Filtering Relocations on a Delaunay Triangulation

Pedro Machado Manhães de Castro, Jane Tournois, Pierre Alliez, Olivier Devillers

Filtering Relocations on a Delaunay Triangulation

Pedro Machado Manhães de Castro Jane Tournois Pierre Alliez
Olivier Devillers
INRIA Sophia Antipolis - Méditerranée, France

September 3, 2009

Abstract

Updating a Delaunay triangulation when its vertices move is a bottleneck in several domains of application. Rebuilding the whole triangulation from scratch is surprisingly a very viable option compared to relocating the vertices. This can be explained by several recent advances in efficient construction of Delaunay triangulations. However, when all points move with a small magnitude, or when only a fraction of the vertices move, rebuilding is no longer the best option. This paper considers the problem of efficiently updating a Delaunay triangulation when its vertices are moving under small perturbations. The main contribution is a set of filters based upon the concept of vertex tolerances. Experiments show that filtering relocations is faster than rebuilding the whole triangulation from scratch under certain conditions.

Classification: Computer Graphics, I.3.5, Computational Geometry and Object Modeling: Geometric algorithms, languages, and systems

1 Introduction

Delaunay triangulation of a point set is one of the most famous and successful data structures introduced in the field of Computational Geometry. Two main reasons explain this success. First, it is suitable to many practical uses such as mesh generation for finite elements methods [Ede01] or surface reconstruction from point sets [CG06]. Second, computational geometers have produced efficient implementations [BDP*02, She96].

In several applications, the triangulation needs to evolve over time. Thus, the vertices of the triangulation - defined by input data points - are moving. This happens for instance in data clustering [HW79, AV07], mesh generation [DBC07], remeshing [VCP08, ACD*03], mesh smoothing [ABE97], mesh optimization [ACYD05, TAD07, Che04], to name a few.

The *Delaunay triangulation* of a set S of n points in $\mathbb{R}^d$ is a *simplicial complex*. It is defined such that no point in S is inside the circumsphere of any *simplex* in the Delaunay triangulation [PS90, Aur91]. Several Delaunay triangulation algorithms have been described in the literature. Many of them are appropriate in the *static* setting [SH75, For87], where the points are fixed and known in advance. There is also a variety of so-called *dynamic* algorithms [GS78, Dev02, DLM04], in which the points are fixed and the triangulation is maintained under point insertions or deletions. Next, if some of the points move continuously and we want to keep track of all topological changes, we are dealing with *kinetic* algorithms [Gui98, Rus07]. Finally, an important variant occurs when the points move and when we are only interested in the triangulation at some discrete timestamps. We call such context *timestamp relocation*.

Notions. In the context of timestamp relocation, a simple method consists of rebuilding the whole triangulation from scratch at every timestamp. We denote by ***rebuilding*** such approach. When the output has an expected linear size, rebuilding can lead to a $O(kn\log(n))$ time complexity, where n denotes the number of vertices of the triangulation and k denotes the number of distinct timestamps. Despite its poor theoretical complexity, the rebuilding algorithm turns out to be surprisingly hard to outperform when most of the points move, as already observed [Rus07]. Rebuilding the triangulation from scratch allows using the most efficient static algorithms. In this paper we use the Delaunay triangulations from the CGAL library [Yvi08,PT08]. The latter sort the points so as to best preserve point proximity for efficient localization, and make use of randomized incremental constructions.

There are a number of applications which require computing the next vertex locations one by one, updating the Delaunay triangulation after each relocation [ACYD05, TAD07, TWAD09]. Naturally, rebuilding is unsuitable for such applications. Another naive updating algorithm, significantly different from rebuilding, is the ***relocation*** algorithm, which relocates the vertices one by one. Roughly speaking, the latter consists of iterating over all vertices to be relocated. For each relocated vertex the algorithm first walks through the triangulation to locate the simplex containing its new position, inserts a vertex at the new position and removes the old vertex from the triangulation. This way each relocation requires three operations for each relocated point: one point location, one insertion and one removal. When the displacement of a moving point is small enough the point location operation is usually fast. In favorable configurations with small displacement and constant local triangulation complexity, the localization, insertion, and deletion operations take constant time per point. This leads to $O(m)$ complexity per timestamp, where m is the number of moving points. Such complexity is theoretically better than the $O(n\log n)$ complexity of rebuilding. In practice however, the deletion operation is very costly and hence rebuilding the whole triangulation is

faster when all vertices are relocated, i.e., when $m = n$.

Algorithms which are not able to relocate vertices one by one are referred to as ***static***. Algorithms relocating vertices one by one are referred to as ***dynamic***. Advantages of being dynamic include to name a few:
–1– the computational complexity depends mostly on the number of moving points (which impacts on applications where points eventually stop moving);
–2– the new location of a moving point can be computed *on-line* which is required for variational methods [DBC07, ACYD05, TAD07, TWAD09];
–3– the references to the memory that the user may have remain valid (conversely to rebuilding).
Rebuilding is static while the relocation algorithm is dynamic.

Previous Work. Several recent approaches have been proposed to outperform the two naive algorithms (rebuilding and relocation) in specific circumstances. For example, *kinetic data structures* [Gui98] are applicable with a careful choice of vertex trajectories [Rus07]. Some work has also been done to improve the way kinetic data structures handle degeneracies [ABTT08]. Approaches based on *kinetic data structures* are often dynamic as they can relocate one vertex at a time. Guibas and Russel consider another approach [GR04] which consists of the following sequence: Remove some points until the triangulation has a non overlapping embedding, *flip* the invalid pairs of adjacent simplices until the triangulation is valid (i.e., Delaunay), and add insert back the previously removed points. In this approach the flipping step may lead to deadlocks in dimensions higher than two, which trigger rebuildings from scratch with huge computational overhead. For their input data however, deadlocks do not happen so often and rebuilding can be outperformed when considering heuristics on the ordering of the points to be removed. Although this method is dynamic when relocating one vertex at a time, it loses considerably its efficiency as it is not allowed to use any heuristic anymore in this case. Shewchuk proposes two elegant algorithms to repair Delaunay triangulations: *star splaying* and *star flipping* [She05]. Both algorithms can be used when *flipping* causes a deadlock, instead of rebuilding the triangulation from scratch. Finally, for applications which can live with triangulations which are not necessarily Delaunay at every timestamp (e.g., almost-Delaunay upon lazy removals [DBC07]), some dynamic approaches outperform rebuilding by a factor of three [DBC07]. It is worth mentioning that dynamic algorithms, which perform nearly as fast as rebuilding, are also very well-suited to applications based on variational methods.

Contributions. We propose to compute for each vertex of the triangulation a safety zone where the vertex can move without changing its connectivity. This way each relocation which does not change the connectivity of the triangulation is *filtered*. We show experimentally that this approach is worthwhile for applications where the points are moving under small perturbations.

Our main contribution takes the form of a filtering method for relocating the points of a Delaunay triangulations when the points move with small amplitude. The noticeable advantages of the filter are: –1– **Simplicity.** The implementation is simple as it relies on well-known dynamic Delaunay triangulation constructions [Dev02, DLM04] with few additional geometric computations; and –2– **Efficiency.** Our filtering approach outperforms in our experiments by at least a factor of four, in two and three dimensions, the current dynamic relocation approach used in mesh optimization [TAD07]. It also outperforms the rebuilding algorithm for several conducted experiments. This opens new perspectives for several applications. For example, in mesh optimization, the number of iterations is shown to impact the mesh quality under converging schemes. The proposed algorithm enables the possibility of going further on the number of iterations while being dynamic.

