ELSEVIER

PII: S0010-4485(97)00082-1

Computer-Aided Design, Vol. 30, No. 5, pp. 333–341, 1998

0010-4485/98/$19.00+0.00

DeWall: A fast divide and conquer Delaunay triangulation algorithm in E^d

P Cignoni†, C Montani† and R Scopigno*

The paper deals with Delaunay Triangulations (DT) in E^d space. This classic computational geometry problem is studied from the point of view of the efficiency, extendibility to any dimensionality, and ease of implementation. A new solution to DT is proposed, based on an original interpretation of the well-known Divide and Conquer paradigm. One of the main characteristics of this new algorithm is its generality: it can be simply extended to triangulate point sets in any dimension. The technique adopted is very efficient and presents a subquadratic behaviour in real applications in E^3, although its computational complexity does not improve the theoretical bounds reported in the literature. An evaluation of the performance on a number of datasets is reported, together with a comparison with other DT algorithms. © 1998 Published by Elsevier Science Ltd. All rights reserved.

Keywords: Delaunay triangulation, divide and conquer, uniform grids

INTRODUCTION

Triangulation is one of the main topics in computational geometry and it is commonly used in a large set of applications, such as computer graphics, scientific visualization, robotics, computer vision and image synthesis, as well as in mathematical and natural science. Given a point set P, the Delaunay Triangulation (DT) is a particular triangulation, built on the points in P, which satisfy the empty circum-circle property: the circum-circle (-sphere in E^3 or -hypersphere in E^d) of each simplicial cell in the triangulation does not contain any input point $p \in P$. Many algorithms were proposed for the DT of a set of sites in E^2, E^3 or E^d, and most of them are reviewed in Ref. [1,2].

Unfortunately there has been little research into implementations and performance evaluations of Delaunay triangulators. Few papers report evaluations of real implementations or give experimental comparisons of different algorithms. Worst case time complexities are generally given, but such analyses, from the point of view of the application programmer, are not always sufficient to make the correct decisions. In fact, theoretically better algorithms can sometimes be outperformed by more naive methods; the theoretical asymptotic worst case complexity sometimes fails to consider the optimization techniques that can be applied to reduce the expected complexity.

A new divide and conquer DT algorithm is proposed in this paper. The algorithm gives a general and simple solution to DT in E^d space and makes use of accelerating techniques which are specific to computer graphics.

The paper is organized as follows: definitions and a taxonomy of Delaunay triangulation algorithms are presented in the second section. The proposed algorithm is described in detail in the third section, together with some optimization techniques. The performances of the proposed solution are evaluated on a number of datasets and compared with other solutions in the penultimate section. Conclusions are drawn in the last section.

DELAUNAY TRIANGULATION

Given a point set P in E^d, a *k-simplex*, with $k \leq d$, is defined as the convex combination of $k + 1$ affinely independent points in P, called vertices of the simplex (e.g. a triangle is a 2-simplex and a tetrahedron is a 3-simplex). An *s-Face* of a simplex is the convex combination of a subset of $s + 1$ vertices of the simplex (i.e. a *2-face* is a triangular facet, *1-face* is an edge, *0-face* is a vertex).

A triangulation Σ defined on a point set P in E^d space is the set of *d-simplices* such that:

(1) a point p in E^d is a vertex of a simplex σ in Σ if $p \in P$;
(2) the intersection of two simplices in Σ is either empty or a common face;
(3) the set Σ is maximal: there does not exist any simplex σ that can be added to Σ without violating the previous rules.

A triangulation Σ is a *Delaunay Triangulation* if the hypersphere circumscribing each simplex does not contain any point of the set P[3,4]. The Delaunay triangulation of a given point set P is unique if these do not exist in P $d + 2$ points lying on the same hypersphere. Such cases, also known as degeneracies, can be managed by using local perturbation schemes[5].

*To whom correspondence should be addressed at: CNUCE—Consiglio Nazionale delle Ricerche, Via S. Maria 36, 56126, Pisa, Italy. Tel: +39 50 593304; E-mail: r.scopigno@cnuce.cnr.it
†I.E.I. - Consiglio Nazionale delle Ricerche, Via S. Maria 46, 56126 Pisa, Italy
Paper Received: 9 August 1996. Revised: 25 October 1997. Accepted: 25 October 1997

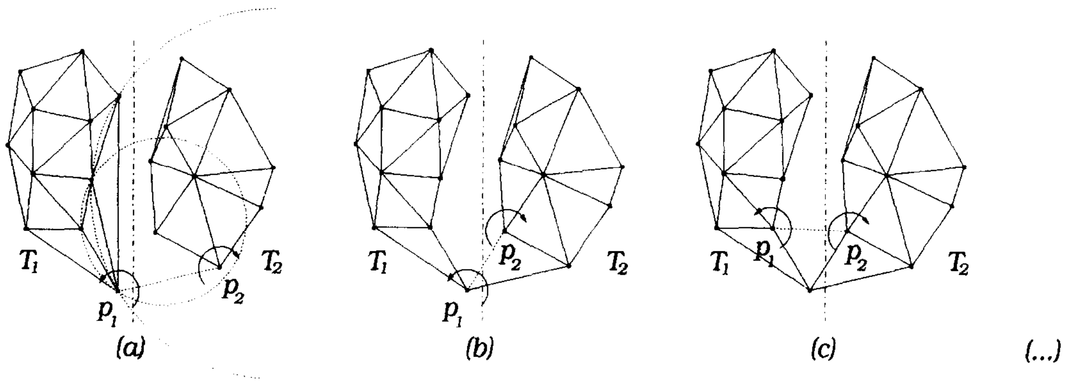

Figure 1 Merging of two partial DT in E^2 space

The duality between DTs and Voronoi diagrams is well known[4] and therefore algorithms are given for the construction of DT from Voronoi diagrams. However, direct construction methods are generally more efficient because the Voronoi diagram does not need to be computed and stored. Direct DT algorithms[1] can be classified as follows:

