Computing the 3D Voronoi Diagram Robustly: An Easy Explanation

Hugo Ledoux
Delft University of Technology (OTB—section GIS Technology)
Jaffalaan 9, 2628BX Delft, the Netherlands
h.ledoux@tudelft.nl

Abstract

Many algorithms exist for computing the 3D Voronoi diagram, but in most cases they assume that the input is in general position. Because of the many degeneracies that arise in 3D geometric computing, their implementation is still problematic in practice. In this paper, I describe a simple 3D Voronoi diagram (and Delaunay tetrahedralization) algorithm, and I explain, by giving as many details and insights as possible, how to ensure that it outputs a correct structure, regardless of the spatial distribution of the points in the input.

1. Introduction

Many computer scientists and mathematicians consider the Voronoi diagram (VD) as being the most fundamental spatial structure because it is very simple, and yet is so powerful that it helps in solving many theoretical problems, and is also useful in a great many application domains [2, 32]. Algorithms that can compute efficiently and robustly the structure are therefore of the utmost importance. Several optimal algorithms based on different paradigms, and also implementations of these algorithms, have been proposed over the years. The majority of these algorithms do not compute directly the VD, but a closely-related structure, the Delaunay triangulation. As explained in Section 2, the two structures are actually dual, which means that the knowledge of one implies the knowledge of the other one (i.e. if one has only one structure, he can always extract the other one). As first pointed out by Boots [5], it is easier, from an algorithmic and data structure point of view, to manage triangles over arbitrary polygons (they have a constant number of vertices and adjacent cells). When the VD is needed, it can simply be extracted from its dual. This has the additional advantage of speeding up algorithms because when the VD is used directly intermediate Voronoi vertices—that will not necessarily exist in the final diagram—need to be computed and stored.

The authors of most papers in the computational geometry literature assume that they are working in a ‘perfect world’, i.e. they state that their algorithms are valid if and only if the input is in *general position* (which can have different meanings depending on the geometric problem at hand). When developing an algorithm and analysing its complexity, this assumption usually makes sense because one wants to avoid being overwhelmed with technical details (that are usually seen as implementation details), and also wants to simplify the complexity analysis (to be able to compare with other algorithms). Moreover, as stated in [8, page 9], “degenerate cases can almost always be handled without increasing the asymptotic complexity of the algorithm”. The result is that the handling of degenerate cases is usually left to the programmer. However, modifying an algorithm to make it robust for any inputs can be in some cases an intricate, time-consuming and error-prone task because degeneracies have the effect of increasing considerably the number of special cases that must be handled. For non-geometric algorithms such as a sorting algorithm, it can be easy since only the case of two keys being equal must be handled, but geometric algorithms can have dozens, or even hundreds of special cases [18].

In some cases, modifying the original algorithm so that it is robust against all input can be even more time-consuming and tedious than the design of the original solution in general position. A simple GIS-related example is that of Douglas [12] who describes his early attempts to implement a routine to determine where two line segments in a plane intersect. What seemed like a simple problem at first, turned out to be an implementation nightmare because of the problems caused by vertical lines, tolerance, segments that are close to parallel, etc. (his routine finally had 36 different cases). He states: “All of these inconsistencies eventually drag the programmer down from his high level math (i.e. algebra), through computer language (i.e. FORTRAN), into the realm of the machine methods actually used to perform arithmetic operations and their restrictions". Admittedly, this problem can now be solved rather easily if vector representations are used instead of mathematical equations [33], but I believe it gives a good example of the difficulties of dealing with special cases.

This paper is concerned with the implementation of an algorithm to construct the 3D Delaunay tetrahedralization (DT), and thus the 3D Voronoi diagram. Geometric computing in 3D space is known to be plagued with special cases, and the robust implementation of the 3D DT is a difficult problem in practice [14, 38]. In the following, I describe, by giving as many details and insights as possible, a 3D DT algorithm and I explain how to ensure that it outputs a correct solution, regardless of the spatial distribution of the points in the input. Note that a certain number of implementations are known (it is for instance implemented in `CGAL`[1]), so that what is presented in this paper is not entirely new. However, there is to my knowledge no comprehensive description that takes into account all the possible degeneracies, and explains how to handle them. The details and insights given in the following are based on my own experience while implementing such an algorithm, and also on the few details that are available in scientific papers and books. It is my hope that this paper will help others implement the 3D VD/DT, and that they will not spend countless hours fixing their code to make it robust.

The algorithm, which is described in Sections 4 and 5, is a flip-based incremental algorithm, as first proposed by Joe [21]. Notice that it is not the fastest solution in practice, but conceptual simplicity and robustness were favoured over theoretical results and solutions that would be intricate to implement. Still, as demonstrated in Section 7, it takes in practice a reasonable time for the problem that needs to be solved, and it is also relatively easy to implement. The important problem of the robustness of the arithmetic used by computer is also discussed in Section 6.

2. Duality Between the VD and the DT

Let S be a set of n points in a 3-dimensional Euclidean space $\mathbb{R}^3$. The Voronoi cell of a point $p \in S$, defined $\mathcal{V}_p$, is the set of points $x \in \mathbb{R}^3$ that are closer to p than to any other point in S. The union of the Voronoi cells of all generating points $p \in S$ form the Voronoi diagram of S, defined VD(S). In three dimensions, $\mathcal{V}_p$ is a convex polyhedron.

[1]The Computational Geometry Algorithm Library.

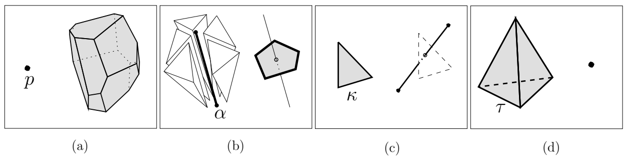

Figure 1. Duality in 3D between the elements of the VD and the DT.

The Delaunay tetrahedralization is dual to the VD, and is defined by the partitioning of the space into tetrahedra—where the vertices of the triangles are the points in S (generating each Voronoi cell)—that satisfy the *empty circumsphere* test (a sphere is empty is no points are in its interior, but points can lie directly on the sphere).

The duality between the VD and the DT in $\mathbb{R}^3$ is simple, each element of a structure corresponds to one and only one element in the dual: each polyhedron becomes a point and each line becomes a face, and vice-versa. For example, as shown in Figure 1: a Delaunay vertex p becomes a Voronoi cell (Figure 1(a)), a Delaunay edge α becomes a Voronoi face (Figure 1(b)), a Delaunay triangular face κ becomes a Voronoi edge (Figure 1(c)), and a Delaunay tetrahedron τ becomes a Voronoi vertex (Figure 1(d)). A Voronoi vertex is located at the centre of the sphere circumscribed to its dual tetrahedron, and two vertices in S have a Delaunay edge connecting them if and only if their two respective dual Voronoi cells are adjacent.

