Available online at www.sciencedirect.com
ScienceDirect
Computational Geometry 38 (2007) 100–110
Computational Geometry
Theory and Applications
www.elsevier.com/locate/comgeo

A generic software design for Delaunay refinement meshing ☆

Laurent Rineau *, Mariette Yvinec

INRIA, BP 93 06902 Sophia Antipolis, France

Received 27 February 2006; received in revised form 26 October 2006; accepted 15 November 2006

Available online 4 April 2007

Communicated by B. Gärtner and R. Veltkamp

Abstract

This paper describes a generic software designed to implement meshing algorithms based on the Delaunay refinement paradigm. Such a meshing algorithm is generally described through a set of rules guiding the refinement of mesh elements. The central item of the software design is a generic class, called a mesher level, that is able to handle one of the rules guiding the refinement process. Several instantiations of the mesher level class can be stacked and tied together to implement the whole refinement process. As shown in this paper, the design is flexible enough to implement all currently known mesh generation algorithms based on Delaunay refinement. In particular it can be used to generate meshes approximating smooth or piecewise smooth surfaces, as well as to mesh three dimensional domains bounded by such surfaces. It also adapts to algorithms handling small input angles and various refinement criteria. This design highly simplifies the task of implementing Delaunay refinement meshing algorithms. It has been used to implemented several meshing algorithms in the CGAL library.

Keywords: Mesh generation; Delaunay refinement; Generic programming

1. Introduction

Delaunay refinement is recognized as one of the most powerful techniques in the field of mesh generation and surface approximation. The Delaunay refinement paradigm yields robust meshing algorithms that are proved to terminate and to offer some guarantee on the quality of the output meshes. Moreover, it allows the user to get a fine control on the size of the mesh elements, for instance through a (possibly non-uniform) sizing field and it constructs meshes with a good grading, that is able to conform to quickly varying sizing fields.

The pioneer works on Delaunay refinement are due to Chew [9] and Ruppert [16]. Ruppert proposed a two-dimensional mesh generator for domains with piecewise linear boundaries and constraints. Provided that the boundaries and constraints do not form angles smaller than $\frac{\pi}{3}$, Ruppert's algorithm guarantees a lower bound on the smallest angle in the mesh. Furthermore, this bound is achieved by adding an asymptotically optimal number of Steiner vertices. Later on, Shewchuk improved the handling of small angles in two dimensions [21] and generalized the method

☆ Work partially supported by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-006413 (ACS—Algorithms for Complex Shapes) and by the European Network of Excellence AIM@SHAPE (FP6 IST NoE 506766).
* Corresponding author.
E-mail address: laurent.rineau@sophia.inria.fr (L. Rineau).

0925-7721/$ – see front matter
doi:10.1016/j.comgeo.2006.11.008

to generate three-dimensional meshes for domains with piecewise linear boundaries [19]. The handling of small angles is more puzzling in three dimensions, where dihedral angles and facet angles come into play. Using the idea of protecting spheres around sharp edges, first proposed by Cohen-Steiner, Colin de Verdière, and Yvinec [2], Cheng and Poon [12] provided a thorough handling of small input angles formed by boundaries and constraints. Cheng, Dey, Ramos, and Ray [5] turned the same idea into a simpler and practical meshing algorithm.

In three-dimensional space, Delaunay refinement is able to produce tetrahedral meshes with an upper bound on the radius–edge ratios of the tetrahedra, where the radius–edge ratio of a tetrahedron is the ratio between its circumradius and the length of its shortest edge. This eliminates from the mesh all kinds of degenerate tetrahedra, except the *slivers*. A sliver can be described as a tetrahedron formed by four vertices close to the equatorial circle of a sphere and roughly equally spaced on this circle. Cheng et al. [4], and later on Cheng and Dey [3], proposed to exude slivers from the mesh by turning the Delaunay triangulation into a weighted Delaunay triangulation with carefully chosen small weights associated to the vertices. Li and Teng [13] proposed to avoid the creation of slivers by relaxing the choice of refinement vertices inside small areas around the circumcenters of the elements to be refined. Recently, Cheng, Dey, Ramos, and Ray [7] combined the weight pumping method for sliver exudation with the protecting spheres methods to handle small input angles. They provide thus a meshing algorithm that is able to handle any polyhedral domain and yields tetrahedra with bounded aspect ratio, except maybe in the neighborhood of sharp input edges.