- *local improvement*—starting with an arbitrary triangulation, these algorithms locally modify the faces of pairs of adjacent simplices according to the circum-sphere criterion;
- *on-line* (or *incremental insertion*)—starting with a simplex which contains the convex hull of the point set, these algorithms insert the points in P one at a time: the simplex containing the currently added point is partitioned by inserting it as a new vertex. The circum-sphere criterion is tested on all the simplices adjacent to the new ones, recursively, and, if necessary, their faces are flipped;
- *incremental construction*—the DT is constructed by successively building simplices whose circum-hyperspheres contain no points in P;
- *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 simply projecting the convex hull into E^d; for a comparison of the different approaches see[6];
- *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. Current algorithms are not generalized to E^d space, but limited to E^2 space alone.

On-line methods[7,8] hold the lower worst case time complexity, $O(n \log n + n^{\lceil d/2 \rceil})$. Moreover, these methods in their naive implementation are simple to program and can be generalized to manage point sets in E^d space.

D&C solutions hold in E^2 the same complexity as *on-line* methods, but a general D&C E^d ($d > 2$) solution has not been proposed yet. The main problem here is the design of the merging phase. Because of the explicit ordering of the edges incident in a vertex (*Figure 1*), the merging phase is simple in E^2[9], but hard to design in E^d where this ordering is not given.

The algorithm proposed in this paper by-passes this problem by reversing the order between the solutions of subproblems and the merging phase. The classical D&C algorithms recursively subdivide the input points, construct two partial DTs and then merge them. Our solution is based on a more complex division phase, in which the input dataset P is split into P_1 and P_2, and a section of the DT is immediately built. This partial triangulation allows the algorithm to recursively triangulate the two point sets P_1 and P_2, taking into account the border of the partial triangulation and avoiding the need for a further merging phase. A "merging" simplex set is thus built before the subproblems are solved: we partition the problem solution, instead of its instance. The partial triangulation can be built very simply using a constructive rule similar to McLain's in its *incremental construction* approach[10]. This means we can specify a general E^d D&C Delaunay triangulator. Its simple structure permits an efficient implementation using some well known optimization techniques.

THE DEWALL ALGORITHM

A new algorithm for the DT of a point set P in E^d is presented in this section. The algorithm is based on the D&C paradigm, but this paradigm is applied in a different way with respect previous DT algorithms[9,11]. The general structure of D&C algorithms is: divide the input data into subset P_1 and P_2; recursively solve on P_1 and P_2; and merge the partial results S_1 and S_2 to build solution S.

In the case of triangulations, the input point set P can easily be divided using a cutting plane such that the two associated halfspaces contain two point sets P_1 and P_2 of comparable cardinality. The problem is how to implement the merging phase, i.e. how to build the union of the two solutions S_1 and S_2. This union requires the triangulation of the space separating S_1 and S_2, and generally also requires a number of local modifications to S_1 and S_2. As previously stated, this problem was efficiently solved for the E^2 case[9,11], but not for the general E^d case.

Our approach to D&C is slightly different. Instead of merging partial results, we apply a more complex dividing phase which partitions the point set and builds, as first step, the *merging* triangulation. The algorithm is then recursively applied to triangulate the two subsets of the input dataset P.

The splitting plane α separates the point set P into two subsets P_1 and P_2. Analogously, the splitting plane α divides a triangulation Σ into three disjoint subsets: the simplices that are intersected by the plane, which we call the *simplex wall* Σ_α, and the two sets of simplices Σ_1 and Σ_2 that are completely contained in the two halfspaces defined by α (*Figure 2*). Σ_α can be chosen as a valid *merging* triangulation: (a) each $\sigma \in \Sigma_\alpha$ is also in Σ and (b) subtracting Σ_α from Σ generates two disconnected simplicial complexes Σ_1 and Σ_2.

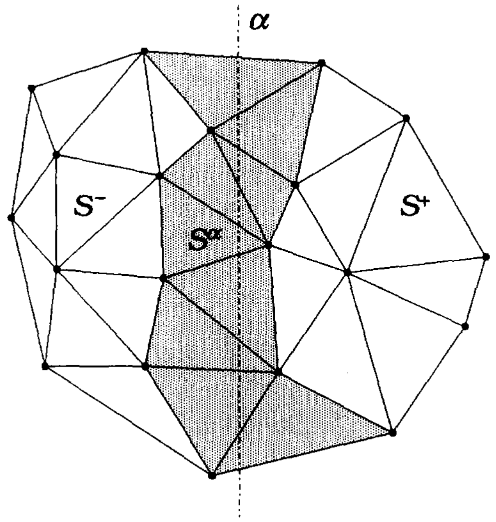

Figure 2 An example of DT in E^2: α is the dividing line,and Σ_α (the set of gray triangles) is the associated simplex wall; Σ_1 and Σ_2 are the triangulations returned by the recursive invocation of the DeWall algorithm on the two point set partitions

The DeWall (Delaunay Wall) algorithm, specified in pseudo-pascal in *Figure 3*, consists of the following steps:

- select the dividing plane α, split P into the two subsets P_1 and P_2 and construct Σ_α;
- starting from Σ_α, recursively apply DeWall on P_1 and P_2 to build Σ_1 and Σ_2;
- return the union of Σ_α, Σ_1 and Σ_2.

The technique used to build the simplex wall Σ_α is a slight variation on an incremental construction algorithm; it is described in the next section.

Incremental construction of the simplex wall

The simplex wall can be simply computed by using an *incremental construction* approach: a starting simplex is individuated and then Σ is built by adding a new simplex at each step and without having to modify the current triangulation. This technique for DT was originally proposed in E^2 by McLain[10], and then applied by Dobkin and Laszlo[12] for E^3 subdivisions.