2.1. Degeneracies

For the VD and the DT, a set S of points is in general position when the distribution of points does not create any ambiguity in the structures. For the VD/DT in $\mathbb{R}^d$, the degeneracies, or special cases, occur when $d+1$ points lie on the same hyperplane and/or when $d+2$ points lie on the same ball. For example, in two dimensions, when four or more points in S are cocircular there is an ambiguity in the definition of DT(S). As shown in Figure 2, the quadrilateral can be triangulated with two different diagonals, and an arbitrary choice must be made since both respect the Delaunay criterion (points should not be on the interior of a circumcircle, but more than three can lie directly on the circumcircle).

This implies that in the presence of four or more cocircular points, DT(S) is not unique. Notice that even in the presence of cocircular points, VD(S) is still unique, but it has different properties. For example, in Figure 2, the Voronoi vertex in the middle has degree 4

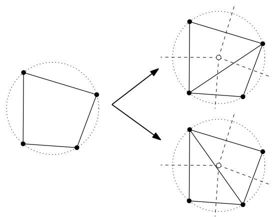

Figure 2. The DT (black lines) for four cocircular points in two dimensions is not unique, but the VD (dashed lines) is.

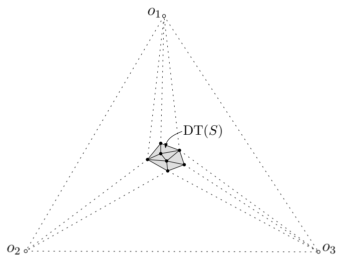

Figure 3. The set S of points is contained by a *big triangle* formed by the vertices o_1, o_2 and o_3.

(when S is in general position, every vertex in VD(S) has degree 3). When three or more points are collinear, DT(S) and VD(S) are unique, but problems with the computation of the structures can arise, as explained in Section 5. All the previous observations generalise straightforwardly to three and higher dimensions.

3. Constructing a DT

Mainly three paradigms of computational geometry can be used for computing a Delaunay triangulation in two and three dimensions: divide-and-conquer, sweep plane, and incremental insertion. Each one of these paradigms yields an optimal algorithm in two dimensions, i.e. a DT of n points is computed in $\mathcal{O}(n \log n)$. Examples of these are the divide-and-conquer and the incremental insertion algorithms of Guibas and Stolfi[17], and the sweep line algorithm that constructs directly the VD [15] .

In three dimensions, things are a bit more complicated. An algorithm, called `DeWall` and based on the divide-and-conquer paradigm, has been developed for constructing the DT in any dimensions [7]. Although the worst-time complexity of this algorithm is $\mathcal{O}(n^3)$ in three dimensions, the authors affirm that the speed of their implementation is comparable to the implementation of known incremental algorithms, and is subquadratic. Shewchuk [36] proposes sweep algorithms for the construction of the *constrained* Delaunay triangulation in $\mathbb{R}^d$, and these can be used for the normal DT. As is the case with `DeWall`, his algorithm is suboptimal for the three-dimensional case.

In $\mathbb{R}^3$, only incremental insertion algorithms have a complexity that is worst-case optimal, i.e. $\mathcal{O}(n^2)$ since the complexity of the DT in $\mathbb{R}^3$ is quadratic. With these algorithms, each point is inserted one at a time in a valid DT and the tetrahedralization is 'updated', with respect to the Delaunay criterion, between each insertion. Observe that the insertion of a single point p in a DT only modifies locally the DT, i.e. only the tetrahedra whose circumsphere contains p need to be deleted and replaced by new one respecting the Delaunay criterion (see Figure 8 for a two-dimensional example). In sharp contrast to this, divide-and-conquer and plane sweep algorithms build a DT in *one* operation (this is a batch operation), and if another point needs to be inserted after this, the whole construction operation must be done again from scratch.

Incremental insertion algorithms are therefore mandatory for building a dynamic spatial model, which is useful in many applications. Consider $\mathcal{T}$, the DT(S) of a set S of n points in $\mathbb{R}^3$. The insertion of a single point p, thus getting $\mathcal{T}^p = \mathcal{T} \cup \{p\}$, can be done with two incremental insertion algorithms: the *Bowyer-Watson* algorithm [6, 39], or one based on flipping in a triangulation.

The idea behind the former algorithm is relatively simple: all the tetrahedra *conflicting* with p, i.e. whose circumsphere contains p, are deleted from $\mathcal{T}$, and the 'hole' thus created is filled by creating edges joining p to each vertex of the hole. It can be argued that this algorithm is more error-prone because a hole in the tetrahedralization is created. It is also intricate from an algorithmic and data structure point of view as the geometric and topological relationships of the tetrahedralization are temporarily destroyed. The rest of the paper focuses on the alternative method: a flip-based algorithm.

4. Basic DT Operations

4.1. Initialisation: Big Tetrahedron

The algorithm in Section 5 assumes that the set S of points is entirely contained in a *big tetrahedron* (τ_{big}) several times larger than the range of S; the convex hull of S, denoted conv(S), therefore becomes τ_{big}. Figure 3 illustrates a two-dimensional example. The construc-

$$\textsc{Orient}(a,b,c,p) = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ p_x & p_y & p_z & 1 \end{vmatrix}$$

$$\textsc{InSphere}(a,b,c,d,p) = \begin{vmatrix} a_x & a_y & a_z & a_x^2 + a_y^2 + a_z^2 & 1 \\ b_x & b_y & b_z & b_x^2 + b_y^2 + b_z^2 & 1 \\ c_x & c_y & c_z & c_x^2 + c_y^2 + c_z^2 & 1 \\ d_x & d_y & d_z & d_x^2 + d_y^2 + d_z^2 & 1 \\ p_x & p_y & p_z & p_x^2 + p_y^2 + p_z^2 & 1 \end{vmatrix}$$

Figure 4. The two predicates needed.

tion of DT(S) is for example always initialised by first constructing τ_{big}, and then the points in S are inserted.

Doing this has many advantages, and is being used by several implementations [30, 4, 29]. First, when a single point p needs to be inserted in DT(S), this guarantees that p is always inside an existing tetrahedron. We do not have to deal explicitly with vertices added outside the convex hull, as in [21]. Second, we do not have to deal with the (nasty) case of deleting a vertex that bounds conv(S), if such an operation is needed. Third, since a triangular face is always guaranteed to be shared by two tetrahedra, point location algorithms never 'fall off' the convex hull.

The main disadvantage is that more tetrahedra than needed are constructed. For example in Figure 3 only the shaded triangles would be part of DT(S). The extra tetrahedra can nevertheless be easily marked as they are the only ones containing at least one of the four points forming τ_{big}.