2 Fundamentals

We now review the necessary background on Delaunay triangulations and introduce the notions of *safe region* and *tolerance region* of a vertex.

2.1 Certificates and Tolerances

A *predicate* is a function on a set of primitives which returns one value in a discrete set of possible results. In this paper, we consider the common case where we compute a numerical value denoted by *predicate discriminant*: the result of the predicate is the sign of this value. When one or more predicates are used to evaluate whether a geometric data structure is valid or invalid, we denote by *certificate* each of those predicates. In the sequel a certificate is said to be *valid* when it is positive.

Let $C : \mathcal{A}^m \to \{-1, 0, 1\}$ be a certificate, acting on a m-tuple of points $\zeta = (\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_m) \in \mathcal{A}^m$, where $\mathcal{A}$ is the space where the points lie. By abuse of notation, $\mathbf{z} \in \zeta$ means that $\mathbf{z}$ is one of the points of ζ. We define the *tolerance* of ζ with respect to C, namely $\varepsilon_C(\zeta)$ or simply $\varepsilon(\zeta)$ when there is no ambiguity, the largest displacement applicable to $\mathbf{z} \in \zeta$ without invalidating C. More precisely, the tolerance, assuming $C(\zeta) > 0$, can be stated as follows:

$$\varepsilon_C(\zeta) = \inf_{\substack{\zeta' \\ C(\zeta') \leq 0}} dist_H(\zeta, \zeta'), \tag{1}$$

where $dist_H(\zeta, \zeta')$ is the Hausdorff distance between two finite sets of points.

Let $\mathcal{X}$ be a finite set of m-tuples of points in $\mathcal{A}^m$. Then, the *tolerance* of an element $\mathbf{e}$ belonging to one or several m-tuples of $\mathcal{X}$, with respect to a given certificate

C and to $\mathcal{X}$, is denoted $\varepsilon_{C,\mathcal{X}}(\mathbf{e})$ (or simply by $\varepsilon(\mathbf{e})$ when there is no ambiguity). It is defined as follows:

$$\varepsilon_{C,\mathcal{X}}(\mathbf{e}) = \inf_{\substack{\zeta \ni \mathbf{e} \\ \zeta \in \mathcal{X}}} \varepsilon_C(\zeta). \tag{2}$$

2.2 Delaunay Triangulations: Certificate and Tolerance

An intersection free triangulation lying in $\mathbb{R}^d$ can be checked to be Delaunay using the *empty-sphere certificate* [DLPT98]. This certificate states that, for each facet of the triangulation, the hypersphere passing through the $d+1$ vertices of a simplex on one side does not contain the vertex on the other side. Therefore, this certificate is applied to any $d+2$ distinct points $\mathbf{z}_1 = (x_1^1, \ldots, x_d^1), \ldots, \mathbf{z}_{d+2} = (x_1^{d+2}, \ldots, x_d^{d+2})$ of the triangulation which belong to the same pair of incident cells. In the sequel, such a pair is called a *bi-cell* (see Figure 1).

The tolerance involved in a Delaunay triangulation is the *tolerance of the empty-sphere certificate* acting on any bi-cell of a Delaunay triangulation. From Equation 1, it corresponds to the size of the smallest perturbation the bi-cell's vertices can undergo so as to become cospherical. This is equivalent to compute the hypersphere that minimizes the maximum distance to the $d+2$ vertices, i.e., the *optimal middle sphere*, which is the median sphere of the d-annulus of minimum width containing the vertices. An annulus, defined as the region between two concentric hyperspheres, is a fundamental object in Computational Geometry [GLR97]).

Dealing with Boundary. In the following we use a simple way to deal with the boundary of the triangulation (i.e., its convex hull) by adding a "point at infinity" ∞ to the initial set of points [AGMR98, Yvi08, PT08]. Let S be the initial set of points. We consider an augmented set $S' = S \cup \{\infty\}$. Let $CH(S)$ be the convex hull of S and $DT(S)$ its Delaunay triangulation, then the *extended Delaunay triangulation* is given by

$$DT(S') = DT(S) \cup \{(f, \infty) \mid f \text{ facet of } CH(S)\}. \tag{3}$$

In addition to $DT(S)$, every point on the boundary of the convex hull $CH(S)$ is connected to ∞ creating new simplices: the infinite simplices. In contrast with $DT(S)$, $DT(S')$ has the nice property that there are exactly $d+1$ simplices adjacent to each simplex in $DT(S')$. This greatly simplifies the following descriptions.

When dealing with infinite bi-cells, namely the ones which include the point at infinity, the *d-annulus of minimum width* becomes the region between two parallel hyperplanes. This happens because the hypersphere passing through ∞ reduces to an hyperplane.

In order to precise which are those parallel hyperplanes, we consider two distinct cases (which coincide in two-dimensions):

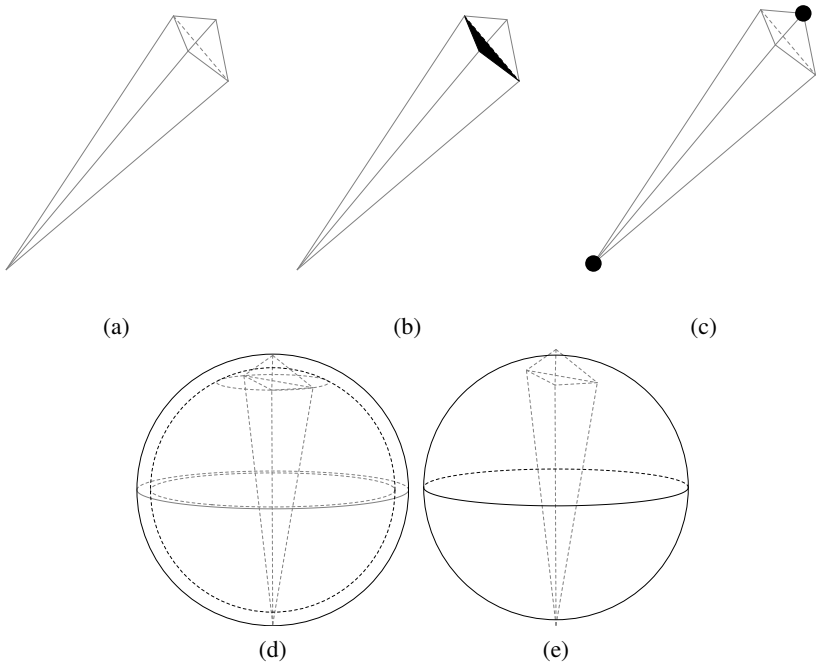


Figure 1: **Definitions.** (a) A three-dimensional bi-cell, (b) its interior facet and (c) opposite vertices. (d) The boundaries of its 3-annulus of minimum width; the smallest boundary passing through the facet vertices and the biggest boundary passing through the opposite vertices. (e) depicts its standard delimiter separating the facet and opposite vertices.