Delaunay refinement has also been used for surface approximation and surface meshing. Delaunay refinement for surfaces has been pioneered by Chew [10]. Boissonnat and Oudot [1] applied Delaunay refinement to the meshing of smooth surfaces and provide a surface meshing algorithm with guarantees on the approximation error as well as on the shape of mesh elements. The method is flexible and can be applied in various situations ranging from implicit surfaces to level-sets in 3D gray scaled images and including point set surfaces. In a recent paper, Oudot, Rineau, and Yvinec [14] used this surface meshing algorithm as a building block to provide a three-dimensional meshing algorithm for domains bounded by smooth surfaces. In the work of Boissonnat and Oudot [1], the Delaunay refinement process is guided by the circumradii of mesh elements that are compared to a local characteristic of the surface, called the local feature size. In a related work, Cheng, Dey, Ramos, and Ray [6] propose to mesh surfaces using a Delaunay refinement process based on topological properties. This last method avoids the need to compute or estimate the local feature size of the surfaces but requires to compute critical points of the surface and some of the silhouette points.

There already exist several popular implementations of Delaunay refinement algorithms, such as Triangle [18], Pyramid [19], and Qualmesh [7]. This paper does not present an implementation of a specific algorithm, but a generic software framework that allows to implement Delaunay refinement based meshing algorithms. In this framework, a Delaunay refinement process is implemented using a few instances of a generic class called *mesher level*. A Delaunay refinement process is generally described by a set of refinement rules which are iteratively applied with some given priority order. Roughly, each instantiated mesher level is associated to one of the rules and takes care of the refinement of mesh elements of a given dimension (edges, triangles or tetrahedra) according to some refinement criteria. The different instantiated mesher levels interact to ensure the representation of lower dimensional features such as the domain boundaries, or sharp features of the approximated surfaces. This framework is already in use in the CGAL library [8], to implement several meshing algorithms [15,17], and has proved its efficiency. The generic meshing design is described in this paper using mainly the features of the C++ programming language, however we claim that such a design could be implemented in other programming languages as well.

To be more concrete and precise in our further description of the framework, we first recall in Section 2 a typical example of a Delaunay refinement algorithm namely the algorithm proposed by Shewchuk [19] to mesh a three dimensional polyhedral domain. Section 3 presents the mesher level class and describes how several mesher levels can be instantiated and tied together to implement a complete refinement process. Section 4 shows how the framework applies to implement various kinds of algorithms using the Delaunay refinement process. The concluding section provides some examples obtained with the CGAL library where the design has been used to implement a surface mesher [17] and a three dimensional mesher for domains with curved boundaries.

2. Delaunay refinement for a 3D polyhedral domain

To serve as a typical example, we recall in this section the algorithm proposed by Shewchuk [21] to mesh a three dimensional polyhedral domain. Let us assume that the domain to be meshed is given through a piecewise linear complex, called a *PLC* for short. In three dimension, a PLC is a collection of faces of dimension 0, 1 or 2, respectively

called vertices, edges and facets. The vertices are points, the edges are segments and the facets are polygonal regions. The facets are not required to be convex nor simply connected, they can have holes or be pierced by slits or isolated vertices. The faces of a PLC satisfy the following conditions:

- the boundary of any face in the PLC is the union of some faces of the PLC.
- the intersection of any two faces of the PLC is either empty or the union of some faces of the PLC.

Once given a PLC C, the domain D to be meshed is described as the union of some bounded components of $\mathbb{R}^3 \setminus C$. Such a description implies, that the domain is bounded and, that its boundary is the union of some faces of the PLC.

The goal is to output a mesh of the domain, that is a three dimensional triangulation T such that the vertices of C are vertices of T, each edge of C is the union of some edges of T and each facet of C is the union of some facets in T. In addition, the tetrahedra of T included in the domain D are required to match some size and shape criteria.

The refinement process maintains a Delaunay triangulation. It starts with the Delaunay triangulation of the vertices of C and iteratively adds new vertices, called Steiner vertices, in the triangulation in order to enforce the representation of the faces of C as union of mesh elements and to discard bad elements. In the present case bad elements are tetrahedra that do not fulfill the size and shape criteria.

To give a more precise description, we recall some definitions given by Shewchuk [21]. Vertices inserted in the edges of C partition those segments into smaller segments that are called *subsegments*. One of the first goals of the algorithm is to achieve and maintain the fact that subsegments are Delaunay edges. Once this fact is granted, each facet F of C appears as a union of facets in the two dimensional Delaunay triangulation $T_2(F)$ of the vertices of the current mesh located in the hyperplane supporting F. The facets of $T_2(F)$ included in F are called the *subfacets* of F. In the sequel, the faces (vertices, edges and facets) of C are called *input constraints*, subsegments and subfacets are called *constrained elements*.

A point is said to *encroach* a subsegment or a subfacet if it lies in its smallest circumscribing sphere. A subsegment or a subfacet is said *to be encroached* if it is encroached by some vertex of the current mesh.

The algorithm refines encroached subsegments by adding their midpoints as vertices. Encroached subfacets and bad tetrahedra are refined by adding their circumcenters. Encroached subsegments have priority over encroached subfacets and bad tetrahedra, and encroached subfacets have priority over bad tetrahedra. Furthermore, the algorithm ensures that no subfacet circumcenter is inserted if it encroaches some subsegment and that no tetrahedron circumcenter is inserted if it encroaches some subsegment or subfacet. More precisely, the algorithm calls in turn the three procedures `refine_edge`, `refine_facet` and `refine_tetrahedron` applying iteratively one of the three following rules. The rules are applied with a priority order which means that Rule i is applied only if no Rule j with $j < i$ applies.