4.2. Predicates

Constructing a DT and manipulating it essentially require two basic geometric tests (called *predicates*): Orient determines if a point p is over, under or lies on a plane defined by three points a, b and c; and InSphere determines if a point p is inside, outside or lies on a sphere defined by four points a, b, c and d. As shown in Figure 4, both tests can be reduced to the computation of the determinant of a matrix [17]. We can state that applying an identical translation to every point will not change the result of the determinant, and, for this reason, after translating all the points by $-p$ the two predicates are respectively implemented as the determinants of 3x3 and 4x4 matrices. I will not prove here the correctness of those predicates (see [17] for the two-dimensional case) but simply state that Orient returns a positive value when the point p is above the plane defined by a, b and c; a negative value if p is under the plane; and exactly 0 if p is directly on the plane. Orient is consistent with the *left-hand rule*: when the ordering of a, b and c follows the direction of rotation of the curled fingers of the left hand, then the thumb points towards the positive side (the *above* side of the plane). In other words, if the three points defining a plane are viewed clockwise from a viewpoint, then this viewpoint defines the positive side the plane. The predicate InSphere follows the same idea: a positive value is returned if p is inside the sphere; a negative if p is outside; and exactly 0 if p is directly on the sphere. Observe that to obtain these results, the points a, b, c and d in InSphere must be ordered such that Orient (a, b, c, d) returns a positive value.

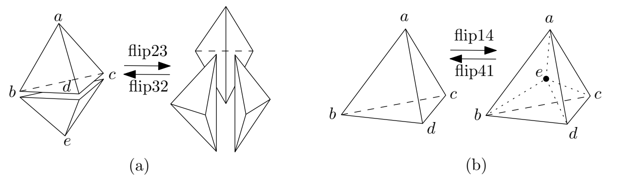

Figure 5. The 3D bistellar flips.

4.3. Three-dimensional Bistellar Flips

A bistellar flip is a local (topological) operation that modifies the configuration of some adjacent tetrahedra [25, 13]. Consider the set $S = \{a, b, c, d, e\}$ of points in general position in $\mathbb{R}^3$ and its convex hull conv(S). There exist two possible configurations, as shown in Figure 5:

1. the five points of S lie on the boundary of conv(S); see Figure 5(a). According to Lawson [25], there are exactly two ways to tetrahedralize such a polyhedron: either with two or three tetrahedra. In the first case, the two tetrahedra share a triangular face bcd, and in the latter case the three tetrahedra all have a common edge ae.

2. one point e of S does not lie on the boundary of conv(S), thus conv(S) forms a tetrahedron; see Figure 5(b). The only way to tetrahedralize S is with four tetrahedra all incident to e.

Based on these two configurations, four types of flips in $\mathbb{R}^3$ can be described: *flip23*, *flip32*, *flip14* and *flip41* (the numbers refer to the number of tetrahedra before and after the flip). When S is in the first configuration, two types of flips are possible: a flip23 is the operation that transforms one tetrahedralization of two tetrahedra into another one with three tetrahedra; and a flip32 is the inverse operation. If S is tetrahedralized with two tetrahedra and the triangular face bcd is not locally Delaunay, then a flip23 will create three tetrahedra whose faces are locally Delaunay. A flip14 refers

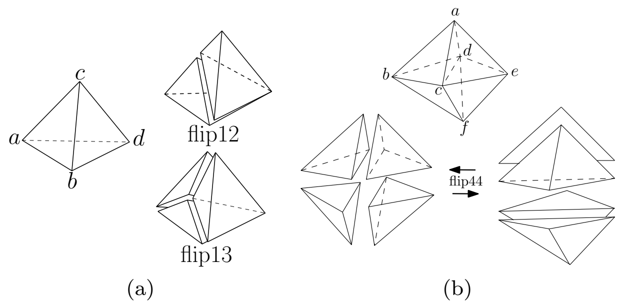

Figure 6. The degenerate flips (a) **flip12 and flip13, and** (b) **flip44.**

to the operation of inserting a vertex inside a tetrahedron, and splitting it into four tetrahedra; and a flip41 is the inverse operation that deletes a vertex.

Bistellar flips do not always apply to adjacent tetrahedra [20]. For example, in Figure 5(a), a flip23 is possible on the two adjacent tetrahedra *abcd* and *bcde* if and only if the line *ae* passes through the triangular face *bcd* (which also means that the union of *abcd* and *bcde* is a convex polyhedron). If not, then a flip32 is possible if and only if there exists in the tetrahedralization a third tetrahedron adjacent to both *abcd* and *bcde*.

4.4. Degenerate cases

To deal with degenerate cases, other flips need to be defined. Shewchuk defines and uses *degenerate flips* for the construction and the manipulation of constrained Delaunay triangulations in $\mathbb{R}^d$ [37]. A flip is said to be degenerate if it is a non-degenerate flip in a lower dimension. It is needed for special cases such as when a new point is inserted directly onto a simplex. For instance, Figure 6a shows the flip12 that splits a tetrahedron *abcd* into two tetrahedra when a new point is inserted directly onto the edge *ac*, and the flip13 that splits *abcd* into three tetrahedra when a new point is inserted directly onto the face *abc*. The flip12 is equivalent to a flip in one dimension, and the flip13 to the one needed to insert a new point in a triangle in two dimensions.

These flips are needed when dealing with constrained DT, but in our case they are not explicitly needed, as the operation they perform can be simulated, as shown in the next sections, with the flip23 and the flip32. The only degenerate flip needed for constructing a DT is the following. Consider the set $S = \{a, b, c, d, e, f\}$ of points configured as shown in Figure 6b, with points b, c, d and e forming a plane. If S is tetrahedralized with four tetrahedra all incident to one edge—this configuration is called the *config44*—then a flip44 transforms one tetrahedralization into another one also having four tetrahedra. Note that the four tetrahedra are in config44 before and after the flip44. A flip44 is actually a combination in one step of a flip23 (that creates a flat tetrahedron *bcde*) followed immediately by a flip32 that deletes the flat tetrahedron; a flat tetrahedron is a tetrahedron spanned by four coplanar vertices (its volume is zero). The flip44 is conceptually equivalent to the well-known flip22 in two dimensions.

4.5. Point Location

Given the DT $\mathcal{T}$ of a set S of n points and a query point p, the point location problem consists of determining inside which tetrahedron of $\mathcal{T}$ lies p. This is a necessary operation for constructing incrementally a DT, and the algorithm described in Section 5 uses it.

Many point location algorithms for the DT are optimal [13, 9], but they use additional storage, they require preprocessing for creating the additional data structure, and they are also often very complicated to implement. As Mücke et al. [31] note, optimal algorithms do not necessarily mean better results in practice because of the amount of preprocessing involved, the extra storage needed, and also because the optimal algorithms do not always consider the dynamic case, where points in the DT could be deleted. For many problems like this one, practitioners have been known to favour sub-optimal algorithms that are easier to implement.

Mücke et al. [31] and Devillers et al. [10] analyse theoretically sub-optimal algorithms, for the DT in three dimensions, that yield fast practical performances. These algorithms are desirable in the case of a dynamic structure because they do not use any additional storage or preprocessing. The adjacency relationships between the simplices in a DT are used to perform the point location.