- If only one cell of the bi-cell is infinite then one hyperplane H_1 passes through the d common vertices of the two cells. The other hyperplane H_2 is parallel to H_1 and passes through the unique vertex of the finite cell which is not contained in the infinite cell (see Figure 2a).

- Otherwise the two vertices which are not shared by each cell form a line L and the remaining finite vertices are a ridge of the convex hull H (an edge in three-dimensions). Consider the hyperplanes H_1 passing through H and parallel to L and H_2 passing through L and parallel to H. They compose the boundary of the d-annulus (see Figure 2b).

2.3 Tolerance Regions

Let $\mathcal{T}$ be a triangulation lying in $\mathbb{R}^d$ and $\mathcal{B}$ a bi-cell in $\mathcal{T}$. The *interior facet* of $\mathcal{B}$ is the common facet of both cells of $\mathcal{B}$. The *opposite vertices* of $\mathcal{B}$ are the remaining

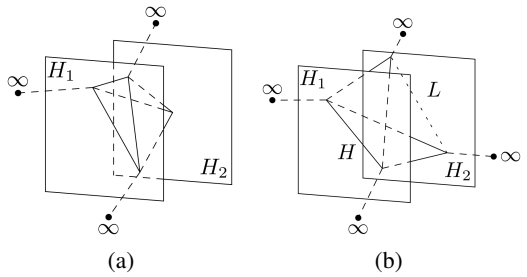


Figure 2: **Infinite bi-cells.** The 3-annulus of minimum-width of a bi-cell containing: (a) one infinite cell and one finite cell, (b) two infinite cells.

two vertices that do not belong to its interior facet. We associate to each bi-cell $\mathcal{B}$ an arbitrary hypersphere $\mathcal{S}$ denoted by *delimiter* of $\mathcal{B}$, see Figure 1. If the interior facet and opposite vertices of $\mathcal{B}$ are respectively inside and outside the delimiter of $\mathcal{B}$, we say that $\mathcal{B}$ verifies the *safety condition*. We call $\mathcal{B}$ a *safe bi-cell*. If a vertex $\mathbf{z}$ belongs to the interior facet of $\mathcal{B}$, then the *safe region* of $\mathbf{z}$ with respect to $\mathcal{B}$ is the region inside the delimiter. Otherwise, the *safe region* of $\mathbf{z}$ with respect to $\mathcal{B}$ is the region outside the delimiter. The intersection of the safe regions of $\mathbf{z}$ with respect to each one of its adjacent bi-cells is called *safe region* of $\mathbf{z}$. If all bi-cells of $\mathcal{T}$ are safe bi-cells we call $\mathcal{T}$ a *safe triangulation*. When a triangulation is a safe triangulation we say that it verifies the *safety condition*.

It is clear that a safe triangulation is equivalent to a Delaunay triangulation as:

- Each delimiter can be shrunk so as to touch the vertices of the interior facet, and thus defines an empty-sphere passing through the interior facet of its bi-cell (which proves that the facets belongs to the Delaunay triangulation).
- The *empty-sphere* property of the Delaunay triangulation facets defines itself empty-spheres passing through the interior facets of the bi-cells. Those empty-spheres are delimiters.

Let $\mathcal{T}$ be a triangulation. We define the graph (V,E) of $\mathcal{T}$, where V and E are the set of vertices and edges of $\mathcal{T}$ respectively, as the *combinatorics* of $\mathcal{T}$. Then we have the following proposition:

Proposition 1 Given the combinatorics of a Delaunay triangulation $\mathcal{T}$, if its vertices move inside their safe regions, then the triangulation obtained while keeping the same combinatorics as in $\mathcal{T}$ in the new embedding remains a Delaunay triangulation.

Proposition 1 is a direct consequence of the equivalence between safe and Delaunay triangulations: If the vertices remain inside their safe regions then $\mathcal{T}$ remains

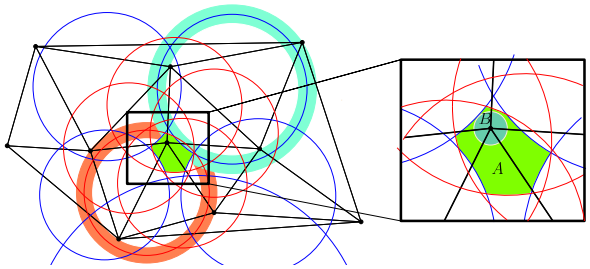


Figure 3: **Safe region and tolerance region.** $\mathbf{z} \in \mathbb{R}^2$ the center of B. The region A is the safe region of $\mathbf{z}$, while B is its tolerance region.

a safe triangulation. As a consequence it remains a Delaunay triangulation.

Note that the safe region of a vertex depends on the choice of delimiters, i.e., one for each bi-cell of the triangulation. We denote by *tolerance region* of $\mathbf{z}$, given a choice of delimiters, the biggest ball centered at the location of $\mathbf{z}$ included in its safe region. More precisely, let $\mathcal{D}(\mathcal{B})$ be the delimiter of a given bi-cell $\mathcal{B}$. Then, for a given vertex $\mathbf{z} \in \mathcal{T}$, the tolerance region of $\mathbf{z}$ is given by:

$$\tilde{\varepsilon}(\mathbf{z}) = \inf_{\substack{\mathcal{B} \ni \mathbf{z} \\ \mathcal{B} \in \mathcal{T}}} dist_H(\mathbf{z}, \mathcal{D}(\mathcal{B})). \tag{4}$$

We have $\tilde{\varepsilon}(\mathbf{z}) \leq \varepsilon(\mathbf{z})$, since the delimiter generated by the minimum-width d-annulus of the vertices of a bi-cell $\mathcal{B}$ maximizes the minimum distance of the vertices to the delimiter (see Figure 3).

Among all possible delimiters of a bi-cell, we define the *standard delimiter* as the median hypersphere of the d-annulus with the inner-hypersphere passing through the interior facet and the outer-hypersphere passing through the opposite vertices. Both median hypersphere and d-annulus are unique. We call the d-annulus the *standard annulus*. If our choice of delimiter for each bi-cell of $\mathcal{T}$ is the standard delimiter, then we have $\tilde{\varepsilon}(\mathbf{z}) = \varepsilon(\mathbf{z})$. Notice that the standard annulus is usually the annulus of minimum-width described in Section 2.2 defined by the vertices of $\mathcal{B}$. In the pathological cases where the minimum-width annulus is not the standard annulus, then the standard delimiter is not safe [GLR97].

Computing the standard annulus of a given bi-cell $\mathcal{B}$ requires computing the center of a d-annulus. This is achieved through finding the line perpendicular to the interior facet passing through its circumcenter (it corresponds to the intersection of the bisectors of the interior facet vertices of $\mathcal{B}$) and intersecting it with the bisector of the opposite vertices of $\mathcal{B}$.

3 Filtering Relocations

As a remainder, the dynamic relocation algorithm which relocates one vertex after another unlike rebuilding, can be used when there is no a priori knowledge about the new point locations of the whole set of vertices. Such approach is especially useful for algorithms based on variational methods, such as [DBC07, ACYD05, TAD07, TWAD09].

In this section we propose two improvements over the naive relocation algorithm for the case of small displacements. Finally, we detail an algorithm devised to filter relocations using the tolerance region of a vertex in a Delaunay triangulation (see Section 2.3). It is worth mentioning that this algorithm can be easily modified so as to incorporate other kinds of filters such as the safe region of a vertex.