Rule 1. If there is an encroached subsegment e
call `refine_edge`(e), i.e.
Compute the midpoint v of e
insert v

Rule 2. If there is an encroached subfacet f
call `refine_facet`(f), i.e.
Compute the circumcenter v of f
if v encroaches some edge e, call `refine_edge(e)`
else insert v

Rule 3. If there is some bad tetrahedron t
call `refine_tetrahedron`(t), i.e.
Compute the circumcenter v of f
if v encroaches some edge e, call `refine_edge`(e)
else if v encroaches some face f, call `refine_facet`(f)
else insert v

For appropriate size and shape criteria, the above algorithm is proved to terminate provided that input constraints do not form angles smaller than 90°. The angles considered here are either dihedral angles formed by two adjacent

facets of C, or edge–facet angles formed by a facet and an edge sharing a vertex or edge–edge angles formed by two edges sharing a vertex.

3. A generic software design

3.1. The mesher level

In algorithms based on Delaunay refinement, the refinement process is guided by a set of rules. In general cases, each rule takes care of the refinement of a different type of mesh elements. These types are subsegments, subfacets and tetrahedra in the algorithm described in Section 2. The rules are applied iteratively and with a precise priority order. The central item of our design is a class, called *mesher level*, which is designed to be associated with one rule of the refinement process and one type of mesh elements. Several instances of mesher levels are stacked and tied together to implement a given refinement algorithm. For instance, the implementation of the algorithm of Section 2 involves three mesher levels, one for each rule.

The main tasks devoted to a mesher level is to maintain a set, called `to_be_refined` in the sequel, of elements to be refined and to achieve, when possible, the refinement of those objects. Elements to be refined are either bad elements which do not fulfill size and shape criteria or constrained elements which have to be refined to enforce the representation of input constraints. To achieve the refinement of a given element, the mesher level has to compute a refinement point, check the agreement of this point by other mesher levels, and if agreed insert it as a new mesh vertex. A refinement point is agreed if it does not encroach constrained elements handled by higher priority mesher levels. The set `to_be_refined` of a mesher level has to be updated upon each insertion of a refinement point, whatever may be the level mesher inserting this point and also upon rejection of a refinement point that encroaches constrained elements handled at this level.

To achieve these tasks a mesher level provides several methods. The methods can be classified into two groups.

- The methods of the first group, called *primitive methods*, are all the methods that need a specialized version for each instance of mesher level because their code depends on the actual mesher level. This group includes in particular the methods handling the set `to_be_refined` which can be stored with or without some ordering. Besides the methods `is_to_be_refined_empty()`, `get_next_element()` and `pop_next_element()` whose functionalities are rather obvious, we find for example the following methods.
  - `scan_triangulation()` scans the whole triangulation, and initializes the `to_be_refined` set.
  - `refinement_point(e)` computes a refinement point for element e.
  - `check_refinement_point(p)` checks the refinement point p computed by this level or by a lower priority level. If p has been computed by a lower priority level and encroaches some constrained element e handled by this level, p is rejected and e is added in the `to_be_refined` set.
  - `after_insertion(p)` updates the `to_be_refined` set after the insertion of point p.
- The method of the second group, called *frame methods*, are shared by all mesher levels. These methods implement the framework of the refinement process. They issue calls to the primitive methods and are responsible for the communication between the different mesher levels involved in the same Delaunay refinement process. Among frame methods are the following methods.
  - `previous()` returns a pointer to the previous mesher level in the same refinement process. The mesher levels involved in a given refinement process are thought of as ordered according to decreasing priority order of the associated rules.
  - `check_refinement_point_by_all_levels(p)` triggers the check of refinement point p by the current level and all higher priority levels.
  - `process_one_element()` handles one element of the `to_be_refined` set. It finds the next element to be refined, computes its refinement point p, calls `check_refinement_point_by_all_levels(p)` and eventually inserts p if agreed.
  - `refine()` drives the refinement process. It calls the `refine()` method of the previous level and `process_one_element()` until the `to_be_refined` sets of this level and of higher priority levels are empty. The whole Delaunay refinement process is launched by a call to the `refine()` method of the mesher level of lowest priority.

Because all mesher levels share a set of common functionalities (the frame methods), it is natural to implement them as derived classes with a common base class, called `Mesher_level_base`. Each derived mesher level class has to provide its own specialized version of primitive methods. However, because the primitive methods are called by the frame methods, they should be implemented as virtual methods. To avoid the run time cost of virtual methods, we have chosen to implement mesher levels using the *Curiously Recurring Template Pattern* (CRTP), described by James O. Coplien in [11].

