and tetrahedra or hexahedra without directional structure in 3D). From a general point of view, unstructured meshes are more flexible for the discretization of complex geometries and mesh adaptation, on the other hand, they require a much more complex data structure for solution.

Structured meshes are not the topic of this thesis.

Unstructured meshes are generated by three methods in general [TSW99]:

- the Delaunay triangulation – first, Delaunay triangulation of the existing points is made; if the mesh is not sufficiently fine, new points are generated, triangles are subdivided and the mesh is refined to make the approximation more precise,
- advancing front techniques – the point generation (and triangulation) goes in the direction from the boundary of the domain inwards,
- finite octree – the sparse regular grid over the domain is subdivided so that at the boundaries the cell size is consistent with the boundary point spacing; the mesh is then further modified by the boundaries.

Although a fully automated mesh generator for arbitrary domain is still only a dream, the Delaunay-based methods are mentioned to be closest methods for black box mesh generation [TSW99]. Although the Delaunay triangulation in 2D needs an extra algorithm to conform to a given boundary, it has a great advantage in its optimal properties, mainly in 2D, where it e.g. maximizes the minimal angle of each triangle and of the whole triangulation or it minimizes the radius of circumscribed circles of triangles (i.e., in other words, produces well shaped triangles). The optimal time complexity of algorithms is $O(n \log n)$ in 2D. Algorithms for other optimal triangulations, which optimize e.g. minmax angle or minmax edge length, are more complex. In 3D space not all the good properties of Delaunay triangulation are preserved: the conformity of boundaries is a far more complex problem, which is not always solvable. On the other hand, if the quality of triangulation is not sufficient, there is always a “cheap” method of further improvement.

Delaunay triangulation got over the borders of the area of mesh generation for numerical computing and entered the area of crystallography, metallurgy, cartography, geology and also the area of computer graphics. There is used in many applications, e.g., object reconstruction from scattered data points, solid modelling, interpolation and iso-surface extraction. It is also used in virtual reality for virtual surgery, etc.

The Delaunay triangulation is the main topic of this thesis. Firstly its properties are presented. Then an overview of existing sequential algorithms for the construction of 3D Delaunay triangulation (without Steiner points and point displacement), which are based on different algorithmic paradigms, is given. Also our experience with the implementation of incremental algorithm is mentioned. A technique for the optimization of tetrahedral meshes is also mentioned and the results of our experiments in this area are presented. One important part of the text is dedicated to an improvement in the numerical stability of triangulation algorithms, where existing solutions are presented and experience with an incorporation of the exact evaluation of geometric predicates into our implementation is commented.

2 Delaunay triangulation

Before we start, let us recall several definitions [GOR97]:

- **simplex** is the convex hull of d+1 affinely independent points in E^d,
- **circumsphere** is the sphere through the vertices of a simplex,
- **flat** is an affine subspace of dimension k.

Triangulation of a given set P of n points in d-dimensional space E^d is a simplicial decomposition of the convex hull of P. For 3D space it means the convex hull of P is decomposed into tetrahedra such that

- vertices of tetrahedra belong to P,
- intersection of two tetrahedra is either empty or a vertex or an edge or a face.

Triangulation in 3D space can be called *triangulation*, *3D triangulation*, *tetrahedralization* [GOR97 p. 423] or *tetrahedrization* [MK00]. The term triangulation will be used in this thesis.

Delaunay triangulation (shortly DT) is a triangulation such that the circumsphere of every d-simplex is empty, i.e., it does not contain any of the given points in its interior. Properties of DT in E^d are as follows [GOR97]:

- the DT is unique, if the point set is in general position (or is non-degenerate): no d+2 points lie on a common d-sphere and no k+2 points lie on a common k-flat, for $k<d$,
- the DT includes at most $O(n^{\lceil d/2 \rceil})$ simplices,
- the DT minimizes the maximum radius of a simplex enclosing sphere (the enclosing sphere is the smallest sphere containing a simplex), see [Raj91],
- the faces of DT hold visibility depth ordering for any viewpoint (i.e. the "in front of" relation of faces is acyclic),
- the edges of a DT form a graph DT; the following subgraph relations holds:

  $EMST \subseteq RNG \subseteq GG \subseteq DT$,

  where EMST is the Euclidean minimum spanning tree, RNG is the relative neighbourhood graph and GG is the Gabriel graph.

For the comparison, in two dimensions there are more optimal properties proven or observed [GOR97]:

- the DT minimizes the maximum radius of a circumcircle,
- the DT maximizes the minimum angle (and, also, lexicographically maximizes the angles from smallest to largest),
- maximizes the sum of inscribed circle radii,
- minimizes the roughness of a piecewise-linear interpolating surface (roughness is the square of the L_2 norm of the gradient of the surface, integrated over the triangulation),
- minimizes the surface area of a piecewise-linear interpolating surface for elevations scaled "sufficiently" small,
- the distance along edges of the DT between any pair of vertices is at most a constant (at most 2.42) times the Euclidean distance between them.

It really seems that two dimensional space is a promised land for Delaunay triangulation. With an increasing dimension the complications appear. Let us make only one

step higher. The first complication in E^3 is that the same point set can be generally triangulated in several ways with different number of tetrahedra. The second bad news is that a triangulation can have as few as $n-3$ or as many as $\binom{n-2}{2}$ tetrahedra [AEG87, GOR97]. For 3D DT it means that unlike the 2D case the amount of tetrahedra is not known in advance and can vary from O(n) to O(n^2). In 2D the number of triangles is always O(n). For example, the expected number of simplices for points chosen uniformly at random from inside a sphere is ~2n in 2D or ~6.77n in 3D. The worst number of simplices mentioned before leads to the worst case time complexity of algorithms, which is O(n log n) resp. O(n^2) for 2D resp. 3D. But many of the 3D algorithms work in linear expected time, when the number of simplices is linear too (which is in most cases).

Two more general properties of DT has not to be forgotten [GOR97]:

- the DT is dual of Voronoi diagram in the same dimension,
- there is a close connection between DT in d-dimensional space and convex hulls in (d+1)-dimensional space.

These properties can be used for non-direct construction of DT. Creation of DT from the convex hull constitutes a special class of algorithms (see below), while in the case of a Voronoi diagram it is simpler to construct a DT directly than with the use of a Voronoi diagram, although it is possible.

3 Algorithms for 3D Delaunay triangulation

In this chapter we give an overview of the existing sequential algorithms for Delaunay triangulation. Firstly we present different classes or general concepts of the algorithms. The algorithms themselves will be discussed and commented in more details later, each of them in the section dedicated to a particular class.

The time complexity of algorithms for 3D DT is derived from the number of simplices in the triangulation. The worst case optimal time is $O(n^2)$, but as we mentioned before, the number of simplices depends on the distribution of data points. Thus the expected time complexity can be even $O(n)$.

As for the data structures, the faces or tetrahedra are often stored in simple or doubly linked lists, or the hash table can be used. The selection of data structure depends on particular algorithm, because, for example, when the searching for faces is not needed, it is useless to manage a hash table for it.

There are several different types of algorithms for creating Delaunay triangulation. According to [CMS98] in general we can classify DT algorithms into several groups:

- *local improvement* – starting with an arbitrary triangulation, these algorithms locally flip the faces of pairs of adjacent simplices according to the circumsphere criterion,
- *incremental construction* (or *shelling* or *face expansion*) – the DT is constructed by successively building simplices whose circumspheres contain no points in P,
- *incremental insertion* – these algorithms insert the points from P one at a time: the simplex containing the currently added point is partitioned by inserting it as a new vertex; the circumsphere criterion is tested on all the simplices adjacent to the new ones, recursively, and, if necessary, their faces are flipped or processed in a particular way,
- *higher dimensional embedding* – these algorithms transform the points into the E^{d+1} space and then compute the convex hull of the transformed points; the DT is obtained by projecting the convex hull into E^d,
- *divide and conquer* (D&C) – this is based on the recursive partition and local triangulation of the point set, and then on a merging phase where the resulting triangulations are joined.

3.1 Local improvement

This method is used mainly in 2D and is described for example in [GOR97]. We use this example to explain the main idea of this class. The initial non-Delaunay triangulation can be computed by some simple algorithm, e.g., a sweep-line one, which adds the points from the input set in an x-coordinate order. After each addition the algorithm joins the new point with the old visible points on the convex hull. The initial triangulation is then optimized to be a Delaunay one as follows.

The main idea is to swap the diagonals of strictly convex quadrilaterals in the triangulation to fulfil the following criteria, either the empty circle or the max-min angle. All internal edges of triangulation are added to the queue. Then they are processed in the following way:

```
while the queue is not empty do
  take the first edge e
  if the triangles sharing the edge e in the convex quadrilateral Q are not
  Delaunay then
    flip the diagonal of Q for the other
    add outside edges of Q to the queue
```

The success of the local improvement by edge flipping is guaranteed and the convergence proven. The generalization of flips into the higher dimensions is not entirely straightforward and the flipping from an arbitrary triangulation can get stuck before reaching the DT [GOR97, p.425]. This is probably the reason, why we did not meet any algorithm based on this method in 3D. The algorithms using the flips have moved to the class of incremental algorithms, where the convergence is sure.

3.2 Incremental construction

As the name of this class says, the triangulation is built step by step by adding the points from the input set into the existing triangulation. Then the added point and the existing triangulation are processed together to give a proper Delaunay triangulation at the end of each step. In the general step of the algorithm no unused point has to be enclosed inside the triangulation.

The whole process of the incremental construction starts with the first tetrahedron, which does not contain any of the input points inside it. The selection of the first tetrahedron depends on the particular method and will be described further. From the first tetrahedron the triangulation is built by adding vertices in a special order to expand this tetrahedron or – in the general step – expand the current triangulation. The vertices are added and the triangulation is processed until the input set is empty.

From above it is clear that this class of algorithms needs to know all the points in advance to create the first tetrahedron and to select the proper next point.

To speed up the algorithm, a preprocessing is often used to organize the points. Thus there are methods based on the lexicographical sorting of the input points [Joe91a] or on storing the input points in the uniform grid [FP95]. Instead of uniform grid, the sparse matrix was also used in 2D algorithms, but no use in 3D was observed. Hierarchical data structures such as oct-tree are not employed. The reason is probably that only local changes in the close neighbourhood of an added point are made during the processing of triangulation and the hierarchical structures are too huge for this purpose.

Let us describe several algorithms and their implementations based on incremental construction in more detail. The first one uses the flips in 3D (generalized idea of edge swapping from 2D). The second one uses no flips. Both of them were presented in [Joe91a]. The third one uses the regular grid to achieve a linear time expected complexity [FP95].

Use of local transformations

We have already mentioned the concept of diagonal flips in 2D, which was first used by Lawson. In 3D the situation is more complex. We have to inspect all possible ways of triangulating a set of five points (while in 2D we inspected all possible triangulations of only four points). This research was done by Joe [Joe91] and several algorithms based on the use of flips – sometimes also called local transformations – were proposed. Both algorithms

described in this section (named TRSPH1 and TRSPH2) have empirical time complexity $O(n^{4/3})$ or $O(n (\log n)^2)$ for sets of random points and $O(n^2)$ in the worst case, thus they are worst case time optimal.

But first, let us explain the principle of local transformations in 3D. The local transformation procedure is based on all possible configurations of five distinct non-coplanar 3D points *a*, *b*, *c*, *d*, *e*. These configurations are depicted in Fig. 3.1.

- configuration 1 – no four coplanar points, all points are in the convex hull; edge *de* intersects an interior of the triangle *abc*; the interior of convex hexahedron can be triangulated in two ways: either two tetrahedra *abcd* and *abce* or three tetrahedra *abde*, *acde* and *bcde* are created,
- configuration 2 – no four coplanar points, only four points are in convex hull; the convex hull creates a tetrahedron which contains the fifth vertex inside; this configuration decomposes the convex hull into four tetrahedra *abcd*, *abce*, *abde* and *acde*,
- configuration 3 – four points are coplanar and form a strictly convex quadrilateral; such a pyramid can be triangulated in two ways: either two tetrahedra *abcd* and *abce* or two tetrahedra *acde* and *bcde* are created,
- configuration 4 – four points are coplanar, but quadrilateral from configuration 3 degenerated into a triangle; there is only one possible triangulation with two tetrahedra *abcd* and *abce*,
- configuration 5 – four points are coplanar, but the strictly convex quadrilateral from configuration 3 is non-convex now; three points of it form a triangle, which contains the fourth point inside; only one possible triangulation with tetrahedra *abcd*, *abce* and *acde* can be created.

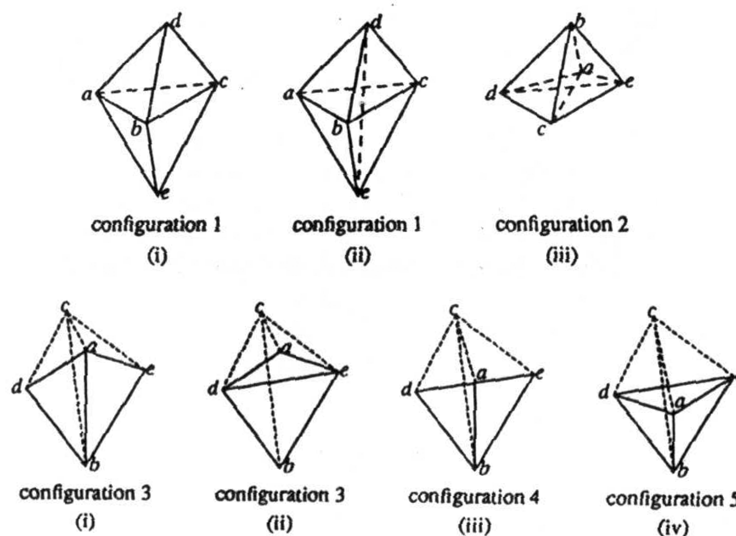


Figure 3.1: All possible configurations of five points [Joe91a].

Joe establishes several definitions:

- Local optimality with respect to sphere criterion – let us have two tetrahedra sharing a common face. Then the shared face is locally optimal if the circumsphere of one tetrahedron does not contain a non-shared vertex of the second tetrahedron in its interior. This automatically holds for both tetrahedra of the shared face. All interior edges in configurations 2, 4, 5 are locally optimal. In configuration 1 either

face *abc* or three faces *abe*, *bde*, *cde* are locally optimal. In configuration 3 either face *abc* or *cde* is locally optimal.

- Transformability – let us have two tetrahedra sharing a common face and let the points be in configurations 1 or 3. Then the shared face is transformable if it is in configuration 1(i) or in configuration 1(ii) where the third tetrahedron needed to fill the convex hull of points has to exist; or in configuration 3 where four coplanar points form boundary faces; or these four coplanar points form an interior faces and then two tetrahedra at the opposite side has to share also a common face. Otherwise the face is non-transformable.
- Pseudo-local-optimality – a triangulation is pseudo-locally-optimal if every non-locally-optimal interior face is non-transformable.

Joe makes several assumptions on his algorithm: the input points have to be sorted in lexicographical order of their coordinates. Then a slight reordering is possible if needed to avoid the first four vertices being coplanar and being able to form a valid tetrahedron. The first tetrahedron has to be empty and the ith point has to be outside the convex hull of the triangulation of the first i–1 points.

Several algorithms are offered in [Joe91a]. The first algorithm, called TRSPH1, is the basic version, which is further improved. The ith step of the algorithm is as follows: if a newly inserted point v_i is visible from a boundary face $v_a v_b v_c$ of the triangulation made in the previous step i–1, the face is pushed onto the stack and the new tetrahedron $v_a v_b v_c v_i$ is added to the triangulation. After processing all old boundary faces, the faces pushed onto the stack are processed in the following way:

```
while the stack is not empty do
  pop interior face vavbvc from the stack
  if vavbvc is still in the triangulation then
    find two tetrahedra vavbvcvd and vavbvcve sharing this face
    if the circumsphere of vavbvcvd contains ve in its interior then
      if vavbvc is transformable then
        apply appropriate local transform. in convex hull of points va…ve
        if neighbouring tetrahedra have to be transformed (config. 3) then
          apply local transformation to vavbvdvf and vavbvevf
        push each face on boundary of union of 2, 3 or 4 transformed
        tetrahedra onto stack if it is an interior face and not yet in
        stack (do the same for vavdve and vbvdve in case of configuration 3)
```

The proof that Delaunay triangulation is constructed by this algorithm is given in [Joe91]. From this proof the improvement of the presented algorithm was found. It lies in the reduction of faces, which are put onto stack during the local transformation procedure. It is sufficient to put onto stack only those faces, which were in the interior of the original triangulation, i.e. no face containing point *i* has to be put onto the stack. The new algorithm is called TRSPH2.

For sets of random points this improvement reduced the number of faces, which were locally optimal when tested for local optimality by about 60% and reduced the number of faces, which were non-locally-optimal and non-transformable when tested for local optimality by about 30%. In runtime the program TRSPH2 took only about 70% of time needed for TRSPH1 for random data, for problems containing O(n^2) tetrahedra the time ratio was closer to 80%. When the data structure for boundary faces was enriched by pointers to three neighbouring faces, the time needed to run TRSPH2 was reduced by a further 10%.

