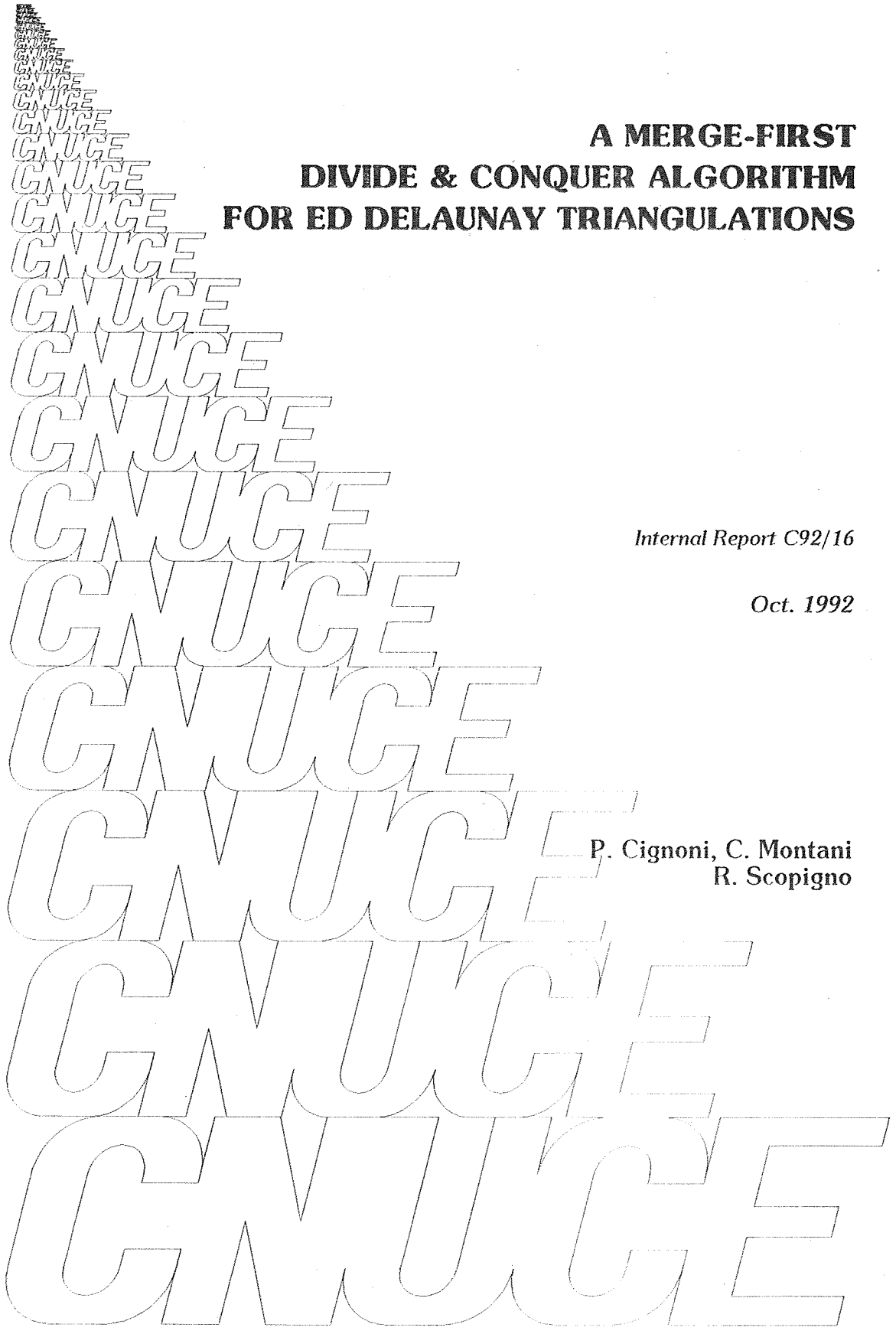

A MERGE-FIRST DIVIDE & CONQUER ALGORITHM FOR ED DELAUNAY TRIANGULATIONS

Internal Report C92/16

Oct. 1992

P. Cignoni, C. Montani
R. Scopigno

A new Merge-First Divide & Conquer Algorithm for E^d Delaunay Triangulations *

P. Cignoni*, C. Montani**, R. Scopigno***

* Dip. di Informatica, Universita' degli Studi, C.so Italia 40, 56100 Pisa, ITALY

** Istituto Elaborazione dell'Informazione, Consiglio Nazionale delle Ricerche

Via S. Maria 46, 56126 Pisa, ITALY

*** Istituto CNUCE, Consiglio Nazionale delle Ricerche

Via S. Maria 36, 56126 Pisa, ITALY

October 16, 1992

Abstract

The paper deals with Delaunay triangulations in E^d space, a classic problem in computational geometry, from the point of view of the efficiency and the easy parallelization of the algorithms. The application field is the processing and visualization of large scattered datasets; in this field, the real time visualization of time-varying phenomena is a typical requirement.

An extension in E^d of an algorithm originally proposed for E^2 Delaunay triangulations and two simple and effective speedup techniques is first proposed and a new algorithm based on a original interpretation of the well-known Divide and Conquer paradigm is then presented. Although the computational complexity of the algorithm does not improve the theoretical results reported in the literature, the technique is very efficient and present a quasi linear behaviour in real applications. Thanks to the use of a D&C technique, the algorithm can be easily parallelized on low grain parallel architectures (such as shared memory MIMD machines or networks of workstations). An evaluation of the performance

*This work was partially funded by the Progetto Finalizzato "Sistemi Informatici e Calcolo Parallelo" of the Consiglio Nazionale delle Ricerche.

on medium resolution datasets is reported.