According to this pattern, the base class `Mesher_level_base` is a class template and the derived class, which will be a mesher level, is plugged as parameter in the base class. For instance, the lowest priority mesher level involved in the algorithm of Section 2, called `Tetrahedra_level`, is defined as:

```
class Tetrahedra_level
  : public Mesher_level\_base<Tetrahedra_level> {
  ...
}
```

In this way, `Tetrahedra_level` both derives from `Mesher_level_base`, thus inheriting the frame methods, and also provides the specialized primitive methods required by the frame methods.[1]

As a matter of fact, the class `Mesher_level_base` has not only one template parameter (`CRTP_Derived`) for the mesher level type but also three other parameters (`Triangulation`, `Element` and `Previous_level`). The parameter `Triangulation` is the type of the triangulation class, `Element` is the type of elements handled by this level and `Previous_level` is the type of the previous mesher level in priority order. A sketch of the `Mesher_level_base` class is given in Appendix A.

3.2. Tying up mesher levels

A whole Delaunay refinement process generally involves several mesher levels which have to be tied up to work together.

For example, the algorithm of Section 2 involves three mesher levels called `Edges_level`, `Facets_level` and `Tetrahedra_level`, each corresponding to a different refinement rule. The `Previous_level` parameter of `Edges_level` is instantiated by a trivial mesher level class. The `Previous_level` parameters of `Facets_level` and `Tetrahedra_level` are instantiated respectively by `Edges_level` and `Facets_level`. Mesher levels are instantiated and linked (via `Previous` template parameters, and a pointer) in order of priority. It means that a mesher level can call methods of higher priority levels. However some more communication between levels is needed. In particular, when a level inserts a refinement point, the `to_be_refined` sets of all others levels may have to be updated. This is achieved using so-called *visitors*.

The visitors pattern is commonly used to customize the behavior of an algorithm [22]. A visitor is similar to a function object but may have several methods, instead of a single `operator()`. In our implementation, the methods `refine()` and `process_one_element()` of the generic class `Mesher_level_base` are parametrized by a `Visitor` type, and take an argument of type `Visitor`. The plugged in visitor class is required to have a few methods among which a method `after_insertion(p)` designed to update the `to_be_refined` sets of all mesher levels after the insertion of a refinement point. In the base class, therefore in each mesher level, the method `visitor.after_insertion()` is called after each insertion of a refinement point (insertions happen in the method `process_one_element()`[2]). Visitors are instantiated and created after the instantiation and creation of the mesher levels. This way, a visitor can store a reference to any mesher level involved in the refinement process and

[1] An alternative design would be to disconnect the two goals (inheriting frame methods and providing primitive methods) and use a policy class to provide the primitive methods:

```
class Tetrahedra_level
  : public Mesher_level_base<Tetrahedra_level_policy>
```

In implementation, `Tetrahedra_level_policy` provides the primitive methods needed by `Mesher_level_base`. The use of the CRTP allows `Tetrahedra_level` and `Tetrahedra_level_policy` to be the same class. This reduces the duplication of information, and prevents a wrong choice of the policy instance.

[2] The code of `refine()` and `process_one_element()` is sketched in Appendix A.

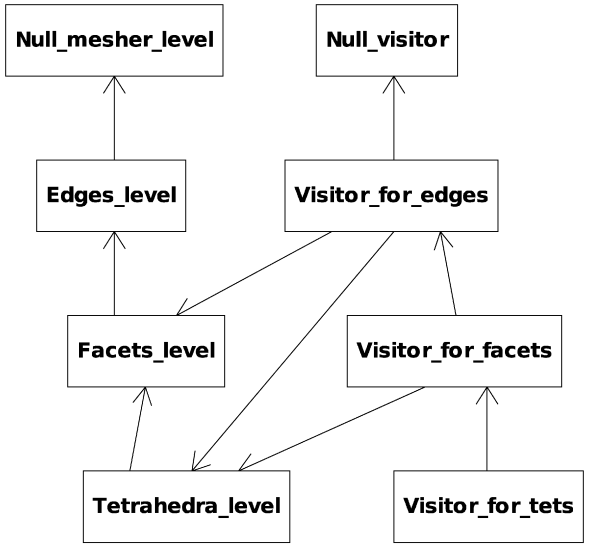

Fig. 1. UML class diagram mesher levels. A link from a class A to a class B means that A has to call methods of B, and thus must be instantiated after B.

the `after_insertion(p)` method of a visitor is able to call the primitive methods `after_insertion(p)` of each mesher level. For technical reasons, visitors of the different mesher levels are also linked together via `previous()` member functions, exactly the same way mesher levels are.

In Fig. 1, an UML diagram shows dependencies and connection between mesher levels and visitors involved in the implementation of the algorithm of Section 2. The classes `Null_mesher_level` and `Null_visitor` are respectively a trivial mesher level and a trivial visitor class whose methods just do nothing.