The incremental construction approach can be easily generalized to E^d triangulations: for each $(d-1)$-face f, which does not lie on the *ConvexHull(P)*, there are exactly two simplices σ_1 and σ_2 in Σ, such that σ_1 and σ_2 share the $(d-1)$-face f. the algorithm starts by constructing an initial simplex σ_i; then, it processes all of the $(d-1)$-faces of σ_i: the simplex adjacent to each of them (if it exists, i.e. the face does not belong to the Convex Hull of P) is built and added to the current list of simplices in Σ. All of the new $(d-1)$-faces of each new simplex are used to update a data structure, here called Active Face List (AFL). Update of the AFL is as follows: if a new face is already contained in AFL, then it is removed from AFL; otherwise, it is inserted in AFL because its adjacent simplex has not yet been built. The process continues iteratively (extract of face f from AFL, build the simplex σ adjacent to f, update the AFL with the $(d-1)$-faces of σ, and then again extract another face from AFL) until AFL is empty. In the implementation of the *AFL* data structure, the efficiency of the most common operations (Insert, Extract, Delete, Member) has to be guaranteed. Our implementation of the AFL data structure is based on hash indexing, making it possible to manage AFL in nearly constant time (an average of 1.15–1.5 accesses to the hash table were measured with the current implementation to solve a query).

Given this general incremental construction algorithm, we only need to specialize it for the construction of Σ_α. In particular we have to detail: (a) how to build the initial simplex (the *MakeFirstSimplex* function), (b) how to build the simplex adjacent to a face f (the *Make Simplex* function), and (c) how this construction process can be limited to the simplices in Σ_α.

Construction of first simplex

The function *MakeFirstSimplex* produces a Delaunay d-simplex which is intersected by the plane α, in order to start from this simplex the incremental construction of the simplex wall Σ_α.

MakeFirstSimplex selects the point $p_1 \in P$ nearest to the plane α. It then selects a second point p_2 such that p_2 on the other side of α. Then, it searches the point p_3 such that the circum-circle around the 1-face (p_1,p_2) and the point p_3 has the minimum radius; (p_1,p_2,p_3) is therefore a 2-face of Σ. The process continues until the required d-simplex is built.

Construction of the generic simplex

Given a face f, the function *MakeSimplex* builds the adjacent simplex by applying the DT definition. For each point $p \in P$, *MakeSimplex* computes the radius of the hypersphere which circumscribes p and the face f. We choose the point p which, generally speaking, minimizes this radius to build the simplex adjacent to f.

MakeSimplex selects the point p which minimizes the function *dd* (Delaunay distance):

$$dd(f,p) = \begin{cases} r & \text{if } c \in \text{Halfspace}(f,p) \\ -r & \text{otherwise} \end{cases}$$

with r and c the radius and the center of the circumsphere around f and p; given the plane on which f lies, Halfspace(f,p) returns the halfspace which contains the new tetrahedra.

Let us introduce the following example to illustrate this definition: Let us assume that there exists a subset of points Q, which are contained in Halfspace(f,p) and are located on a straight line which intersects the face f. If these points are processed in order of decreasing distances from f, and the centers of the circumspheres are computed, we can observe that these circumsphere radii will decrease until we get the first point $q_i \in Q$ whose circumsphere center is located in the opposite halfspace (the one which does not contains the points in Q). Then, for all successive points q_k, $k > i$, the radii will start to increase. The *dd* distance defined previously takes into account this decreasing–increasing behaviour of the circumsphere radius.

The analysis of the points $p \in P$ is limited to the points which lie in the *outer* halfspace with respect to face f (i.e. the halfspace which does not contain the previously generated simplex that originates face f).

The outer halfspace associated with f contains no point of P *if* face f is part of the Convex Hull of P (the faces on the Convex Hull are the only faces that belong to just one simplex in the triangulation). In this case the algorithm correctly returns no adjacent simplex and, in this case only, *MakeSimplex* returns *null*.

A simple solution to reduce the cost of *MakeSimplex* function is to take into account, for each point p in P, the

```
Function DeWall (P : point_set, AFL : (d-1)face_list) : d-simplex_list;
    var f : (d-1)face; AFLα, AFL1, AFL2 : (d-1)face_list;
        t : d-simplex; Σ : d-simplex_list; α : splitting_plane;
    begin
        AFLα, AFL1, AFL2:=emptylist;
        Pointset_Partition(P, α, P1, P2);
    /* Simplex Wall Construction */
        if AFL = ∅ then
                t:=MakeFirstSimplex(P, α);
                AFL:=(d-1)faces(t); Insert(t,Σ);
        for each f ∈ AFL do
                if IsIntersected(f,α) then Insert(f, AFLα);
                if Vertices(f) ⊂ P1 then Insert(f, AFL1);
                if Vertices(f) ⊂ P2 then Insert(f, AFL2);
        while AFLα ≠ ∅ do
                f:=Extract(AFLα);
                t:=MakeSimplex(f, P);
                if t ≠ null then
                        Σ:=Σ ∪ {t};
                        for each f': f' ∈ (d − 1)faces(t) AND f' ≠ f do
                            if IsIntersected(f',α) then Update(f',AFLα)
                            if Vertices(f') ⊂ P1 then Update(f',AFL1)
                            if Vertices(f') ⊂ P2 then Update(f',AFL2);
        /* Recursive Triangulation */
        if AFL1 ≠ ∅ then Σ:=Σ ∪ DeWall(P1,AFL1);
        if AFL2 ≠ ∅ then Σ:=Σ ∪ DeWall(P2,AFL2);
        DeWall:=Σ;
    end;

Procedure Update (f :face, L : face_list);
    begin;
        if Member(f,L) then Delete(f, L)
                else Insert(f, L);
    end;
```

Figure 3 DeWall algorithm

current triangulation progress status. As soon as all of the simplices incident in p were built, p may be removed from P and it will no longer be tested in the further invocations of *MakeSimplex*. The control on the number of incident simplices was implemented with a counter associated with each vertex p, increased each time a new face incident in p is built and decreased for each invocation of *MakeSimplex* on an incident face; as soon as the counter returns zero, p may be deleted from P.