1 Introduction

Triangulation is a main topic in computational geometry and is commonly used in a large set of applications, such as robotics, computer vision, image synthesis, as well as in mathematics and natural science. Delaunay Triangulation (DT) is a particular triangulation which holds the property that the circumcircle of any triangle in the triangulation contains no point in its interior. Many algorithms have been proposed for the DT of a set of sites in E^2, E^3 or E^d [2]. Volume Rendering is one of the more recent applications of DT. A *volume* dataset consists of sampled points in E^3 space, with one or more scalar or vector sample values associated with each point. The need for a visual representation of the content of such huge datasets has encouraged a substantial research effort and has led to a new computer graphics area, known as Volume Rendering [5] [9]. The spatial arrangement of the point set can be either structured, with implicit or explicit topological relations between the sites, or unstructured. In the latter case, the triangulation of the set of points in E^3 is a prerequisite to the execution of a class of surface reconstruction or direct rendering algorithms. The large number of sites common by present in Volume Rendering applications imposes strong efficiency constraints on the triangulator used.

Unfortunately for the application programmer, implementation evaluations of Delaunay triangulators are lacking in the literature. Few papers report evaluations of real implementations or give experimental comparisons of different algorithms. Asymptotic time complexities are generally given, but such analyses are not always sufficient to make the correct decisions. In fact, theoretically better algorithms can sometimes be outperformed by more naive methods, because the theoretical asymptotic complexity do not always consider the management of the complex data structures needed and the optimization techniques that can be applied to reduce the mean case complexity.

A new Divide & Conquer DT algorithm is proposed in this paper. This algorithm gives a general and extremely simple Divide & Conquer solution to the DT in E^d space. The paper is also an attempt to use some classical computer graphics techniques to increase the performance of the triangulator. Our attempt follows Guibas's proposal for greater cooperation and integration between computer graphics and computational geometry [6].

This new algorithm is not an advance in terms of asymptotic complexity, but gives good results

for empirically measured performances showing nearly linear run times.
Moreover, the intrinsic parallelizability of the Divide & Conquer paradigm is one point of strenght of our proposal with respect to the asymptotically optimal but inherently sequential "on line" DT algorithm [8].

In the paper we define the DT in E^d space and give a brief description of Delaunay triangulation algorithms. We introduce an incremental construction algorithm and some optimization tecniques which are at the base of the Divide & Conquer algorithm then described in detail. Finally, an evaluation of the measured performance of the algorithms is reported and some conclusions are presented.

2 Delaunay Triangulation

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

The *Delaunay Triangulation* (*DT(P)*) [14] of the E^d space defined on a point set P in E^d is the set of d-simplices such that:

1. a point p in E^d is vertex of a simplex in DT(P) iff $p \in P$;
2. the intersection of two simplices in DT(P) is either a empty or common face;
3. the hyper-sphere circumscribed around the $n+1$ vertices of each simplex contains no other point of the set P.

The DT is the dual of the Voronoi diagram, and therefore algorithms are given for the construction of DT from Voronoi diagrams. But there are also direct methods for the construction of DT and they are generally more efficient because the Voronoi diagram does not need to be computed and stored.
DT algorithms [2] can be classified as follows:

- *local improvement* methods start with an arbitrary triangulation and then locally modify the faces of pairs of adjacent simplices according to the equiangularity criterion;

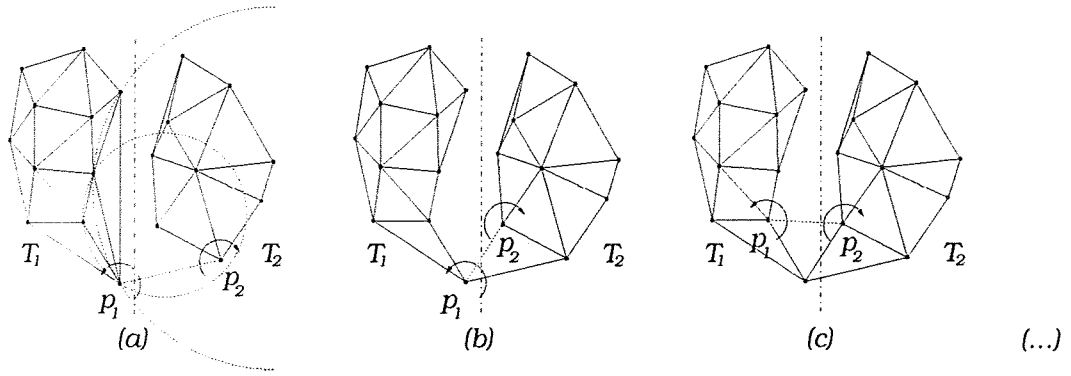

Figure 1: Merging of two partial DT in E^2 space.

- *on-line* (or *incremental insertion*) methods start with a simplex which contains the *Convex-Hull*(P); for each newly added point in P, the simplex containing that point is partitioned by inserting the new vertex. All the simplices adjacent to the new one are then recursively tested for equiangularity in topological order and, if necessary, their faces are flipped;

- *incremental construction* methods use the empty circle property to construct the DT by successively adding simplices whose circum-circles/spheres contain no points in P (Figure 2);

- *higher dimensional embedding* methods transform the points in 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 in E^d;

- *divide & conquer* (D&C) methods are based on the recursive partition and local triangulation of the point set, and then on a phase in which the resulting subset of simplices are merged. Current algorithms are not generalized to E^d space, but limited to the E^2 space only.

On-line methods [7] [8] hold the lower asymptotic time complexity, $\mathcal{O}(n\ log\ n + n^{\lceil \frac{d}{2} \rceil +})$. Moreover, these methods extremely simple to program and can be simply generalized to manage pointset in E^d space.

The increasing complexity of the input dataset and the current technological trend are giving ever increasing importance to parallel solutions. From this point of view, *on-line* algorithms are clearly penalized for being inherently sequential. On the other hand, D&C methods can easily be parallelized and have been proved to be optimal for the E^2 space [10]. Unfortunately, no general D&C solution has been proposed to manage E^3 or E^d pointsets. The problem here

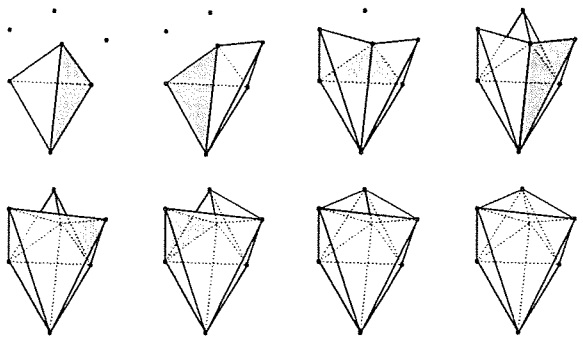

Figure 2: An example of incremental DT construction in E^3 space.

is in the specification of the merging phase, which is simple in E^2 due to the explicit ordering of the edges incident in a vertex(Figure 1). In E^d this ordering is not given, and the specification of the merge phase is thus hard.