To store interior faces, which are to be checked, a stack or later a queue are used. The boundary faces are kept in a doubly linked list. A hash table with direct chaining and with a face as a key is used to store faces. The indices of the fourth vertices of tetrahedra incident to

a face are stored in each hash table entry. Hash table size is proportional to the number of faces in Delaunay triangulation. As a hash function was used $h(a,b,c)=(an^2+bn+c) \bmod M$, where $a<b<c$ are indices of three vertices of a face and the hash table size M is a prime number. So searching, insertion and deletion of a face can be performed in constant time on average.

The time relations of this algorithm are confirmed by our implementation experience – the main part of the running time is spent on flips. By speeding up other than flipping parts of the algorithm, no greatly better time is achieved. Although it is possible to generalize the flips into higher dimensions, this does not seem to be practical as it is too complicated. The parallelization of the algorithm seems improbable.

The algorithm without local transformations

The algorithm presented in the previous section can be also modified for use without local transformations, in [Joe91] it is called TRSPH3. The triangulation is constructed in a way similar to Watson's algorithm (see page 12), where non-locally-optimal and non-transformable interior faces do not occur and thus no local transformations have to be explicitly performed. The assumption of lexicographically sorted points with possible reordering holds here as well. The empirical time complexity of the algorithm is $O(n^{4/3})$ or $O(n (\log n)^2)$ for sets of random points and $O(n^2)$ in the worst case, thus it is worst case time optimal.

The ith step of the algorithm is as follows: all boundary faces $v_a v_b v_c$, which are visible from the ith point (remember that this point is always outside the current triangulation and the triangulation is Delaunay), are stored at the end of the queue. Then the faces in the queue are processed further to remove those tetrahedra, which are no more Delaunay after the ith point has been added. This step is similar to creating the gap in Watson's algorithm.

```
while queue is not empty do
  remove face vavbvc from the head of the queue
  if vavbvc is still in the triangulation then
    find the tetrahedron vavbvcvd incident on face vavbvc
    if the circumsphere of vavbvcvd does not contain vi in its interior then
      push face vavbvc onto stack
    else
      delete vavbvc and vavbvcvd from triangulation
      for each of the faces vavbvd, vavcvd, vbvcvd do
        if the face is a boundary face of i–1 and i triangulation then
          push face onto stack
        else if the face is already in the queue then
          delete face from triangulation
        else
          add the face to the end of the queue
```

Then the faces stored in the stack are processed in order to construct a new triangulation.

```
while stack is not empty do
  pop face vavbvc from the stack
  if vavbvc is a boundary face of i–1 triangulation with vi on the plane of
  vavbvc then
    delete the face from vavbvc triangulation
  else
    add tetrahedron vavbvcvd and 3 faces involving vi to the triangulation
```

The implementation of this algorithm is simpler and the code is shorter than in the algorithms mentioned in the previous sections, because part of the local transformations was omitted. The program is also faster than TRSPH2 by about 20%, although the time complexity remains the same.

The disadvantage of this algorithm, as Joe says himself, comes with degenerated cases when five or more vertices are cospherical. In finite precision arithmetic with roundoff errors this algorithm can lead to an invalid triangulation, in which there are overlapping tetrahedra or gaps. On the other hand Joe mentioned, that it is not possible for the algorithm TRSPH2 to produce invalid triangulation, since local transformations only change the tetrahedra in a constant volume. At worst it could lead to a triangulation which is not Delaunay in the floating-point arithmetic.

This algorithm is quite simple and easy to implement. Its parallelization seems improbable but generalization to a higher dimension is possible. Its unstability in degenerated or close-to-degenerated cases makes it less suitable for practical purposes. The use of some kind of exact arithmetic will make it really usable.

Shelling algorithm with a uniform grid

This algorithm was presented in [FP95]. It is a generalization of an algorithm for 2D Delaunay triangulation made by the same authors. The same main idea is also used in the InCoDe algorithm (Incremental Construction of Delaunay triangulation), see [CMS92], but the implementation is a bit different (see the end of this section for comparison). The referred time performance of the described algorithm is close to linear. The worst case time complexity is $O(n^{\lceil d/2 \rceil + 1})$, thus $O(n^3)$ for 3D space.

This algorithm uses a system of 3D cells to store the points from the input set. The number and size of the cells are derived from the number n of input points and their maximal and minimal coordinates (in other words from the min-max box, which is amplified a little, because no point must lie on its boundary). The cell size is computed as follows:

$$size = \sqrt[3]{\frac{(x\max - x\min)(y\max - y\min)(z\max - z\min)}{n}}.$$

It means that on average no more than one point lies within each cell. The authors note, that if the distribution of data points is known in advance, it is reasonable to adjust the *size* up or down to affect the performance of the algorithm.

The algorithm begins with creating a starting tetrahedron, which has to fulfil the Delaunay criterion. The first point can be selected arbitrarily, but a point “somewhere in the middle” is usually chosen because of program efficiency. The second point has to be the one closest to the first point, because the sphere with a diameter equal to the distance of the first and second points must not contain any other point from the input set in its interior. The system of cells prepared during the preprocessing stage is used to investigate the space in the vicinity of the first point to find the second one. The searching for the third point is similar. At the end of this stage we have the first triangle, whose circumsphere is empty.

The process of *range searching* is a general step of the algorithm, which finds the fourth point to any given triangular face, and makes it possible to create a Delaunay tetrahedron (the authors call such points Delaunay points). Because most of the time of the program is spent in searching for points to fulfil the Delaunay condition, the range searching is of crucial importance. The authors offer two versions of the searching, box-based and tunnel-based.

The first method is slower. It searches many more cells than is necessary, but it is easy to implement. It is based on a bounding box of cells, which is growing from the cell containing the centre of the triangle. In addition to the growing the bounding box has also changed according to the points already found and the circumscribed spheres they created. As well as the next method it uses the searching only in the outer halfspace of the face (i.e. the halfspace, where no tetrahedron is sharing this face so far).

The second method computes the normal vector of the face at first. Then it determines the *search direction* as the axis-direction of the fastest growth of the normal vector ($\pm x$, $\pm y$, $\pm z$). The cells are investigated in tunnels of search direction starting with the triangle centre. Also the neighbouring tunnels are visited and all of them work in cooperation with the bounding boxes of the spheres again. This method tries to search for points in the close vicinity of the face and thus to minimize the number of cells visited. It is about twice as fast as the previous one. But the program is long and complicated.

The final step of the algorithm is called *shelling*. For the shelling procedure let us imagine the current triangulation as only the boundary faces of it. If a new Delaunay point is found (remember it always lies outside the current triangulation), the new tetrahedron has to be built in. If the new tetrahedron touches some of the old ones except its own face, it causes some special cases to be solved. There are three special cases of touches:

- one-face touch – the base face (i.e. the boundary face, which was expanded) and the shared face are eliminated, two new faces are added,
- two-face touch – the base face and both of the shared faces are eliminated, one new face is added,
- edge-and-face touch – creating a new tetrahedron is not possible, because it will cause a gap in the mesh; the fourth point is stored and the next face from the list is then processed.

No other touch cases affect the algorithm.

To store points, a linked list of points is connected with each cell. It stores information such as point number, flag – whether it is used or not, point coordinates. Each cell contains an information about the number of its points. For the shell procedure the dynamic circular doubly linked list of faces is maintained. The face information consists of face pointers to neighbours, vertex pointers, boundary flag (boundary face will never have a fourth point), pointer to the fourth point. For checking the edge touch, the list of edges is also maintained.

This algorithm has good properties to be parallelized, e.g. see [TSBP93, CMPS92]. It works for arbitrary dimension [CMS92]. The authors do not say anything about numerical stability, which should be discussed in such algorithms. From our experiences from implementing the 2D version of the algorithm [KS98] we think that this algorithm is not very stable in degenerated cases when more than four input points are cospherical. The complications, which are elegantly avoided in Barry Joe's similar algorithm, come if the Delaunay point, which has already been used in the triangulation, is found. In degenerated cases, where more Delaunay points are available, the "bad" one can be chosen, which implies that the different touches do not need to be recognized properly and the algorithm can create overlapping tetrahedra or gaps in the triangulation.

The InCoDe algorithm in [CMS92] is nearly the same as the one described. It also uses the uniform grid to speed up the search for Delaunay points, but the shelling procedure is simpler. The faces are managed in a hash table. The shelling procedure runs in the same way, except all new faces that arise from a face expansion are stored in a so-called active face list. If a face appears in the list for the second time, such a face is removed, because it has changed from the external to the internal face and cannot be expanded anymore. This approach thus

does not need to test any type of touch, nor does it affect the principle of numerical unstability in degenerated cases of this type of algorithms.

Both of the mentioned algorithms were implemented by the students as semestral works for selected algorithmical methods course. Both of them were considered to be fairly difficult to implement and not too stable (especially the one with different cases of touches). Also the speed of them did not come up to our expectations.

3.3 Incremental insertion

This class of algorithms, similar to the previous one, works on the principle of adding the points from the input set one by one to the existing triangulation. But there are two fundamental differences. Firstly, the new points are inserted *inside* the current triangulation. Secondly, the order of added points is not important, which implies that the whole set of input points do not need to be known in advance. The triangulation is usually constructed in such a way, that all points are enclosed within a boundary tetrahedron (thus four new boundary points have to be added), so only the range of the data set need be known before the algorithm starts.

The common part of this class of algorithms is the location of the tetrahedron within the current triangulation, which contains the added point. This search can be done by simply inspecting all tetrahedra, or by using some data structure to speed up the searching process. Hierarchical structures are welcomed because they are natural in such tasks. This type of algorithm is naturally sequential.

Main types of algorithms of this class will be presented in the following sections; the algorithm, which uses retriangulation of a hole to insert a new point into the current triangulation, and the algorithm, which uses flips to achieve the same goal.

Watson's algorithm

The algorithm was presented in [Wat81]. It is also know under the name Bowyer-Watson [TSW99]. It works for arbitrary dimension d. The time complexity is $O(n^{(2d-1)/d})$ and space complexity $O(n^{(d-1)/d})$ for an input set of n points.

The algorithm uses a boundary tetrahedron to enclose the whole set of points. In Watson's implementation, an orthogonal tetrahedron was used, but any other one could be used as well.

After one point is inserted into the triangulation, all existing tetrahedra must be searched to find those, which contain the inserted point within their circumsphere. These tetrahedra will be deleted. Before doing this, the list of their 2-faces (triangles) is created. If any triangle occurs twice in the list, which means it is shared, it is dropped from the list. The rest of the triangles composes a boundary of simplicial polyhedron within the 3-dimensional triangulation – this is the "hole" or the "gap" from the words above. The inserted point lies inside the simplicial polyhedron.

If the problems with numerical precision occur in this part of program and it cannot be determined whether the inserted point lies inside or outside the sphere, computation in higher precision arithmetic could be used to make the right decision. If no alternative treatment is ready for use, the inserted point and the created list of triangles are abandoned. Maybe it is strange, but in its original version this algorithm does not accept such points: "Any data point found to be in non-general position is rejected because it leads to a non-unique tesselation. A point is in non-general position when the resolution of the software is not sufficient to

determine whether or not an intersection with a given circumsphere has actually occurred." [Wat81].

In the next step the polyhedron with boundary triangles has to be decomposed into new tetrahedra. This is simply done by joining the vertices from each boundary triangle and the newly inserted point into the new tetrahedra.

At the end of the algorithm the boundary tetrahedron must be removed. The Delaunay tetrahedra of the original data set are well separated from those boundary ones, because the boundary tetrahedra contain at least one of the boundary points as a vertex.

The algorithm uses a list of data points and a list of 4-tuples of indices of data points, which represent Delaunay tetrahedra. Each 4-tuple of indices is associated with a 4-tuple of real values representing the circumsphere: coordinates of its centre and the square of the circumsphere radius (for speeding up the execution by eluding the computation of square root).

Another implementation of Watson's algorithm for decomposition of the models into tetrahedra for finite elements analysis is mentioned in [Fie86]. This approach tries to cope with the problem of points in a non-general position, so called *insertion degeneracy*. The author offers three methods to solve the case when a point lies on a circumsphere. The point is "on" a sphere when its distance to a circumcentre is within a prescribed small epsilon of the circumradius. For checking whether the insertion of a point was correct, the volumes of the polyhedra before and after the insertion are compared. So called *volume degeneracy* is reported to tend to occur when the inserted point is nearly in insertion degeneracy. Also *secondary degeneracy* was reported, which means that more than four points are on or inside the circumsphere of newly created tetrahedra. Then the volume degeneracies tend to occur later in the algorithm.

Three methods for solving degenerated cases together with their results are as follows:

- definition of the point to be always inside or outside the circumsphere – fewer volume degeneracies occurred by taking points "on" as "inside" rather than "outside" the circumsphere,
- postponing of insertion of problematic points – new list of postponed points was created and after the original list of points was empty, the insertion started from the beginning of the new list; at the end only points which always cause the degeneracy are in the list; if used with previous strategy, it rarely failed,
- shifting and reinserting of the points – the shift is made in random direction ten times the epsilon as long as the point lies on the circumsphere; this approach is reported to sometimes create very thin tetrahedra, so called slivers; the creation of slivers by Watson's algorithm is also mentioned in [GD97]; both approaches solve this problem by their own scheme of generating the new points, which are then added to the original triangulation to avoid mentioned problems.

The author [Fie86] also notes it is not sure that by removing the boundary tetrahedron, the part of the convex hull of the original point set will not also be removed. He recommends selecting the boundary points carefully, but does not say how.

In fact, the use of boundary tetrahedron brings a problem how to choose its points. Let us assume, that their coordinates are always sufficient to enclose the original set. But still one problem remains. If the boundary points are chosen too close to the original set, tiny non-convexities can appear after removing the boundary tetrahedra instead of the assumed convex hull of original point set, because too close points influence the computation of empty circumsphere of the original point set. If points are chosen in greater distance, the algorithm becomes more unstable in usual floating-point arithmetic. There is a method in 2D [ŽKP01],

which solves this problem by modification of circumsphere test for boundary triangles, but we have not been able to develop its generalization into 3D yet.

Incremental insertion with local transformations

The idea of local transformations presented in the previous section can be combined with an incremental insertion to construct a 3D Delaunay triangulation in the following way. This algorithm is presented in [Joe91], where it is called TRSPH4. There are also alternative algorithms [Fac95]. The input points can be sorted, but it is not necessary; the selection of input points can be also randomized [Fac95].

Let us describe the *i*th step of the algorithm. To insert *i*th point into the existing triangulation the tetrahedron, which contains this point, has to be found. This search can be done naively by inspecting all existing tetrahedra. Then this step could take $O(i^2)$ time in the worst case, which could lead to the worst case time complexity of $O(n^3)$. Joe proposes a faster way of searching: to take any tetrahedron containing point $i-1$ as one of its vertices and walk towards a tetrahedron containing vertex *i*. A successful termination of this process is guaranteed.

When a tetrahedron *abcd* containing point *i* is found, there are three possibilities of mutual position of a point and a tetrahedron.

- if *i* lies in the interior of tetrahedron, then four new tetrahedra *abci*, *abdi*, *acdi* and *bcdi* are created and faces *abc*, *abd*, *acd* and *bcd* are put into the queue,
- if *i* lies on a face of tetrahedron, let say *abc*, then three tetrahedra *abdi*, *acdi*, *bcdi* are created and faces *abd*, *acd*, *bcd* are put into the queue; the same process is done with the tetrahedron *abce*, which is sharing the face *abc*,
- if *i* lies on an edge of tetrahedron, let say *ab*, then all tetrahedra sharing this edge (i.e. tetrahedra $abc_0c_1,\ldots, abc_{m-1}c_m$ where $c_0=c_m$) has to be divided; tetrahedra $ac_jc_{j+1}i$, $bc_jc_{j+1}i$ are created and faces ac_jc_{j+1}, bc_jc_{j+1} where $j=0,\ldots, m-1$, are put into the queue.

The faces stored in the queue are processed in a way mentioned before and local transformations are applied until the queue is empty. At the end of each step a 3D Delaunay triangulation of *i* points plus four boundary points is created.

At the end of the algorithm all tetrahedra containing one or more boundary points as a vertex are deleted. Joe like Watson notes that boundary points have to be chosen carefully to keep the convex hull of original points unbroken. Joe tested the algorithm on random sets up to 5000 points and on sets with $O(n^2)$ tetrahedra up to 250 points and he concludes, that coordinates of boundary points may have to be up to 10^5 times greater than those from the original set.

Two versions of the given algorithm were tested – one with sorted and one with unsorted input points. For random data sets the unsorted version is faster, it takes less than 80% of the time needed for the sorted version. For data sets with $O(n^2)$ tetrahedra the sorted version is the faster one by about 10%. But this algorithm always takes longer than the algorithm called TRSPH2. Joe also notes, that much more time is spent on making local optimality tests and applying local transformations than in walking through the tetrahedra.

This algorithm is described in more detail in the chapter dedicated to our own implementation of the DT triangulator.

3.4 Higher dimension embedding

As we have mentioned before, there is a close relation between Delaunay triangulation in d-dimensional space and convex hulls in $(d+1)$-dimensional space. This means that a Delaunay triangulation in 3D space can be constructed from the convex hull in 4D space. We now introduce the main idea of this approach [GOR97]. Because most people still live in 3D space and find it easier to imagine something in this rather than in higher dimensions, we are concentrating only on a 3D convex hull and thus on 2D Delaunay triangulation.