Based on experimental results, both conclude that the *walking* (refer to as WALK in the following) strategy is the fastest solution. It was described in the earliest papers about the construction of the DT in two dimensions [16]: in a DT in $\mathbb{R}^d$, starting from a d-simplex σ, we move to one of the neighbours of σ (σ has $(d+1)$-neighbours; we choose one neighbour such that the query point p and σ are on each side of the $(d-1)$-simplex shared by σ and its neighbour) until there is no such neighbour, then the simplex containing p is σ. The algorithm, illustrated in Figure 7, is not detailed here because of space constraints, but can be easily implemented after reading [31] and [10]. Also,

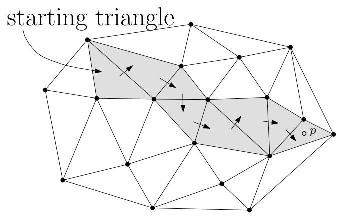

Figure 7. The WALK algorithm for a DT in two dimensions. The query point is p.

making WALK robust is trivial, as it is not affected by degenerate cases.

5. An Incremental Flip-based Algorithm

The alternative to creating a hole is to use bistellar flips to modify the configuration of tetrahedra in the vicinity of p. It should first be noticed that flips are operations valid in any dimensions, and not only in three dimensions [25]. They permit us to keep a complete tetrahedralization during the insertion process, and hence algorithms are relatively simple to implement and also numerically more robust. Although a flip-based algorithm requires somewhat more work than an algorithm where a hole is created—Liu and Snoeying [29] state that on average 1.5–2 times more tetrahedra are created and that flippability tests also slow down the whole process—its implementation is simplified and less error-prone since the maintenance of adjacency relationships in a DT is encapsulated in the flip operations.

5.1. Two Dimensions

The first flip-based algorithm was designed to construct the DT in two dimensions. It was developed by Lawson [24] who proved that, starting from an arbitrary triangulation of a set S of points in the plane, flipping edges (i.e. replacing the diagonal of a quadrilateral as in Figure 2) can transform this triangulation into any other triangulation of S, including the Delaunay triangulation. The total running time of this algorithm is $\mathcal{O}(n^2)$, since there is $\mathcal{O}(n)$ triangles that must be tested against each other.

An incremental insertion algorithm based on edge flipping can improve this to $\mathcal{O}(n \log n)$ [17]. What follows are the main steps for the insertion of a single point p in a DT, and the process is shown in Figure 8. First, the triangle τ containing p is identified and then split into three new triangles by joining p to

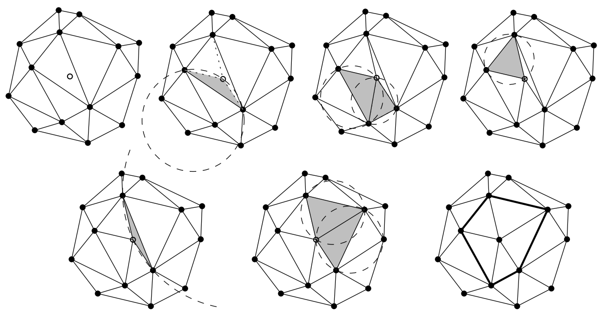

Figure 8. Step-by-step insertion, with flips, of a single point in a DT in two dimensions.

every vertex of τ. Second, each new triangle is tested—according to the Delaunay criterion—against its opposite neighbour (with respect to p); if it is not a Delaunay triangle then the edge shared by the two triangles is flipped and the two new triangles will also have to be tested later. This process stops when every triangle having p as one of its vertices respects the Delaunay criterion.

5.2. Three Dimensions

While the concept of flipping generalises to three dimensions, Lawson's algorithm [24] does not, as Joe [20] proves. Indeed, starting from an arbitrary tetrahedralization, it is possible that during the process of flipping, a non-locally Delaunay facet be impossible to flip (this case happens when the union of two tetrahedra is concave). Joe [21] nevertheless later circumvented this problem by proving that, given a DT(S) and a point p in $\mathbb{R}^3$, there always exists at least one sequence of bistellar flips to construct DT($S \cup \{p\}$). In this case, there will be non-Delaunay facets impossible to flip, but he proved that there will always be a flip possible somewhere else such that the algorithm progresses. This can form the basis of an incremental insertion algorithm for the construction of the Delaunay tetrahedralization, that is a straightforward generalisation of the two-dimensional case. Let S be a set of points in $\mathbb{R}^3$ and let $\mathcal{T}$ be DT(S), Figure 9 shows the algorithm, called INSERTONEPOINT, needed to restore the 'Delaunayness' in $\mathcal{T}$ when a single point p is inserted. The algorithm is adapted from [21], and is conceptually the same as [13]. As is the case with the two-dimensional algorithm, the point p is first inserted in $\mathcal{T}$ with a flip (flip14 in the case here), and the new tetrahedra created must be tested to make sure they are Delaunay. The sequence of flips needed is controlled by a stack containing all the tetrahedra that have not been tested

```
Input: A DT(S) $\mathcal{T}$ in $\mathbb{R}^3$, and a new point $p$ to insert
Output: $\mathcal{T}^p = \mathcal{T} \cup \{p\}$
 1: $\tau \leftarrow$ WALK {to obtain tetra containing $p$}
 2: insert $p$ in $\tau$ with a flip14
 3: push 4 new tetrahedra on stack
 4: while stack is non-empty do
 5:     $\tau = \{p, a, b, c\} \leftarrow$ pop from stack
 6:     $\tau_a = \{a, b, c, d\} \leftarrow$ get adjacent tetrahedron of $\tau$
        having $abc$ as a facet
 7:     if $d$ is inside circumsphere of $\tau$ then
 8:         FLIP($\tau$, $\tau_a$)
 9:     end if
10: end while
```

Figure 9. Algorithm InsertOnePoint($\mathcal{T}, p$).

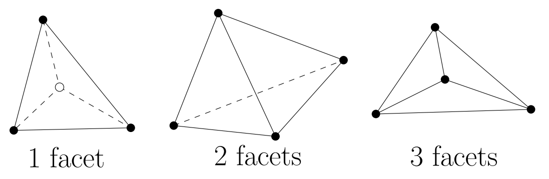

Figure 10. In 3D, three types of tetrahedra are possible, when viewed from a fixed viewpoint (the reader in this case).

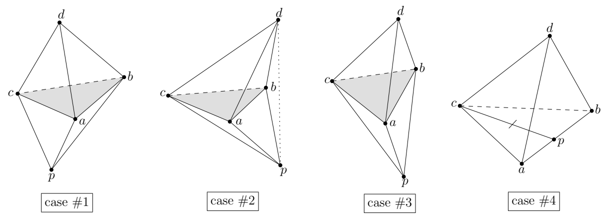

Figure 11. Four cases are possible for the Flip algorithm. For case #3, vertices p, a, b and d are coplanar. For case #4, the tetrahedron $abcp$ is flat.

yet. The stack starts with the four resulting tetrahedra of the flip14, and each time a flip is performed, the new tetrahedra created are added to the stack. The algorithms stops when all the tetrahedra incident to p are Delaunay, which also means that the stack is empty.