Construction of simplices in Σ_α alone

A slight modification to the canonical incremental construction approach is needed to build only those simplices intersected by the splitting plane α. Instead of using a single list of active faces (AFL), the algorithm uses three disjoint lists containing:

- AFL_α: the $(d-1)$-faces intersected by plane α;
- AFL_1: the $(d-1)$-faces with all of the vertices in P_1;
- AFL_2: the $(d-1)$-faces with all of the vertices in P_2;

For each simplex σ, the algorithm inserts its $(d-1)$-faces in the suitable face list. It then extracts faces (on which the next simplices will be built) from the AFL_α alone; this ensures that each simplex built is part of the simplex wall Σ_α.

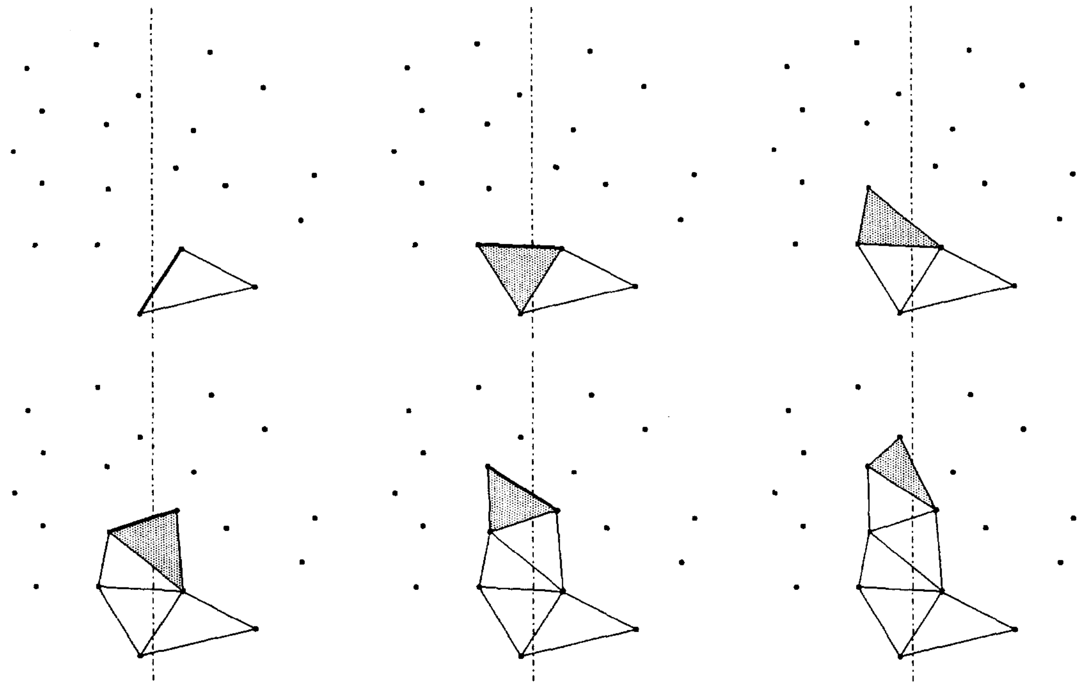

Figure 4 Incremental construction of the simplex wall (first steps in a 2D example)

The simplex wall construction process terminates when AFL_α is empty. This process returns both Σ_α and the pair of active face lists AFL_1 and AFL_2. DeWall is then recursively applied to the pairs (P_1,AFL_1) and (P_2,AFL_2), unless all the active face lists are empty. The splitting plane α is cyclically selected as a plane orthogonal to the axes of the E_d space (X, Y or Z in E^3), in order to recursively partition the space with a regular pattern. Two-dimensional examples of the simplex wall construction and of the recursive application of the algorithm are shown in *Figures 4 and 5*, respectively.

Uniform grid

The DeWall algorithm is simple and easy to implement although in its naive implementation the asymptotic time complexity is not optimal nor is its practical efficiency good. An analysis of the algorithm shows that the main inefficiency is in the *MakeSimplex* function.

Each simplex is constructed from an adjacent simplex face, by finding the *dd*-nearest point (i.e. the nearest according to the *dd* metric). This search entails performing an $O(n)$ test for each simplex, where n is the number of sites in P.