Let us identify E^2 with the plane spanned by the first two coordinate axes of E^3 and call the direction of third coordinate the *vertical* direction. Let us define the **lifting map**:

$$\lambda : E^2 \rightarrow E^3 \text{ by } \lambda(x_1, x_2) = (x_1, x_2, x_1^2 + x_2^2).$$

$\Lambda = \lambda(E^2)$ is a paraboloid of revolution about the vertical axis. Let CH be the convex hull of the lifted points $\lambda(P)$.

The Delaunay triangulation of P is exactly the orthogonal projection into E^2 of the lower faces of CH (a face is lower if it has a supporting plane with inward normal having a positive vertical coordinate). For a better explanation let us imagine that triangle $\lambda(s)\lambda(t)\lambda(u)$ is a lower facet of CH and that plane π passes through $\lambda(s)\lambda(t)\lambda(u)$. The intersection of π with Λ is an ellipse that projects orthogonally to a circle in E^2. Since all other lifted points are above the plane, all other unlifted points are outside the circle and *stu* is a Delaunay triangle.

This is the principle of creating a Delaunay triangulation from a convex hull. The convex hull algorithms are a fundamental part of computational geometry. There are many different convex hull algorithms, see e.g. [GOR97 chap.19]. The properties of DT algorithms by using this class of method, e.g. time and space complexity, possible parallelization of the algorithm, the necessity to know all the data in advance, etc., totally depend on the convex hull algorithm.

The approach of higher dimension embedding is used e.g. in Qhull implementation, see [Bar96].

3.5 Divide and conquer

Divide and conquer is a general approach for many types of different algorithms. The main idea is to split the problem to be solved into two subproblems and solve them in the same way. At the lowest level the problem is simply solved and then comes the necessity to merge the solutions at the same level to provide the result for the level one step higher. Thus, there are two main parts of this algorithm class: recursive splitting and merging.

Divide and conquer algorithms are very often worst case time optimal, but it of course depends on the algorithms used inside. D&C algorithms can be also naturally parallelized and used in higher dimensions. The implementation is usually difficult. It is always good to know something about the input data in order to split the set into similarly large subsets, otherwise the parallel execution is not balanced.

In the following section a DeWall algorithm is presented, which uses a clever idea to avoid the complications of joining phase in 3D – it sounds foolish, but it joins the halves before their splitting.

DeWall algorithm

As a member of a group of D&C algorithms the DeWall (Delaunay Wall) algorithm from [CMS98], which is presented here, is based on the idea of InCoDe algorithm [CMS92]. The DeWall algorithm is generally used in arbitrary dimension *d*. Its worst case time complexity is $O(n^3)$, the expected time complexity is subquadratic. The main idea of D&C algorithms is to divide an input set of points P into subsets P_1 and P_2, make recursively a solution of these subsets and merge the partial results S_1 and S_2 to build solution S. The main problem of such a solution is the merging phase of two sub-solutions, because both of them have to be modified. In 2D it does not cause much trouble thanks to the possibility of explicitly ordering the edges incident to a vertex, but in higher dimensions this ordering is not possible.

The solution in [CMS98] exchanges the phases of the classical D&C concept. The set of points is split by the splitting plane into two subsets of points. The tetrahedra of the imaginary triangulation are thus separated into three disjointed subsets:

- tetrahedra intersected by the plane α (they are called the *simplex wall*),
- tetrahedra in the left halfspace of α,
- tetrahedra in the right halfspace of α.

The simplex wall can be chosen as a valid merging triangulation if

- each tetrahedron of it is also in the final triangulation,
- by subtracting the simplex wall from the final triangulation we get two disconnected simplicial complexes.

The whole DeWall algorithm can be briefly written as follows (see also Fig. 3.2 below):

```
select the dividing plane α
split P into two subsets P1 and P2 and construct triangulation Σα
recursively apply DeWall on P1 and P2 to build triangulations Σ1 and Σ2,
return the union of Σα, Σ1 and Σ2
```

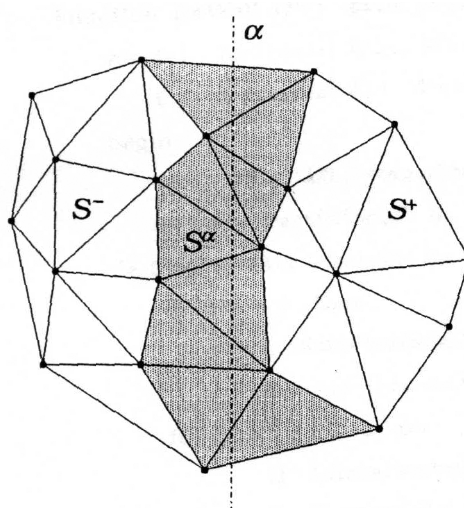


Figure 3.2: An illustration of the divide and conquer approach [CMS98].

Let us describe each step in more detail.

The splitting plane α is cyclically selected as a plane orthogonal to the axes of the E^3 space (i.e. x, y, z). Two subsets of input points obtained by the splitting should be of comparable cardinality. The median of the set is usually used for this purpose.

The simplex wall (tetrahedra intersected by α) is constructed by a simple incremental algorithm (already mentioned algorithm InCoDe). The whole process starts with the following construction of the first tetrahedron. The point p_1 is the one closest to α. The point p_2 has to lie on the other side of α and the third point p_3 is selected in such a way that circumcircle of triangle $p_1p_2p_3$ has the minimal radius. Thus the triangle (i.e. 2-face) is obtained from the edge (i.e. 1-face) and the whole process continues until the required tetrahedron (i.e. d-simplex in general dimension) is constructed.

Then the adjacent simplices are constructed according to the Delaunay definition by searching for the Delaunay points. The point, which is selected as the Delaunay one, is the point, which minimizes the function *dd* (Delaunay distance). The $dd(f,p)$ for face f and point p is defined as

- r if the centre of the circumscribed sphere is in the halfspace of f and p,
- $-r$ otherwise,

where r is the radius of the circumscribed sphere around f and p.

To speed up the searching for Delaunay points, the uniform grid is used. The space of input points is divided into a uniform grid where the number of cells is equal to the number of points. The search is then restricted to those cells contained in the bounding box of the circumscribed sphere with minimal radius. Another technique is also mentioned, where the bounding boxes are replaced by the union of cells made by a sphere scan conversion algorithm. This approach is not used in implementation, because the number of searched cells is not significantly less, while the overheads of this technique are too big.

In the implementation the lists are used to store the faces. The following three disjoint so-called active faces lists (AFL) are used:

- AFL_α: the triangles intersected by the plane α,
- AFL_1: the triangles with all of the points in P_1,
- AFL_2: the triangles with all of the points in P_2.

For each tetrahedron from the simplex wall its triangles are stored in the suitable face list. The simplex wall is built by removing and expanding the triangles from AFL_α. The wall is finished when the AFL_α is empty. The algorithm is then recursively applied to the pair (P_1,AFL_1) and (P_2,AFL_2), unless all the active face lists are empty.

Both the parallelization and the use in higher dimension are natural for this type of algorithm. The knowledge of all input data is always necessary. The numerical stability of it depends on partial algorithms and as we mentioned before, the expected numerical stability of InCoDe algorithm is not so great.

The chapter concerning the algorithms of Delaunay triangulation is at an end. We have presented different approaches for the construction of DT in 3D space. We have shown, that not each idea (local improvement) can be generalized from 2D space to a higher dimension. We also met the question of numerical stability, which will be discussed in more detail at the end of the thesis.

4 Our implementation of DT triangulator

We decided to implement the algorithm based on the incremental insertion with local transformations (flips). Reasons for this choice were as follows: this type of algorithm is one of the most popular and most used in practice [Joe91a, Fac95, Müc95]. The algorithms of this class always converge. Incremental algorithms with the flips are known to be more robust than the same algorithms without them. Input can be randomized to achieve good expected time complexity. The algorithm is relatively easy to implement. It can also be modified for other types of triangulations (e.g. regular triangulations [Fac95]). Moreover, we have had good experience with this type of algorithm from 2D.

4.1 Explanation of the algorithm

Let us describe the main points of the algorithm, which was briefly mentioned in the chapter dedicated to DT algorithms. It is based on *incremental insertion*, i.e., it works on the principle of inserting new points one at a time into the existing triangulation. For each inserted point it is necessary to find the tetrahedron which contains this point. This tetrahedron must be divided; it becomes a *parent* (it means, that it is no more valid in the triangulation and it is useful only for searching purposes). Newly created tetrahedra become its *children*. The search can be done naively by inspecting all existing tetrahedra – the complexity of this process can be seen in the worst case $O(n^2)$ because the number of the tetrahedra can be $O(n^2)$. It is better to use some data structure for speeding up the searching process. A proper structure for this purpose is **DAG, directed acyclic graph.** More precisely, it is the tree-like structure, where the root represents a big tetrahedron comprising the whole scene, each node corresponds to a parent tetrahedron and the sons of it correspond to the children tetrahedra. Leaves belong to those tetrahedra, which have not been divided (i.e., current triangulation). Although the DAG in this case is similar to a tree, it cannot be simplified to it, because one child can have more than one parent in our application.

If we want to find the tetrahedron containing the inserted point, we take the root in DAG and then test the mutual position of the point and children of this tetrahedron. When the relevant child is found, the whole process is repeated until a leaf of DAG is reached. This last tetrahedron contains the inserted point and must be divided. The method of its division depends on the mutual position of the tetrahedron and the point.

There are three such positions, see Fig. 4.1. If the point lies inside the tetrahedron, this tetrahedron is divided into four new tetrahedra. If the point lies on the surface of the tetrahedron (on one of the boundary triangles), this tetrahedron is divided into three new tetrahedra (if there is a neighbour which shares divided triangle, such tetrahedron must be divided, too, see Fig. 4.1b). If the point lies on one of the edges, the tetrahedron is divided into two new tetrahedra. In addition, every tetrahedron that shares this edge must be divided, too. This operation doubles the number of the tetrahedra around the edge. That is the reason why the number of the tetrahedra can be $O(n^2)$ in relation to the number of the points if too many points lie on the edges of the triangulation.

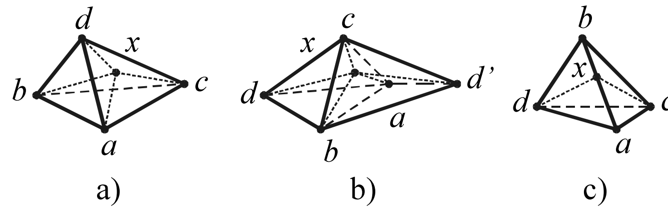


Figure 4.1: A point inside tetrahedron.

If we use only this incremental method, empty circumsphere property is not ensured and the resulting triangulation has a lot of thin tetrahedra, which are bad for applications. To reach the Delaunay triangulation, the idea of local transformations [Joe91a] is incorporated. After a new point is inserted, the obtained geometrical configuration is inspected to see whether or not it fulfills the Delaunay criterion. This means that each new tetrahedron has to have an empty circumsphere – if no five points lie on the sphere then the sphere circumscribed to each tetrahedron must not contain any other point inside. If there are five or more points lying on the sphere, the triangulation is not unique – there exist several triangulations and each of them can be called Delaunay's. If Delaunay property is not fulfilled, the surroundings of newly created tetrahedra should be locally transformed.

However, the transformation is sometimes impossible. The decision whether the transformation is possible is based on the tetrahedra configuration formed by the set of five points. As we have seen in Fig. 3.1 and as it was described before, there are five such configurations.

From Fig. 3.1 it is clear that configurations 1 and 3 can be made in two ways. The convex hulls of these configurations are the same, while the interior is different. Which one of two possibilities is better is determined by the value of the optimization criterion, e.g., the Delaunay criterion.

These two ideas (the incremental algorithm and the local transformations) are joined in one algorithm as follows. First it is necessary to put the whole set of the input points into one large tetrahedron – we must add four special points to the input set. Coordinates of these points are obviously chosen as closely to the point set as possible to avoid problems with numerical precision and so distantly to enclose all the set. Points $(kM,0,0)$, $(0,kM,0)$, $(0,0,kM)$, $(-kM, -kM, -kM)$ are suitable, when M is taken as the maximal absolute value of the extremes in x, y, z directions and k is in the order hundreds or thousands.

The incremental algorithm can start now by inserting the first point from the set. It divides the first tetrahedron in the way described above. After creating new tetrahedra and new triangles, it is necessary to optimize the current triangulation as a newly inserted point might cause non-optimality of some triangles. Triangles that must be checked form the convex hull of the current configuration. All these triangles are inserted into the queue and step by step are checked to see if they are changeable – i.e. that they form a changeable configuration (configuration 1 or 3). If they are, a Delaunay test is made. If the current configuration does not pass, it is changed to the other possibility and all convex hull triangles of this configuration are inserted into the queue, because the flip could cause another non-optimality in the mesh.

In this way, local optimizations are processed after one point is inserted. The convergence of this process and avoidance of the cycling of flips is proven. If the current mesh is locally optimal and nothing can be improved, the next point can be inserted. The whole algorithm can be written as follows:

```
1. while (input set is not empty) do
2.   insert a new point
3.   find and divide the tetrahedron
4.   do local transformations
```

Step 4 is more detailed as follows:

```
insert the triangles from the convex hull of the divided tetrahedron into
the queue
while (the queue is not empty) do
  take a triangle from the queue
  get triangle configuration
  if (configuration is 1 or 3) then
    if (triangle is not Delaunay) then
      change the configuration
      remove old triangle from the queue
      insert new triangles into the queue
```

At the end of the algorithm, it is necessary to remove the big boundary tetrahedron, created at the beginning, and all the tetrahedra that coincide with any of four vertices of the big tetrahedron. This is very easy because Delaunay triangulation forms the convex hull of the input point set, so the above condition can be satisfied by removing all those tetrahedra and triangles that coincide with one of the four added points.

The theoretical complexity of this algorithm can be derived in the following way: suppose the worst case in the number of tetrahedra, which is $O(n^2)$. If we use DAG for the search speedup, the proper tetrahedron can be found in $\log(n^2)$, i.e. $O(\log n)$. Subdivision of a tetrahedron is done in constant time, so an important part of the time will be spent only on local optimizations. Suppose again the worst case, i.e., when all tetrahedra must be optimized at each step. So the resulting worst-case time complexity for n points is $O(n(n^2 + \log n))$, i.e. $O(n^3)$. The expected time complexity is $O(n \log n)$, because of the expected linear number of tetrahedra in the mesh and constant number of tetrahedra to be optimized for the set of uniformly distributed points.

Now several words about the data structures used in our implementation. At the beginning, only the number of input vertices and their coordinates are known. Hence we used the array allocated one time in runtime to store all the points. As we already know, the number of the tetrahedra or the triangles is not known and cannot be computed in any way in advance. Therefore, bidirectionally linked lists with mutual references were used. The tetrahedron structure was adapted for composing DAG (to speed up search).

4.2 Results of implementation

The whole algorithm was implemented in Borland C++ Builder 3.0 environment under Windows NT and tested on PC PIII 450 MHz, 1GB RAM. Randomly generated data sets of uniform, gaussian and cluster distribution were used, while the number of points varied from 5000 to 100000 (cluster distribution means, that points are distributed into clusters, each cluster has a gaussian distribution of points).

Computation time, obtained as an average for five data sets, is shown in Table 4.1 and Fig. 4.2.

Vertices no. / distribution	5000	10000	50000	100000
Uniform	4.407	9.203	51	108.1
Gauss	4.453	9.265	51.05	110.9
Clusters	4.407	9.578	54.72	111.3

Table 4.1: Computation time in seconds measured on sets with a different number of points and different point distribution.

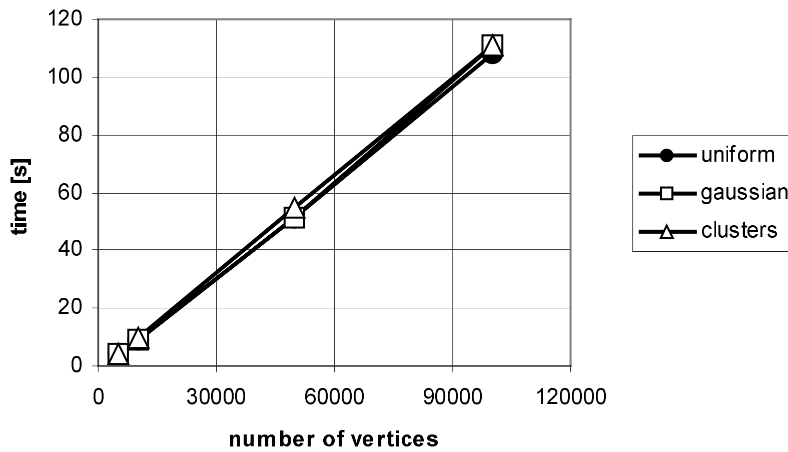


Figure 4.2: Computation time from Table 4.1.