The Flip method in InsertOnePoint needs to be refined because, unlike in two dimensions, different geometric cases yield different flips. Assume that p was inserted in $\mathcal{T}$, and that a certain number of flips have been performed but that $\mathcal{T}^p = \mathcal{T} \cup \{p\}$ is not Delaunay yet. All the tetrahedra incident to p, which form the star[2] of p, must be tested to ensure that they are Delaunay, and notice that the ones that have not been tested yet are in the stack. Let $\tau = \{p, a, b, c\}$ be the next tetrahedron popped from the stack. To ensure τ is Delaunay, we need only to test it with the tetrahedron $\tau_a = \{a, b, c, d\}$ outside star(p) and incident to the facet abc; we are actually testing if abc is locally Delaunay. If the circumsphere of τ contains d, different options for the flip required are possible, according to the geometry of τ and τ_a. Observe that if we look from p, we can see one, two or even three facets of τ_a (depicted in Figure 10). When S is in general position, two cases are possible (both cases refer to Figure 11):

Case #1: only one face of τ_a is visible, and therefore the union of τ and τ_a is a convex polyhedron. In this case, a flip23 is performed.

Case #2: two faces of τ_a are visible, and therefore the union of τ and τ_a is non-convex. If there exists a tetrahedron $\tau_b = abpd$ in $\mathcal{T}^p$ such that the edge ab is shared by τ, τ_a and τ_b, then a flip32 can be performed. If there exists no such tetrahedron, then no flip is performed. The non-locally Delaunay facet abc will be 'rectified' by another flip performed on adjacent tetrahedra, as [21] proves.

When three faces of τ_a are visible, no action is taken and the next tetrahedron in the stack is processed.

The implementation of the test that determines which flip should be applied for τ and τ_a does not test if their union is convex or concave. What is used instead is a test that determines if the edge joining the two apexes of τ and τ_a crosses the interior of their common facet. If the edge does, it means the union is convex, and if it does not the union is concave.

5.3. Time Complexity

When an incremental insertion algorithm based on InsertOnePoint is used to compute DT(S) of a set S of n points in $\mathbb{R}^3$, the algorithm takes $\mathcal{O}(n^2)$, which is worst-case optimal [13]. However, in practice, the algorithm will most likely be faster. It was proved that if the n points are uniformly distributed in a unit cube, then the expected running time goes down to $\mathcal{O}(n \log n)$, provided that the history of the flips is used for point location [13].

Furthermore, the work needed to insert a single point p in $\mathcal{T}$ is proportional to s, the number of tetrahedra conflicting with p [13]. Indeed, notice that every flip in InsertOnePoint is actually deleting one and

[2]The star of a vertex v in a d-dimensional triangulation, denoted star(v), consists of all the simplices that contain v; it forms a star-shaped polytope.

```
Input: Two adjacent tetrahedra $\tau$ and $\tau_a$
Output: A flip is performed, or not
 1: if case #1 then
 2:     flip23($\tau$, $\tau_a$)
 3:     push tetrahedra $pabd$, $pbcd$ and $pacd$ on stack
 4: else if case #2 AND $\mathcal{T}^p$ has tetrahedron $pdab$ then
 5:     flip32($\tau$, $\tau_a$, $pdab$)
 6:     push $pacd$ and $pbcd$ on stack
 7: else if case #3 AND $\tau$ and $\tau_a$ are in config44 with $\tau_b$ and $\tau_c$ then
 8:     flip44($\tau$, $\tau_a$, $\tau_b$, $\tau_c$)
 9:     push on stack the 4 tetrahedra created
10: else if case #4 then
11:     flip23($\tau$, $\tau_a$)
12:     push tetrahedra $pabd$, $pbcd$ and $pacd$ on stack
13: end if
```

Figure 12. Algorithm FLIP(τ, τ_a). Note that the 4 cases refer to Figure 11.

only one non-Delaunay tetrahedron, and replacing it with some new ones incident to p. The first flip14 removes only one tetrahedron and replaces it by four new ones; a flip23 deletes a conflicting tetrahedron τ_a and replaces τ (the tetrahedron incident to p in star(p)) and τ_a by three tetrahedra all incident to p; and a flip32 also removes one conflicting tetrahedron (the one outside star(p)) and creates two new tetrahedra incident to p. Also, once a tetrahedra is deleted after a flip, it is never re-introduced in $\mathcal{T}^p$. Therefore, if s tetrahedra in $\mathcal{T}$ are conflicting with p, then exactly s flips are needed to obtain $\mathcal{T}^p$.

5.4. Degenerate Cases

Joe [21] not only proved that a flip-based incremental insertion works in three dimensions, he proved it even when S has degeneracies. He detailed the different configurations possible when a flip is to be performed. I present in the following the many degeneracies that can arise and I describe how to solve them. These solutions are mainly taken from [21, 35, 37].

When S contains degenerate cases, the intersection tests between the two apexes of τ and τ_a can return other results (both cases refer to Figure 11):

Case #3: the line segment pd intersects directly one edge of abc (assume it is ab here). Thus, vertices p, d, a and b are coplanar. Observe that a flip23, that would create a flat tetrahedron $abdp$, is possible if and only if τ and τ_a are in config44 (see Figure 6b) with two other tetrahedra τ_b and τ_c, such that the edge ab is incident to τ, τ_a τ_b and τ_c. Since the four tetrahedra are in config44, a flip44 can be performed. If not, no flip is performed.

Case #4: the vertex p is directly on one edge of the face abc (assume it is ab here). Thus, p lies in the same plane as abc, but also as abd. The tetrahedron $abcp$ is obviously flat. The only reason for such a case is because p was inserted directly on one edge of $\mathcal{T}$ in the first place (the edge ab), and the first flip14 to split the tetrahedron containing p created two flat tetrahedra. Because p was inserted on the edge ab, all the tetrahedra in $\mathcal{T}$ incident to ab must be split into two tetrahedra. This could be done with the degenerate flip12, but it can also be done simply with a sequence of flip23 and flip32. When the case #4 happens, it suffices to perform a flip23 on τ and τ_a. Obviously, another flat tetrahedron $abdp$ will be created, but this one will deleted by another flip.

Observe that although flat tetrahedra are not allowed in a DT (the circumsphere of a flat tetrahedron is undefined and contains all the points in a set), they are allowed during the process of updating a tetrahedralization, as long as the combinatorial structure stays coherent.

The only degenerate cases remaining to handle are: (1) there exists a vertex at the exact location where p was supposed to be inserted; (2) p lies directly on the circumsphere of a tetrahedron in $\mathcal{T}$; (3) p was inserted directly on a face of a tetrahedron in $\mathcal{T}$. The first case can be handled trivially: simply test the distance from p to each vertex of the tetrahedron returned by WALK against a tolerance. If one distance is smaller than the tolerance, do not insert p at all. The second case is also easy to handle: do not perform a flip since this is an unnecessary operation that will slow down the algorithm. Since p is directly on the circumsphere and not inside, no operation will still result in a correct DT. The last case does not require any special treatment: the flip14 that inserts p in a tetrahedron will create one flat tetrahedron, and this tetrahedron will be deleted when tested against the tetrahedron outside star(p) sharing the same face.

5.5. Extracting the VD from the DT

Let $\mathcal{T}$ be the DT of a set S of points in $\mathbb{R}^3$. The simplices of the dual $\mathcal{D}$ of $\mathcal{T}$ can be computed as follows (all the examples refer to Figure 1:

Vertex: a single Voronoi vertex is easily extracted:it is located at the centre of the sphere passing through the four vertices of its dual tetrahedron τ.