3.1 Improving the Relocation Algorithm for Small Displacements

In two-dimensions a small modification of the relocation algorithm leads to a substantial acceleration, by a factor of two in our experiments. This modification consists of flipping edges when a vertex displacement does not invert the orientation of any of its adjacent triangles. The key idea is to avoid as many removal operations as possible, as they are the most expensive. In three-dimensions, repairing the triangulation is far more involved [She05].

A weaker version of this improvement consists of using relocation only when at least one topological modification is needed; otherwise the vertex coordinates are simply updated. Naturally, this additional computation leads to an overhead, though our experiments show that it pays off when displacements are small enough. When this optimization is combined with the algorithms described next, our experiments show evidence that it is definitely a good option.

3.2 Filtering Algorithm

We now redesign the relocation algorithm so as to take into account the tolerance region of every relocated vertex. The proposed algorithm, denoted by *filtering algorithm*, is capable of correctly deciding whether or not a vertex displacement requires an update of the connectivity so as to trigger the trivial update condition. It is dynamic in the sense that it preserves all benefits from the relocation algorithm compared to rebuilding.

Data structure. Consider a triangulation $\mathcal{T}$, where to each vertex $\mathbf{z} \in \mathcal{T}$ we associate two point locations: $\mathbf{f_z}$ and $\mathbf{m_z}$. We denote them respectively by the *fixed* and the *moving* position of a vertex. The fixed position is used to *fix* a reference position for a moving point. The moving position of a given vertex is its actual

position, and changes at every relocation. Initially the fixed and moving positions are equal. We denote by $\mathcal{T}_{\mathbf{f}}$ and $\mathcal{T}_{\mathbf{m}}$ the embedding of $\mathcal{T}$ with respect to $\mathbf{f_z}$ and $\mathbf{m_z}$ respectively. For each vertex, we store two numbers: $\varepsilon_{\mathbf{z}}$ and $D_{\mathbf{z}}$ which represent respectively the tolerance value of $\mathbf{z}$ and the distance between $\mathbf{f_z}$ and $\mathbf{m_z}$.

Pre-computations. We initially compute the Delaunay triangulation $\mathcal{T}$ of the initial set of points S, and for each vertex we set $\varepsilon_{\mathbf{z}} = \varepsilon(\mathbf{z})$ and $D_{\mathbf{z}} = 0$. For a given triangulation $\mathcal{T}$, the tolerance of each vertex is computed efficiently by successively computing half the width of the standard annulus of each bi-cell of $\mathcal{T}$, and keeping the minimum value on each of its vertices.

The *filtering algorithm* performs as follows for every vertex displacement:

Input: Triangulation $\mathcal{T}$ after pre-computations, a vertex $\mathbf{z}$ of $\mathcal{T}$ and its new location $\mathbf{p}$.
Output: $\mathcal{T}$ updated after the relocation of $\mathbf{z}$.

$(\mathbf{m_z}, D_{\mathbf{z}}) \leftarrow (\mathbf{p}, dist(\mathbf{f_z}, \mathbf{p}))$;
if $D_{\mathbf{z}} < \varepsilon_{\mathbf{z}}$ **then** we are done;
else
  insert $\mathbf{z}$ in a queue $\mathcal{Q}$;
  while $\mathcal{Q}$ *is not empty* **do**
    remove $\mathbf{h}$ from the head of $\mathcal{Q}$;
    $(\mathbf{f_h}, \varepsilon_{\mathbf{h}}, D_{\mathbf{h}}) \leftarrow (\mathbf{m_h}, \infty, 0)$;
    update $\mathcal{T}$ by relocating $\mathbf{h}$ with the relocation algorithm;
    foreach *new created bi-cell* $\mathcal{B}$ **do**
      $\varepsilon' \leftarrow$ half the width of the standard annulus of $\mathcal{B}$;
      foreach *vertex* $\mathbf{w} \in \mathcal{B}$ **do**
        if $\varepsilon_{\mathbf{w}} > \varepsilon'$ **then**
          $\varepsilon_{\mathbf{w}} \leftarrow \varepsilon'$;
          if $\varepsilon_{\mathbf{w}} < D_{\mathbf{w}}$ **then** insert $\mathbf{w}$ into $\mathcal{Q}$;
        end
      end
    end
  end
end

The algorithm is shown to terminate as each processed vertex $\mathbf{z}$ gets a new displacement value $D_{\mathbf{z}} = 0$ and thus $\leq \varepsilon_{\mathbf{z}}$. At the end of this algorithm all vertices are guaranteed to have their $D_{\mathbf{z}}$ smaller or equal than their $\varepsilon_{\mathbf{z}}$. In such a situation, from Proposition 1, $\mathcal{T}_{\mathbf{m}}$ is the Delaunay triangulation of the points located at moving positions. The tolerance algorithm has the same complexity as the relocation algorithm. Let n be the number of vertices of $\mathcal{T}$. Although in the worst case we have n relocation calls for a single call of the filtering algorithm, if all points move, the total number of calls to the relocation algorithm is reduced to at most $2n$. This

is due to the fact that when a vertex is relocated its D value is set to 0.

When relocating the vertices with a convergent scheme we can run rebuilding for the first few timestamps until the points are more or less stable, then switch to the filtering algorithm. As a drawback, the algorithm is no longer dynamic during the first timestamps. We give more details on this approach in the next section.

A natural idea is to replace the tolerance test (which checks if the moving position of a vertex stays within the tolerance distance from its fixed position) by a more involved test which checks if it stays within its safe region. We could also run this *safety test* in case of failure of the first test but the safety test is rather involved and does not save any computation time in practice, even when using first order approximations of its computation.

Another point concerns robustness issues. Computing the tolerance values using floating point computations may in some special configurations yield to rounding errors and hence to wrong evaluation of ε_z. The algorithm ensures that T_f, the embedding at fixed position, is always correct while the embedding T_m at moving position may be incorrect. A certified correct Delaunay triangulation for T_m is obtained through a certified lower bound over ε_z. As expected the numerical stability of ε_z depends on the *quality of the simplices*. To explain this fact, consider the smallest angle θ between the line perpendicular to the interior facet of a bi-cell passing through its circumcenter and the bisector of the opposite vertices, see Figure 4. The numerical stability is proportional to the size of this angle. For convex bi-cells, if θ is too small, the two simplices are said to have a *bad shape* [CDE*99]. It is worth saying that the computation of ε_z is rather stable in the applications which would benefit from filtering since the shape of their simplices improve over time.

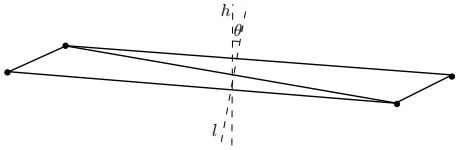


Figure 4: **Numerical stability.** If the smallest angle θ between the $(d-2)$-dimension flat h and the line l is too small, the simplices are *badly shaped* in the sense of being non isotropic.

4 Experimental Results

This section investigates through several experiments the size of the vertex tolerances and the width of the standard annulus in two- and three-dimensions. We

discuss the performance of the rebuilding, relocation and filtering algorithms on several data sets.