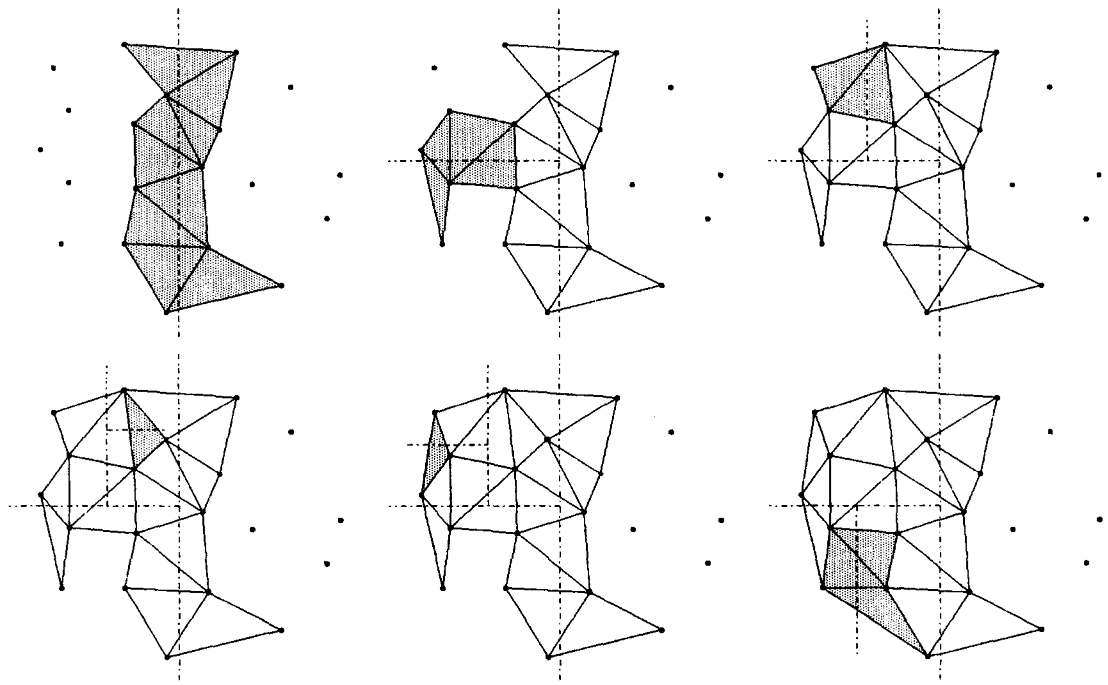

Figure 5 Some steps of the DeWall algorithm on a point set in E^2

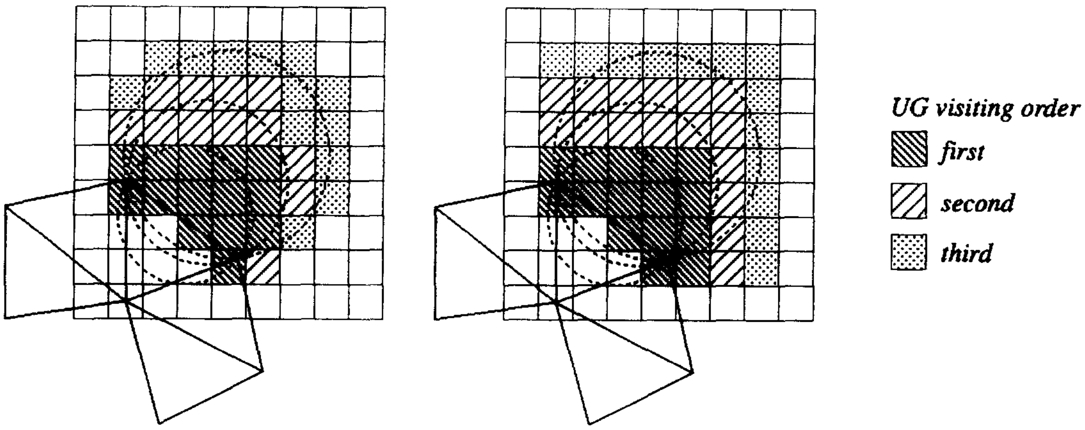

Figure 6 On the left, a 2D example of the cell visiting order of Maus (sphere scan conversion) and, on the right, our technique (based on the analysis of all the UG cells contained in the bounding box of each sphere)

However, the construction of a new simplex in expected constant time is possible.

The concept of *local processing* is often adopted in computer graphics either to speed up sequential algorithms or to achieve parallelism. The speed-up technique proposed here is based on the E^d extension of the *uniform grid* (UG)[13]; for simplicity, the use of the UG is described here for the case of DT in E^3, supporting a regular partition of the space into hexahedral cells;

$$UG = \{c_{ijk}\};\ i, j, k \in [0..N] \tag{1}$$

The main reason why uniform grid techniques are effective in geometric computations is that two points, which are far apart, generally have little or no effect on each other. A large class of geometric algorithms possesses this property, ranging from visibility, to modeling (boolean operations, intersection detection, etc.) and computational geometry (point location, triangulation, etc.)[14].

The *uniform grid* is used as an indexing scheme for the fast detection of the *dd*-nearest point. A similar technique was also used by Fang and Piegl[15,16] to speed up incremental 2D and 3D Delaunay triangulation.

The space E^3 is partitioned into cubic cells following a regular pattern. The *UG* structure is built in a preprocessing phase, by computing for each cell c_{ijk} the subset of points in P contained in c_{ijk}.

The *MakeSimplex* function is designed such that, analogously to Maus's proposal[17], the UG is scanned in order of increasing distance from f. Given this partial ordering of the sites, not all the points in P have to be analyzed for each face f. In fact, given a point p_1 such that $dd(f,p_1) = d_1$, all the points which are not contained in the sphere around f and p_1 will certainly have a dd value greater than d_1, and it is pointless to evaluate their dd value. The analysis of the cells of UG can be stopped when there are no more cells contained in the circumsphere around f and the current *dd*-nearest point (*Figure 6*).

The cells scanning order used is simpler than that proposed by Maus. Indeed we do not test the cells contained in circumspheres with increasing radius (the sphere to cells conversion is not a simple task) but we simply select and test all of the cells contained in the smallest cube circumscribed to each circumsphere. This method is simpler because it avoids the scan-conversion of spheres, and the number of cells selected is not much higher. Note that if the sphere radius selected is small (up to three times the cell edge length) the discretized circumsphere and the circumcube are identical.

The choice of the right resolution for the uniform grid space crucially affects the efficiency of the algorithm. In the reported implementation, the resolution of the UG is defined such that the number of cells is equal to the number of sites.