4. Applications and examples

This section shows how the framework described in the previous section can be used to implement several meshing algorithms based on Delaunay refinement.

4.1. A 2D mesher for planar straight line graphs

In [20], Shewchuk presents an algorithm for meshing domains, whose boundaries and constraints are described by planar straight line graphs. The algorithm is based on two rules, the first one refines constrained edges and the other one refines bad facets. The algorithm is therefore implemented with two mesher levels, respectively called `Edges_level_2` and `Facets_level_2`.

The algorithm in [20] is able to handle small input angles, i.e. two constraints sharing a vertex may form a small angle. The algorithm has two features to prevent infinite cascade of insertions caused by small input angles. First, constrained edges sharing the same vertex are gathered in *clusters* such that two consecutive edges in a cluster form an angle smaller than 60°. Edges in a cluster are not refined at their midpoint but along their intersection with an appropriate set of concentric shells centered on the common vertex of the cluster. In that way, after a few insertions, edges of a cluster have equal lengths. When an edge of a cluster is refined, the whole cluster is refined but the cascade of insertions for constraints enforcing stops there. This feature is encoded in the `compute_refinement_point()` primitive of `Edges_level_2`. The second feature concerns the refinement of bad facets. When the refinement point of a bad facet encroaches a constrained edge in a cluster, the algorithm tests if the refinement of the cluster could jeopardize the termination of the algorithm. In such a case, the refinement point of the bad facet is still rejected but the refinement of the cluster is refused, the bad facet is definitely discarded from the `to_be_refined` set and kept in the final mesh. To implement this feature, the method `check_refinement_point()` has a return type with three possible values `PERMITTED`, `REJECTED` and `REJECTED_FOREVER`. The above behavior is triggered by a return of the value `REJECTED_FOREVER`.

4.2. A 3D mesher for polyhedra with small angles

The algorithm of Cheng et al. [5] differs from the one described in Section 2, in its ability to handle small input angles. The edges that are intersections of two facets forming a dihedral angle smaller than 90° are called *sharp edges*, and their subsegments are called *sharp subsegments*. Vertices that are endpoints of sharp segments or apex of small input angles are called *sharp vertices*. The idea is to protect sharp vertices and sharp edges with unions of balls inside which the insertion of tetrahedra circumcenter is disallowed.

The first phase of the algorithm, called CONFORM, enforces the representation of constraints in the mesh and ensures a refinement of subsegments in sharp edges suitable to compute the protecting spheres. In [5], this first phase is described by three rules, however it is convenient to split the third rule (that handles both constrained edges and constrained facets) in two subrules and to implement this phase using four mesher levels. The first two rules are quite similar to those of Section 2 and the associated mesher levels, `Edges_level` and `Facets_level`, are almost identical to those described in Section 3.2. There are nevertheless two main differences. The first one concerns the refinement point of edges lying in the protecting ball of sharp vertices. This point is not the midpoint of the edge but the midpoint of the spherical arc with the same endpoints (according to the SOS strategy defined in [2]). This is taken into account by the `compute_refinement_point` method of `Edges_level`. The second difference is that subfacets are not required to be free of encroachment but just to be present in the mesh. Only subfacets of the bounding box are required to be free of encroachment. The `scan_triangulation()`, `check_refinement_point()` and `after_insertion()` methods of `Facets_level` are specialized to take this into account.

At the end of CONFORM phase, *protecting balls* are defined from the subsegments in sharp edges. The aim of the two last rules in the CONFORM phase is to refine subsegments in sharp edges so that the size of protecting balls roughly match the local feature size.

Rule 3. If the midpoint of a sharp subsegment s encroaches a subsegment e
and if s and e are contained in disjoint elements of the PLC,
call `refine_edge`(e)

Rule 3′. If the midpoint of a sharp subsegment s encroaches a subfacet f
and if s and f are contained in disjoint elements of the PLC,
call `refine_facet`(f)

These two rules are implemented by two mesher levels respectively similar to `Edges_level` and `Facets_level`. The only difference resides in the selection of elements to be refined in these levels. This is taken into account in the `scan_triangulation` and `after_insertion` methods.

When the CONFORM phase is over, the set of protecting balls is fixed and the algorithm enters the REFINE phase. The refine phase is a Delaunay refinement process analog to the algorithm presented in Section 2 except that no refinement point of a tetrahedra can be inserted in a protecting ball. The REFINE phase is implemented with three mesher levels. The levels `Edges_level` and `Facets_level` of the CONFORM phase are reused coupled with a `Tetrahedra_level`. This `Tetrahedra_level` has a specialized version of the `check_refinement_point` method, returning the value `REJECTED_FOREVER` if the candidate refinement point lies in a protecting ball.