Edge: a Voronoi edge, which is dual to a triangular face κ, is formed by the two Voronoi vertices dual to the two tetrahedra sharing κ.

Face: a Voronoi face, which is dual to a Delaunay edge α, is formed by all the vertices that are dual to the Delaunay tetrahedra incident to α. The idea is simply to 'turn' around a Delaunay edge and extract all the Voronoi vertices. These are guaranteed to be coplanar, and the face is guaranteed to be convex.

Polyhedron: the construction of one Voronoi cell $\mathcal{V}_p$ dual to a vertex p is similar: it is formed by all the Voronoi vertices dual to the tetrahedra incident to p. Since a Voronoi cell is convex by definition, it is possible to collect all the Voronoi vertices and then compute the convex hull (e.g. see [3] for an algorithm); the retrieval of all the tetrahedra incident to p can be done by performing a BFS-like algorithm on the graph dual to the tetrahedra. A simpler method consists of first identifying all the edges incident to p, and then extracting the dual face of each edge.

Given $\mathcal{T}$, we must obviously visit all its 3-simplices to be able to extract $\mathcal{D}$. This means that computing $\mathcal{D}$ from $\mathcal{T}$ has a complexity of $\Theta(n)$ when S contains n points.

6. Robustness

Consider an admittedly simple algorithm that adds the value 0.1 one hundred times, and then returns `true` if the total is 10.0, and `false` otherwise. It should obviously always returns `true`, but if it is implemented on a computer with floating-point arithmetic, the odds that it will return `true` are rather low (if not nil). This is because floating-point arithmetic offers only an approximation to real numbers, which are always rounded to the closest possible value in the computer. When one chooses to implement an algorithm with floating-point arithmetic, it is important he understands the consequences of his choice. Floating-point arithmetic to represent real numbers is ubiquitous because it has many advantages: it is available almost on every platform, and, more importantly, it has been highly optimised so that arithmetic operations are performed very fast.

I began implementing the algorithm described in this paper under the naïve assumption that floating-point arithmetic was 'good enough' and that I would be able to understand the special cases, and fix them by adding some more code. My method was the most widely used to fix numerical non-robustness: the *tolerance*. In other words, if two values are very close to each other, then they are equal. Tolerances can probably yield satisfactory results for simple problems (e.g. the intersection of two line segments), but for more complex ones like the construction of the DT, where the combinatorial structure could be invalidated by small movements of the vertices, it is risky. How should one define the 'optimal' tolerance? I observed that tweaking the tolerance to fix a given problem was usually easy, the problem being that it was also creating another problem somewhere else. Even with the tolerance well-defined (or so I thought), during the development of algorithms I became frustrated because my program would sometimes crash, or even output something that was not a valid tetrahedralization. "Was it the arithmetic? My algorithm? Or a mistake I made while coding it?" I wondered all the time. The development of the algorithm was hindered by the fact that the source of the problem was never known. The only solution was to use exact arithmetic: my first question was therefore answered—that left two unanswered but that was better than nothing!

The major obstacle to using exact arithmetic is the speed of computation: it is very slow [40]. Unlike floating-point arithmetic, an arithmetic operation can not be assumed to be performed in constant time. The complexity of an operation depends on the numbers n of bits used to store a number, and the multiplication of two numbers can have for instance a complexity of $\mathcal{O}(n^2)$. Karasick et al. [23] reported that the naïve implementation of a divide-and-conquer algorithm to construct the two-dimensional DT was slowed down dramatically by a factor of 10 000, although they showed that by carefully selecting when to use exact arithmetic, they could reduce the factor to around 5 for points uniformly distributed.

As mentioned in Section 4.2, the incremental insertion algorithm makes its only important decisions based on the result of two geometric predicates. For this reason, if one wants to build a robust algorithm, only these two predicates need to be implemented with exact arithmetic. Observe that we are not interested in the exact values returned by Orient and InSphere, but rather by the sign of the result (although we must be able to detect when the value of the determinant is exactly 0). Floating-point arithmetic will most likely compute correctly the sign of the determinant when the points involved in a predicates are 'clearly' in general position, but when four points are nearly coplanar, there is a chance that Orient returns that they are coplanar. Similarly, five cospherical points can be mistakenly considered not cospherical by InSphere because the result of the determinant is not exactly 0, but perhaps something like $1,555234 \times 10^{-18}$. The problem of robustness of an algorithm is therefore tightly linked to the problems of degeneracies, as it is special cases that will create problems when computing a predicate.

An easy solution for the robustness problem is due to Shewchuk [34, 35]. He developed and implemented—his code is publicly available on the Internet[3]—fast geometric predicates that are 'adaptive', which means that their speed is inversely proportional to the degree of uncertainty of the result. His method works like a filter that will activate exact arithmetic only when it is needed, but since the exact result of the predicate is not sought, as soon as the sign of the determinant can be correctly computed, the process stops. He shows that using his adaptive predicates for building a DT in two dimensions slows down by about 8% the total running time for 1 000 000 randomly generated points, and by about 30% when those points form a tilted grid. In three dimensions, his predicates slows down by about 35% the total running time for 10 000 randomly generated points, and by a factor of 11 when those are generated on the surface of a sphere. As he states: "these predicates cost little more than ordinary nonrobust predicates, but never sacrifice correctness for speed' [35].

7. Practical Performances

The algorithm InsertOnePoint, as described in Section 5, was implemented with the Borland Delphi environment, the object-oriented version of the language `Pascal`. I report in this section experiments that were made on a Pentium 4 (2.8 GHz), with 1 GB of RAM and running the Windows XP Professional operating system. The practical performances of InsertOnePoint are analysed with different datasets, and compared with what is arguably the *de facto* standard in computational geometry implementations, `CGAL` [4]. The source code of `CGAL` was written in `C++`, and can be compiled under different platforms, including Windows. It should be noticed that `CGAL` is not the fastest implementation for the computation of a DT in $\mathbb{R}^3$, as there are reports of implementations that outperform it, e.g. [1] and [28]. The aim of this section is not to show that InsertOnePoint is the fastest—because it is not—but to demonstrate that it is comparable to other solutions available and can realistically be used for real-world applications.

[3]`www.cs.cmu.edu/~quake/robust.html`

For the experiments described in the following, the running time shown is the time only for the operation itself, i.e. the time to read the input file and output the results are not taken into account. Unless explicitly mentioned, the adaptive predicates of Shewchuk [34] are used for both Orient and InSphere for all the operations (which includes for instance Walk), but standard floating-point arithmetic is used for all the other operations. The `CGAL` code (release 3.1, December 2004) was compiled under Windows with `MinGW` (a port to Windows for the `gcc` compiler), and was made 'comparable' to my implementation. In other words, the predicates Orient and InSphere also use robust arithmetic but all other operations are made with floating-point arithmetic[4]. Also, all the 'checks' in `CGAL` have been disabled; these are functions verifying that the combinatorial structure of the DT stays coherent after an operation (disabling them sped up the insertion and the deletion by around 20%).