We run our experiments on a Pentium 4 at 2.5 GHz with 1GB of memory, running Linux (kernel 2.6.23). The compiler used is g++4.1.2; all configurations being compiled with -DNDEBUG -O2 flags (release mode with compiler optimizations enabled). *CGAL* 3.3.1 is used along with an *exact predicate inexact construction* kernel [BFG*08, FT06].

The initialization costs of the filtering algorithm accounting for less than 0.13% of the corresponding total running time of the experiments, we consider them as negligible.

4.1 Clustering

In several applications such as image compression, quadrature, and cellular biology, to name a few, the goal is to partition a set of objects into k clusters, following an optimization criterion. Usually such criterion is the squared error function. Let S be a measurable set of objects in $\mathcal{A}$ and $\mathcal{P} = \{S_i\}_1^k$ a k-partition of S. The squared error function associated with $\mathcal{P}$ is defined as:

$$\sum_{i=1}^{k} \int_{S_i} dist(\mathbf{x}, \mu_i)^2 d\mathbf{x}, \tag{5}$$

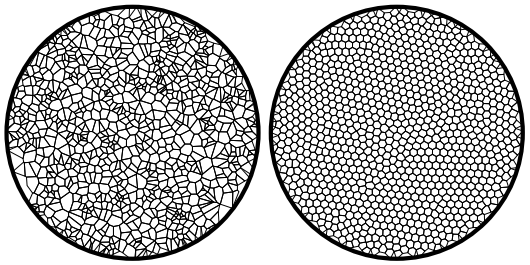
where μ_i is the centroid of S_i and $dist$ is a given distance between two objects in $\mathcal{A}$.

One relevant algorithm to find such partitions is the *k-means* algorithm. The most common form of the algorithm uses the Lloyd iterations [Llo82,SG86,DEJ06, ORSS06]. The Lloyd iteration starts by partitioning the input domain into k arbitrary initial sets. It then calculates the centroid of each set and constructs a new partition by associating each point to the closest centroid. The algorithm is repeated by alternate application of these two steps until convergence.

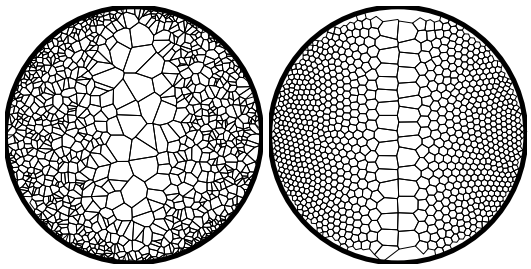
Let $S = \mathbb{R}^d$, $dist \in L^2$ and ρ a density function in $\mathbb{R}^d$, the latter being used to compute centroids. The Lloyd is described as follows: First, compute the *Voronoi diagram* of an initial set of k random points following the distribution ρ. Next, each cell of the Voronoi diagram is integrated so as to compute its center of mass. Finally, each point is relocated to the centroid of its Voronoi cell. Note that, for each iteration, the Delaunay triangulation of the points is updated and hence each iteration is considered as a distinct timestamp.

The convergence of the point locations evolving through the Lloyd iteration is proven for $d = 1$ [DEJ06]. Although only weaker convergence results are known for $d > 1$ [SG86], the Lloyd iteration is commonly used for dimensions higher than 1 and experimentally converges to "good" point configurations.

We consider in $\mathbb{R}^2$ one uniform density function ($\rho_1 = 1$) as well as three non-uniform density functions: $\rho_2 = x^2 + y^2$, $\rho_3 = x^2$ and $\rho_4 = \sin^2\sqrt{x^2+y^2}$. We apply the Lloyd iterations to obtain evenly-distributed points in accordance with the above-mentioned density functions, see Figure 5. The standard annuli widths and tolerances increase up to convergence, while the average displacement size quickly decreases, see Figure 6. In addition the number of near-degenerate cases tends to decrease along with the iterations.



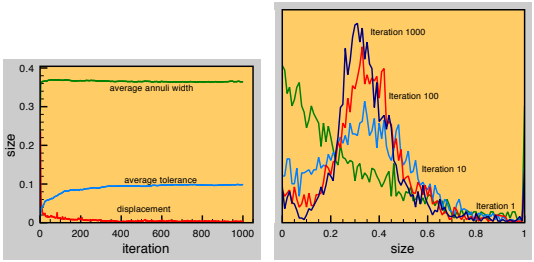
(a) Uniform density: $\rho = 1$



(b) Non-uniform density: $\rho = x^2$

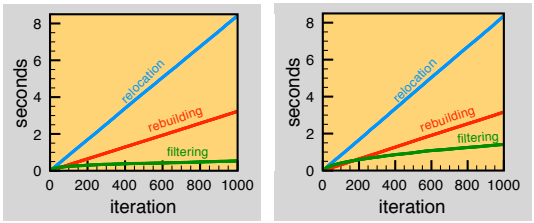
Figure 5: **Point distribution before and after Lloyd's iteration.** $1,000$ points are sampled in a disc with (a) uniform density, and (b) $\rho = x^2$. The point sets are submitted to $1,000$ Lloyd iterations with their respective density function.

As shown by Figure 7 the proposed filtering algorithm outperforms both the relocation and rebuilding algorithm. When the iteration number is really high, filtering becomes several times faster than rebuilding even though being a dynamic algorithm. At this point it is worth saying that other strategies for accelerating the Lloyd algorithm, orthogonal with respect to this one, exist. We can cite, e.g., the Lloyd-Newton method, devised to reduce the overall number of iterations [DE06a, DE06b, LWL*08].

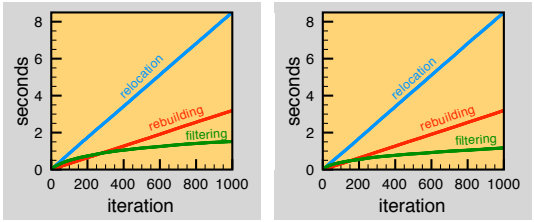


(a) evolution (b) distribution

Figure 6: **Statistics in 2D.** Consider a disc such that the quantity representing the number of points divided by its surface is equal to 1. The square root of this quantity is the unity of distance. The curves depict for $1,000$ Lloyd iterations with $\rho = x^2$: (a) the evolution of the average displacement size, the average standard annulus width and the average tolerance of vertices; (b) the distribution of the standard annulus width for iteration 1, 10, 100 and $1,000$.



(a) uniform ($\rho = 1$) (b) $\rho = x^2 + y^2$



(c) $\rho = x^2$ (d) $\rho = \sin^2\sqrt{x^2+y^2}$

Figure 7: **Computation times in 2D.** The curves depict the cumulated computation times for running up to 1,000 Lloyd's iterations with: (a) uniform density ($\rho = 1$); (b) $\rho = x^2 + y^2$; (c) $\rho = x^2$; and (d) $\rho = \sin^2\sqrt{x^2+y^2}$. The filtering algorithm consistently outperforms rebuilding for every density function. Note how the slope of the filtering algorithm's curve decreases over time.