The graph in Fig. 4.3 tries to examine the time complexity of the program. On the x-axis there is the number of points in the set. On the y-axis there is the ratio of time actually spent and the expected time complexity. If the resulting curve is decreasing, the time complexity of the program is better than the one in the formula denominator. Depicted curves show that for the uniform distribution of points, the expected time complexity is as good as in [Joe91a], i.e. $O(n(\log n)^2)$ or $O(n^{4/3})$ and, moreover it is better than $O(n \log n)$. It is also visible from Fig. 4.2 that the expected time complexity seems to be linear.

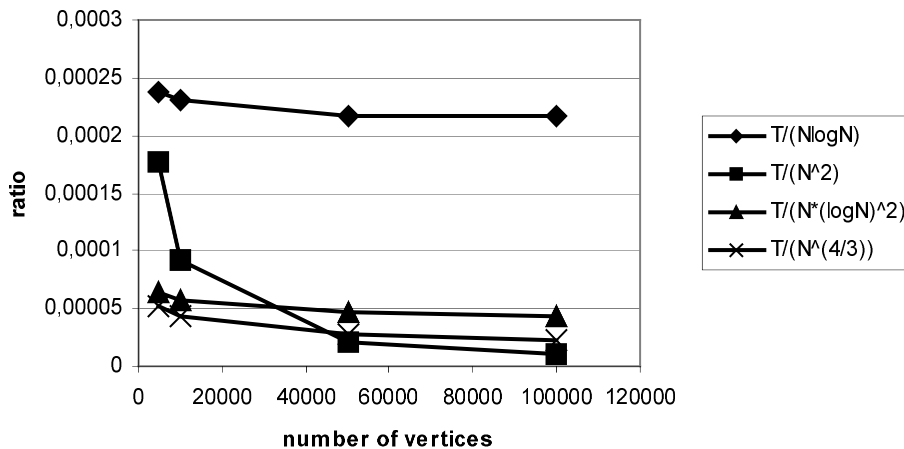


Figure 4.3: Time complexity.

Fig. 4.4 examines time relations in the main parts of the program. *IncStep* is the program part which searches for the adequate tetrahedron in the DAG and subdivides it. *OptimizeMesh* makes the main local optimization loop. And *CleanTheMesh* is the part destroying the DAG and removing the boundary tetrahedron. It can be seen that the main part of the time is spent on optimizing. The importance of the time needed for removing auxiliary data structures grows with the increasing number of points – this observation leads to the idea of using block-allocation of memory instead of member-allocation.

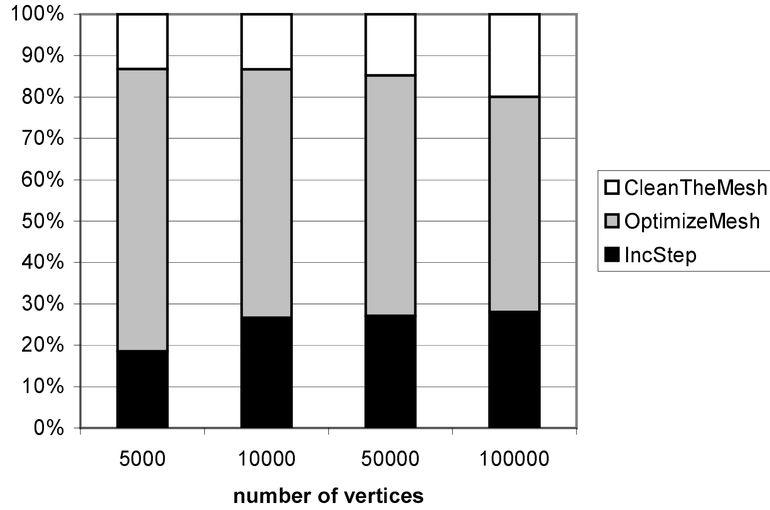


Figure 4.4: Time relations in the program.

The next observations from implementation concern the number of nodes generated in the DAG data structure. As previously mentioned, linear time complexity was observed as well as the linear dependence of the resulting number of tetrahedra. The number of tetrahedra was about 6.5 times the number of points. The number of all nodes generated in the DAG structure was not greater than 50 times the number of points, but on most occasions it was lower (between 40 and 50). These numbers can serve for memory requirement estimation. The number of performed flips was on average also nearly constant for every data set.

5 The quality of the mesh

5.1 Measuring tetrahedra properties

How can we evaluate the quality of the created triangulation? The mesh has good quality if the shape of the tetrahedra is good. Which shape is good? For example, for interpolation on tetrahedral mesh it is bad to have tetrahedra with very long edges because the distant vertices would have a weak mutual influence. The best shape is the equilateral tetrahedron. How "far" a created tetrahedron is from this ideal shape is told in different ways by the following properties [Joe91b, LJ94].

- **(Relative) volume** This property has no exact meaning in the sense written above, it only informs us about the size of created tetrahedra. The relative volume of the current tetrahedron is computed as its real volume divided by the value of maximal volume in triangulation.
- **Radius ratio** Computed as $3*r/R$ where r is the radius of the inscribed sphere, R is the radius of the circumscribed sphere. For the equilateral tetrahedron this value is equal to 1, for any other shape it is somewhere between 0 and 1.
- **Solid angle** Because the angle properties are very important for the shape of a tetrahedron, we decided to watch the behaviour of the solid angles of a tetrahedron. The solid angle is the area of a spherical triangle created on the unit sphere whose centre is in the tetrahedron vertex [Rek95]. Vertices of the spherical triangle are found as the intersection of the unit sphere and the edges incidating with the tetrahedron vertex. This criterion is a naturally generalized feature of 2D triangulations – the inner angle of a triangle.
- **Minimum solid angle** This property tells us whether the tetrahedron is flat or not because the sum of all four solid angles in a tetrahedron is less or equal to 2π. This property was also used because of observation of the behaviour of the minimal inner angle of a triangle generalized into 3D, which is a very important feature of Delaunay triangulation.
- **Maximum solid angle** This property is also very important, because it reports whether the tetrahedron is flat or not, but from the opposite point of view of the minimum solid angle. If all four vertices of the tetrahedron are coplanar, the maximum solid angle is equal to 2π.
- **Edge ratio** Computed as E/e, where E, e is the length of the longest, shortest edge of the tetrahedron, respectively, and is always greater than (or equal to) 1. It tells us how big is the difference between the longest edge E and the shortest edge e, and so how "nonequilateral" this tetrahedron is.
- **Aspect ratio** Computed as R/E, a ratio of the circumsphere radius and the longest edge of a tetrahedron. A high value tells us that the tetrahedron is close to being flat and is "pressed" near to the surface of the circumsphere.

5.2 Properties of Delaunay tetrahedra

All the properties listed above were tested on sets of points with a different number of points and with a different distribution of points. Most of the graphs do not differ as much as

expected. It seems that distribution of vertices has no great influence tetrahedra shape if the number of created tetrahedra is close to linear.

Most of the triangulations have similar histograms of **volume**. They contain most of the small tetrahedra and very few greater tetrahedra.

You can see typical histograms of the **radius ratio** made from five different data sets in Fig. 5.1. The typical shape is that the maximum of the graph is almost at the same place in all graphs. The maximum is closer to higher values – it means that there are more tetrahedra with good ratios than those with bad ratios.

Figure 5.1: Radius ratio.

Histograms of **aspect ratio** and **edge ratio** are not easily readable because of their great variation in values. In edge ratio, see Fig. 5.2, most of the values are placed in low ratios, very few tetrahedra have a ratio greater then 10. The histograms of these properties did not bring any benefit to us. Thus, we decided not to use them in further experiments.

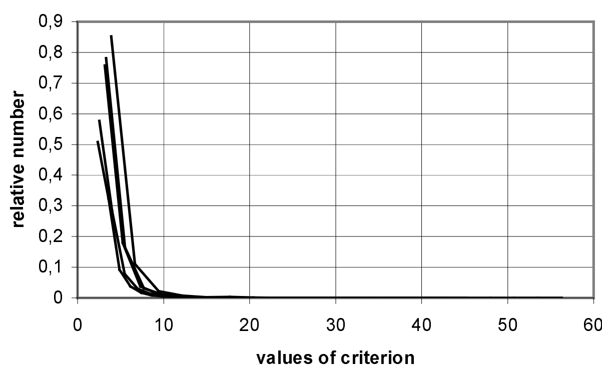


Figure 5.2: Edge ratio.

Minimum solid angle has its own typical graph, much like that of radius ratio, see Fig. 5.3. Most of the tetrahedra are placed at lower values. This criterion says that these tetrahedra are quite different from equilateral ones.

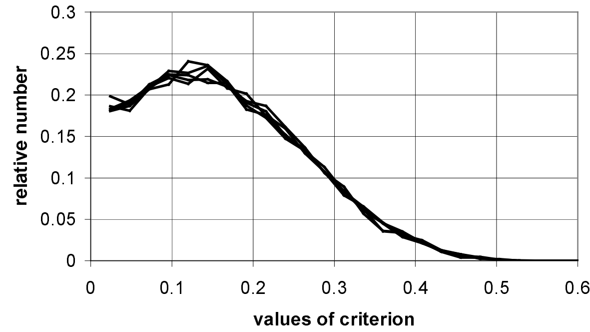


Figure 5.3: Minimum solid angle.

The graph of **maximum solid angle**, see Fig. 5.4, is only presented here to illustrate the maximum solid angle situation and to compare its behaviour with the minimum solid angle (see Fig. 5.3).

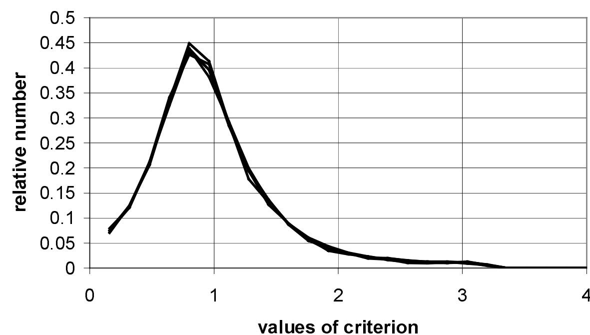


Figure 5.4: Maximum solid angle

We can say that by measuring presented tetrahedron properties it was found that Delaunay triangulation produces mostly good shapes of the tetrahedra (a small number of tetrahedra have a bad edge or aspect ratio, mainly maximal values of these criteria, but we think those tetrahedra are the ones in convex hull and their quality is thus limited by the boundary). Histograms of all criteria for different distribution of the vertices and for their different numbers were surprisingly similar when the number of created tetrahedra was nearly linear.

6 Post-optimization of DT

The previous chapter was closed by the statement, that 3D Delaunay triangulation produces mostly good shapes. But from the histograms presented there, we can imagine a triangulation whose quality can be better. At the beginning of this thesis we also said, that properties of DT in 3D are not as brilliant as those in 2D, for example it seems that in 3D the DT does not satisfy any optimal angle condition [Joe91b].

Thus it seems that there is a possibility of improving the quality of Delaunay triangulation. It is reported in [Joe91b] that a sort of post-processing can be applied to a completed Delaunay triangulation to improve the quality of the tetrahedra. The main idea is simple: the use of the flips on a completed Delaunay triangulation, but now with other than empty circumsphere criterion. After this post-optimization is finished, we obtain a triangulation, which is locally optimal to the given criterion. By using this process, two things are important for the resulting mesh:

- starting mesh of post-optimization,
- the order of performed flips.

It is noted in [Joe95] that if the starting triangulation is far from Delaunay, results are poor. But if the starting triangulation is Delaunay or nearly Delaunay, the quality of resulting mesh is better.

As mentioned before, this approach to an improvement of Delaunay mesh was presented in [Joe91b]. As a post-optimization criterion it was used only the maximizing of the minimal solid angle there. The Delaunay mesh was really improved in both of the observed properties – relative volume and minimum solid angle. We decided to extend this method by using various post-optimization criteria – mainly those, which directly improve some of the properties presented in the previous chapter, but also others – to find out, whether post-optimization used according to different criteria gives better or worse shapes of the tetrahedra. We also watched more properties than Joe in order to find out more about the quality of the resulting meshes.

The whole post-optimization of Delaunay triangulation can be described by the following algorithm.

```
put all non convex hull triangles into the queue
while (queue is not empty)
  do flips according to a selected criterion
```

We decided to use three main groups of criteria. First we started to improve the geometrical properties directly. For example, we would like to have a triangulation, which locally fulfils the condition of maximal volume (we want to reach locally a maximal value of volume). Thus, we use this condition in a local transformation decision, i.e. in one step of the algorithm the flip is done and the configuration is changed if the other configuration has a greater value of maximal volume than the current configuration. Furthermore these geometrical criteria can be divided into two groups – *simple* and *compound* ones. Both of them are described later.

The second group of criteria does not deal with any geometrical feature. It only increases or decreases the number of tetrahedra in the current mesh.

The third group of criteria (in fact there is only one criterion) deals directly with badly shaped tetrahedra. It is well known, that Delaunay triangulation is not able to remove special types of tetrahedra – so called *slivers*, which have a flat shape and small solid and dihedral angles. The reason for this is that Delaunay criterion optimizes the radius of circumsphere to

be minimal, but these badly shaped tetrahedra have small radius of circumsphere by themselves. Thus they cannot be removed and they remain in the Delaunay mesh. Our goal in this branch of post-optimization is to find such tetrahedra and try to remove them. How can we recognize this type of tetrahedra? We decided to create a tetrahedra classifier, whose classification is based on the number of short edges of a tetrahedron. There are also other methods of tetrahedra classification, see [CDEFT, BCER95].

6.1 Post-optimization by geometrical criteria

Simple criteria

In this case it is enough for us if the measure or value of a geometrical property satisfies only one simple condition – it means that by making the flips we watch only the maximal or minimal values. In other words we want to achieve an extreme value for this property. Thus, a configuration with a greater maximal value (for maximizing) or a smaller minimal value (for minimizing) is selected.

An expected improvement in the shape of the tetrahedra with such criteria is not very high, because if we watch only one extreme, the second extreme tends to get worse. The criteria below were selected:

- MaxVol – maximizing relative volume property of tetrahedron,
- MaxRR – maximizing radius ratio,
- MinAR – minimizing aspect ratio,
- MinER – minimizing edge ratio.

Compound criteria

On the other hand, compound criteria used in the post-optimization move extreme values in the opposite direction. By making the flips, for example, maximal values are watched, but the one with a lower value is selected as a good configuration.

The expected improvement in tetrahedra shape with this type of criteria is greater, because the main idea of this process is to move the level of all tetrahedra in the mesh towards a better quality. Selected criteria are:

- MaxMinVol – maximizing minimal volume property of tetrahedron,
- MaxMinRR – maximizing minimal radius ratio,
- MinMaxAR – minimizing maximal aspect ratio,
- MinMaxER – minimizing maximal edge ratio,
- MaxMinSA – maximizing minimum solid angle,
- MinMaxSA – minimizing maximum solid angle.

6.2 Changing the number of tetrahedra

Next, there are two criteria, which only change the number of tetrahedra towards the minimum or maximum values by performing only one type of flips. To minimize the number of tetrahedra, only the flips, which transform three tetrahedra into two are made, but to maximize the number of tetrahedra, only the flips transforming two tetrahedra into three are made.

- MinThNo – minimizing number of tetrahedra,
- MaxThNo – maximizing number of tetrahedra.

6.3 Removing of slivers

This type of criterion was introduced to try another approach towards improving Delaunay triangulation. The condition used in deciding whether or not to make the flip is simple. Both possible configurations are inspected for the number of slivers, which are provided by the tetrahedra classifier. Finally, the flip is done if the number of slivers in the other configuration is smaller than in the current configuration. It means we make the flip if we are able to decrease the number of slivers.

The classification of tetrahedra shapes is based on the number of short edges and by using such a classification four classes can be declared.

- **class 0**: no short edge or more than three short edges – in other words tetrahedra, which are classified as equilateral or nearly equilateral,
- **class 1**: one short edge,
- **class 2**: two short edges,
- **class 3**: three short edges.

Classes 1, 2 and 3 belong to badly shaped tetrahedra (the tetrahedra are too flat or too sharp). Nearly equilateral tetrahedra in class 0 seem to be well shaped– but only for the first sight (see below). It is good to say that a tetrahedron cannot have more than three short edges.

The procedure of classification is as follows. Firstly the ratio of the shortest to the longest edge is computed. If this ratio is greater than the *coefficient of equilaterality* (which is the first parameter of classifier and which belongs to the interval <0,1>) then the tetrahedron is classified as equilateral. Next, the edges are separated into two clusters. The rule for the separation is the greatest difference between two following edges, which are sorted in increasing order according to their lengths. The number of edges in the cluster of shorter ones gives the number of the appropriate class of the tetrahedron.

Sliver tetrahedra are classified in the second step. The tetrahedra, whose edges have similar lengths, seem to be shaped well. But such tetrahedra can have small height and thus they can be flat, i.e. poorly shaped. According to this condition, the classifier has to have a second parameter, which will serve for recognition of slivers. This parameter is tetrahedron height, which is built from a face with the largest surface. As before, this parameter is again in relation to the longest edge: if the height is smaller than the longest edge multiplied by the *coefficient of height*, then the tetrahedron is classified as a sliver. In fact, class for slivers is a part of class 0, but it is separated from it and constitutes its own class. “Equilateral” tetrahedra, which are not slivers, remain in class 0.