7.1. Datasets Used

InsertOnePoint was tested with four different datasets, which represent a variety of spatial distributions that one is likely to find, and where degeneracies are present:

50k: a set of 50 000 points randomly distributed in a unit cube;

sphere: a set of 25 000 points randomly distributed on the surface of a sphere;

cubes: a set of 15x15x15 points regularly distributed in the $x-y-z$ directions, forming a cube with 3 375 points;

ocean: a real oceanographic dataset of the Bering Sea[5]. Figure 13 shows the dataset, where the points are distributed along water columns. It contains many of such water columns, and the points are regularly distributed along each column (at regular intervals of depth), although this is not always the case as some data points were removed because of instrument errors. Also, notice that the water columns do not all go to the same depth.

Table 1 contains the details of the four datasets. The number of tetrahedra is the total number including the ones existing because of the big tetrahedron (see Section 4.1), and k is the degree of a vertex in the

[4]The `CGAL` kernel `Exact_predicates_inexact_constructions_kernel` was used.

[5]Taken from the Oceanographic In-Situ Data Access: `http://www.epic.noaa.gov/epic/ewb/`

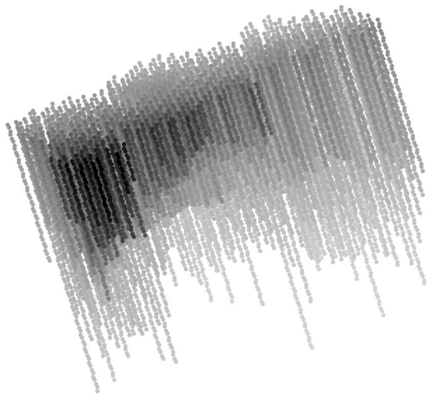

Figure 13. Perspective view of the ocean dataset, formed by many water columns.

	# pts	# tet	k_{min}	k_{max}	k_{avg}
50k	50 000	335 731	5	37	15.4
sphere	25 000	126 392	4	81	11.1
cubes	3 375	18 946	4	14	12.9
ocean	14 550	86 338	6	123	13.8

Table 1. Details concerning the datasets used for the experiments.

DT. Observe that in the case of the ocean dataset, the anisotropic distribution means that one vertex has a degree of 123, but that the number of tetrahedra is still linear.

It should be noticed that the ocean and cubes datasets have a strong *spatial coherence*: in the original ocean file, the water columns are listed one after the other, and in each water column the points are listed from the surface to the bottom of the sea; and the cube file was constructed using three imbricated loops (so the dataset is constructed 'line by line'). For this reason, the order of the points in the two datasets were randomly shuffled to obtain two new datasets: ocean–r and cubes–r.

7.2. Results

To construct a DT, `CGAL` also uses an incremental insertion algorithm, but instead of flipping, the Bowyer-Watson algorithm is used. For the point location strategy, `CGAL` implements the Delaunay hierarchy [9]: for a set S of points, the first DT(S) is constructed, and then other levels are created by sampling some points in S and creating the other DTs. The tetrahedra sharing the vertices between two levels are linked together. The point location involves walking (with WALK) to the tetrahedron at the top level containing the target point, and then going down one level and continuing this way until the tetrahedron in DT(S) containing the target point is found; this is theoretically faster than the algorithm WALK described earlier.

	mine (robust)	mine (float)	CGAL
50k	10.6	7.6	9.6
sphere	20.6	—	49.5
cubes	6.6	1.1	37.9
cubes–r	1.7	0.4	23.3
ocean	16.9	—	58.1
ocean–r	5.4	—	55.8

Table 2. Running times (in seconds) for the construction of the DT of the datasets.

The running times of my implementation versus `CGAL` for the six datasets are reported in Table 2. The results obtained are rather surprising as my implementation is faster for five of the datasets, and by a factor of almost 14 for the cubes–r dataset; the only dataset where my implementation is slower is for 50k, but only by 10%. Notice also that shuffling the cubes and the ocean datasets improved the construction of the DT of my implementation by a factor of respectively 4 and 3; while with `CGAL` the factors were respectively 1.6 and almost no change.

The reasons of such important differences between my implementation and `CGAL` are not totally clear to me because in many reports `CGAL` performs rather well, and can compare easily with the other implementations available. For instance, Boissonnat et al. [4] report constructing the DT of 100 000 points randomly distributed in a cube in 12.1s, on a computer relatively slower than the one used for my experiments (a Pentium 3, 500 MHz with 512 MB of RAM). By comparison, for 100 000 points randomly distributed, `CGAL` takes 19.7s on the computer I used. The major differences were that they used `CGAL` under Linux, and also used integers to store the coordinates of the 100 000 points. The use of integers can speed up an implementation (computation with integers is faster), but unfortunately can not be used in all situations. When dealing with real-world data, the coordinates of the points are usually converted to integers by multiplying them by a constant; this can lead to problems in a dynamic context if the next points to be added are unknown, as there might be a big difference in the order of magnitude of the precision of a point, which would invalid the constant used. Integers are also limited to 4 bytes, which could be problematic for some datasets.

Two reasons could explain the somewhat poor results of `CGAL` I obtained during my experiments. Firstly, because `CGAL` performs very poorly for datasets containing degeneracies (even when they are shuffled), the robust arithmetic used by `CGAL` is probably the bottleneck. The arithmetic used in my implementation is adaptive, while `CGAL` uses a completely different scheme [22], which might not be as fast, especially under Windows. Secondly, it seems that the compiler used, the compiler options, and the platform itself, can have an influence on the speed of the code. Indeed, Liu and Snoeyink [28] report having differences in `CGAL`'s running time of a factor of as much as 2.5 when it was compiled on Linux with different kernels, and that similar differences were found between versions of `CGAL`.

The speed of my implementation, with robust and with standard floating-point arithmetic, was also compared to verify the claims made in Shewchuk [34]. The running times for each dataset are given in Table 2 (a dash '—' means that the program crashed). As expected, the DT of the datasets containing many degenerate cases could not be constructed, although, surprisingly, the cubes and cubes–r datasets could be constructed (this by no means implies that all regularly spaced datasets could have been constructed, since during other experiments many regularly-spaced datasets crashed). The construction of the DT for the 50k dataset is around 40% slower when robust arithmetic is used, and the cubes and cubes–r datasets are slower by a magnitude of respectively 6 and 4.6. These numbers corroborate the results in [34].

More details concerning these experiments can be found in [26].

8. Discussion

Although many aspects of the computation of a 3D VD/DT have been discussed, several others were left out, mostly because of space constraints, but also because they are full research topics on their own.

One of them is the data structure that can be used to store the DT (or the VD). A very simple one, the tetrahedron-based data structure, was used to implement the algorithm and perform the experiments. It considers the tetrahedron as being its atom and stores each tetrahedron with four pointers to its vertices and four pointers to its adjacent tetrahedra. Most implementations use that structure, because it is space-efficient and fast. `CGAL` notably uses it but has added to each vertex a pointer to one of its incident tetrahedra, to speed up the extraction of the Voronoi cells [4]. It is also possible to store the tetrahedra implicitly by considering a data structure where the atom is a triangular face having three pointers to its vertices and six pointers to its adjacent faces [35]. Furthermore, it is possible to store simultaneously, both the DT and the VD [11, 27].

