Decoupling the CGAL 3D Triangulations from the Underlying Space

Manuel Caroli, Nico Kruithof, Monique Teillaud

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Decoupling the CGAL *3D Triangulations from the Underlying Space*

Manuel Caroli — Nico Kruithof — Monique Teillaud

N° 6318

Octobre 2007

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6318--FR+ENG

INRIA
SOPHIA ANTIPOLIS

Decoupling the CGAL 3D Triangulations from the Underlying Space

Manuel Caroli * , Nico Kruithof † , Monique Teillaud*

Thème SYM — Systèmes symboliques
Projets Géométrica

Rapport de recherche n° 6318 — Octobre 2007 — 12 pages

Abstract: The *Computational Geometry Algorithms Library* CGAL currently provides packages to compute triangulations in $\mathbb{R}^2$ and $\mathbb{R}^3$. In this paper we describe a new design for the 3D triangulation package that permits to easily add functionality to compute triangulations in other spaces. These design changes have been implemented, and validated on the case of the periodic space $\mathbb{T}^3$. We give a detailed description of the realized changes together with their motivation. Finally, we show benchmarks to prove that the new design does not affect the efficiency.

Key-words: Triangulation, Tetrahedrization, Torus, Periodic space

* INRIA Sophia-Antipolis (`Firstname.Lastname@sophia.inria.fr`).
† Nico Kruithof worked on this during his stay at INRIA Sophia-Antipolis (`Kruithof@jive.nl`).

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Découplage des triangulations 3D de CGAL de l'espace sous-jacent

Résumé : La bibliothèque CGAL (*Computational Geometry Algorithms Library)* fournit actuellement des modules de calcul de triangulations dans $\mathbb{R}^2$ et $\mathbb{R}^3$. Nous décrivons ici une nouvelle architecture pour le module de triangulations 3D, qui permet d'ajouter facilement les fonctionnalités nécessaires au calcul de triangulations dans d'autres espaces. Ces changements d'architecture ont été implantés, et validés sur le cas de l'espace périodique $\mathbb{T}^3$. Nous donnons une description détaillée des changements réalisés ainsi que de leur motivation. Finalement, nous présentons des mesures expérimentales prouvant que la nouvelle architecture n'affecte pas l'efficacité du code.

Mots-clés : Triangulation, Tore, Espace périodique

1 Introduction

Computing Delaunay triangulations and the more general regular triangulations is a well-studied subject in Computational Geometry. There are many algorithms available [dBvKOS00, ES96] as well as several implementations [cga, She96, Hel01, qhu]. The algorithms work in general for the d-dimensional Euclidean space $\mathbb{R}^d$, and the implementations are usually restricted to $\mathbb{R}^2$ or $\mathbb{R}^3$. However, there are also applications for triangulations in spaces other than $\mathbb{R}^d$. For instance, in simulation, one is typically interested in having no boundary conditions. This can be achieved by computing a triangulation in the periodic space denoted by $\mathbb{T}^d$. Currently, the CGAL triangulation package is generic with respect to the implementation of the basic arithmetic operations and geometric tests and the triangulation data structure (Section 3) but the embedding space is bound to be $\mathbb{R}^3$. The goal of this work is to extend the triangulation package such that it is possible to easily add functionality to compute in other spaces. For the implementation, we will restrict ourselves to the case of three-dimensional triangulations, also referred to as tetrahedrizations.

The CGAL 3D triangulation package implements two types of triangulations: *Delaunay triangulations* and *regular triangulations* [PT07b]. We shortly introduce both triangulations in the following section. The original triangulation package has been designed for computing in $\mathbb{R}^3$ only; we show how to add one more layer of genericity, which implies considerable modifications in the package design. However, we require the package to remain backward compatible and not to lose performance. The package modifications will be discussed in detail in the third section. The described changes enable us to add functionality for computing triangulations in the periodic space $\mathbb{T}^3$. In Section 4, we give more details about the properties of $\mathbb{T}^3$ and the implementation. Finally, we present some benchmarking results of the current design and the new design[1] and for different spaces in Section 5.

2 Delaunay Triangulation and Regular Triangulation

Given a set $\mathcal{S}$ of points, a triangulation partitions the space into cells (tetrahedra in 3D) whose vertices are the given points. The *Delaunay triangulation* has the property that the circumsphere of each cell does not contain any other point of $\mathcal{S}$ [BY98, dBvKOS00].