The parameters coefficient of equilaterality and coefficient of height were empirically set to the values 0.1 and 0.25, respectively.

Before presentation of the results of post-optimization by removing slivers, let us take a look at the situation of tetrahedra classes in Delaunay and post-optimized meshes. In Fig. 6.1 you can see on the left graph situation of classes **slivers** and **equilateral** and on the right graph situation of classes with **one**, **two** and **three** short edges. There are the columns, which represent the amount of tetrahedra that belong to each class.

The first column called Delaunay depicts the situation of original mesh. The second column called RemSliv shows the number of tetrahedra after removing those tetrahedra that belong to sliver class. Third column called RemSlivAll is the result of our next experiment, not to remove only slivers, but all tetrahedra that do not belong into the equilateral class. As you can see, the number of bad tetrahedra in classes one, two and three was reduced slightly by this method. But it did not have any influence on the resulting histograms, because the number of such tetrahedra was minimal compared to zero class.

These graphs can be understood as an experimental proof of the quality of Delaunay triangulation, which is reported to have a minimal number of tetrahedra in classes one, two and three. Also you can see, that not all slivers were removed, due to the kind of optimization method, which is not global. And thus only the local extreme could be reached.

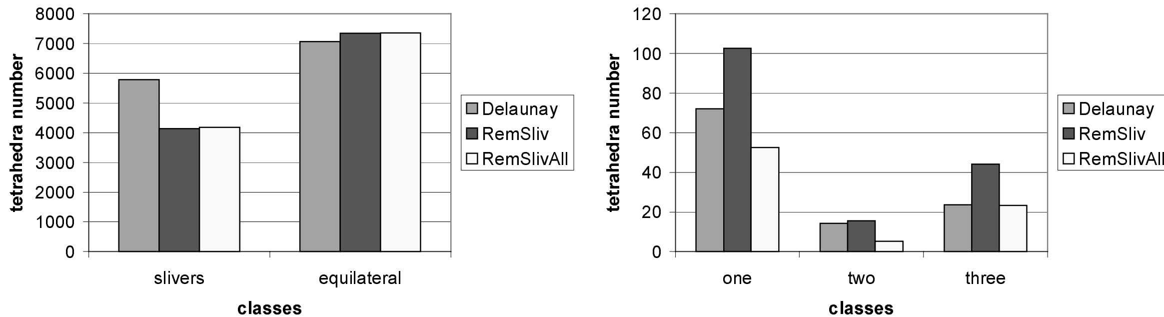


Figure 6.1: Classification of tetrahedra.

6.4 Results of geometrical criteria use

The properties, which were used in the previous chapter to measure the quality of Delaunay triangulation, are now used to measure the quality of post-optimized triangulation. The results are presented as a comparison of the properties of Delaunay and the post-optimized meshes to make a clear image of differences and improvement.

All post-optimization criteria mentioned before were used in order to create improved meshes the properties of which were inspected. Our goal is to find such post-optimization criteria which most are able to improve each property.

The tests were performed on five different data sets of 2000 uniformly distributed points as a representative example. The graphs are very similar for a different number of points.

Volume

Two types of histograms were reached by post-optimization. Both of them are shown on the graphs in Fig. 6.2. The first one compares the Delaunay mesh to the mesh post-optimized to MaxRR criterion. One of them is a smooth curve, while the other is a curve with a small peak at the lowest values. As it can be seen, Delaunay tetrahedra are better than those created with MaxRR criterion because there is less tetrahedra with small volume in Delaunany case. And our desire is to have as much tetrahedra with big volume as possible.

Slight improvement of volume histogram in the part with the peak is reached by MaxMinVol criterion. It is not a general rule but most of the simple criteria lead to a histogram looking like a smooth curve while compound criteria create a curve with a peak.

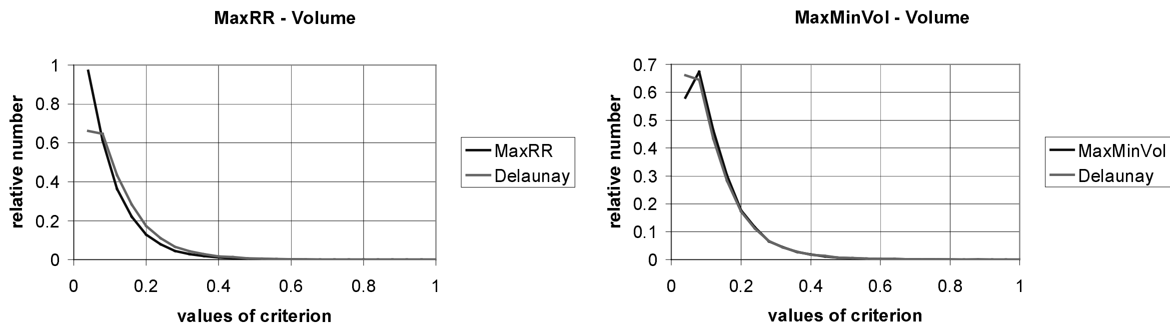


Figure 6.2: Histograms of volume.

Radius ratio

This property has three characteristic histograms. First one is a really improved Delaunay curve, as it is shown on the first graph in Fig. 6.3. It is clear that some of the tetrahedra were moved from the lower ratios towards the higher – better – ones, which was our goal. This improvement is reached by all of the MaxMin, MinMaxAR and MinThNo criteria.

The second type of curve is not shown here – the improvement was not successful, wrong tetrahedra has moved in the bad direction. There are more tetrahedra with lower ratios than in the Delaunay mesh. But still, the curve has a small local minimum in the right part. Such curves are produced by MaxRR, MinAR or MinMaxER criteria.

The third type of curve, which is shown on the second graph in Fig. 6.3, misses even this local minimum and only decreases. It is the worst type of histogram because all the tetrahedra are pressed close to very low ratios. It is produced for example by MaxThNo or MaxVol criteria.

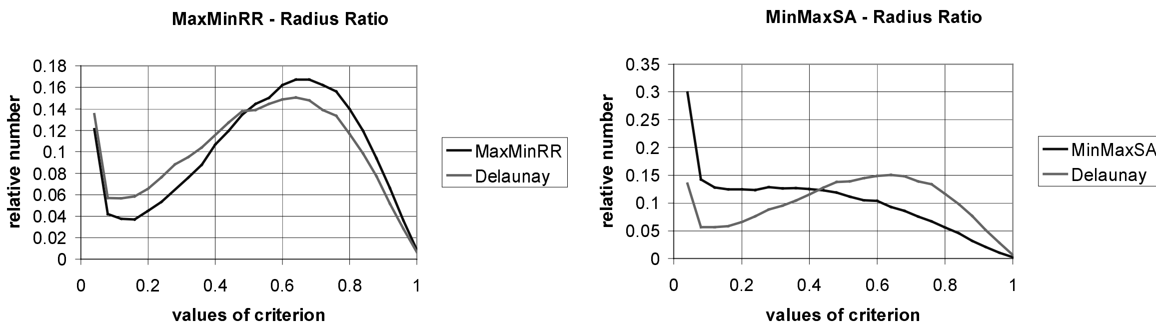


Figure 6.3: Histograms of radius ratio.

Solid angle

Histograms of solid angles produced by different post-optimization criteria are very similar in their shapes to the histograms of volume. Just for the illustration the same graphs as in the case of volume are shown in Fig. 6.4 to document this fact. This property does not show anything new or surprising.

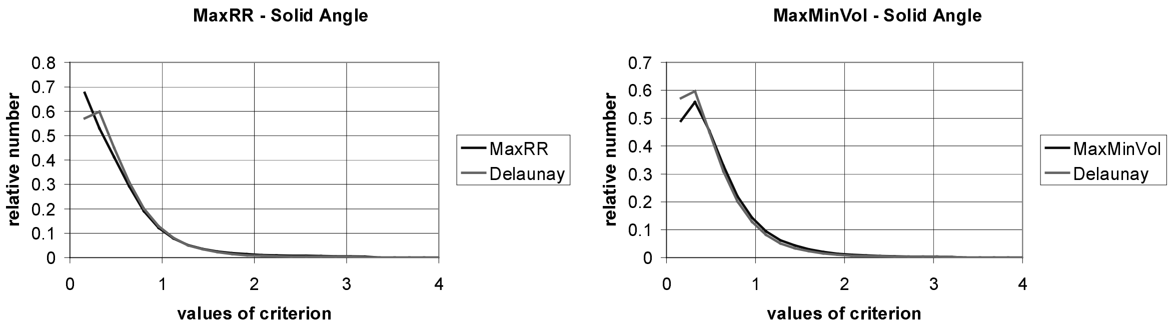


Figure 6.4: Histograms of solid angle.

Minimum solid angle

Two main types of graph appear here. The first curve, such as the one on the first graph in Fig. 6.5, has a decreasing trend. It illustrates typical results of the simple criteria including MaxThNo and MinMaxSA.

The second one is nearly the same as the curve of Delaunay triangulation, but some tetrahedra were slightly moved towards the greater values, while some tetrahedra disappeared from the area of smaller values of minimum solid angle, which was our goal. This small improvement is reached by most of the compound criteria including MinThNo.

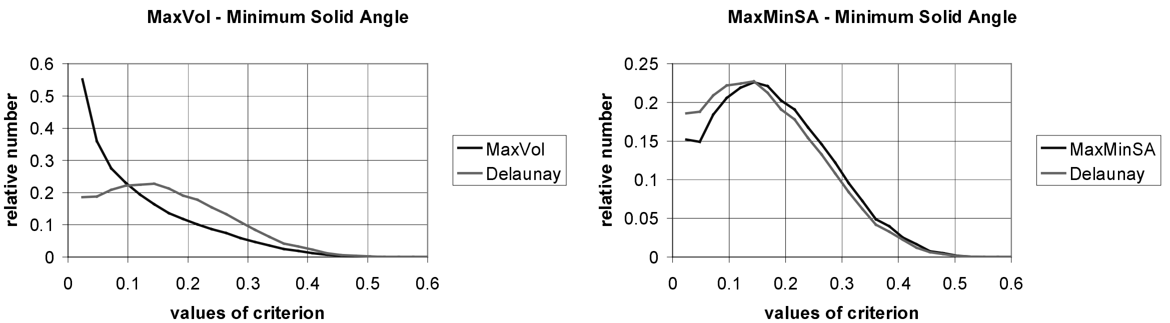


Figure 6.5: Histograms of minimum solid angle.

Maximum solid angle

Unlike the minimum solid angle property, where the moving of the tetrahedra towards higher values was wanted, the main goal here is to move the tetrahedra towards the lower values. All criteria used in post-optimization produced histograms like that on the first graph in Fig. 6.6 – values were moved in the wrong direction. The only one criterion, which is able to move the curve towards the lower values, is direct minimization of maximum solid angle. The curves reached by this type of post-optimization are shown on the second graph.

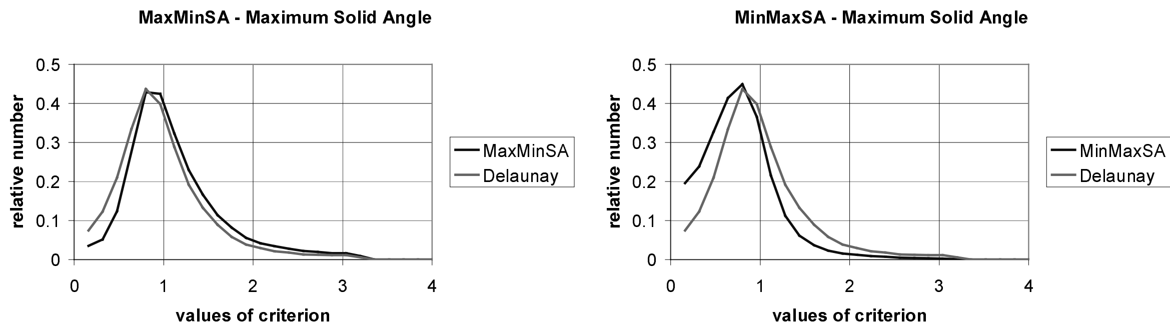


Figure 6.6: Histograms of maximum solid angle.

6.5 Results of criteria changing the number of tetrahedra

Now, after inspecting all of the properties, we would like to focus on two more post-optimization criteria, which only change the number of tetrahedra towards higher or lower values. Which results provide these criteria? MinThNo behaves almost in the same way as compound criteria – it tends to improve histograms, except in one case. This case is the histogram of maximum solid angle, whose curve is moved by this criterion towards the higher values (like by MaxMinSA).

On the contrary, MaxThNo behaves rather like simple criteria and it creates more tetrahedra of poor shape, with one exception – this case is again the histogram of maximum solid angle. Its curve has a shape, which is not produced by any of other criteria. This graph is shown in the Fig. 6.7.

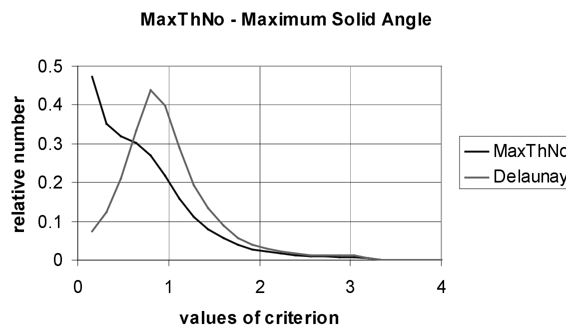


Figure 6.7: Special histogram of maximum solid angle.

6.6 Results of removing slivers

We tested both of the variants of removing badly shaped tetrahedra – variant with removing only those slivers, which originally belonged to zero class (RemSliv) and variant with removing all those tetrahedra, which don't belong to equilateral class (RemSlivAll). The method RemSlivAll produces slightly better histograms than RemSliv. In fact, both of them bring no new surprises. This is the reason, why no graph with properties produced by these criteria is shown here. Briefly said, both of them tend to optimize tetrahedra shapes in a similar way as e.g. MinThNo does, but the results are slightly worse.

6.7 Summary

For all properties, except maximum solid angle, the graph in Fig. 6.8 approximately shows the move of the level of average values of all of the properties. From this graph it is clear that the criteria for post-optimization like MinMaxSA, MaxThNo and MaxVol make the shape of the tetrahedra more or less worse. The MinMaxSA is the most time-consuming criterion, because of the computing of solid angles, while MaxThNo is only slightly slower than generating of Delaunay mesh. It also produces the greatest number of tetrahedra and the worst average at all.

On the opposite, the post-optimization criteria MaxMinSA, MinThNo, MaxMinVol, MaxMinRR and RemSliv really improve the quality of the mesh. This fact can be seen from the graph below. All of them have nearly the same time demand, all of them improve the average and produce less tetrahedra than Delaunay produce. The last criterion MaxRR has only slight influence on the quality of tetrahedra shapes.

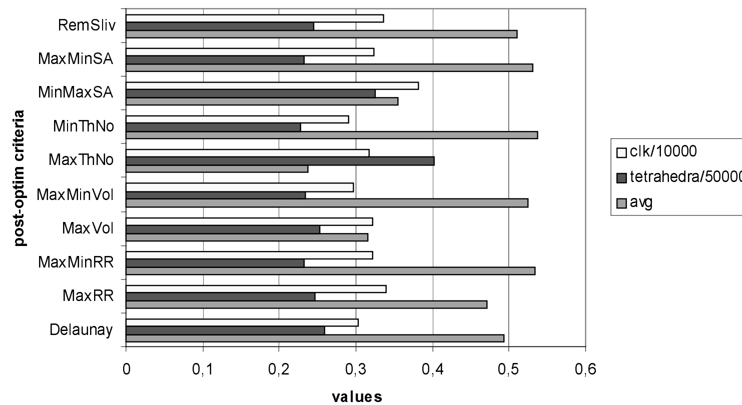


Figure 6.8: Typical statistics (*clk* means time in milliseconds, *tetrahedra* is number of tetrahedra, *avg* denotes average value of radius ratio property).

The behaviour of the maximum solid angle property is different. The only two criteria which are able to decrease the values of this criterion are MinMaxSA, that means direct post-optimization of itself, and MaxThNo, as you can see in Fig. 6.9. It seems good, but we should know that the number of tetrahedra in these two cases is the biggest of all criteria. And what is more, if we take a look back on the graph in Fig. 6.8, we will find out that these two criteria make the average considerably worse. Thus, this way doesn't seem to be a good approach to minimize maximum solid angle.

Table 6.1 shows movements of tetrahedra in some intervals of values of properties. It can be seen that the compound criteria MaxMinRR, MaxMinVol and MaxMinSA have a positive influence on tetrahedra shapes in given intervals. The same effect has criterion MinThNo, which only minimizes the number of tetrahedra. On the opposite, it is clear, that criterion MinMaxSA has bad influence on every watched shape criteria, except its own – this situation has been already explained above.