4.2 Mesh Optimization

We choose as experiment a recent work on isotropic tetrahedron mesh generation [TWAD09] based on a combination of Delaunay refinement and optimization. We focus on the optimization part of the algorithm, based upon an extension of the

Optimal Delaunay Triangulation approach [Che04], denoted by NODT for short. We first measure how the presented algorithm accelerates the optimization procedure, assuming that all vertices are relocated. We consider the following meshes (see Figure 8):

- SPHERE: Unit sphere with $13,000$ vertices (Figure 8a);
- MAN: Human body with $8,000$ vertices (Figure 8b);
- BUNNY: Stanford Bunny with $13,000$ vertices (Figure 8c);
- HEART: Human heart with $10,000$ vertices (Figure 8d).

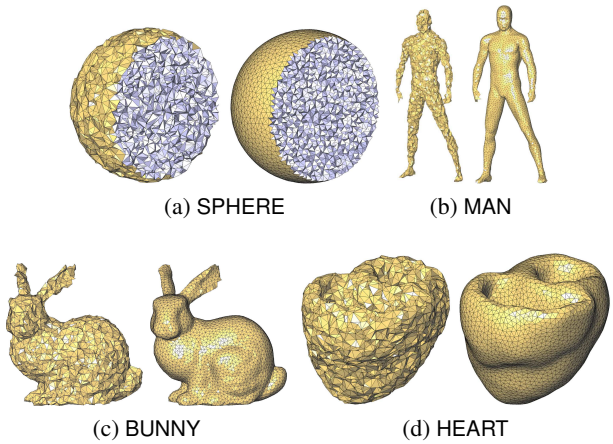


Figure 8: **Mesh optimization based on Optimal Delaunay Triangulation.** (a) Cut-view on the SPHERE initially and after $1,000$ iterations; (b) MAN initially and after $1,000$ iterations; (c) BUNNY initially and after $1,000$ iterations; (d) HEART initially and after $1,000$ iterations.

Figure 9 shows the MAN mesh model evolving over $1,000$ iterations of NODT optimization (the mesh is chosen intentionally coarse and uniform for better visual depiction). The figure depicts how going from 100 to $1,000$ iterations brings further improvements on the mesh quality (confirmed by improvements over distribution of dihedral angles and over number of remaining slivers). In meshes with variable sizing the improvement typically translates into 15% less leftover slivers. These experiments explain our will at accelerating every single optimization step.

Experiments in 3D show a good convergence of the average standard annulus width and average tolerance of vertices for the NODT mesh optimization process. However, the average tolerance of vertices converges in 3D to a proportionally

smaller value than the average standard annulus width compared with the 2D case (see Figures 6a and 10a). This is an effect of increasing the dimension and can be explained as follows: the average number of bi-cells including a given vertex is larger than 60 in three-dimensions (compare with the two dimensional case, which is 12). As explained in Section 2.3, the tolerance of a vertex $\mathbf{z}$ is half the minimum of the standard annulus width of its adjacent bi-cells. In other words, the tolerance of a vertex is proportional to the minimum value of around 60 distinct standard annulus widths. Figure 10b quantizes how the standard deviation of the standard annulus widths in three-dimensions is larger than in two-dimensions. Figures 11a and 11b show respectively the percentage of failures and the amount of tolerance updates per failure in two- and three-dimensions.

The rebuilding algorithm is considerably harder to outperform in three-dimensions for two main reasons. First, the removal operation is dramatically slower and the number of bi-cells containing a given point is five times larger than in two-dimensions. Nevertheless the filtering algorithm outperforms rebuilding for most input data considered in our experiments. In both two- and three-dimensions, it accelerates when going further on the number of iterations (see Figure 4.2). In three-dimensions, the main difficulty of this approach lies into the persistent quasi-degenerate cases. As the name suggests such cases consist of almost co-spherical configurations of the vertices of a bi-cell which persist across several iterations. The persistence mostly happens because the magnitude of the relocation moves decreases over time. Such cases lead to several consecutive filter failures, which themselves trigger expensive point relocations. In practice however, the amount of such degenerate configurations is rather small (see Figure 11c and Figure 11d).

Lastly we implement a small variation of the tolerance algorithm suggested in Section 3.2. The latter consists of rebuilding for the first few iterations before switching to the filtering algorithm. The switching criterion is more or less heuristic. Experimentally, we found that generally when 75% of the vertices remain inside their tolerance (55% in three dimensions), the performance of the filtering algorithm is more or less the same as rebuilding. This percentage can be used as a switching criterion. In our implementation we sample 40 random vertices at each group of four iterations and compute their tolerance. We compare these tolerances with their displacement sizes, and check if at least 75% of those vertices have their tolerance larger than their displacement size. In the positive we switch from the rebuilding to the filtering the algorithms. With this variant of the filtering algorithm, which is not dynamic for the first few iterations, we obtain an improvement in the running time of, e.g., 10% in the mesh optimization process of the HEART model.

Mesh optimization schemes such as NODT [TWAD09] are very labor-intensive due to the cost of computing the new point locations and of moving the vertices. However, as illustrated by Figure 9, a large number of iterations provides us with

higher quality meshes. When optimization is combined with refinement, performing more iterations requires not only reducing computation time of each iteration, but also additional experimental criteria. One example is the lock procedure which consists of relocating only a fraction of the mesh vertices [TWAD09]. In this approach, the locked vertices are the ones which are incident to only high quality tetrahedra (in terms of dihedral angles). Each time a vertex move or a *Steiner vertex* is inserted into the mesh so as to satisfy user-defined criteria by Delaunay Refinement (sizing, boundary approximation error, element quality), all impacted vertices are unlocked. Our experiments show that more and more vertices get locked as the refinement and optimization procedures go along, until 95% of them are locked. In this context the dynamic approach is mandatory as the mesh refinement and optimization procedure may be applied very locally where the user-defined criteria are not yet satisfied. Note also that the status of each vertex evolves along the iterations as it can be unlocked by the relocation of its neighbors. A static approach would slow down the convergence of the optimization scheme. In this context accelerating the dynamic relocations makes it possible to go further on the number of iterations without slowing down the overall convergence process, so as to produce higher quality meshes.

5 Conclusion

This paper deals with the problem of updating Delaunay triangulations for moving points. We introduce the concepts of tolerance and safe region of a vertex, and put them at work in a dynamic filtering algorithm which avoids unnecessary insert and remove operations when relocating the vertices.

We conduct several experiments to showcase the behavior of the algorithm for a variety of data sets. These experiments show that the algorithm is particularly relevant when the magnitude of the displacement keeps decreasing while the tolerances keep increasing. Such configurations translate into convergent schemes such as the Lloyd iteration. For the latter, and in two-dimensions, the algorithm presented performs up to six times faster than *rebuilding*.

In three-dimensions, and although rebuilding the whole triangulation at each time stamp can be faster than our algorithm when all vertices move, our solution is fully dynamic and outperforms previous dynamic solutions. Such a dynamic property is required for practical variational mesh generation and optimization techniques. This result makes it possible to go further on the number of iterations so as to produce higher quality meshes.

Acknowledgments The authors wish to thank the anonymous referees for their insightful comments and Hans Lamecker for the HEART model. This work has been supported by the *ANR Triangles*, contract number ANR-07-BLAN-0319, and Région PACA.

References

[ABE97] AMENTA N., BERN M., EPPSTEIN D.: Optimal point placement for mesh smoothing. In *Symp. on Discrete Algorithms* (1997), 528–537.

[ABTT08] ACAR U. A., BLELLOCH G. E., TANGWONGSAN K., TÜRKOĞLU D.: Robust kinetic convex hulls in 3D. In *European Symp. on Algorithms* (2008), 29–40.