In [7], Cheng and al. extend their meshing algorithm, adding a PUMPING phase to get rid of sliver tetrahedra, the only type of nearly degenerated tetrahedra which are not disallowed by Delaunay refinement. This phase does not add or move any vertex of the mesh. The pumping phase consists in changing the triangulation from a Delaunay triangulation into a weighted Delaunay triangulation, assigning suitable weights to the vertices to pump slivers out of the mesh. Clearly, the pumping phase is not a refinement and mesher levels are not relevant to implement this phase. Nevertheless, if a pumping phase is performed the REFINE phase has to be modified to prevent the pumping to ruin the representation of constraints. A new refinement rule anticipating the weighting of vertices is added. This rule concerns either subsegments or subfacets and is also conveniently split into two subrules with low priority (Rule 4 and Rule 4′) that can be implemented with two mesher levels. These two levels are respectively analog to `Edges_level` and `Facets_level` except for the selection of to be refined elements which takes into account the future weighting of vertices. To be more precise, we need to generalize the notion of encroachment in the case where the vertices of the mesh are assigned weights. The orthosphere of a subsegment e (resp. of a subfacet f) is the smallest sphere orthogonal

to the weighted points corresponding to the vertices of e (resp. of f). A subsegment (resp a subfacet) is said to be encroached by a weighted point if the weighted point has a negative weighted distance to its orthosphere. Rule 4 and Rule $4'$ refine respectively subsegments and subfacets that would be encroached if the vertices of tetrahedra that are slivers, were assigned a weight equal to the maximum weight they can be assigned during the pumping phase.

4.3. A mesher for smooth surfaces

Boissonnat and Oudot [1] proposed a Delaunay refinement process to compute a two dimensional mesh approximating a smooth surface. The algorithm maintains a three dimensional Delaunay triangulation and its restriction to the surface which is obtained by selecting in the Delaunay triangulation, the facets whose dual Voronoi edges intersect the surface.

Because the surface is assumed to be smooth there is no constrained edge and the whole refinement process can be implemented using a single mesher level called `Surface_mesher`. The criteria to refine triangles take into account the shape of triangles and their size relative to the so-called *local feature size* of the surface. The refinement point of a facet is a point on the surface where the dual Voronoi edge intersects the surface. The requirements on the surface reduce to an oracle able to detect the intersections of the surface with a segment and to compute an intersection point. This fact makes the algorithm flexible enough to be applied in various cases: surfaces described by an implicit equation, level-sets in 3D gray scaled images, point set surface etc.

In [14], the surface mesher of [1] is extended into a three dimensional mesher able to mesh a domain bounded by smooth surfaces. The extension is performed by adding to the previous algorithm a rule to refine tetrahedra. The implementation of this new algorithm mainly consists in combining the mesher level `Tetrahedra_level` with `Surface_mesher`, as previous mesher level.

Note that, although we have not implemented it, the topology based Delaunay refinement process proposed by Cheng, Dey, Ramos and Ray [6] could also be implemented using mesher levels. A natural implementation would involve one mesher level for each topological rule of the algorithm (VorEdge, TopoDisk, FacetCycle and Silhouette). Two more mesher levels with lower priority would be involved to implement the rules that ensures the quality and smoothness of the generated mesh.

5. Results and conclusion

The mesher level framework has been used to develop a mesh generation package in the CGAL library [8]. Currently this package includes a two dimensional mesher [15] as described in Section 4.1, a surface mesher [17] for smooth surfaces and a three dimensional mesher for volume bounded by smooth surfaces as described in Section 4.3. Figs. 2 and 3 show two meshes generated with this package. The first one triangulates a three dimensional domain bounded by an implicit surface, called the tangle cube, and described by an algebraic equation. No size criterion was imposed on the tetrahedra of this mesh. The mesh includes 57293 vertices and has been generated in 23 s on a Pentium 4 (3.06 GHz). The second one triangulates a three dimensional volume bounded by some level set in the 3D gray scaled image of a skull. We used a uniform size criterion on tetrahedra. This mesh includes 89245 vertices and has been generated in 38 s on the same Pentium 4.

The three dimensional mesh generation package of CGAL is still under development. The next step is to mesh domains bounded by piecewise smooth surfaces with a good approximation of the one dimensional singular features. Of course, the plan is just to add before the `Surface_mesher` level a highest priority level to capture those one dimensional singular features.

The mesher level framework highly simplifies the task of implementing Delaunay refinement meshing algorithms. Indeed, the development of a mesher level reduces to the implementation of a few primitive methods. The tying up of several mesher levels, to implement a meshing algorithm involving several rules, reduces to implement the appropriate visitor classes. Furthermore the common interface of mesher levels allows the reuse of mesher levels and the use of mesher levels created by different developers. Therefore this framework is a useful tool to experiment on Delaunay refinement algorithms and develop mesh generators.