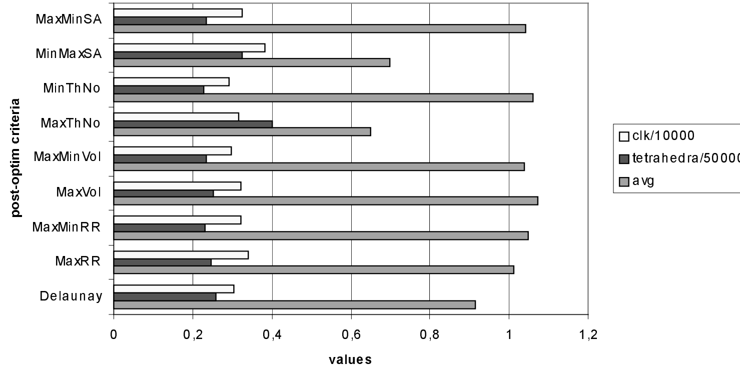


Figure 6.9: Statistics of maximum solid angle property (*clk* means time in milliseconds, *tetrahedra* is number of tetrahedra, *avg* denotes average value of maximum solid angle).

	MSA (0.168, 0.408)	**RR (0.44, 1.0)**	**SA (0.48, 2.4)**	**Vol (0.04, 0.2)**	**XSA (0.8, 3.2)**
Delaunay	40.28	60.8	34.7	61.41	52.86
MaxMinRR	44.45	68.35	39.34	64.28	64.72
MaxMinVol	42.66	66.31	39.13	64.68	63.6
MaxMinSA	44.47	67.43	38.54	63.29	63.34
MinThNo	45.31	69.02	39.23	64.12	64.62
MinMaxSA	22.23	37.11	27.46	22.4	34.95

Table 6.1: Percentual improvement of properties in intervals made by some of criteria

As it can be seen, the greatest benefit comes from compound criteria. The improvement – presented in graphs above and in Table 6.1 – is not as big as it was expected. This is probably due to “too good” quality of Delaunay triangulation, or due to simplicity of the algorithm. But there is an improvement up to 9 percent. Surprisingly nearly the same improvement as by compound criteria is done by criterion which only minimizes number of tetrahedra. It seems that for fast improvement of Delaunay tetrahedra it is enough only to decrease number of tetrahedra as much as possible. But for reaching the top of improvement of a property, the best way is to use direct post-optimization of this property.

As presented before, simple criteria mostly degrade the shape of tetrahedra and they cannot be recommended for the practical use. Created tetrahedra are flat, which is not desirable shape. Compound criterion MinMaxSA can be counted to this group, too.

We do not present here any comparison to results reached in [Joe91b] because Joe generates extra (Steiner) points inside the convex region, while our method uses only points that belong to the given data set.

Time demand of post-optimization criteria always depends on computational complexity of them. Most time consuming criterion seems to be such, which uses solid angles. But always the time of post-optimization is less than 10 percent of the main Delaunay algorithm. This time loss is not too important, because in most of the applications the mesh is constructed only once and the main goal is to have a mesh with the best possible quality.

We think that with this sort of post-optimization – it means local post-optimization with inspecting only configurations of five points and only one flip in one step of algorithm – it is not possible to get significantly better results. To reach them, one would have to use either more flips in one step as in [Joe95] penalted by more complicated algorithm or replacement of the points, such as in [ELMS00].

7 Numerical stability and robustness

"If you decide to use the efficient built-in number types, you have to cope with numerical problems. Be aware of this. For example, you can compute the intersection point of two lines and then check whether this point lies on the two lines. With floating-point arithmetic, roundoff errors may cause the answer of the check to be *false*. With the built-in integer types overflow might occur."

CGAL support library reference manual about built-in number types

Although our implementation of incremental algorithm works well for point sets distributed uniformly up to 200000 points (on a machine equipped with 1GB of memory), it can sometimes fail. The errors occur more often in the gaussian distribution of points or when distributing points in clusters. If the point set is more regular, especially when the points lie on the uniform grid, the algorithm fails even for very small data sets (hundreds of points). In our implementation we use so called ε arithmetic – computing with small tolerance ε. The idea is simple: for example if the point is found to lie not exactly (in a usual floating-point arithmetic) on the plane, but not further than in ε distance, it is taken as though it was lying on the plane. The results are not taken exactly, but with a small ε tolerance around the exact value. But it seems that this method cannot sufficiently solve the problems with numerical stability that we have. To remove the unpleasant unstable behaviour of the implementation, we had to look for a solution, which would improve the numerical stability and robustness of the implementation.

In the available literature we have found several solutions, which should guarantee an improvement in the area of numerical stability. There are solutions based on both types of arithmetic – integer and floating-point (we assume 32bit signed integer arithmetic and binary floating-point arithmetic according to IEEE Standard 754 [Hol98], which are the most usual on current PCs). The following lines give a short overview of them, more information from this area can be found e.g. in [GOR97 chap. 35].

Two examples of huge packages, which provide integer arithmetic in arbitrary precision as well as floating-point arithmetic, are LEDA (Library of Efficient Data types and Algorithms) and CGAL (Computational Geometric Algorithms Library). They offer a number of different program classes for exact computation. As a basic type an integer is offered, which is implemented by a vector of unsigned long numbers; the sign and the size are stored in extra variables. Derived types are rational numbers, which consist of the numerator and the denominator, both of them are integers. The real numbers, in LEDA called bigfloats, are also implemented as two integers – one for the mantissa one for the exponent. The precision of result and rounding method can be given by creating a bigfloat variable. CGAL for example offers a data type for specialized predicates with fixed precision, given by integers in the range $(-2^{24}, 2^{24})$ and multiplying factor 2^b. All these datatypes are very comfortable to use, because the routines for conversion among these types as well as input and output operations are provided. Moreover, these packages offer a floating-point filter with error bound computation or geometric predicates even with some perturbation schemes (see below).

Another approach to improve the reliability of the results is the solution mentioned in [DP98], which is based on an analysis of the input set of points and on an analysis of geometric predicates computed on common floating-point arithmetic. The assumptions of the analysis are 3D error-free input data and input points distributed uniformly in unit cube. The analysis of an insphere predicate is provided for 24 bit and 53 bit mantissa. The result is: if the absolute value of the insphere test is greater than $6\ 10^{-14}$ for 53 bits or $3\ 10^{-5}$ for 24 bits then the sign is reliable. The probability of failure is less than $6\ 10^{-11}$, resp. 0.011. This solution

shows, how often it is necessary to employ more powerful (or exact) computation methods if we want numerically stable algorithms.

Solutions for integer arithmetic are also provided. The most notable is the so-called SoS – Simulation of Simplicity described in [EM90]. It is designed for removing the degenerated cases from the input data by symbolical perturbations in arbitrary dimension. The points have to be moved a sufficient distance so as to destroy the degeneration but not so far that they destroy the relations among the points. Each point from the input data set is perturbed by adding a polynomial in ε to each coordinate. For example, the original 3D point $p_i = (p_{i,1}, p_{i,2}, p_{i,3})$ is replaced by a perturbed point $p_i(\varepsilon) = (p_{i,1}(\varepsilon), p_{i,2}(\varepsilon), p_{i,3}(\varepsilon))$, where $p_{i,j}(\varepsilon) = p_{i,j} + \varepsilon(i, j)$. The term $\varepsilon(i, j)$ is a polynomial in ε that goes to 0 as ε goes to 0. It is defined as $\varepsilon(i, j) = \varepsilon^{2^{di-j}}$, where d depends on the dimension of the matrix. The reason for evaluating the determinant of the matrix is because we use geometric predicates (see below), which are able to provide the results of different geometric tests, e.g. whether the point lies above or below the plane. The algorithm, which uses SoS predicates, is then run on perturbed points, but the result is presented as the solution of the original point set. The results have to be used carefully. Some kind of post-processing is needed sometimes (for example in convex hull algorithms, where the degenerated points can disappear from the convex hull by perturbing them). The low-level computation in SoS is based on integer arithmetic. Because the determinant evaluation results into finite expression, the finite number of bits is sufficient to evaluate the determinant exactly.

The employing of so called constraints for robust and accurate computation is used in [Hub96] for 3D Voronoi diagram construction. The inexact arithmetic is assumed and because of this reason and to avoid topological inconsistencies in the resulting diagram topological constraints are involved in the set of vertices to be deleted. Checking these constraints requires no numerical computation. The vertex is deleted only if it satisfies both the numerical and the topological criteria.

One of the possible approaches that can be used to obtain really exact result with integer arithmetic on current computers is used in [Sug92] for construction of 2D Voronoi diagram. The main idea lies in the range of restriction of the input values of expressions, which has to be evaluated exactly. By making the analysis of the expression structure the maximal values of inputs are derived. For example the expression for computing whether a point lies inside, on or outside the sphere needs input values less than 23 170 if it should be evaluated exactly on 64 bit integer arithmetic, or less than 5 931 640 if 96 bit integer arithmetic is used. Otherwise the hardware equipment is not adequate to give an exact result because of overflow.

For the exact decisions in triangulation algorithms it is possible to use the approaches described above, but if we do not want to put any restrictions on input points in coordinate values or their distribution, there are two more solutions, which offer exact evaluation of geometric predicates both for integer and floating-point arithmetic. Both of them are described below and practical experience of their use is mentioned at the end of the section.

7.1 Geometric predicates

Geometric predicates provide results of various geometrical tests (e.g., mutual position of a point and a line). They are computed as a value of a determinant of a particular matrix. The advantage of such an approach is that it can be generalized into an arbitrary dimension. For the purpose of a triangulation algorithm in 3D space, the two following predicates are of main interest:

- the test of a mutual position of a point *d* and a plane given by three oriented points *a*, *b*, *c*, the so called *orient3d* predicate; this predicate is given by the following matrix of dimension 4 or after some algebraic transformations by a matrix of dimension 3

$$\begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix}$$

$$= \begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix}$$

- the test of a mutual position of a point *e* and a sphere given by four oriented points *a*, *b*, *c*, *d*, so called *insphere* predicate; this predicate is given by matrix of dimension 5 or 4

$$\begin{vmatrix} a_x & a_y & a_z & a_x^2 + a_y^2 + a_z^2 & 1 \\ b_x & b_y & b_z & b_x^2 + b_y^2 + b_z^2 & 1 \\ c_x & c_y & c_z & c_x^2 + c_y^2 + c_z^2 & 1 \\ d_x & d_y & d_z & d_x^2 + d_y^2 + d_z^2 & 1 \\ e_x & e_y & e_z & e_x^2 + e_y^2 + e_z^2 & 1 \end{vmatrix}$$

$$= \begin{vmatrix} a_x - e_x & a_y - e_y & a_z - e_z & (a_x - e_x)^2 + (a_y - e_y)^2 + (a_z - e_z)^2 \\ b_x - e_x & b_y - e_y & b_z - e_z & (b_x - e_x)^2 + (b_y - e_y)^2 + (b_z - e_z)^2 \\ c_x - e_x & c_y - e_y & c_z - e_z & (c_x - e_x)^2 + (c_y - e_y)^2 + (c_z - e_z)^2 \\ d_x - e_x & d_y - e_y & d_z - e_z & (d_x - e_x)^2 + (d_y - e_y)^2 + (d_z - e_z)^2 \end{vmatrix}$$

To make a correct decision about the mutual position of geometric primitives, these matrices have to be evaluated exactly. Unlike the usual computation, which can be never exact because of limited memory space, the evaluation of such predicates can be done exactly, because the evaluation of determinant results in an expression, which is composed from a finite number of terms. Thus we know how much memory space is needed to store the result exactly. Moreover, the methods for predicate evaluation usually use some speedup techniques, which means that the result is not exact in the numerical sense, but is always correct in the geometrical sense.

Two such methods of predicate evaluating are presented in following sections in more detail. The first one is based on integer arithmetic, while the second one on floating-point arithmetic.

7.2 LN generator

The method for exact computation of several geometric predicates in integer arithmetic is mentioned in [FVW96]. The program called LN generator was created and it serves for the purpose of symbolical writing of an arbitrary predicate in the LN language. Then the symbolical input is transformed into the C++ source code. The output of this process is a routine for the exact computation of the predicate enriched by the so-called floating-point filter for checking whether the result is exact enough in usual arithmetic or if it is necessary to employ the exact arithmetic procedure.

As we mentioned before, the coordinates of the points in integer arithmetic are used as the input for the computation of a predicate. The maximal bit-length of each coordinate is 53 bits. It is due to the internal processing of exact computation, where no integers are used, because they have only 32 bit-length in current computers. Instead of them, the mantissa of

floating-point data types is used, which have 53 bit-length. The fear of slowing down the computation in floating-point arithmetic is described by the authors as useless, because this type of computation is highly optimized on current computers.

The number in the exact arithmetic is represented as an unevaluated sum of double precision floating-point variables

$$a = a_0 + a_1 + \ldots + a_m,$$

where $a_i = a_i' \times 2^{ri}$, a_i' is a particular integer number and r is the bit-length of the radix. The number of variables a_i depends on the radix length and the expected bit-length of the resulting value. Any integer up to 2^{1024} can be represented in this way. The limit comes from the limit on exponents in IEEE double precision. An exact value is normalized if each a_i' is at most 2^r in absolute value.

Let us have two exact numbers $a = a_0 + \ldots + a_m$ and $b = b_0 + \ldots + b_m$. The exact sum $c = a + b$ can be simply implemented as the sum of floating-point numbers:

$$c_0 = a_0 + b_0,\ \ c_1 = a_1 + b_1, \ldots,\ \ c_m = a_m + b_m.$$

The exact product $c = ab$ is implemented by using the usual $O(m^2)$ algorithm:

$$c_k = \text{sum}(a_j b_{k-j}),\ \ 0 \le j \le m.$$

During the process of calculating the sum, very often by calculating the product, the situation appears, when the term in a partial result cannot be represented exactly as a floating-point value. Then the operands has to be normalized before the operation, which guarantees that for each i, a_i' is at most 2^r in absolute value.

The choice of the radix bit-length may lie in the following:

- to minimize the number of components of exact number representation, the radix bit-length has to be as big as possible,
- the evaluation of the determinant contains several additions and several multiplications and our goal is to minimize the number of necessary normalizations, which slow down the evaluation; this point of view leads to the choice of minimal radix bit-length.

It is recommended by the authors to use 23 bit radix which is a good compromise between the above mentioned requirements.

The floating-point filter is used to filter out the results, which are exact enough in usual arithmetic. The filter is based on the computation of a static error in advance. The following formulae for the maximal bit-lengths of results for the addition, subtraction and multiplication operations are used

$$\text{maxbitlen}(a \pm b) = 1 + \max(\text{maxbitlen}(a), \text{maxbitlen}(b)),$$
$$\text{maxbitlen}(a \times b) = \text{maxbitlen}(a) + \text{maxbitlen}(b).$$

The static upper bound of the error is based on the structure of a particular expression (particular geometric predicate) and on the bit-length of the variables. It can be computed as follows: if maxbitlen(x) ≤ 53, then maxerr(x) = 0, else:

$$\text{maxerr}\,(a \pm b) = \text{maxerr}(a) + \text{maxerr}(b) + 2^{\text{maxbitlen}(a + b) - 53},$$
$$\text{maxerr}(a \times b) = \text{maxerr}(a)\text{maxbitlen}(b) + \text{maxerr}(b)\text{maxbitlen}(a) + 2^{\text{maxbitlen}(a \times b) - 53}.$$

The great advantage of the static upper bound of the error is that it can be computed statically before the program is run. In the runtime the program only compares the absolute value of the determinant computed in usual floating-point arithmetic with the value of maxerr and if the determinant is greater than maxerr, the result is sufficiently exact. In other way, the process of exact computation starts.

7.3 Adaptive floating-point geometric predicates

The previous approach can be also called a *multiple-digit* format. In this section a method of evaluating geometric predicates based on *multiple-term* format, which was described in [She96] is presented. This technique uses all 64 bits of floating-point double precision numbers. The advantage of this approach can be seen for example in adding two numbers of very different magnitudes – if we want to evaluate the expression $2^{300} + 2^{-300}$, only two words in memory are enough to store the result while the multiple-digit approach has to use at least 601 bits of memory, because all intermediate zero values have to be stored.

The computation in this method is based on a nonoverlapping expansion of the number, where the number x is stored as

$$x = x_n + ... + x_2 + x_1,\ \ (x_i < x_{i+1}),$$

where the values of all components x_i do not overlap and components are ordered by magnitude (x_n largest, x_1 smallest). The values of two floating-point numbers are not overlapping if the least significant nonzero bit of x is more significant than the most significant nonzero bit of y, or vice versa. For example the binary values 1100 and −10.1 are not overlapping, whereas 101 and 10 overlap.

The elementary functions, which transform the addition, subtraction and multiplication of two floating-point numbers into the nonoverlapping expansion of the result (that are also two floating-point numbers) are presented in the [She96]. As an illustration of such function a FastTwoSum is depicted here, which transforms value $a + b$ into a nonoverlapping result $x + y$, where the x is an approximation to $a + b$ and y represents the roundoff error in the calculation of x. The assumption $|a| \geq |b|$ holds. The operations are usual floating-point operations.

```
FastTwoSum(a,b)
  x = a + b
  b_virtual = x − a
  y = b − b_virtual
  return (x,y)
```

Such elementary functions are then used to build more complex functions for addition or subtraction of two expansions or for the multiplication of expansion by a floating-point number. The important notice is, that after these complex operations many zero components can appear in the resulting expansion, which can slow down the next computation. It is recommended to perform explicit zero component elimination.

Fig. 7.1 represents a scheme of addition of two expansions e and f, which are first processed into one expansion g. The result is expansion h. Elementary functions TwoSum and FastTwoSum are represented by the boxes.