DeWall time complexity

The worst-case time complexity of the DeWall algorithm may be misleading: neither of the two techniques used (D&C strategy and Uniform Grid optimization) guarantee worst case optimality whilst they do offering good performances in practical situations. It is possible to define patological datasets which cancel the efficiency of both the D&C strategy and the UG: if DeWall is applied to the dataset depicted in *Figure 7*, the construction of the first wall originates the entire triangulation (all the simplices in the triangulation intersect the splitting plane α); analogously, it is possible to choose site distributions that make the Uniform Grid not useful at all. In these pathological

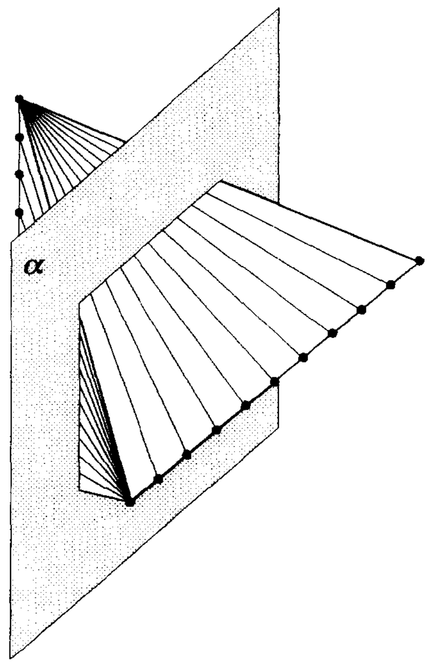

Figure 7 The worst-case input dataset for the DeWall algorithm

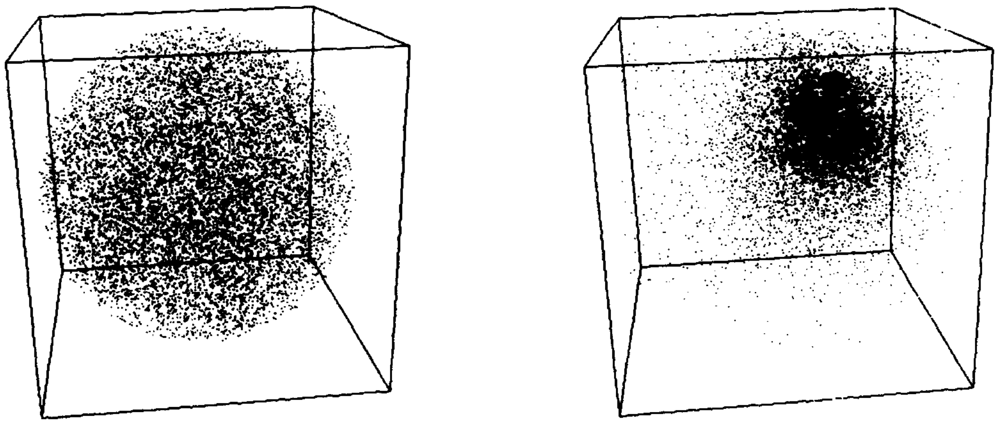

Figure 8 Spatial distribution of the sites: *uniform* dataset on the left, *bubbles* on the right

situations the DeWall algorithm reduces itself to an incremental construction algorithm, yielding a $O(n^{\lceil d/2 \rceil + 1})$ worst case time complexity. In spite of this result, the algorithm behaves well in practical cases (as shown in Section 4) yielding, in the three-dimensional case, a plain subquadratic behaviour versus a $O(n^3)$ worst case complexity.

DeWall space complexity

The algorithm space requirements are bounded by the space complexity of:

- the point set P;
- the active fact list AFL; each $AFL(n,d)$ is always a set of connected $(d-1)$-faces forming a unique $(d-1)$ surface in E^d. Recalling that the number of $(d-1)$-faces of a polytope in E^d of n vertices is at most $O(n^{\lfloor d/2 \rfloor})$, the worst case space complexity of $AFL(n,d)$ is $O(n^{\lfloor d/2 \rfloor})$;
- the outcoming triangulation; however, like the incremental construction algorithms, DeWall can return each simplex as soon as it is built, avoiding explicitly storing the triangulation at run time.

Therefore, the worst case space complexity of DeWall is $O(n^{\lceil d/2 \rceil})$, so it is interesting to note that the maximum space required by the algorithm in this worst case is lower than (or at most equal to in E^2) the size of the outcoming triangulation. In contrast, *on line* triangulators need the current triangulation to be stored which is generally represented by the use of a hierarchical structure which holds the history of the construction process for fast point-in-triangle computations.

RESULTS AND EMPIRICAL EVALUATION

The performance of the algorithm was tested on two classes of datasets. The first class consists of *uniform* datasets, where the locations of sites are generated using a uniform probability distribution function (*Figure 8*). In the second dataset class, the sites are organized into a number of *bubbles* with the density of sites decreasing as the distance from the bubble center increases (*Figure 8*). The sites in each bubble are generated using an approximation of a normal probability distribution function.

For each dataset class and for each resolution (number of sites), a number of different datasets were generated in E^3; the times reported in *Table 1* and *Table 2* are the means of the run times measured on each dataset. The machine used for the timings was an SGI Indigo workstation (MIPS R4000 cpu); the times include the uniform grid preprocessing. The results obtained show an empirically estimated complexity which is clearly subquadratic in E^3.