Another aspect is the construction of the DT for very large datasets, and we are talking about many millions of points here. The algorithm as described is not optimised in any ways for datasets for which the data structures do not fit in main memory. Several solutions to this problem have been proposed in the last few years. The most promising methods modify the order of insertion of the points, so that the swapping between the memory and the disk is minimised, see for instance [1, 29, 19]. The good news is that the use of these methods requires only slight modifications to the algorithm described in this paper—no rewriting of a new algorithm or data structure is necessary.

Finally, it should be said that if one does not want to implement the 3D VD/DT algorithm himself, he can always use one of the freely available implementations or libraries, such as `CGAL`!

References

[1] N. Amenta, S. Choi, and G. Rote. Incremental constructions con BRIO. In *Proceedings 19th Annual Symposium on Computational Geometry*, pages 211–219, San Diego, USA, 2003. ACM Press.

[2] F. Aurenhammer. Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.

[3] C. B. Barber, D. P. Dobkin, and H. T. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, 1996.

[4] J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, and M. Yvinec. Triangulations in CGAL. *Computational Geometry—Theory and Applications*, 22:5–19, 2002.

[5] B. Boots. Delaunay triangles, an alternative approach to point pattern analysis. In *Proceedings, Association of American Geographers, 6*, pages 26–29, 1974.

[6] A. Bowyer. Computing Dirichlet tessellations. *Computer Journal*, 24(2):162–166, 1981.

[7] P. Cignoni, C. Montani, and R. Scopigno. DeWall: A fast divide & conquer Delaunay triangulation algorithm in E^d. *Computer-Aided Design*, 30(5):333–341, 1998.

[8] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational geometry: Algorithms and applications.* Springer-Verlag, Berlin, second edition, 2000.

[9] O. Devillers. The Delaunay hierarchy. *International Journal of Foundations of Computer Science*, 13(2):163–180, 2002.

[10] O. Devillers, S. Pion, and M. Teillaud. Walking in a triangulation. *International Journal of Foundations of Computer Science*, 13(2):181–199, 2002.

[11] D. P. Dobkin and M. J. Laszlo. Primitives for the manipulation of three-dimensional subdivisions. *Algorithmica*, 4:3–32, 1989.

[12] D. Douglas. It makes me so CROSS. Unpublished manuscript from the Harvard Laboratory for Computer Graphics and Spatial Analysis. Reprinted in Peuquet DJ, Marble, DF, editors, *Introductory Readings in Geographic Information System*, pages 303–307. Taylor & Francis, London, 1974.

[13] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15:223–241, 1996.

[14] D. A. Field. Implementing Watson's algorithm in three dimensions. In *Proceedings 2nd Annual Symposium on Computational Geometry*, volume 246–259, Yorktown Heights, New York, USA, 1986. ACM Press.

[15] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[16] C. M. Gold, T. D. Charters, and J. Ramsden. Automated contour mapping using triangular element data structures and an interpolant over each triangular domain. In *Proceedings Siggraph '77*, volume 11(2), pages 170–175, 1977.

[17] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4:74–123, 1985.

[18] C. M. Hoffmann. The problems of accuracy and robustness in geometric computation. *Computer—IEEE Computer Society Press*, 22:31–42, 1989.

[19] M. Isenburg, Y. Liu, J. R. Shewchuk, and J. Snoeyink. Streaming computation of Delaunay triangulations. *ACM Transactions on Graphics*, 25(3):1049–1056, 2006.

[20] B. Joe. Three-dimensional triangulations from local transformations. *SIAM Journal on Scientific and Statistical Computing*, 10(4):718–741, 1989.

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

[22] V. Karamcheti, C. Li, I. Pechtchanski, and C. K. Yap. A CORE library for robust numeric and geometric computation. In *Proceedings 15th Annual Symposium on Computational Geometry*, volume 351–359, Miami, Florida, USA, 1999. ACM Press.

[23] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, 1991.

[24] C. L. Lawson. Software for C^1 surface interpolation. In J. R. Rice, editor, *Mathematical software III*, pages 161–194. Academic Press, 1977.

[25] C. L. Lawson. Properties of n-dimensional triangulations. *Computer Aided Geometric Design*, 3:231–246, 1986.

[26] H. Ledoux. *Modelling three-dimensional fields in geoscience with the Voronoi diagram and its dual.* PhD thesis, School of Computing, University of Glamorgan, Pontypridd, Wales, UK, 2006.

[27] H. Ledoux and C. M. Gold. Simultaneous storage of primal and dual three-dimensional subdivisions. *Computers, Environment and Urban Systems*, 2006. In press.

[28] Y. Liu and J. Snoeyink. A comparison of five implementations of 3D Delaunay tessellation. In J. E. Goodman, J. Pach, and E. Welzl, editors, *Combinatorial and Computational Geometry*, volume 52, pages 439–458. Cambridge University Press, 2005.

[29] Y. Liu and J. Snoeyink. TESS3: A program to compute 3D Delaunay tessellations for well-distributed points. In *Proceedings 2nd International Symposium on Voronoi Diagrams in Science and Engineering*, pages 225–234, Seoul, Korea, 2005.

[30] E. P. Mücke. A robust implementation for three-dimensional Delaunay triangulations. *International Journal of Computational Geometry and Applications*, 8(2):255–276, 1998.

[31] E. P. Mücke, I. Saias, and B. Zhu. Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. *Computational Geometry—Theory and Applications*, 12:63–83, 1999.

[32] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial tessellations: Concepts and applications of Voronoi diagrams.* John Wiley and Sons, second edition, 2000.

[33] A. Saalfeld. It doesn't make me nearly as CROSS: Some advantages of the point-vector representation of line segments in automated cartography. *International Journal of Geographical Information Systems*, 1(4):379–386, 1987.

[34] J. R. Shewchuk. Adaptive precision floating point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18:305–363, 1997.

[35] J. R. Shewchuk. *Delaunay Refinement Mesh Generation.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburg, USA, 1997.

[36] J. R. Shewchuk. Sweep algorithms for constructing higher-dimensional constrained Delaunay triangulations. In *Proceedings 16th Annual Symposium on Computational Geometry*, pages 350–359, Hong Kong, 2000. ACM Press.

[37] J. R. Shewchuk. Updating and constructing constrained Delaunay and constrained regular triangulations by flips. In *Proceedings 19th Annual Symposium on Computational Geometry*, pages 181–190, San Diego, USA, 2003. ACM Press.

[38] K. Sugihara and I. Hiroshi. Why is the 3D Delaunay triangulation difficult to construct? *Information Processing Letters*, 54:275–280, 1995.

[39] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *Computer Journal*, 24(2):167–172, 1981.

[40] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–486. World Scientific Press, second edition, 1995.