[ACD*03] ALLIEZ P., COLIN DE VERDIÈRE E., DEVILLERS O., ISENBURG M.: Isotropic Surface Remeshing. In *Shape Modeling International* (2003), 49–58.

[ACYD05] ALLIEZ P., COHEN-STEINER D., YVINEC M., DESBRUN M.: Variational Tetrahedral Meshing. *ACM Transactions on Graphics 24* (2005), 617–625.

[AGMR98] ALBERS G., GUIBAS L. J., MITCHELL J. S. B., ROOS T.: Voronoi diagrams of moving points. *Int. J. of Computational Geometry and Applications 8* (1998), 365–380.

[Aur91] AURENHAMMER F.: Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Computing Surveys 23*, 3 (1991), 345–405.

[AV07] ARTHUR D., VASSILVITSKII S.: K-means++: The advantages of careful seeding. In *Symp. on Discrete Algorithms* (2007), 1027–1035.

[BDP*02] BOISSONNAT J.-D., DEVILLERS O., PION S., TEILLAUD M., YVINEC M.: Triangulations in CGAL. *Computational Geometry: Theory and Applications 22* (2002), 5–19.

[BFG*08] BRÖNNIMANN H., FABRI A., GIEZEMAN G.-J., HERT S., HOFFMANN M., KETTNER L., SCHIRRA S., PION S.: 2D and 3D geometry kernel. In *CGAL User and Reference Manual*, Board C. E., (Ed.), 3.4 ed. 2008.

[CDE*99] CHENG S.-W., DEY T. K., EDELSBRUNNER H., FACELLO M. A., TENG S.-H.: Sliver exudation. In *Symp. on Comp. Geometry* (1999), 1–13.

[CG06] CAZALS F., GIESEN J.: Delaunay triangulation based surface reconstruction. In *Effective Computational Geometry for Curves and Surfaces*, Boissonnat J.-D., Teillaud M., (Eds.). Springer-Verlag, Mathematics and Visualization, 2006.

[Che04] CHEN L.: Mesh smoothing schemes based on optimal Delaunay triangulations. In *International Meshing Roundtable* (2004), 109–120.

[DBC07] DEBARD J.-B., BALP R., CHAINE R.: Dynamic Delaunay tetrahedralisation of a deforming surface. *The Visual Computer* (2007), 12 pp.

[DE06a] DU Q., EMELIANENKO M.: Acceleration schemes for computing centroidal Voronoi tessellations. *Numerical Linear Algebra with Applications 13* (2006).

[DE06b] DU Q., EMELIANENKO M.: Recent progress in robust and quality Delaunay mesh generation. *J. of Computational and Applied Mathematics 195*, 1 (2006), 8–23.

[DEJ06] DU Q., EMELIANENKO M., JU L.: Convergence of the Lloyd algorithm for computing centroidal Voronoi tessellations. *SIAM J. on Numerical Analysis 44*, 1 (2006), 102–119.

[Dev02] DEVILLERS O.: The Delaunay hierarchy. *Int. J. Found. Comput. Sci. 13* (2002), 163–180.

[DLM04] DEVROYE L., LEMAIRE C., MOREAU J.-M.: Expected time analysis for Delaunay point location. *Computational Geometry: Theory and Applications 29* (2004), 61–89.

[DLPT98] DEVILLERS O., LIOTTA G., PREPARATA F. P., TAMASSIA R.: Checking the convexity of polytopes and the planarity of subdivisions. *Computational Geometry: Theory and Applications 11* (1998), 187–208.

[Ede01] EDELSBRUNNER E.: *Geometry and Topology for Mesh Generation*. Cambridge, 2001.

[For87] FORTUNE S. J.: A sweepline algorithm for Voronoi diagrams. *Algorithmica 2* (1987), 153–174.

[FT06] FOGEL E., TEILLAUD M.: Generic programming and the CGAL library. In *Effective Computational Geometry for Curves and Surfaces*, Boissonnat J.-D., Teillaud M., (Eds.). Springer-Verlag, Mathematics and Visualization, 2006.

[GLR97] GARCIA-LOPEZ J., RAMOS P. A.: Fitting a set of points by a circle. In *Symp. Comp. Geometry* (1997), 139–146.

[GR04] GUIBAS L., RUSSEL D.: An empirical comparison of techniques for updating Delaunay triangulations. In *Symp. on Comp. Geometry* (2004), 170–179.

[GS78] GREEN P., SIBSON R.: Computing Dirichlet tessellations in the plane. *The Computer J 21*, 2 (1978), 168–173.

[Gui98] GUIBAS L. J.: Kinetic data structures: a state of the art report. In *Workshop on the Algorithmic Foundations of Robotics* (1998), 191–209.

[HW79] HARTIGAN J. A., WONG M. A.: Algorithm AS 136: A k-means clustering algorithm. *Applied Statistics 28*, 1 (1979), 100–108.

[Llo82] LLOYD S.: Least squares quantization in pcm. *Information Theory, IEEE Transactions on 28*, 2 (1982), 129–137.

[LWL*08] LIU Y., WANG W., LÉVY B., SUN F., YAN D.-M., LU L., YANG C.: *On Centroidal Voronoi Tessellation—Energy Smoothness and Fast Computation*. Tech. Rep. TR-2008-18, Dept. of Computer Science, Univ. of Hong Kong, 2008.

[ORSS06] OSTROVSKY R., RABANI Y., SCHULMAN L. J., SWAMY C.: The effectiveness of Lloyd-type methods for the k-means problem. In *Symp. FOCS* (2006), 165–176.

[PS90] PREPARATA F. P., SHAMOS M. I.: *Computational Geometry: An Introduction*, 3rd ed. Springer-Verlag, Oct. 1990.

[PT08] PION S., TEILLAUD M.: 3D triangulations. In *CGAL User and Reference Manual*, Board C. E., (Ed.), 3.4 ed. 2008.

[Rus07] RUSSEL D.: *Kinetic Data Structures in Practice*. PhD thesis, Stanford University, 2007.

[SG86] SABIN M. J., GRAY R. M.: Global convergence and empirical consistency of the generalized Lloyd algorithm. *IEEE Transactions on Information Theory 32*, 2 (1986), 148–155.

[SH75] SHAMOS M. I., HOEY D.: Closest-point problems. *Symp. FOCS* (1975), 151–162.

[She96] SHEWCHUK J. R.: Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, vol. 1148. Springer-Verlag, 1996, 203–222.

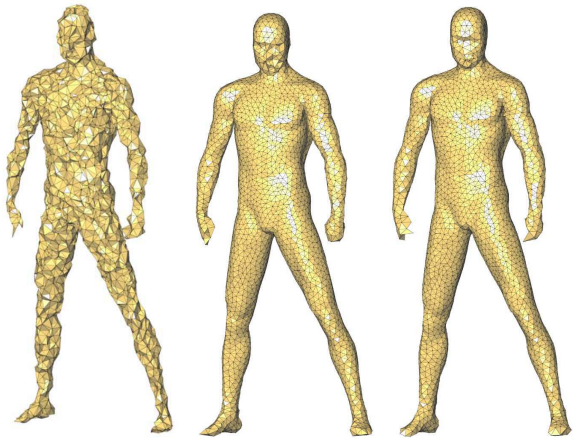
[She05] SHEWCHUK R.: Star splaying: an algorithm for repairing Delaunay triangulations and convex hulls. In *Symp. on Comp. Geometry* (2005), 237–246.