Table 1 Processing times, in seconds, required to triangulate the *uniform* dataset with various triangulations, plus statistical information. [$\#(\sigma \in \Sigma)$: number of tetrahedra in the final triangulation; $\#(\sigma \in$ *first* $\Sigma_\alpha)$: number of tetrahedra on the first simplex wall; #(*cels visited*): mean number of cells visited to build a single tetrahedra; *max*(*sites per cell*): maximum number of sites contained in each UG cell]

Uniform dataset (No. of sites)	2000	4000	6000	8000	10000
De Wall					
times (no opt.)	32.7	100.3	211.1	352.7	516.4
times (UG opt.)	4.4	9.4	14.8	20.7	26.5
$\#(\sigma \in \Sigma)$	12642	25736	39024	52390	65469
$\#(\sigma \in$ *first* $\Sigma_\alpha)$	1497	2396	3106	3666	4385
#(*cells visited*)	12.86	13.15	14.43	14.15	14.15
max(*sites per cell*)	8	8	9	8	10
Incode					
times (no opt.)	218.8	976.	2306.	4433.	
times (UG opt.)	5.8	13.8	22.7	32.6	43.1
Qhull					
times	5.34	23.33	29.88	44.64	71.96
Detri					
times	33.11	64.59	101.36	144.87	169.41

Table 2 Triangulation of the *bubble* datasets using different triangulators (processing times in seconds)

Bubble dataset(No. of sites)	2000	4000	6000	8000	10000
De Wall					
times (UG opt.)	8.3	20.6	24.6	31.1	56.0
#(cells visited)	14.70	13.55	13.21	12.40	16.47
max(sites per cell)	250	496	1,178	536	200
Incode					
times (UG opt.)	10.7	33.0	38.9	53.3	96.2
Qhull					
times	5.10	12.00	18.04	23.15	30.47
Detri					
times	32.55	67.51	105.82	156.00	188.14

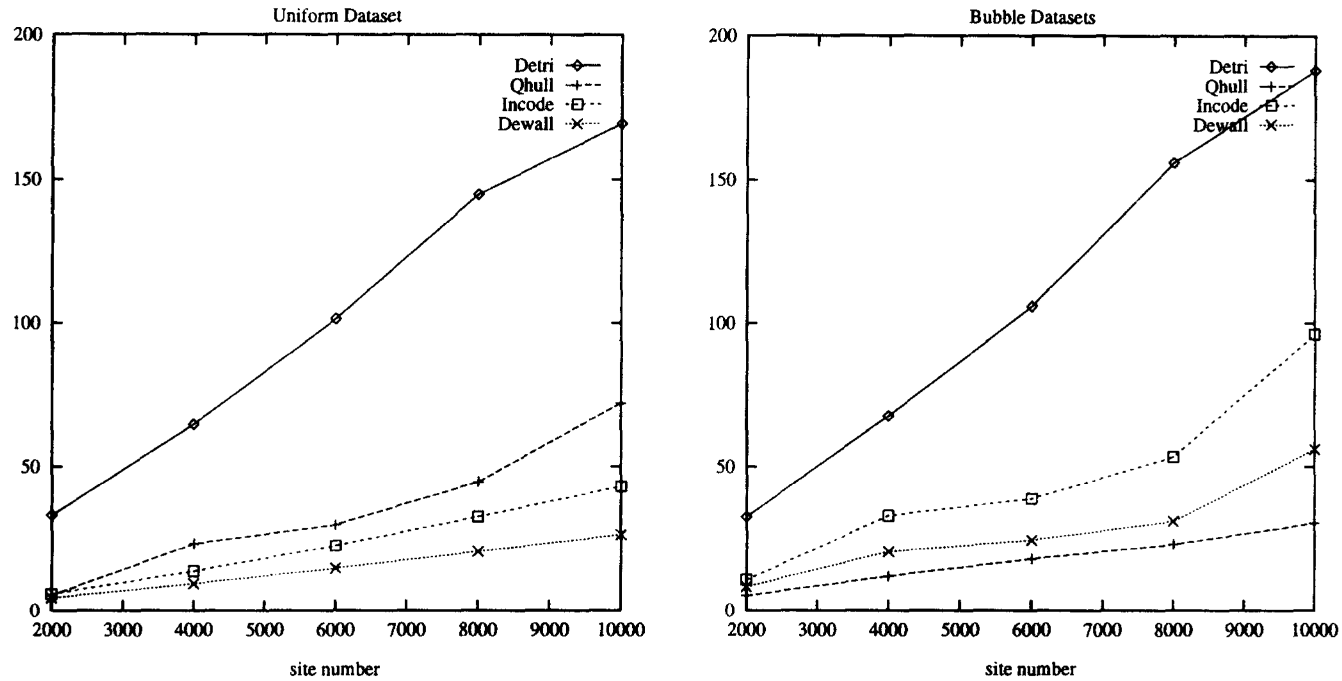

Figure 9 The algorithm times in seconds: *uniform* datasets on the left, *bubble* on the right

Another way to empirically evaluate DeWall is to compare it with other implementations. We tested DeWall against two efficient Delaunay triangulators that are publicly available:

- Incode: a totally *incremental construction* algorithm, with and without the use of the UG optimization technique*. Incode was implemented by using most of the DeWall's code;
- Qhull: a general dimension code for computing convex hulls and Delaunay triangulations. It is an implementation of the Quickhull algorithm[19] for computing the convex hull†. It was chosen because it qualifies as the fastest convex hull code for large datasets defined in low dimension spaces;
- Detri: as part of the alpha-shape software, Detri builds the 3D DT by adopting an *incremental insertion and flip* approach[7]‡.

The results in *Table 1* and their graphical representation in *Figure 9* show that DeWall is the most efficient of the four software programs on regularly distributed datasets, while it gives slightly slower times than Qhull on the bubble datasets. This is justified by the lower speed-up obtained by adopting a UG on irregularly distributed datasets; the bubble datasets contain the worst distribution of sites for algorithms that use a UG (and therefore the DeWall algorithm).

Some statistics on the execution of the DeWall algorithm on the *uniform* dataset are also reported in *Table 1*. The total number of tetrahedra returned is considerably lower than the theoretical upper bound in E^3, $O(n^2)$: it was linear with the number of sites (approximately $7*n$) in our experiments. The growth of the number of tetrahedra in the first wall is clearly sublinear (approximately $O(n^{2/3})$).