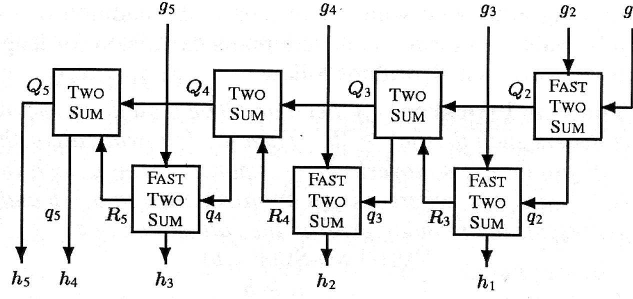


Figure 7.1: Scheme of addition of expansions [She96].

The same function written in pseudo-code is in the Fig. 7.2.

```
LINEAR-EXPANSION-SUM(e, f)
1    Merge e and f into a single sequence g,
         in order of nondecreasing magnitude
2    (Q_2, q_2) ⇐ FAST-TWO-SUM(g_2, g_1)
3    for i ⇐ 3 to m + n
4        (R_i, h_{i-2}) ⇐ FAST-TWO-SUM(g_i, q_{i-1})
5        (Q_i, q_i) ⇐ TWO-SUM(Q_{i-1}, R_i)
6    h_{m+n-1} ⇐ q_{m+n}
7    h_{m+n} ⇐ Q_{m+n}
8    return h
```

Figure 7.2: Function of addition of expansions [She96].

Unlike the previous approach to geometric predicates evaluation, where the static evaluation of error bound was used, in this method an adaptive scheme is incorporated, which tests the resulting values at several levels. Let us show it in an example.

Let us have an expression for computing the distance of two points in the plane: $(a_x - b_x)^2 + (a_y - b_y)^2$. Let us replace the subtractions in parentheses by their expansions $(x_1 + y_1)$ and $(x_2 + y_2)$, see Fig. 7.3 part a) and b). The resulting expression is

$$(x_1^2 + x_2^2) + (2x_1y_1 + 2x_2y_2) + (y_1^2 + y_2^2).$$

By the IEEE arithmetic it is guaranteed that $|y_i| \leq \varepsilon|x_i|$, where $\varepsilon = 2^{-p}$ is called the machine epsilon and it is equal to 2^{-53} for double precision or 2^{-34} in single precision arithmetic. The expression above can be divided into three parts, having magnitudes of O(1), O(ε) a O(ε²). The ith adaptive approximation of the exact result with the error of magnitude $O(\varepsilon^i)$ is then understood as a sum of the intermediate parts up to magnitude $O(\varepsilon^{i-1})$.

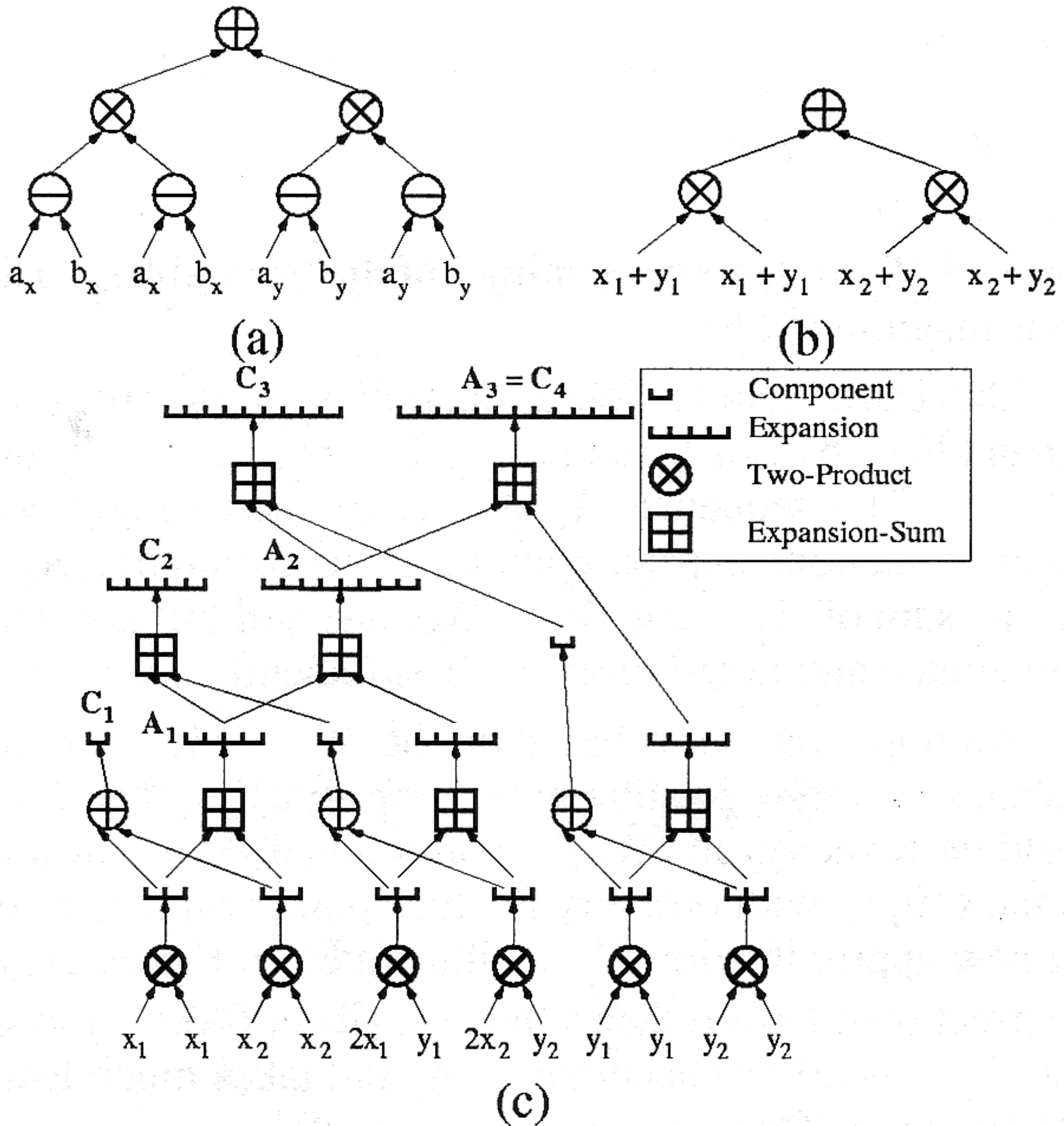


Figure 7.3: Exact evaluation of simple expression [She96].

As can be seen from the picture, the first adaptive approximation of the result is C1, which simply comes from the multiplication of the most significant part of the expansion with error bound O(1) – this approximation is identical to the static floating-point filter from the approach of Fortune and Van Wyk. If this value is not sufficiently exact, the computation continues. In the next step the part with error bound O(1) is computed exactly into the first exact approximation A1 and the part with error bound $O(\varepsilon)$ is computed inexactly. The sum of these two results is a new adaptive approximation of the result C2, which is also examined to see, if it is exact enough. In a similar way the whole expression is processed, until one of the adaptive approximations is found to be exact enough or until the exact result is computed. In practice the cases where exact results are needed are quite rare – very often the approximations are sufficient.

The computation of real predicates is much more complicated and thus they are not presented here. To give a better idea the exact result of the mutual position of a point and a plane may have up to 192 components.

7.4 Practical experience

This section describes our practical experience with the afore mentioned methods for the evaluation of geometric predicates. One more notice about the original implementation of our tests before the presentation of our results. At the beginning we did not use the predicates in our implementation of DT 3D algorithm. In the test of the mutual position of a point and a plane, the coefficients of the plane equation of each face of a tetrahedron were stored. The position of the point was computed only by putting its coordinates into the stored plane equation. Instead of insphere predicate, we computed the centre of a sphere circumscribed to the four points of a tetrahedron. Then the radius of a sphere was used to find out if the point lies inside or outside or on a sphere. We had to change both of these tests if we wanted to use an exact evaluation of the predicates.

The first part of the tests was made on artificial data sets with different distribution of points. Particularly we used points with uniform and gaussian distribution, points distributed in clusters, on the surface of the sphere and points on regular grid. The number of points in each data set was 50, 100, 200 and 240 thousands. The results produced from artificial data sets were taken as an average of results from five different data sets. As the second type of input data we used 17 real objects, particularly the points on surface of the real models. There were used objects such as chair, dino, teapot, CTHead and others [Stan, Geor], the number of their vertices was up to 277 thousands.

To provide the same input data for all tested functions, all input data had to be transformed onto integer numbers. It was done by simple scale. The reason for this was that the Fortune's predicates work only with integers. This correction of the range of the data did not affect their data type. Each tested function used the data type suited for it.

For the testing, a simple algorithm was used. In the following example the processing of input points is shown. The *orient3d* predicate is used, which needs only four input points. If we used *insphere* predicate to test, five input points are needed.

```
take the points v1,v2,v3,v4 from the input set
for (the number of input points - 3) do
  result = orient3d(v1,v2,v3,v4)
  v1 = v2
  v2 = v3
  v3 = v4
  v4 = next point
```

Thanks to Mr. Fortune and Mr. Shewchuk we obtained the source codes of their predicates implementations. In the case of Fortune's predicates two versions of them were provided – for 31 bit-length and for 53 bit-length. To achieve the equal conditions for all functions in the tests, we had to use the same data structures and the same function calls. Therefore we had to modify the source code of Fortune's predicates to use arrays instead of structures to store the coordinates of the input points. Thus we obtained four functions to be tested. Fortune's exact predicates for two different bit-lengths, named as *Fort31* and *Fort53* in the following text. Then Shewchuk's adaptive predicates named as *Shew* and, finally, pure unexact computation of determinant called *det*. We also separated our own test-functions from our implementation to use them in the tests. These functions are referred as *orig*. The last common property of all

functions, which had to be united, was the returned value. The output of all functions was restricted to values –1, 0 and 1.

After making these arrangements we performed the following tests on all mentioned functions, i.e. exact Shewchuk's predicates, exact Fortune's predicates for 31 and 53 bit-lengths, usual floating-point computation of determinant and original tests from our own implementation. These functions will be called as "tested functions" in following text.

All experiments were executed on Pentium 3, 450 MHz equipped with 1GB of memory under Windows 2000 operating system. Microsoft Visual C++ was used as the compiler.

Test of precision

To test the precision of tested functions we used the following approach: in each step of testing cycle we obtained the results of all tested functions and then we compared the results among them. As the correct one, the result of exact predicates was taken. Differences in results of other functions are depicted in Table 7.1 and Table 7.2.

It was surprising that on artificial data sets only the input points distributed on uniform grid produced wrong results. On the other data sets the inexact functions gave the same results as the exact ones. This held for orient3d tests as well as for insphere tests.

points	**Fort31**	**Fort53**	**Shew**	**det**	**orig**
50000	0	0	0	0.0012	0.0024
100000	0	0	0	0.0002	0.0008
200000	0	0	0	0.0002	0.0003
240000	0	0	0	0.0004	0.0006

Table 7.1: Percentage of bad decisions of orient3d tests on grid data.

points	**Fort31**	**Fort53**	**Shew**	**det**	**orig**
50000	0	0	0	0	0.0060
100000	0	0	0	0	0.0032
200000	0	0	0	0.0001	0.0019
240000	0	0	0	0	0.0014

Table 7.2: Percentage of bad decisions of insphere tests on grid data.

The result of precision tests on real data sets is as follows. Results of exact predicates were the same for all input data as it was expected. For the orient3d tests the percentage of bad decisions made by the determinant was only 0.0002, the percentage of original test was worse: 0.0026. For the insphere predicate the percentage of bad decisions made by the determinant was 0.0878, the percentage of our test was surprisingly high: 10.69.

Test of speed

To test the speed of all tested functions we simply measured the time of each function needed for processing the whole input data set. To show the results of these tests, we present here only the following representative selection.

In Fig. 7.4 and Fig. 7.5 you can see the time needed for executing all tested methods on several artificial data sets. As the representative number of points 100000 was chosen. It is quite strange that the times are nearly equal for all data sets. We expected that execution on sphere or grid data would be significantly slower. The reason for the speed similarity is

probably in the random placement of the points in the data sets. If the points were placed in a "more organized" way, e.g. the close points were also "near" in the input file, there would be probably more differences among the different distribution of points.

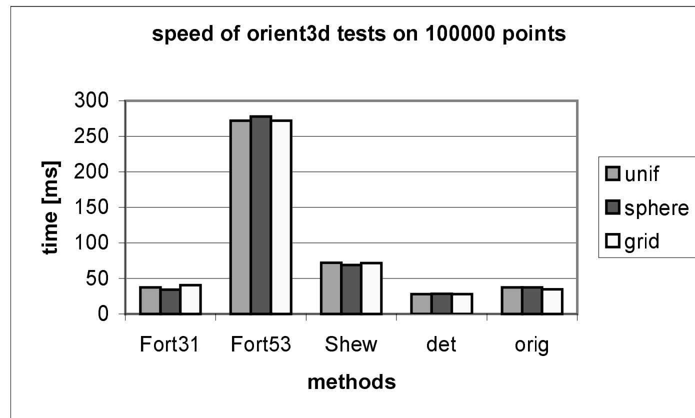


Fig.7.4: Orient3d tests on artificial data sets.

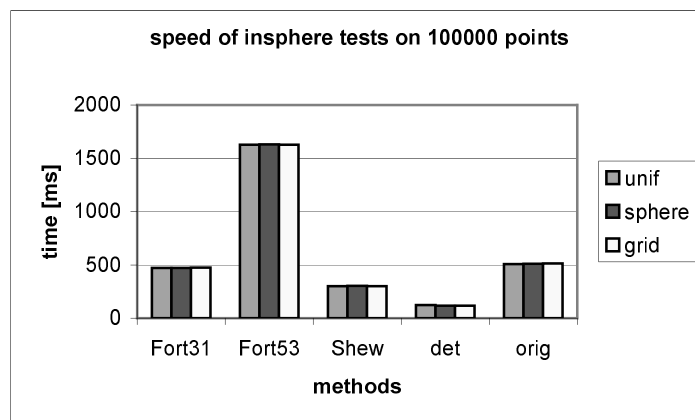


Fig. 7.5: Insphere tests on artificial data sets.

The situation is quite different for real data sets. As it can be seen from Table 7.3 and Table 7.4, inexact tests – such as determinants and original tests – are always faster ones. But the situation in the field of exact predicates is not as clear as for the artificial data. Sometimes the Fortune's predicates are faster than Shewchuk's one sometimes the situation is reversed. In general, Fortune's predicates are slower in cases, when the exact arithmetic is needed. It is due to the fact that the floating-point filter marks more cases as inexact than necessary. On the other side, Shewchuk's predicates use exact computation in less cases, but there is more time spent on making the decision, whether the exact computation is needed or not. This decision is made faster in Fortune's predicates, as the error bound is computed in advance.

	points	**Fort31**	**Fort53**	**Shew**	**det**	**orig**
teapot	80203	94	219	141	15	31
CTHead	230328	172	641	1094	62	78
14spheres	277228	296	765	2780	78	109

Table 7.3: Time of orient3d tests spent on real data.

	points	Fort31	Fort53	Shew	det	orig
teapot	80203	375	1312	719	94	469
CTHead	230328	1094	3750	5047	281	1750
14spheres	277228	1313	4516	14500	328	2860

Table 7.4: Time of insphere tests spent on real data.

We decided to use Shewchuk's predicates for the replacement of our unexact computation tests. We have replaced all tests of mutual position of a point and a plane with orient3d predicates, all computation of empty circumsphere with insphere predicates and the detection of tetrahedra configuration with several orient3d tests. The results are as follows.

For a clear idea of how much the exact evaluation of predicates slows down the implementation of DT triangulation algorithm, we made a speed comparison between the inexact and exact evaluation of a determinant, as it is shown in Fig. 7.6 on the left graph. The tests were made on uniform data sets of various sizes. Surprisingly the slowdown is only about 30 percent. In the next experiment, comparisons of exact predicates speed on different types of data sets were executed to see the slowdown on "bad" distribution of input points. The results are shown on the right graph of Fig. 7.6. The slowdown is much higher in this case, nearly four times – but it is no surprise. Grid data sets include a lot of degenerated cases, thus the exact evaluation of predicates has to employ more adaptive steps to reach a proper decision.