The algorithm proposed in this paper bypasses this problem by reversing the order of the sub-problem solution and the merging phase. The original D&C algorithm recursively subdivides the input dataset, constructs the two DT's and then merges them. The solution here first subdivides the input dataset, then builds the part of the DT that would be built in the merging phase of a classical D&C algorithm and finally recursively triangulates the two semi–spaces taking into account the border of the previously computed merging triangulation. In this way, the merging phase is executed before the subproblem solution.

The merging triangulation can be built in a very simple manner using a constructive rule similar to that proposed by Mc Lain [12]. This makes possible the specification of a general E^d D&C triangulator, with a simple structure that permits an efficient implementation using some well known optimization techniques.

In order to simplify the description of the algorithm, we first present a generalized E^d extension of the Mc Lain algorithm in the next section togheter with a description of the speedup techniques adopted.

3 An Incremental Construction Algorithm

An algorithm for DT in E^2 based on the incremental construction approach was proposed by McLain [12]. In this section we present the **InCoDe** (Incremental Construction of Delaunay triangulation) algorithm, that is a generalized E^d extension of the Mc Lain's algorithm.

The algorithm starts from a simplex s in DT(P), where P is the target point set. It incrementally builds the DT(P), by adding a new simplex in each step and without having to modify the current triangulation.

The algorithm is based on the existence of a pair of adjacent simplices for each (d-1)face which does not lie in the $ConvexHull(P)$. Starting from the initial simplex, the (d-1)faces are visited and the simplex adjacent to each of them is added to the current list of simplices. All of the (d-1)faces of each added simplex are inserted into the Active Face List (AFL). Insertion in the AFL is as follows: if the new face f is already contained in AFL, then f is removed from AFL; otherwise, f is inserted in AFL and this implies that the simplex adjacent to f has not yet been built. The process continues iteratively (extract a face f' from AFL, build the simplex s' adjacent to f', insert the (d-1)faces of s' in AFL, and then again extract another face from AFL) until AFL is empty.

The algorithm can be specified in pseudo–Pascal as follows:

```
Function InCoDe (P : set_of_points) : d-simplex_list;
    var f : (d-1)face; AFL : (d-1)face_list;
        t : d-simplex; DT : d-simplex_list;
    begin
        AFL:=∅; DT:=∅;
        t:=MakeFirstSimplex(P);
        Insert(DT,t);
        for each f ∈ t do Insert(AFL,f);
        while AFL ≠ ∅ do
            begin
                f:=Extract(AFL);
                t:=MakeSimplex(f, P);
                if t ≠ null
                    then begin
                        Insert(DT,t);
                        for each f': f' ∈ (d − 1)face(t) AND f' ≠ f do
                            if f' ∈ AFL
                                then Delete(AFL,f')
                                else Insert(AFL,f')
```

```
                end;
            end;
        InCoDe:=DT;
    end;
```

Two questions arise:
(a) how to build the simplex adjacent to a face f (the *MakeSimplex* function);
(b) how to identify the initial simplex (the *MakeFirstSimplex* function).

Given a face f, the adjacent simplex can be simply identified by using the Delaunay simplex definition. For each point $p \in P$, the ray of the hypersphere which circumscribes p and the d vertices of the face f is computed. The point p which, generally speaking, minimizes this ray is chosen to build the simplex adjacent to f.
We limit our analysis of the points $p \in P$ by considering only those 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 contains face f).
The outer halfspace associated with f contains any point *iff* face f is part of the Convex Hull of the pointset P: in this case the algorithm correctly returns no adjacent simplex and, in this case only, *MakeSimplex* returns *null*. The faces on the Convex Hull are the only faces that belong to only one simplex of the DT(P).
For each point p in the outer halfspace of f, the radius of the circumsphere is evaluated. The algorithm selects the point which minimizes the function *dd* (Delaunay distance):

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

with r and c the ray and the center of the circumsphere around f and p.

Two approaches can be adopted to determine the first simplex (MakeFirstSimplex function). The first is based on a technique used to identify the first face of the convex hull of a set of points [14], from which the first simplex can be simeasilyply built.
The second approach is more simple: having randomly chosen a point $p_1 \in P$, search the point $p_2 \in P$ such that the Euclidean distance $d(p_1, p_2)$ is minimal; then, search the point p_3 such that the circum-circle about the 1-face (p_1, p_2) and the point p_3 has minimum ray: the points (p_1, p_2, p_3) are a 2-face of the DT. The process continues in the same way until the required (d-1)face is built.

The InCoDe algorithm is simple and easy to implement though neither is its asymptotic time complexity optimal nor its practical efficiency good.

3.1 Speedup techniques

An analysis of the algorithm shows that there are two main bottlenecks: the *MakeSimplex* function (given face f, $dd(f,p)$ must be computed for each point $p \in P$) and the *Active Face List* management (each insertion/extraction has a complexity which is proportional to the current number of stored faces).

3D Gridding

In the InCoDe algorithm a new simplex is constructed from a simplex face, by finding the dd-nearest point (i.e. the nearest according the *dd* metric). This search entails scanning the whole dataset and an $\mathcal{O}(n)$ test for each simplex. However, the construction of a new simplex in constant time is possible.