The regular triangulation generalizes the Delaunay triangulation by associating a weight with every point. Let $\mathcal{S}^{(w)}$ be a set of weighted points $p^{(w)} = (p, w_p) \in \mathbb{R}^3 \times \mathbb{R}$. The weighted point $p^{(w)}$ can also be seen as a sphere of center p and radius $\sqrt{w_p}$. To define the regular triangulation of $\mathcal{S}^{(w)}$, we first need to introduce the notion of *power product* and *power sphere*.

Definition 1

- *The* power product *of two weighted points $p^{(w)} = (p, w_p)$ and $q^{(w)} = (q, w_q)$ is defined as* $\Pi(p^{(w)}, q^{(w)}) := \|p - q\|^2 - w_p - w_q$.
- *The* power test *checks the sign of the power product of two weighted points.*
  The power product is zero if the two spheres corresponding to $p^{(w)}$ and $q^{(w)}$ intersect orthogonally; in this case, the two weighted points are said to be orthogonal.
- *Four weighted points have a unique common orthogonal weighted point called the* power sphere.

A sphere $s^{(w)}$ is said to be *regular* if the inequality $\Pi(p^{(w)}, s^{(w)}) \geq 0$ holds for all $p^{(w)} \in \mathcal{S}^{(w)}$. A triangulation of $\mathcal{S}^{(w)}$ is regular if the power sphere of each cell of the triangulation is regular.

A cell and a point $p^{(w)}$ are said to be *in conflict* if $\Pi(p^{(w)}, s) < 0$, where s is the power sphere of the tetrahedron. See Figure 1 for an illustration in 2D.

It is easy to see that if all points have equal weights, a sphere is regular if it does not contain any other point, and a sphere is in conflict with a point if it contains it; in this case, the regular

[1] We will use the term *current* to refer to the implementation available in the public release 3.3 of CGAL, and the term *new* to refer to the implementation we present in this paper, which is available only in internal releases for the moment.

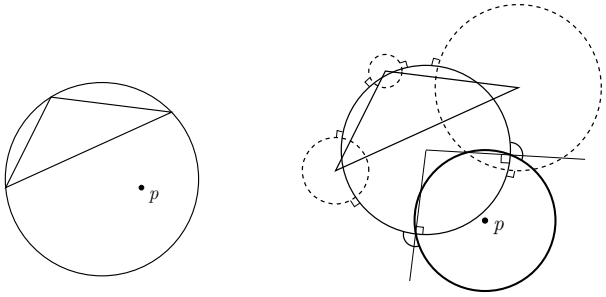

Figure 1: A cell and a point in conflict in Delaunay triangulation (top) and regular triangulation (bottom)

triangulation is the Delaunay triangulation of the set of non weighted points. The regular triangulation is also referred to as *weighted Delaunay triangulation* or *power triangulation*. To know more about regular triangulations, see for example [ES96, Aur87].

2.1 Algorithm.

The current implementation of the CGAL triangulation package uses an incremental approach to compute the triangulation. This means the points are added one by one. The algorithm for inserting a single point works as follows:

- `locate`: Locate the cell containing the point. Also report degeneracies, e.g. point on a facet or edge.
- `find_conflicts`: Mark all the cells that are in conflict with the newly added point.
- `insert`: Call `find_conflicts`, delete all cells in conflict, which creates a "hole" in the triangulation, then create new cells to fill the hole.

Additionally, there are many auxiliary functions providing access to the triangulation. Most of these functions slightly modify already computed and internally stored properties [PT07b]. Therefore, they are not considered in the further discussion.

2.2 Particularities of $\mathbb{R}^3$.

In the current CGAL triangulation package, several aspects are specialized to $\mathbb{R}^3$:

A triangulation in $\mathbb{R}^3$ partitions the convex hull[2] of the point set. To handle the unbounded cell outside the convex hull, an infinite vertex is added to the triangulation, and all the facets of the convex hull are connected with this infinite vertex. In this way, all cells have four vertices and four adjacent cells. The underlying combinatorial graph of the triangulation of $\mathbb{R}^3$ can be seen as a triangulation of the topological sphere $\mathbb{S}^3$ in $\mathbb{R}^4$.

Furthermore, we have to deal with degenerate dimensions when computing triangulations in $\mathbb{R}^3$. For instance, if all the points lie in a plane, a two-dimensional triangulation must be computed and stored. This requires additional implementation effort for some functions.