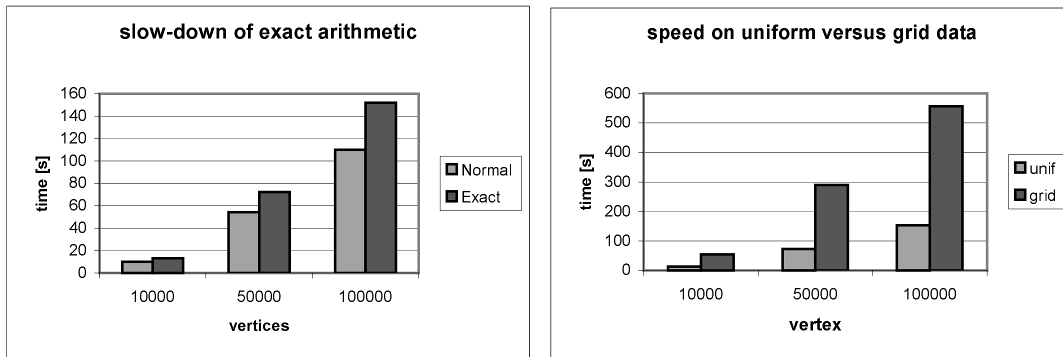
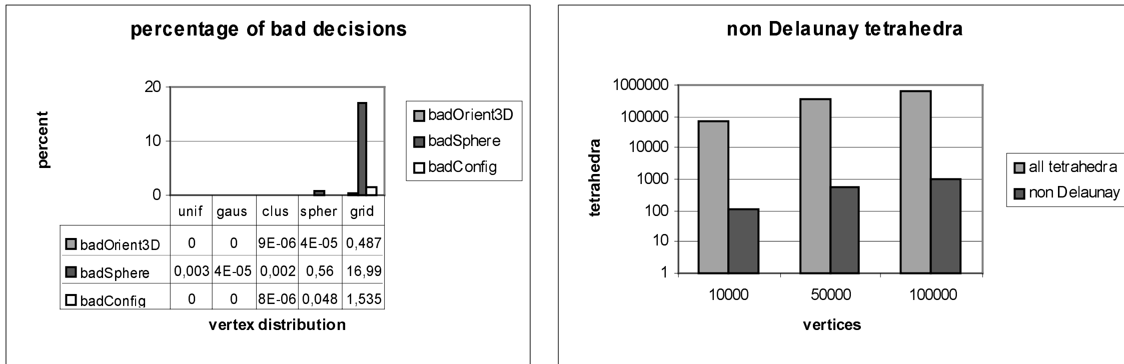


Figure 7.6: Time comparisons.

In Fig. 7.7 on the left graph the percentage of bad decisions in our original tests in orient3d, insphere and detection of configuration is shown. The great number of bad decisions in the grid distribution of points was made due to the impossibility of our test to compute the circumsphere. With inexact arithmetic the program would fail in such cases. The right picture shows that the number of non-Delaunay tetrahedra in inexact versions of the program is only about tenths of percent for uniform data sets.



	unif	gaus	clus	spher	grid
badOrient3D	0	0	9E-06	4E-05	0,487
badSphere	0,003	4E-05	0,002	0,56	16,99
badConfig	0	0	8E-06	0,048	1,535

Figure 7.7: Quality comparison.

Although the Shewchuk's predicates are adaptive, in the cases when the approximation of the result is not sufficient and exact result is needed, the computation takes quite a long time. Also the computation of error bound in the runtime takes some time. Our idea to speed up those predicates is in the use of such a floating-point filter, the error bound of which is computed in advance. This filter would be placed before the procedure of adaptive computation to make a first step of filtering the values, which are exact enough.

However, it is not easy to make such a filter for floating-point numbers – remember that Fortune's predicates are restricted on integers with maximal bit-length 53. For arbitrary floating-point numbers it is impossible to give any general restriction on their values. This fact leads to the idea of adaptive floating-point filter.

What does it mean – the adaptive floating-point filter? It means the static error bound produced by the filter is dependent on the range of the input data. In our explanation we restrict ourselves only on integer numbers. But the main idea works in general for any kind of input – even for fixed point and floating-point numbers. The only thing we have to know is the range, i.e. the maximal bit-length of the data values.

In Shewchuk's predicates the decision whether to go to the next stage of adaptive computation or to finish the computation (when the result is exact enough) depends on comparison between the actual adaptive result and error bound, which is computed in runtime similarly as the actual adaptive result. The possibility of speedup is in computation of the error bound in advance – the values for the static error bound are not derived from actual determinant, but from the whole data set. In other words it means we simulate the worst case in the input data for the computation of determinant.

The algorithm for getting the error bound is as follows:

```
get maxbitlength for x, y, z coordinates
compute maxbitlength for predicate expression
if (maxbitlength <= 53)
  errbound = 0.0
else
  compute maximal value of expression from maxbitlength
  compute errbound
```

The computation of maxbitlength is based on formulae mentioned in section about LN generator. If the maxbitlength is less or equal to 53, the whole expression can be stored exactly in the memory, thus no error bound is needed. Else the error bound is computed according to the rules derived in [She96].

The static error bound is then used as the entering point for the original Shewchuk's predicates. We tested mentioned method on all predicates provided in [She96], i.e. orient2d,

orient3d, incircle and insphere. Our improved predicates took only about 50% of time of original predicates on artificial data. But this speed up was achieved mainly due to the fact that exact arithmetic was not needed too often – we were able to filter out such cases quickly. On the real data, where the necessity of exact computation is greater, the results of our predicates are similar to the original ones, as you can see in Table 7.5. It is clear, because in such cases, actually original predicates are used plus the time needed for two comparisons of floating-point filter.

	teapot	**CTHead60**	**14spheres**
orient2d	0.5161	0.8297	0.8623
orient3d	1.128	0.9694	0.9996
incircle	1.0683	0.9880	1.1063
insphere	1.0208	0.9811	1.0033

Table 7.5: The time ratio of improved to original predicates.
Bit-length of input data is 40.

In conclusion we have to say that the exact evaluation of geometric predicates makes our implementation really numerically stable and far more reliable for any input data sets. The non-Delaunay tetrahedra completely disappeared from the resulting mesh when we started to use exact evaluation of predicates. In addition, the slowdown of the implementation is not so great if we take into account that more important than the speed of the program is the quality of the mesh. Moreover, we presented the predicates, which are enriched with the floating-point filter. However, from our experiments it is clear, that such enrichment is good only for cases, where the exact arithmetic is needed only rarely.

8 Our goals and future work

The main task of this thesis was to summarize existing algorithms for Delaunay triangulation in 3D and present the results of our experiments with tetrahedra shape improvement. According to our own implementation of the incremental insertion algorithm we concluded that for correct functionality and reliable results of the implementation, usual floating-point arithmetic is not sufficient. Thus we were forced to incorporate methods for the exact evaluation of geometric predicates. This approach made our implementation really more stable.

We are now at the point, where the Delaunay generator is implemented, it is reasonably fast and also sufficiently stable. It is used for data reconstruction and iso-surfaces extraction applications in our computer graphics centre. But for those applications that use solid models in practice it is still not sufficient. Such applications require also tetrahedrization of non-convex shapes or incorporation of special edges or faces, e.g. for breaks in geology. For this purpose, the so-called constrained Delaunay triangulation (shortly CDT) is required. CDT is a natural extension of DT in the sense that there are edges or faces, which are forced to be included in the DT. In 2D, the existence of CDT is proven for any given boundary. But in 3D, as we are aware, there are such objects, which cannot be triangulated in any way. There not much work has been done in this area, although the use of CDT is obvious for the creation of solid models. Hence we will focus on constrained Delaunay triangulation in our future research.

Furthermore, we are not fully satisfied with the improvement of tetrahedra shape, which we were able to reach. But recently in this area, theoretical computational geometry came up with new suggestions on how to reduce the amount of badly shaped tetrahedra or how to reach a sliver-free triangulation. These approaches are theoretically derived, most of them are based on interesting ideas, but they still remain theoretical – no algorithms or practical solutions are offered. This area is very attractive for us, because it is a typical task for research in applied computer sciences.

There is another extension, or let us say generalization of DT, so called regular triangulation. This type of triangulations assign a weight to each point from the input set and the criterion of empty sphere of DT is replaced by more complex criterion. Such triangulations are often mentioned as providing very good resulting meshes. There are algorithms for the construction of regular triangulations based on similar principles to DT algorithms. This means that our implementation of the Delaunay triangulation generator can be relatively simply modified to produce more powerful regular triangulation. It is probably clear from the previous lines, that the second area of our research will be dedicated to the improvement of tetrahedra shape.

9 References

[AEG87] Avis, D., ElGindy, H.: Triangulating Point Sets in Space. Discrete and Computational Geometry, Vol. 2, 1987, pp. 99–111.

[Bar96] Barber, C. B.: The Quickhull Algorithm for Convex Hulls. ACM Transactions on Mathematical Software, Vol. 22, No. 4, 1996, pp. 469–483.

[BCER95] Bern, M., Chew, P., Eppstein, D., Ruppert, J.: Dihedral Bounds for Mesh Generation in High Dimensions. In: Proc. 6th Ann. ACM-SIAM Sympos. Discrete Algorithms, 1995, pp. 189–196.

[CDEFT] Cheng, S.-W., Dey, T. K., Edelsbrunner, H., Facello, M. A., Teng, S.-H.: Sliver exudation. Proc. 15th ACM Symp. Comp. Geometry, 1999.

[CMPS92] Cignoni, P., Montani, C., Perego, R., Scopigno, R.: Parallel 3D Delaunay Triangulation. Internal Report C92/20, 1992.

[CMS92] Cignoni, P., Montani, C., Scopigno, R.: A Merge–First Divide & Conquer Algorithm for Ed Delaunay Triangulations. Internal Report C92/16, 1992.

[CMS98] Cignoni, P., Montani, C., Scopigno, R.: DeWall: A fast divide and conquer Delaunay triangulation algorithm in E^d. Computer-Aided Design, 30, 1998, pp. 333–341

[DP98] Devillers, O., Preparata, F. P.: Further Results on Arithmetic Filters for Geometric Predicates. INRIA, 1998.

[ELMS00] Edelsbrunner, H., Li, X.-Y., Miller, G., Stathopoulos, A., Talmor, D., Teng, S.-H., Üngör, A., Walkington, N.: Smoothing and Cleaning up Slivers. In ACM Symposium on Theory of Computing (STOC00) (2000).

[EM90] Edelsbrunner, H., Mücke, E. P.: Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms. ACM Transactions on Graphics, 9(1), 1990, pp. 66–104.

[Fac95] Facello, M. A.: Implementation of a randomized algorithm for Delaunay and regular triangulations in three dimensions. Computer Aided Geometric Design, Vol. 12, 1995, pp. 349–370

[Fie86] Field, D. A.: Implementing Watson's algorithm in three dimensions. Proc. Second Ann. ACM Symp. Comp. Geom., 1986.

[FP95] Fang, T. P., Piegel, L. A.: Delaunay triangulation in Three Dimensions. IEEE Computer Graphics and Applications, 1995, pp. 62–69.

[FVW96] Fortune, S., Van Wyk, C. J.: Static Analysis Yields Efficient Exact Integer Arithmetic for computational Geometry. ACM Trans. Graph. 15(3), 1996, 223–248.

[GD97] Golias, N. A., Dutton, R. W.: Delaunay triangulation and 3D adaptive mesh generation. Finite Elements in Analysis and Design 25, 1997, pp. 331–341.

[Geor] Georgia Institute of Technology. Large geometric models archive. http://www.cc.gatech.edu/projects/large_models

[GOR97] Handbook of Discrete and Computational Geometry. Edited by Goodman, J. E. and O'Rourke, J., CRC Press, 1997.

[Hol98] Hollasch, S: IEEE Standard 754 Floating Point Numbers. 1998. http://research.microsoft.com/~hollasch/cgindex/coding/ieeefloat.html

[Hub96] Hubbard, P. M.: Improving Accuracy in a Robust Algorithm for Three-Dimensional Voronoi Diagrams. The Journal of Graphics Tools, 1(1), 1996, pp. 33–47.

[Joe91a] Joe, B.: Construction of three-dimensional Delaunay triangulations using local transformations. Computer Aided Geometric Design, Vol. 8, 1991, pp. 123–142.

[Joe91b] Joe, B.: Delaunay versus max–min solid angle triangulations for three-dimensional mesh generation. International Journal for Numerical Methods in Engineering, Vol. 31, 1991, pp. 987–997.

[Joe95] Joe, B.: Construction of three-dimensional improved-quality triangulations using local transformations. SIAM Journal on Scientific Computing, 16, 1995, pp. 1292–1307.

[KS98] Kolingerová, I., Skala, V.: Delaunay Triangulation in Linear Expected Time. In Proceedings of seminars on Computational Geometry, SCG'98, Vol. 7, pp. 79–84, 1998.

[LJ94] Liu, A., Joe, B.: Relationship between tetrahedron shape measures. BIT, Vol. 34, 1994, pp. 268–287.

[Müc95] Mücke, E. P.: A robust implementation for three-dimensional Delaunay triangulations. Presented at the Geometric Software Workshop, The Geometry Center, Minneapolis, 1995.

[Raj91] Rajan, V. T.: Optimality of the Delaunay Triangulation in Rd. Proceedings of the Seventh Annual Symposium on Computational Geometry, ACM, 1991, pp. 357–363.

[Rek95] Rektorys, K. et al.: Přehled užité matematiky. (Survey of applied mathematics, in Czech) Prometheus, Prague, 1995.

[She96] Shewchuk, J. R.: Robust Adaptive Floating-Point Geometric Predicates. Proceedings of the Twelfth Annual Symposium on Computational Geometry, ACM, 1996.

[Stan] Stanford Computer Graphics Laboratory. `http://graphics.stanford.edu/data/3Dscanrep`

[Sug92] Sugihara, K.: A Simple Method for Avoiding Numerical Errors and Degeneracy in Voronoi Diagram Construction. IEICE Trans. Fundamentals, vol. E75-A, No. 4, 1992.

[TSBP93] Teng, Y. A., Sullivan, F., Beichl, I., Puppo, E.: A Data-parallel Algorithm for Three-dimensional Delaunay Triangulation and its Implementation. In SuperComputing 93, ACM, 1993, pp. 112–121.

[TSW99] Handbook of Grid Generation. Edited by Thompson, J. F., Soni, B. K., Weatherill, N. P., CRC Press, 1999.

[Wat81] Watson, D. F.: Computing the n-dimensional Delaunay tesselation with application to Voronoi polytopes. Comput. J. 24, 1981, pp. 167–172.

[ŽKP01] Žalik, B., Kolingerová, I., Podgorelec, D.: An Incremental Construction Algorithm for Delaunay Triangulation Based on Two-level Uniform Subdivision. 2001, Submitted to International Journal of Geographical Information Science.

Appendix A: Other DT implementations

There are several implementations of algorithms for the construction of 3D Delaunay triangulation, which are freely available from the Internet.

Qhull

Qhull is able to compute Delaunay triangulation in arbitrary dimension. It is based on a higher dimension embedding algorithm. For computing a convex hull the divide and conquer quickhull algorithm is used. The program is written in C and it is a console application. The solving of degenerated cases in input data is left to the user. It can either select an option for randomizing the input points or an option for joggling (a kind of perturbation) the points. More about Qhull can be found in [Bar96].

The package includes documentation and source files. The homepage of Qhull is at

`http://www.geom.umn.edu/software/qhull/`

Detri

The author is Ernst P. Mücke. Detri is a program for the direct construction of 3D Delaunay triangulation based on flips using an incremental insertion algorithm. It also uses randomization of input points. The numerical problems are solved by the use of Simulation of Simplicity. For details see [Müc95].

Detri is available via following URL, source codes are included.

`http://www.cs.ust.hk/faculty/edels/alpha3d.html`

DeWall, InCoDe

This implementation is made by the Visual Computing Group, which is a part of the Italian National Research Council. The first implementation is based on the divide and conquer paradigm, the second one is an incremental construction algorithm. Both of them use the usual floating-point arithmetic. More details are in [CMPS92, CMS92, CMS98].

`http://vcg.iei.pi.cnr.it/swOnTheWeb.html`

Geompack

The author is Barry Joe. Geompack is a mesh generator for general polyhedral regions based on 3D DT. It decomposes a general polyhedral region into simple or convex polygons, then it uses Steiner points for their triangulation. It uses usual floating-point arithmetic. It is written in FORTRAN 77, source code is available but no longer supported.

Geompack is available via

`ftp://menaik.cs.ualberta.ca/pub/geompack`

Appendix B: Activities

Publications

[Mau00] Maur, P.: Generování tetrahedronových sítí z rozptýlených dat. (Tetrahedral Mesh Generation from Scattered Data, in Czech) Diploma thesis, University of West Bohemia, Pilsen, 2000, supervisor I. Kolingerová.

[MK00] Maur, P., Kolingerová, I.: Properties of Delaunay Tetrahedra. In: Proceedings of International Conference ECI 2000, Herlany, Slovakia, pp. 55-61, ISBN 80-88922-25-9.

[MK01a] Maur P., Kolingerová I.: Post-optimization of Delaunay Tetrahedrization. In Proceedings of Spring Conference on Computer Graphics, Budmerice, Slovak Republic, 2001, ISBN 80 223 1606-7, pp.76-85.

[MK01b] Maur P., Kolingerová I.: Post-optimization of Delaunay Tetrahedrization. In: SCCG IEEE proceedings, ISBN 0-7695-1215-1, Los Alamitos, USA, 2001, pp.31–38.

Lectures, presentations, conferences abroad

- February-August 2000: Study stay at University of Ioannina, Greece.
- September 2000: Presentation of [MK00] at ECI’2000, Herľany, Slovakia.
- November 2000: Delaunay triangulation in 3D. Lecture given during a one-week study stay at the University of Maribor, Slovenia.
- April 2001: CESCG 2001, Budmerice, Slovakia.
- April 2001: Presentation of [MK01] at SCCG2001, Budmerice, Slovakia.