The mean number of cells visited for the construction of each simplex is not constant but shows a low increase with the dataset resolution. This is because for each face f on the *ConvexHull(P)* all of the cells contained in the positive half-space of f have to be tested.

The simplices which do not lie on the *ConvexHull(P)* need, on average, a constant number of cell tests. The

* Incode and DeWall are available in public domain at the address http://miles.cnuce.cnr.it/cg/swOnTheWeb.html

† Qhull is provided by the Geometry Center, University of Minnesota; the Qhull software may be downloaded from the WWW site http://freeabel.-geom.umn.edu/software/download/qhull.html

‡ Detri is provided by the Software Development Group at the National Center for Supercomputing Applications (NCSA); info may be downloaded from the WWW site http://www.ncsa.uiuc.edu/SDG/Software/Brochure/Overview/ALVIS.overview.html

increase in the mean number of cells visited is therefore justified by the increase in the faces on the *ConvexHull(P)*. Finally, the maximum number of sites per cell is reported in *Tables 1 and 2*.

CONCLUSIONS

The DeWall algorithm was presented as an original solution to Delaunay triangulation, based on a particular interpretation of the D&C paradigm. This new approach has greatly simplified the merging phase and makes it possible to define a general D&C solution for point sets defined in any dimension.

Optimization techniques were designed to speed up the proposed algorithm. Our results show how common computer graphics techniques (e.g. data indexing and optimized point selection) can dramatically increase the efficiency of a typical computational geometry task. The optimality of the DeWall algorithm from the viewpoint of asymptotic complexity is hard to prove. However, the experimental results are interesting and show an empirically estimated complexity which is clearly subquadratic in E^3.

REFERENCES

1. Aurenhammer, F., Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Survey*, 1991, **2**(3), 345–405.
2. Su, P. and Drysdale, S. R. L., A comparison of sequential delaunay triangulation algorithms. In *11th ACM Computational Geometry Conf. Proc. (Vancouver Canada)*, ACM Press, 1995, 61–70.
3. Delaunay, B., Sur la sphere vide. *Bull. Acad. Science USSR VII: Class. Sci. Mat. Nat.*, 1934, 793–800.
4. Preparata, F. P. and Shamos, M. I., *Computational Geometry: an Introduction*. Springer-Verlag, 1985.
5. Edelsbrunner, H. and Mücke, E. P., Simulation of simpicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transaction on Graphics*, 1990, **9**(1), 66–104.
6. Avis, D. and Bremner, D., How good are convex hull algorithms? In *Proceedings 11th A.C.M. Symposium on Computational Geometry*, Vancouver, Canada, 1995. ACM Press, pp. 20–28.
7. Edelsbrunner, H. and Shah, N. R., Incremental topological flipping works for regula triangulaions. In *Proceedings of the 8th Annual ACM Symposium on Computational Geometry*, June 1992, pp. 43–52.
8. Guibas, L. J., Knuth, D. E. and Sharir, M., Randomized incremental construction of Delaunay and voronoy diagrams. In *Automata, Languages and Programming, LNCS N.443*. Springer-Verlag, 1990, pp. 414–431.
9. Lee, D. T. and Schachter, B. J., Two algorithms for constructing a Delaunay triangulation. *Int. J. of Computer and Information Science*, 1980, **9**(3), 219–242.
10. McLain, D. H., Two dimensional interpolation from random data. *The Computer J.*, 1976, **19**(2), 178–181.
11. Dwyer, R. A., A faster divide and conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 1987, **2**, 137–151.
12. Dobkin, D. P. and Laszlo, M. J., Primitives for the manipulation of three-dimensional sub-divisions. *Algorithmica*, 1989, **4**, 3–32.
13. Akman, V., Franklin, W. R., Kankanhalli, M. and Narayanaswami, C., Geometric computing and uniform grid technique. *Computer-Aided Design*, 1989, **21**(7), 410–420.
14. Narayanaswami, C., *Parallel Processing for Geometric Applications*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, December 1990.
15. Fang, T. P. and Piegl, L. A., Delaunay triangulation using a uniform grid. *IEEE Computer Graphics and Applications*, 1993, **13**(3), 36–47.
16. Fang, T. P. and Piegl, L. A., Delaunay triangulation in three dimensions. *IEEE Computer Graphics and Applications*, 1995, **15**(5), 62–69.
17. Maus, A., Delanuay triangulation and the convex hull of n points in expected linear time. *Bit.*, 1984, **24**, 151–163.
18. Edelsbrunner, H., *Algorithms in Combinatorial Geometry*. Springer-Verlag, Berlin, 1987.
19. Bradord Barber, C., Dobkin, D. P. and Huhdanpaa, H. The quickhull algorithm for convex hull. Tech. Rep. GCG53-93, Geometry Center, University of Minnesota, July 1993.

Paolo CIGNONI is research scientist at the Istituto di Elaborazione della Informazione of the National Research Council in Pisa, Italy. His research interests include computational geometry and its interaction with computer graphics, scientific visualization and volume rendering. Cignoni received in 1998 a PhD degree in Computer Science from the University of Pisa.

Claudio MONTANI is a research director with the Istituto di Elaborazione della Informazione of the National Research Council in Pisa, Italy. His research interests include data structures and algorithms for volume visualization and rendering of regular or scattered datasets. Montani received an advanced degree (Laurea) in Computer Science from the University of Pisa in 1977. He is a member of IEEE.

Roberto SCOPIGNO is senior scientist at the Istituto CNUCE of the National Research Council in Pisa, Italy; since 1990 he has had a joint appointment at the Department of Computer Engineering of the University of Pisa. His research interests include interactive graphics, scientific visualization, volume rendering and parallel processing. Scopigno received an advanced degree (Laurea) in Computer Science from the University of Pisa in 1984. He is a member of IEEE and Eurographics.