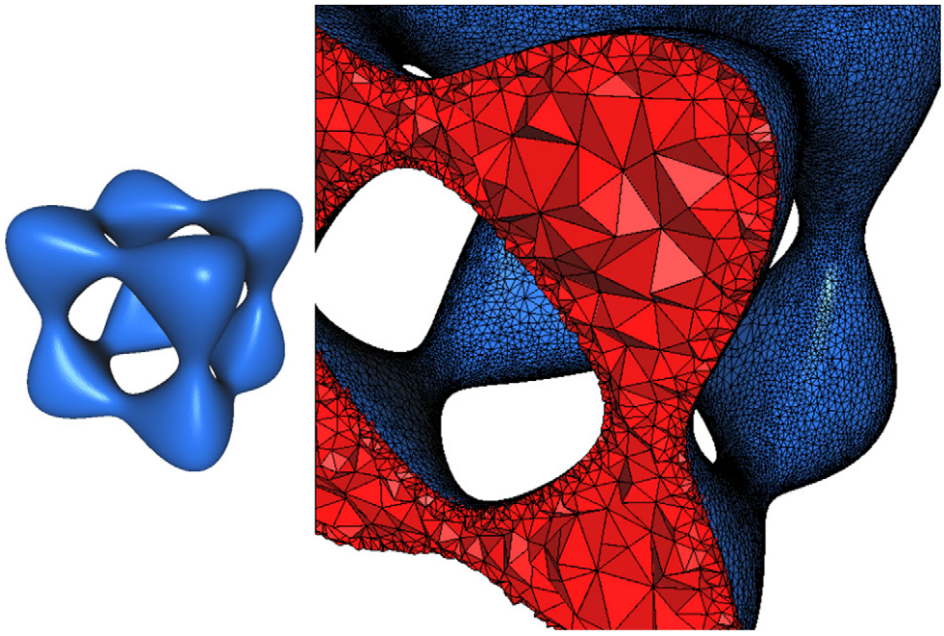

Fig. 2. Mesh of a 3D domain bounded by an algebraic surface, the tangle cube.

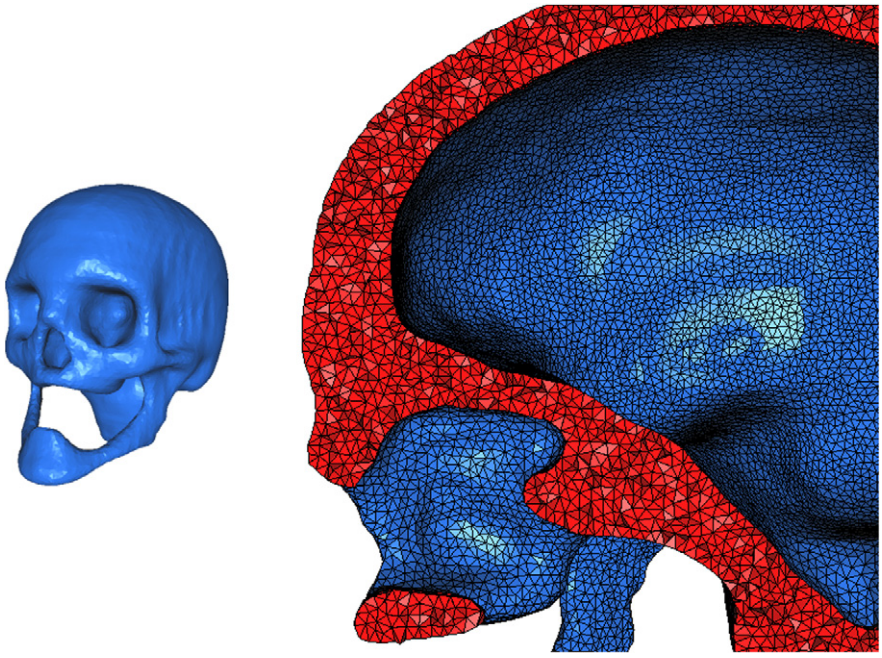

Fig. 3. Mesh of a 3D domain bounded by a level set in a 3D grey-scaled image.