The basic concept of *local processing* is often adopted in computer graphics either to speedup sequential algorithms or to achieve parallelism. The speedup technique proposed here is based on the 3D extension of the *uniform grid* [1]

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

i.e., a regular, non hierarchical 3D partition of the dataset space. The grid cells partition the dataset space without overlap or omission, and the grid resolution adapts to the data. For simplicity, the use of the UG is described here in the case of DT in E^3, but it can be easily generalized to E^n.
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 possess this property, ranging from visibility (hidden surface removal), to modeling (boolean operations, intersection detection, etc.) and to computational geometry (point location, triangulation, etc.) [13]. Local processing can, therefore, generally be adopted to tackle geometric problems.

The *uniform grid* is used as an indexing scheme for the fast detection of the *dd*-nearest point. The space E^3 is partitioned into cubic cells following a regular pattern. Given the dataset P, it is possible to associate each cell c_{ijk} with the list of points in P that are contained in c_{ijk}.

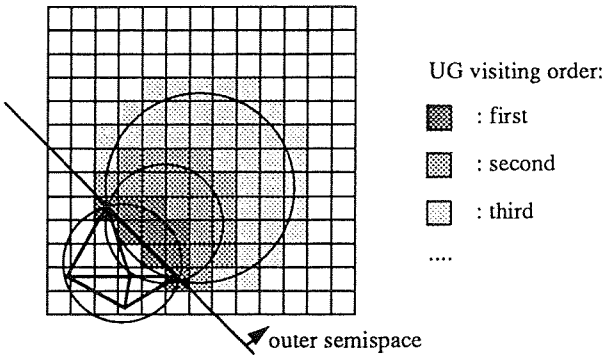

Figure 3: An example of the cell visiting order, shown for simplicity in the case of the 2D Delaunay triangulation.

A preprocessing phase builds the UG structure.

The idea is then to organize the *MakeSimplex* function so that the points in P are examined in order of increasing dd-distance from face f. Analogously to Maus's approach [11], the UG can be 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_i such that $dd(f, p_i) = d_i$, all the points which are not contained in the sphere around f and p_i will certainly have a dd value greater than d_i, 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 3).

The choice of the right resolution for the uniform grid space crucially affects the efficiency of the algorithm. In the implementation reported the resolution of the UG is defined as follows. The length e of the edge of the UG cell is calculated in terms of the volume of the 3D bounding box ($BBox$) of the dataset P :

$$e = \sqrt[3]{\frac{Vol(BBox(P))}{n}} \tag{2}$$

with n the cardinality of P.

Face Lists Management via Hash Tables

The following operations are executed on the AFL data structure are:

- *Insert(f, AFL)*: insertion of a face f into the list;
- *Extract(f, AFL)*: extraction of a face f from the list;
- *Delete(f, AFL)*: removal of a face f from the list;

- *Member(f, AFL)*: true IF face $f \in AFL$.

A time linear to the number of elements in the list considerably reduces the overall efficiency of the algorithm. Runs of the algorithm with the AFL implemented as a simple list showed that $O(n)$ accesses to the list were required on n points dataset.
The current implementation of the AFL data structure is based on hash coding: the hash code associated to each face makes it possible to manage AFL in nearly constant time (1.15 - 1.5 accesses for each operation were measured with the current implementation).

3.2 InCoDe complexity

The asymptotic time complexity $C_{InCoDe}(n)$ of the InCoDe algorithm can be evaluated as follows. Given:

- n : number of points in P;
- $F(n)$: number of (d-1)faces in the DT(P), with F(n)=$\mathcal{O}(n^{\lceil d/2 \rceil^+})$ in E^d;
- *MakeSimplex(n)* : complexity of the function which builds the Delaunay simplex given a face $f \in DT(P)$;
- $AFL(m)$: complexity of the function which manages the active face list (insertion, extraction and deletion) in terms of the number m of faces contained;

the complexity of the InCoDe algorithm is:

$$C_{InCoDe}(n) = F(n) * (MakeSimplex(n) + AFL(m)) \tag{3}$$

If hash coding is used to manage the AFL, then $AFL(m)$ can be considered a constant time operation.
Moreover, if a uniform grid is applied to reduce the MakeSimplex cost, the construction of a new simplex s over the (d-1)face f implies the computation of the *dd* distance:
(a) for each grid cell c contained in the cubic subvolume which bounds the circum-sphere over the simplex s;
(b) for each $p \in P$ contained in c.

Given k_c the mean number of cells visited in (a), k_p the mean number of sites in each cell and assuming that $dd(f, p_i)$ is a constant time operation, the mean complexity of the MakeSimplex function is:
MakeSimplex(n) = $\mathcal{O}(k_c * k_p)$.

Mean number of sites contained in each cell. Assuming that the probability that a site p falls in cell c is $p = 1/n$ (with n both the number of sites in P and of cells in the UG), the probability $P(k, n)$ that k sites from a n site dataset are contained in a cell c can be evaluated as a binomial distribution as follows:

$$\mathrm{P(k,n)} = \binom{n}{k} \mathrm{p^k (1-p)^{n-k}}$$

$$\mathrm{P(k,n)} = \binom{n}{k} \frac{1}{\mathrm{n^k}} \left(1 - \frac{1}{\mathrm{n^k}}\right)^{\mathrm{n-k}} = \binom{n}{k} \frac{(\mathrm{n^k} - 1)^{\mathrm{n-k}}}{\mathrm{n^{k(n-k)+k}}}$$

$$\mathrm{P(k,n)} \leq \frac{\mathrm{n!}}{\mathrm{k!(n-k)!}} \frac{1}{\mathrm{n^k}} \leq \frac{1}{\mathrm{k!}}$$

Therefore the mean number of sites contained in a cell is:

$$\mathrm{k_p} = \sum_{k=1}^{n} kP(k,n) \leq \sum_{k=1}^{n} k \frac{1}{k!} < 3$$

Mean number of cells visited. The number k_c of cells visited to construct a simplex s adjacent to the (d-1)face f is $\mathcal{O}(\frac{r^d}{vol(c)})$, with r the ray of the circum-sphere over s and $vol(c)$ the volume of a single cell of the uniform grid. In the worst case of Figure 6 each circum-sphere contains most of the cells, and therefore $k_c = \mathcal{O}(n)$. For a more general and probable spatial arrangement of the sites in P, the number k_c is at least sublinear with n because the size of the simplices, and therefore the rays of the associated circum-spheres, decreases as the number of sites rises.