We want to allow implementations of triangulations in different spaces in the new triangulation package. One of them is homeomorphic to the hypersurface $\mathbb{T}^3$ of a torus in 4D (see Section 4 for more details). The previously mentioned properties of $\mathbb{R}^3$, currently hard-coded, do not hold in $\mathbb{T}^3$.

3 Design

In this section, we first describe the design of the current triangulation package. Then we introduce the changes that have to be done in order to enable triangulations in different spaces.

[2] The convex hull of a set of points is the smallest convex set containing the points.

Like the overall CGAL library, the triangulation package follows the Generic Programming paradigm [BKSV00, FT06]. The policy of CGAL is to provide code that is generic, flexible, and easily adaptable to specific needs of the user. The main class `Triangulation_3` is built as a layer on top of the triangulation data structure that stores the combinatorial structure of the computed triangulation. This allows a separation between the geometry and the combinatorics, which is reflected by the fact that the triangulation class takes two template parameters:

- the **geometric traits** class, which defines basic geometric objects (e.g. points, segments, triangles, tetrahedra) and predicates (e.g. orientation test, in_sphere test for Delaunay triangulation and power test for regular triangulation). A default traits class is provided in the package, but it can also be substituted by user provided traits.
- the **triangulation data structure** class, which stores the combinatorial structure and takes care of its validity [PT07a]. It is described in more detail in the following section.

3.1 The Triangulation Data Structure.

To explain a data structure for storing triangulations, we need a more precise definition of a triangulation. The notion of simplicial complex must be recalled first. More details can be found in [Zom05, RV06].

Definition 2

- *A* k-simplex *is the convex hull of a set of* $k+1$ *affinely independent points. A 0-simplex is called a* vertex.
- *If* σ *is a simplex defined by a finite point set* $\mathcal{S}$*, then any simplex* τ *defined by* $\mathcal{T} \subset \mathcal{S}$ *is called a* face *of* σ.
- *A* simplicial complex K *is a finite collection of non-empty simplices such that the following two conditions hold:*
  1. *if* $\sigma \in K$ *and* τ *is a face of* σ*, then* $\tau \in K$,
  2. *if* $\sigma_1, \sigma_2 \in K$*, then their intersection* $\sigma_1 \cap \sigma_2$ *is either empty or a face of both* σ_1 *and* σ_2.

Now we can define a triangulation as follows:

Definition 3 *Let* $\mathcal{S}$ *be a finite point set in some space* $\mathbb{X}$*. A simplicial complex* K *is a* triangulation *of* $\mathcal{S}$*, if*

1. *each point in* $\mathcal{S}$ *is a vertex of* K,
2. $\bigcup_{\sigma \in K} \sigma$ *is homeomorphic to* $\mathbb{X}$.

The triangulation data structure stores a purely combinatoric graph without any geometric information. It consists of a container of cells (3-faces) and a container of vertices (0-faces).

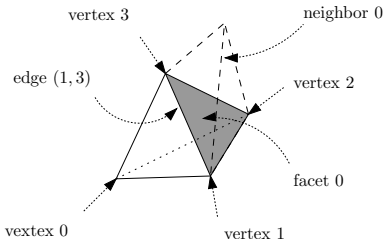

Figure 2: Representation.

Each cell stores pointers to its four vertices and its four adjacent cells. Each vertex stores one of its incident cells (Figure 2).

In order to achieve a high flexibility of design, the classes that store the cells and vertices can be specialized or even completely replaced by the user.

3.2 The Triangulation Class and its Specializations.

The main class `Triangulation_3` is specialized to `Delaunay_triangulation_3` and `Regular_triangulation_3` as shown in the derivation diagram in Figure 3.

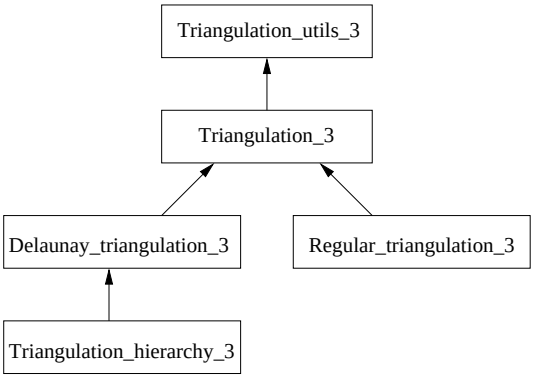

Figure 3: The current design.