[TAD07] TOURNOIS J., ALLIEZ P., DEVILLERS O.: Interleaving Delaunay refinement and optimization for 2D triangle mesh generation. In *Meshing Roundtable Proc.* (2007), 83–101.

[TWAD09] TOURNOIS J., WORMSER C., ALLIEZ P., DESBRUN M.: Interleaving Delaunay refinement and optimization for practical isotropic tetrahedron mesh generation. *ACM/SIGGRAPH Transactions on Graphics 28(3)* (2009).

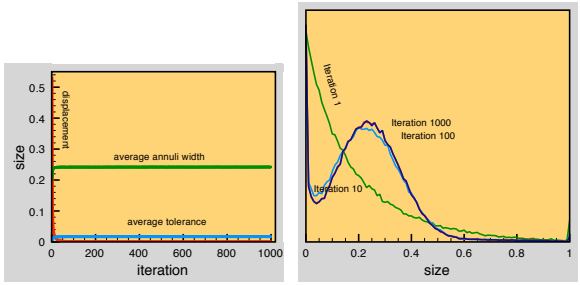
[VCP08] VALETTE S., CHASSERY J. M., PROST R.: Generic remeshing of 3D triangular meshes with metric-dependent discrete Voronoi diagrams. *IEEE Transactions on Visualization and Computer Graphics 14*, 2 (2008), 369–381.

[Yvi08] YVINEC M.: 2D triangulations. In *CGAL User and Reference Manual*, Board C. E., (Ed.), 3.4 ed. 2008.



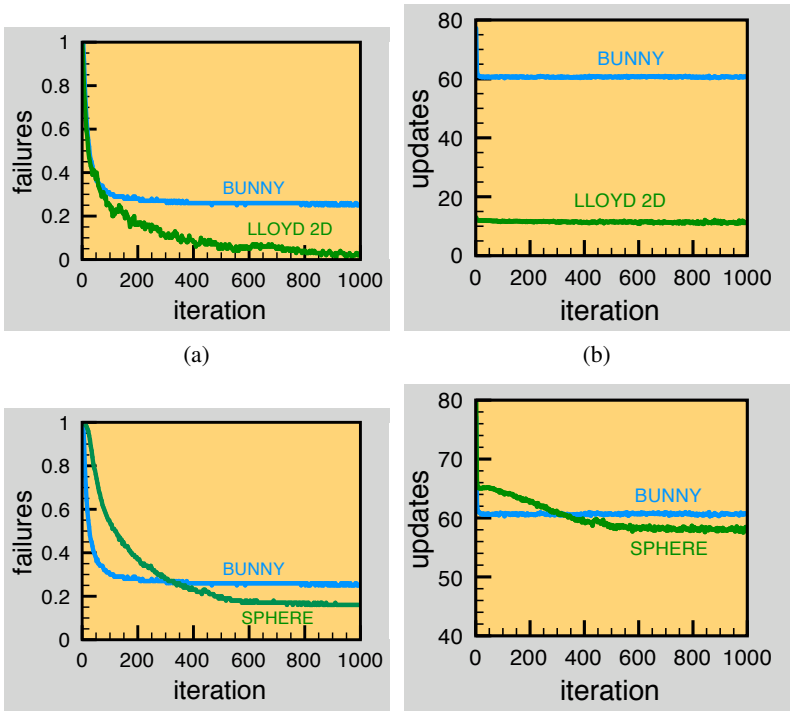
(a) (b) (c)

Figure 9: **Mesh quality improvement.** (a) MAN initially; (b), (c) MAN after 100 and 1,000 iterations respectively.



(a) evolution (b) distribution

Figure 10: **Statistics in 3D.** Consider a sphere such that the quantity representing the number of points divided by its volume is equal to 1. The cubic root of this quantity is the unity of distance. The figures depict 1,000 iterations of the meshing optimization process on SPHERE: (a) evolution of the average displacement size, average standard annulus width and average tolerance of vertices over 1,000 iterations; (b) distribution of the standard annulus width for 1, 10, 100 and 1,000 iterations.



(a) (b) (c) (d)

Figure 11: **Filter failures and number of tolerance updates.** Consider the execution of the filtering algorithm for: the two-dimensional data set of Lloyd's iterations with $\rho = x^2$; BUNNY and SPHERE. The curves depict the percentage of relocations for which the filter fails along the iterations, for (a) BUNNY and Lloyd's iteration with $\rho = x^2$, (c) BUNNY and SPHERE; the average number of tolerance updates done per filter failure, for (b) BUNNY and Lloyd's iteration with $\rho = x^2$, (d) BUNNY and SPHERE. Observe how the complexity is higher for the three-dimensional cases. Also note how the number of filter failures is higher for BUNNY compared to SPHERE (around 25% for BUNNY against 16% for SPHERE at the 1000th iteration).

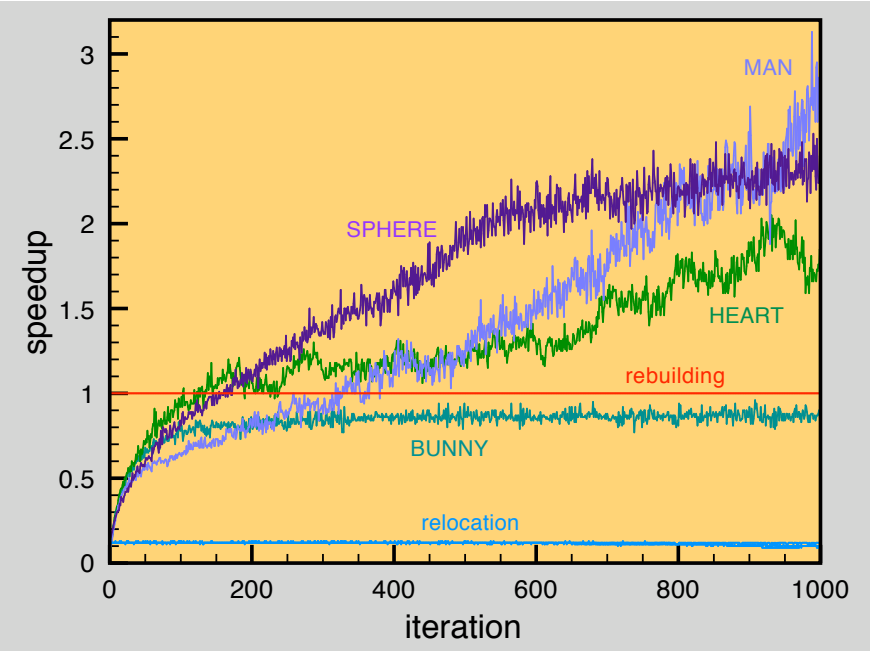


Figure 12: **Speedup factors: NODT.** The figure represents the speedup factor of each algorithm with respect to the rebuilding algorithm along the iterations, for a given input. Names with capital letters (e.g. "SPHERE") in the figure, means the filtering algorithm working in the respective input data (e.g. SPHERE). The "speedup" of the relocation algorithm with respect to rebuilding is more or less constant for each input data.