From the above assumptions and from (3), the complexity of the InCoDe algorithm can be estimated:

$$\mathcal{C}_{InCoDe}(n) \in \mathcal{O}(n^{\lceil d/2 \rceil^+} * k_c) \tag{4}$$

The empirical k_c values measured while running the algorithm are reported in Section 5.

4 DeWall: a Divide and Conquer Evolution of the InCoDe Algorithm

A new algorithm for the DT of a pointset P in E^d is presented in this section. The algorithm is based on the D&C paradigm, applied different by than the usual D&C DT algorithms [10] [4]. The general structure of the Divide and Conquer algorithms is:

```
Div&Conq (P : problem_instance, var S : problem_solution);
    begin
        if | P | ≤ k
            then  S:=Solve(P)
            else begin
                Split(P, P1, P2);
                Div&Conq(P1, S1);
                Div&Conq(P2, S2);
                S:=Merge(S1, S2);
                end;
    end;
```

In our problem, the point set P can easily be split using a cutting plane such that the two associated halfspaces contain two pointsets 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 triangulations S_1 and S_2. This union requires the triangulation of the space which separates S_1 and S_2. This triangulation generally requires a number of local modifications on S_1 and S_2. As described in Section 2, the merging phase can efficiently be solved for Delaunay triangulation in E^2 [10] (Figure 1), whereas there is still no solution for generalized Divide and Conquer DT in E^d.

Our approach to Divide and Conquer is slightly different and it emerged from an attempt to design an efficient merging phase for generalized DT in E^d space.
The main idea is simple. Instead of merging partial results, we first built the merging triangulation and then compute separate triangulations on the two subsets of the input dataset P. The resulting **Merge-FirstD&C** paradigm is applied as follows:

```
Merge-firstD&C (P : problem_instance, M : partial_solution, var S : problem_solution);
    begin
        if | P |≤ k
            then  S:=Solve(P, M)
            else begin
                Split(P, P1, P2);
                SMerge:=Merge(P1, P2, M);
                Merge-firstD&C(P1, SMerge, S1);
                Merge-firstD&C(P2, SMerge, S2);
                S := SMerge ∪ S1 ∪ S2;
                end;
    end;
```

The efficiency of this approach depends on the complexity of the construction of the Delaunay simplicial complex S_{Merge}. The efficiency is increased if the computations of the *Merge* phase are exploited in the subsequent recursive invocation of Merge-firstD&C.

The DeWall Algorithm

The **DeWall** algorithm is an implementation of the above Merge-firstD&C paradigm.

A splitting plane α divides the space E^d into two halfspaces, called PosHalfspace(α) and NegHalfspace(α).

A simplex s intersects the plane α *iff* $\exists\ f\ \mid\ f \in faces(s)\ \wedge\ Is\_Intersected(f, \alpha)$.

A splitting plane α divides a triangulation DT(P) into three disjoint subsets: the simplices that are intersected by the plane, which we call the simplex wall S^α, the simplices S^+ that are completely contained in the PosHalfspace(α), and those completely contained in the NegSemispace(α). S^- (Figure 4).

S^α is a valid candidate for the required S_{Merge}: it is part of a valid DT(P) and it partitions the DT(P) such that the resulting S^+ and S^- have no intersection, because (a) for each simplex $s \in DT(P)$ s is intersected by α iff $s \in S^\alpha$ and (b) if $S^+ \bigcap S^- \neq \{\}$ then at least a simplex s' must exist such that $s' \in S^+ \bigcap S^-$ and s' is intersected by α, which is absurd by definition of simplex wall.

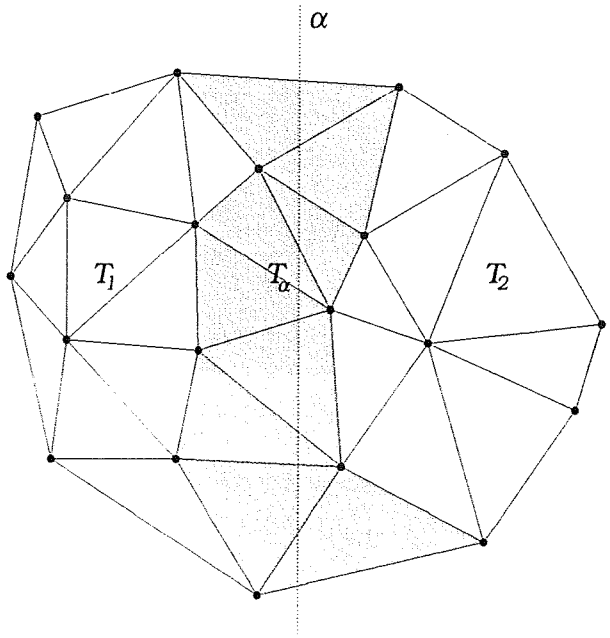

Figure 4: An example of DT in E^2: α is the splitting line, and T_α (the set of gray triangles) is the associated simplex wall; T_1 and T_2 are the triangulations returned by the recursive invocation of the DeWall algorithm on the two pointset partitions.

The efficiency of the algorithm depends on how well balanced is the partition of P operated by α.

The DeWall algorithm consists of the following steps:

- select the splitting plane α;
- construct S^α and the two subsets P_1 and P_2;
- recursively apply DeWall on P_1, starting from S^α, and build DT_1;
- recursively apply DeWall on P_2, starting from S^α, and build DT_2;
- return the union of S^α, DT_1 and DT_2.