These three classes provide high-level geometric functionality as member functions, such as location of a point in the triangulation [DPT02], point insertion, and vertex removal [DT03, DT06], and are responsible for the geometric validity. As mentioned before, they are parameterized by the geometric traits and by the triangulation data structure. This diagram shows two other classes:
- `Triangulation_utils_3` provides a set of tools operating on the indices of vertices in cells,
- `Triangulation_hierarchy_3` implements a hierarchy of triangulations suitable for speeding up point location [Dev02].

The geometric functions in the triangulation class then trigger the combinatorial functions in `Triangulation_data_structure_3`. For instance, `insert` first performs the point location and then computes the conflict region (Section 2.1). These tasks involve only geometric predicates and do not change the combinatorial triangulation. Once the conflict region is known, its cells are deleted and replaced by new cells, which is a purely combinatorial operation performed by `Triangulation_data_structure_3`.

3.3 The New Design.

The goal of the new design is to make it possible to easily extend the current implementation by several different spaces with the least possible redundancy in code. The basic idea of the new

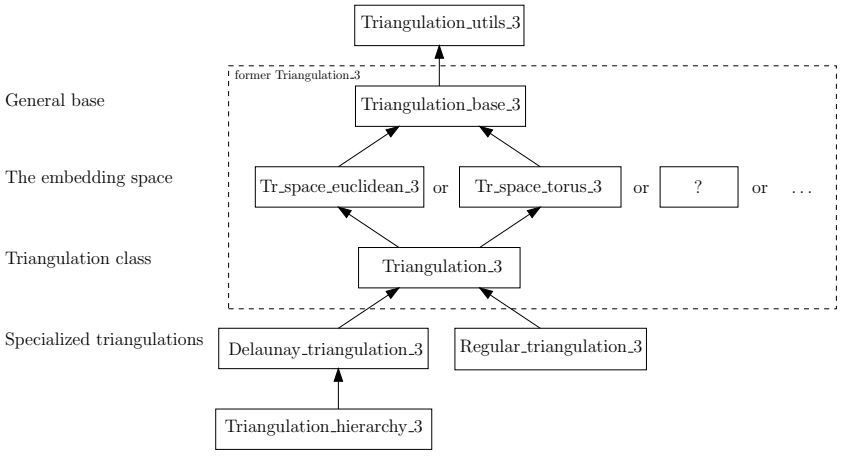

Figure 4: The new design.

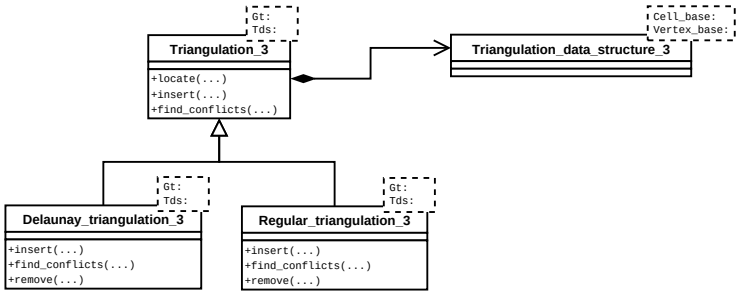

Figure 5: Distribution of functions in the current package (simplified).

design is to split the current class `Triangulation_3` into three classes related by inheritance (cf. Figure 4). The embedding space class provides all functionality that depends on the space. It is now a template parameter of the triangulation, together with the triangulation data structure and the geometric traits.

More details are given in Section 3.4. Let us now list the affected classes in more detail and emphasize on the changes to the current design:

Triangulation data structure: The triangulation data structure does not change in the new design, since the invariant of storing a simplicial complex does not depend on the space.

Geometric traits: The geometric traits class can be substituted by an appropriate traits class if needed by the embedding space, in the same way as it can be modified by the user in the current design.

Triangulation embedding space: This class is new in the design and is used to handle everything that depends on the embedding space of the triangulation.
There is functionality that depends on both space and triangulation type. In these cases we use visitors that modify the functionality relevant to the triangulation type in the space class. This design pattern is described in more detail in Section 3.5.

Triangulation: The Triangulation class provides the same interface as in the current design, independent of the embedding space. It provides generic algorithms for point location, point insertions and flips. This class is finally specialized to Delaunay triangulation and regular triangulation.

Introducing new spaces becomes easy with this design: only the space class (and possibly the geometric traits) is needed, since the algorithms provided by the triangulation classes are fully generic.