Appendix A. Sketch of the `Mesher_level_base` class

This appendix provides a sketch of the `Mesher_level_base` code. This is not actually real C++ code but rather a simplified pseudocode.

```
template <class Triangulation,
          class Element,
```

```
          class CRTP_Derived,
          class Previous_level = Null_mesher_level>
class Mesher_level_base {

  Previous_level& previous; // reference to the previous mesher level
  Triangulation& tr;        // reference to the triangulation

public:
  CRTP_Derived& derived() {
    return static_cast<CRTP_Derived&>(*this);
  }

  bool are_lists_empty() {
    return derived().is_list_empty() &&
       previous().are_lists_empty();
  }

  Encroaching_status check_refinement_point_by_all_levels(Point p) {
    Encroaching_status status =
          previous.check_refinement_point_by_all_levels(p);
    if( status == NO_ENCROACHMENT )
      return derived().check_refinement_point(p);
    else
      return status;
  }

  template <class Visitor>
  void process_one_element(Visitor visitor = Null_visitor) {
    Element e = derived().get_next_element();
    Triangulation::Point p = derived().refinement_point(e);
    Encroachment_status point_status =
                          check_refinement_point_by_all_levels(p)
    if( point_status == ACCEPTED ) {
      visitor.before_insertion(p);
      Triangulation::Vertex_handle vh = tr.insert(p);
      visitor.after_insertion(vh);
      pop_next_element();
    } else if( point_status == REJECTED_FOREVER ) {
      pop_next_element();
    }
  }

  template <class Visitor>
  void refine(Visitor visitor = Null_visitor) {
    previous().refine(visitor.previous());
    while( ! derived().is_list_empty() ) {
      process_one_element(visitor);
      previous().refine(visitor.previous());
    }
  }
}
```

References

[1] J.-D. Boissonnat, S. Oudot, Provably good sampling and meshing of surfaces, Graphical Models 67 (2005) 405–451.
[2] D. Cohen-Steiner, É. Colin de Verdière, M. Yvinec, Conforming Delaunay triangulations in 3d, Comput. Geom. 28 (2–3) (2004) 217–233.
[3] S.-W. Cheng, T.K. Dey, Quality meshing with weighted Delaunay refinement, SIAM J. Comput. 33 (1) (2003) 69–93.
[4] S.-W. Cheng, T.K. Dey, H. Edelsbrunner, M.A. Facello, S.-H. Teng, Sliver exudation, J. ACM 47 (5) (2000) 883–904.
[5] S.-W. Cheng, T.K. Dey, E.A. Ramos, T. Ray, Quality meshing for polyhedra with small angles, in: SCG '04: Proceedings of the Twentieth Annual Symposium on Computational Geometry, ACM Press, 2004, pp. 290–299.
[6] S.-W. Cheng, T.K. Dey, E.A. Ramos, T. Ray, Sampling and meshing a surface with guaranteed topology and geometry, in: Proc. 20th Sympos. Comput. Geom., 2004, pp. 280–289.
[7] S.-W. Cheng, T.K. Dey, E.A. Ramos, T. Ray, Weighted Delaunay refinement for polyhedra with small angles, in: Proceedings 14th International Meshing Roundtable, IMR2005, 2005.
[8] CGAL, Computational Geometry Algorithms Library, http://www.cgal.org.
[9] L.P. Chew, Guaranteed-quality triangular meshes, Technical Report TR-89-983, Dept. Comput. Sci., Cornell Univ., Ithaca, NY, April 1989.
[10] L.P. Chew, Guaranteed-quality mesh generation for curved surfaces, in: SCG '93: Proceedings of the Ninth Annual Symposium on Computational Geometry, ACM Press, 1993, pp. 274–280.
[11] J.O. Coplien, Curiously recurring template patterns, in: S.B. Lippman (Ed.), C++ Gems, vol. 5, SIGS Publications, Inc., 1996, pp. 135–144.
[12] S.-W. Cheng, S.-H. Poon, Graded conforming Delaunay tetrahedralization with bounded radius-edge ratio, in: SODA '03: Proceedings of the Fourteenth Annual ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, 2003, pp. 295–304.
[13] X.-Y. Li, S.-H. Teng, Generating well-shaped Delaunay meshes in 3d, in: SODA '01: Proceedings of the Twelfth Annual ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, 2001, pp. 28–37.
[14] S. Oudot, L. Rineau, M. Yvinec, Meshing volumes bounded by smooth surfaces, in: Proc. 14th International Meshing Roundtable, 2005, pp. 203–219.
[15] L. Rineau, 2d conforming triangulations and meshes, in: CGAL Editorial Board (Ed.), CGAL-3.2 User and Reference Manual, 2006.
[16] J. Ruppert, A Delaunay refinement algorithm for quality 2-dimensional mesh generation, J. Algorithms 18 (1995) 548–585.
[17] L. Rineau, M. Yvinec, 3d surface mesher, in: CGAL Editorial Board (Ed.), CGAL-3.2 User and Reference Manual, 2006.
[18] J.R. Shewchuk, Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator, Appl. Comput. Geom. 1148 (1996) 203–222.
[19] J.R. Shewchuk, Tetrahedral mesh generation by Delaunay refinement, in: Proc. 14th Annu. ACM Sympos. Comput. Geom., 1998, pp. 86–95.
[20] J.R. Shewchuk, Mesh generation for domains with small angles, in: Proc. 16th Annu. ACM Sympos. Comput. Geom., 2000, pp. 1–10.
[21] J.R. Shewchuk, Delaunay refinement algorithms for triangular mesh generation, Comput. Geom. 22 (2002) 21–74.
[22] J.G. Siek, L.-Q. Lee, A. Lumsdaine, The Boost Graph Library: User Guide and Reference Manual, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2002.