Simplex Wall Construction

The technique used is a slight variation of the InCoDe algorithm described in Section 3. The algorithm works on (d-1)faces and incrementally builds simplices on them. The difference is

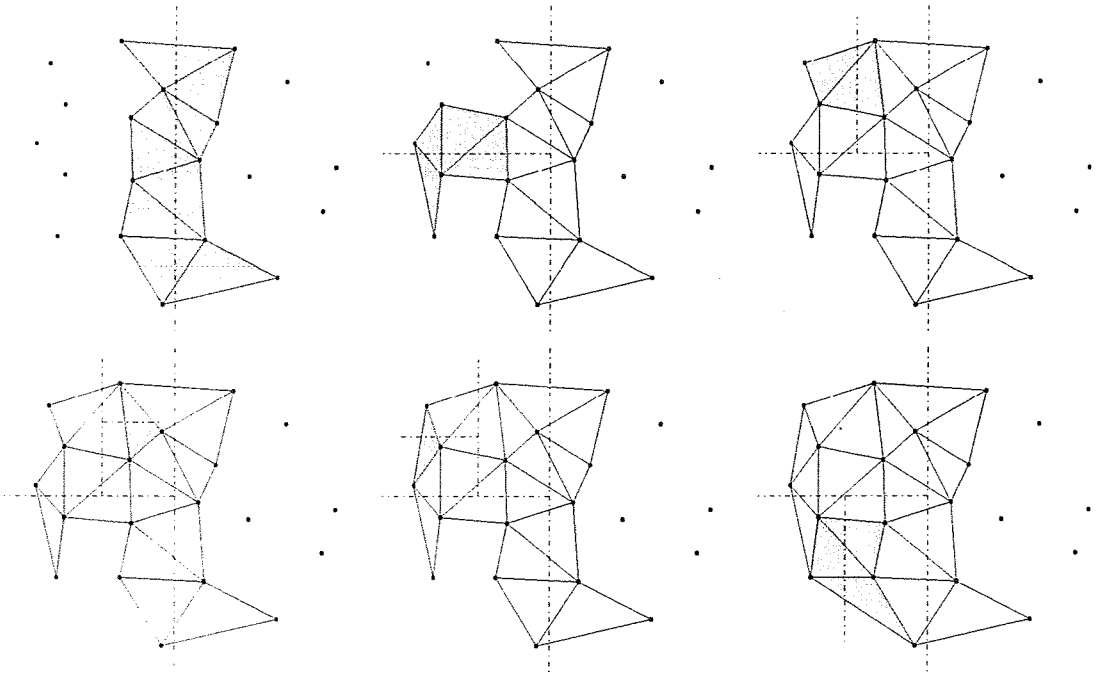

Figure 5: Some phases of the DeWall algorithm on a pointset in E^2.

that instead of using a single list of active faces (AFL), three different lists are defined, which contain:

- AFL_α : the faces which are intersected by the plane α;
- AFL_+ : the faces which are completely contained in the *positive* halfspace defined by the plane α;
- AFL_- : the faces which are completely contained in the *negative* halfspace defined by the plane α.

For each simplex s, the algorithm inserts each (d-1)face of s in one of the above face lists. It then extracts faces (on which the next simplices will be searched) from the AFL_α only; this ensures that each simplex is part of the simplex wall S^α.
The simplex wall construction process terminates when the AFL_α is empty. The process returns both the S^α) and the pair of active face lists AFL_+ and AFL_-.
DeWall is then recursively applied to the tuple (P_1, AFL_-) and (P_2, AFL_+), unless all the active face list are empty.

```
Function DeWall (P : set_of_points, AFL : (d-1)face_list) : d-simplex_list;
    var f : (d-1)face; AFL_α, AFL_-, AFL_+ : (d-1)face_list;
        t : d-simplex; DT : d-simplex_list;
        α : splitting_plane;
    begin
        AFL_α, AFL_-, AFL_+:=∅; DT:=∅;
        Pointset_Partition(P, α, P_1, P_2);
        if AFL = ∅ then
                t:=MakeFirstWallSimplex(P, α);
                AFL:=d-faces(t); DT:=t;
        for each f ∈ AFL do
            if IsIntersected(f,α) then Insert(f, AFL_α)
            else if f ⊂ NegHalfSpace(α) then Insert(f, AFL_-)
            else Insert(f, AFL_+);
        while AFL_α ≠ ∅ do begin
            f:=Extract(AFL_α);
            t:=MakeSimplex(f, P);
            if t ≠ null then begin
                DT:=DT ∪ t;
                for each f': f' ∈ (d-1)faces(t) AND f' ≠ f do
                    if IsIntersected(f',α) then Update(f',AFL_α)
                    else if f' ⊂ NegHalfSpace(α) then Update(f',AFL_-)
                    else Update(f',AFL_+);
                end;
            end;
        if AFL_- ≠ ∅ then DT:=DT ∪ DeWall(P_1,AFL_-);
        if AFL_+ ≠ ∅ then DT:=DT ∪ DeWall(P_2,AFL_+);
        DeWall:=DT;
    end;

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

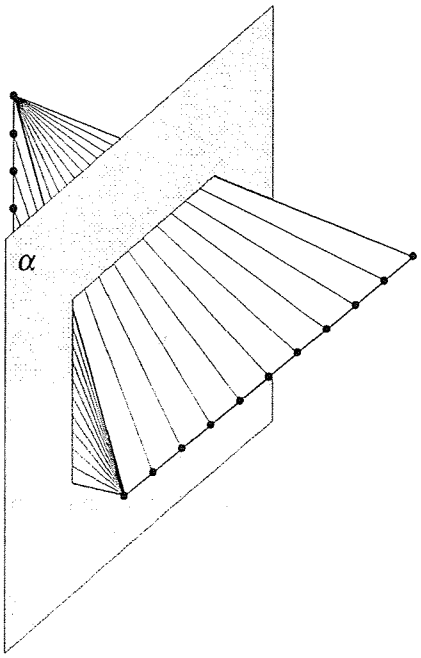

Figure 6: The worst-case input dataset for the DeWall algorithm.

end;

The splitting plane α is cyclically selected as a plane which is 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 (Figure 5).

The function *MakeFirstWallSimplex* returns a Delaunay d-simplex which is intersected by the plane α, the first simplex on which the simplex wall is constructed. This function is a slight modification of the *MakeFirstSimplex* used in the InCoDe algorithm. The first simplex returned must intersect the plane α, so the function selects the point $p_1 \in P$ nearest to the plane α. It then selects a second point p_2 such that (a) p_2 is on the other side of α from p_1 and (b) p_2 is the nearest point to p_1. The first d-simplex is constructed with the same algorithm used in InCoDe, starting from the (1)face (p_1, p_2).

4.1 DeWall complexity

The asymptotic time complexity $C_{DeWall}(n)$ of the DeWall algorithm can be evaluated as follows.

In the worst case (Figure 6), the selected splitting plane can define a simplex wall which is equal to the DT(P). In this case, the DeWall algorithm reduces to the InCoDe algorithm, and the complexity is that reported in Section 3.2. But the probability of such a worst case input set is very low in real applications.

Given:

- *SimplexWall(n)* : number of simplices intersected by the plane α in the DT(P);
- *PPart(n)* : the function PointsetPartition on P, which can be considered a $\mathcal{O}$(n log n) sorting process;

the complexity of the DeWall algorithm is:

$$C_{DeWall}(n) = PPart(n) + SimplexWall(n)(MakeSimplex(n) + AFL(m)) + 2\, C_{DeWall}(n/2) \quad (5)$$

with MakeSimplex(n) and AFL(m) defined as in Subsection 3.2. The number of d-simplex SimplexWall(n) is equal to the mean number of simplices intersected by a plane in a triangulation in E^d. No bound has been proposed in literature for this value. Empirically speaking, an acceptable estimate for this value is $\mathcal{O}(T(n)^{\frac{d-1}{d}})$ with $T(n)$ the number of d-simplex in the triangulation of n sites. Given $T(n) \in \mathcal{O}(n^{\lceil \frac{d}{2} \rceil^+})$ in E^d space, we estimate here $SimplexWall(n) \in \mathcal{O}(n^{\frac{d-1}{d}\lceil \frac{d}{2} \rceil^+})$.

Under these assumptions and given the results in Subsection 3.2, we have:

$$C_{DeWall}(n) \in \mathcal{O}(n log\, n) + k_c * \mathcal{O}(n^{\frac{d-1}{d}\lceil \frac{d}{2} \rceil^+}) + 2\, C_{DeWall}(n/2) \quad (6)$$

$$C_{DeWall}(n) \in \mathcal{O}(n log\, n) + k_c * \mathcal{O}(n^{\frac{d-1}{d}\lceil \frac{d}{2} \rceil^+}) \quad (7)$$

with k_c the mean number of cells visited by MakeSimplex.

Under these assumptions, the DeWall algorithm results asymptotically optimal if $k_c \in \mathcal{O}(n^{\frac{1}{d}\lceil \frac{d}{2} \rceil^+})$. The empirical results obtained running our implementation of DeWall for E^3 space, and reported in Section 5, verify this bound for k_c.

Random dataset (numb. of sites)	2000	4000	6000	8000	10000	20000
InCoDE	796	3620	8838	16242		
DeWall	119	372	809	1292	1905	5930
InCoDE + Hash + UG	33	82	139	198	263	685
DeWall + Hash + UG	25	56	91	126	162	347

Table 1: Processing times, in seconds, required to triangulate the *random* dataset (Hash:using hashing, UG:using 3D gridding).

4.2 Parallel DeWall

The parallelization of the DeWall algorithm is simplified, with respect to other D&C algorithms, because of the pre-merging. A usual D&C algorithm requires two synchronization points: the first at the partitioning of the input dataset, before process splitting, and the second at merging time, when both the splitted processes terminate. In the case of the DeWall algorithm, data partitioning and merging are performed in the same time stemp, and the two subprocesses can then run in an asynchronous way. This implies simpler implementation and load balancing.

5 Results and empirical evaluation

The performances of the algorithms were tested on two classes of datasets.
The first class consisted of *random* datasets, built using a random number generator. The distribution of the sites in the unitary cube is, therefore, nearly uniform.
In the second dataset group, the sites are organized in a number of *bubbles* with the density of each site decreasing as the distance from the bubble center. The point locations in each bubble are generated randomly.
For each dataset class and for each resolution (number of sites) a number of different dataset were generated; the times reported in Tables 1 and 3 are the means of the run times measured on each dataset.
Each time reported in Table 1 is the mean of the times on four different *random* datasets. The machine used for timings was a SUN Sparcstation. As can be seen in the graph in Figure 7, the optimized algorithms yield a near linear performance.

Some statistics on the execution of the DeWall algorithm on the *random* dataset are reported in Table 2.

The total number of tetrahedra returned is considerably lower than the upper bound, $\mathcal{O}(n^2)$: it is linear with the number of sites (approximately $7 * n$).

The number of tetrahedra in the first wall, $SimplexWall(n)$, has been estimated in Subsection 4.1 as $\mathcal{O}(T(n)^{\frac{d-1}{d}})$; the experimental values reported in Table 2 verify this assumption (with constant factor equal to 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 due to the fact that, for each face f on the $ConvexHull(P)$ all of the cells contained in the positive halfspace 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 increase in the mean number of cells visited is therefore justified by the increase with n of the faces on the $ConvexHull(P)$. In Section 4.1 it has been stated that, under some assumptions, DeWall results asymptotically optimal if $k_c \in \mathcal{O}(n^{\frac{1}{d}\lfloor\frac{d}{2}\rfloor^+})$. The empirical results show that in E^3 $k_c \leq n^{\frac{1}{d}\lfloor\frac{d}{2}\rfloor^+} = n^{\frac{2}{3}}$.

Finally, the maximum number of sites per cell is reported in the last row of Table 2.

These results show how common computer graphics techniques (e.g. gridding used to give an indexing scheme and therefore optimized point selection) can solve problems in computational geometry with great efficiency. As shown in Section 3.2 and 4.1 where some not proven assumptions were used, the optimality of the DeWall algorithm from the viewpoint of asymptotic complexity is hard to prove. However, the experimental results are extremely interesting and show an empirical complexity lower than $\mathcal{O}(nlogn)$ in E^3.

Another way to empirically evaluate DeWall is to compare it with other implementations. Unfortunately, implementations and evaluations of E^3 Delaunay triangulators are not so common; the authors are only acquainted with one paper which reports timings and evaluation of a DT implementation [3]. The paper reports on an *incremental construction* method which needs 220 sec. triangulate 1000 points on an IBM R6000, i.e. more than an order of magnitude slower than the optimized DeWall (without considering differences in machine performances).

6 Concluding remarks

The paper has addressed the requirements of unstructured dataset rendering and has explained the need for fast and general triangulation algorithms.

Random dataset (numb. of sites)	2000	4000	6000	8000	10000	20000
total tetrahedra	12976	26257	39677	52974	66394	133490
first wall tetrahedra	1371	2113	2811	3330	3901	6059
cells visited	74	82	96	96	100	116
max no. of sites per cell						

Table 2: Number of tetrahedra in the final triangulation, number of tetrahedra on the first simplex wall, mean number of cells visited to build a single tetrahedra, and maximum number of sites per cell.

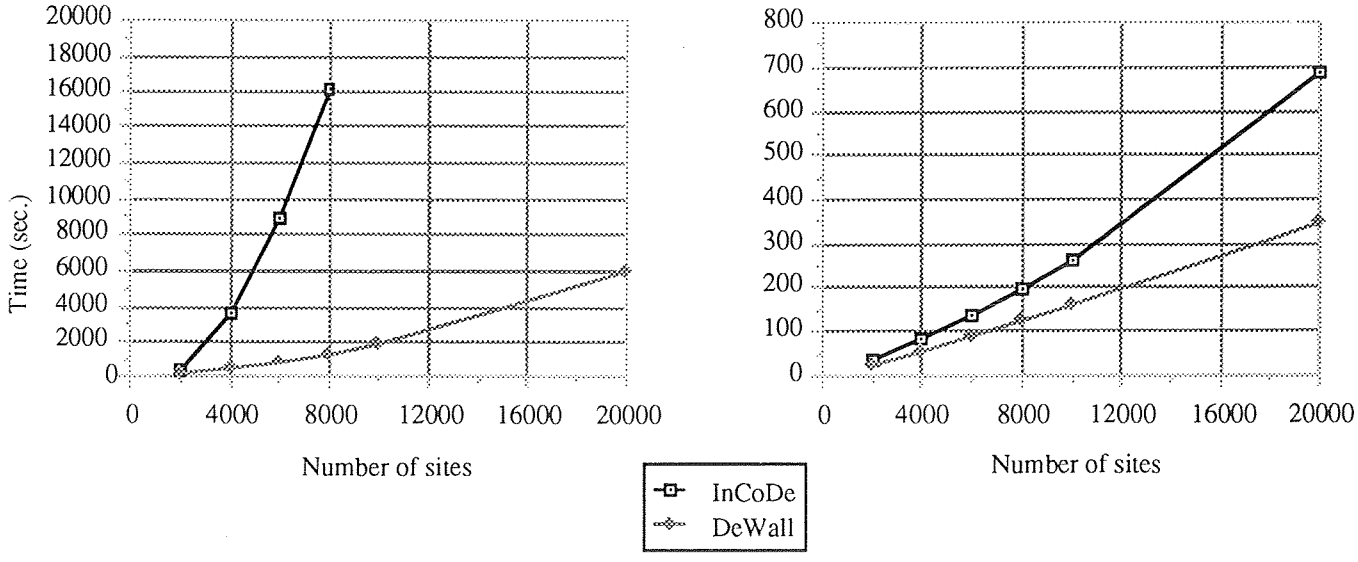

Figure 7: Graph of the algorithm timings: on the right the optimized timings.

Two different Delaunay triangulation algorithms have been presented. The second one is an original solution to the Delaunay triangulation, based on the Divide and Conquer paradigm. Optimization techniques are proposed and implemented. The timings presented show the effectiveness of the solution proposed.

The DeWall algorithm can easily be made parallel: each recursion in the Divide and Conquer paradigm can be assigned to a different processor. The structure of the algorithm is well suited for an implementation on a low grain parallel architecture (such as a shared memory MIMD architecture or a workstation network).

References

[1] V. Akman, W.R. Franklin, M. Kankanhalli, and C. Narayanaswami. Geometric computing and uniform grids tecnique. *Computer-Aided Design*, 21(7):410–420, Sept. 1989.

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

[3] I. Beichl and F. Sullivan. Parallelizing computational geometry: First steps. *SIAM News*, 24(6):1–17, 1991.

[4] R.A. Dwyer. A faster divide and conquer algorithm for constructing delaunay triangulations. *Algorithmica*, 2:137–151, 1987.

[5] K.A. Frenkel. Volume rendering. *Comm. ACM*, 32(4):426–435, April 1989.

[6] L.J. Guibas. Computational geometry and visualization: Problems at the interface. In N.M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*, pages 45–59. Springer-Verlag, 1990.

[7] L.J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of delaunay and voronoy diagrams. In *Lect. Note Comp. Science 443*, pages 414–431. Springer-Verlag, 1990.

[8] H. Hedelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. In *Proceedings of the 8th Annual ACM Symposium on Computational Geometry*, pages 43–52, June 1992.

[9] A. Kaufman. *Volume Visualization.* IEEE Computer Society Press, Los Alamitos, CA, 1990.

[10] D.T Lee and B.J. Schachter. Two algorithms for constructing a delaunay triangulation. *Int. J. of Computer and Information Science*, 9(3):219–242, 1980.

[11] A. Maus. Delaunay triangulation and the convex hull of n points in expected linear time. *Bit*, 24:151–163, 1984.

[12] D.H. McLain. Two dimensional interpolation from random data. *The Computer J.*, 19(2):178–181, 1976.

[13] C. Narayanaswami. *Parallel Processing for Geometric Applications.* PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, December 1990.

[14] F.P. Preparata and M.I. Shamos. *Computational Geometry - An Introduction.* Springer-Verlag, 1985.