3.4 Redistributing the Functionality of `Triangulation_3`.

In the current triangulation package, the functionality is distributed as shown in Figure 5.

In the Delaunay triangulation as well as in the regular triangulation, the `find_conflicts` function and the `insert` function overload the corresponding functions of `Triangulation_3`. Additionally, both specialized triangulation classes implement their own `remove` method: it removes the given vertex from the data structure and uses `insert` to retriangulate the hole.

In the new design, the functionality is redistributed as shown in Figure 6. All of the four functions mentioned in the figures depend on the space used. That means that we must provide a different implementation for each space. Therefore, the class `Triangulation_base_3` provides only basic functionality that is independent of space and triangulation type used. The space classes contain the actual geometric functionality: point location, conflict search, point insertion, and point removal. Out of these four functions, only the point location does not depend on the triangulation type. This means that we need to implement each of the remaining three functions twice for each space (once for Delaunay and once for regular triangulation). To minimize code redundancy, we use the design pattern of visitors as explained in the next section.

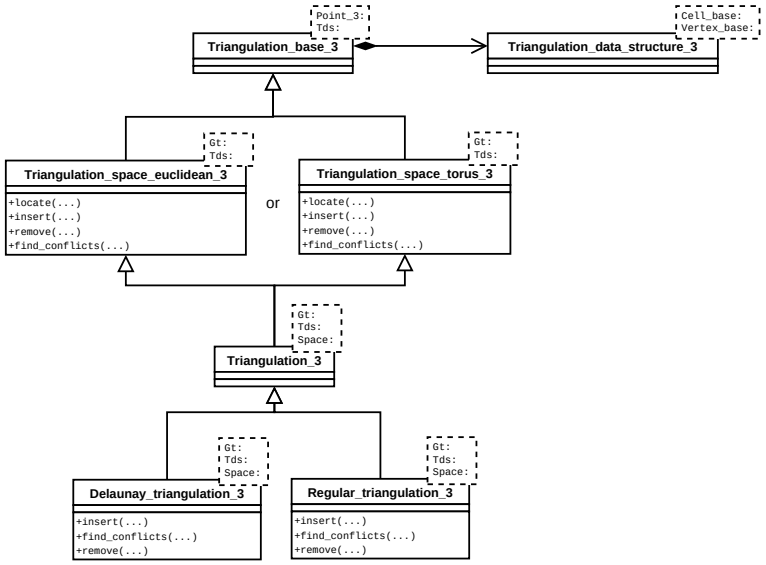

Figure 6: Distribution of functions in the new design (simplified).

3.5 Visitors.

In a class hierarchy, it is usually hard to extend subclasses with new functions, because the new functions must be added to *each* subclass. The idea of the visitor design pattern [GHJV95] is to move functions operating on objects from the class hierarchy into their own classes, called *visitors*. In this way, functions can be reused for several types of objects, and adding a new function corresponds to only adding a new visitor.

```
class Conflict_tester {
  typename Conflict_vertices_iterator;
  typename Hidden_points_iterator;

  // Constructor: the visitor is initialized with the point to test
  Conflict_tester(Point, Embedding_space<Traits, Tds>)

  // returns true if the point is in conflict with the cell
  bool operator()(Cell_handle);

  // access functions
  Conflict_vertices_iterator conflict_vertices_begin();
  Conflict_vertices_iterator conflict_vertices_end();
  Hidden_points_iterator hidden_points_begin();
  Hidden_points_iterator hidden_points_end();
};

template <class ConflictTester, class Iter1, class Iter2, class Iter3>
Triple<Iter1,Iter2,Iter3> find_conflicts(
    Cell_handle,
    ConflictTester,
    Triple<Iter1,Iter2,Iter3>);
```

Figure 7: Pseudo-code listing to illustrate the visitor `ConflictTester`.

In our implementation we modify the presented ideas of the visitor pattern to meet the generic programming paradigm. Visitors have been used in CGAL before, e.g. to permit the user to provide

his own functions that must be applied during the algorithm run [WFZH07]. In our approach, we use visitors to exchange triangulation type dependent functionality in the functions from the space class.
The naive solution to our problem would be to implement four different classes for

- Delaunay triangulation in $\mathbb{R}^3$,
- Regular triangulation in $\mathbb{R}^3$,
- Delaunay triangulation in $\mathbb{T}^3$,
- Regular triangulation in $\mathbb{T}^3$.

This solution would require another pair of triangulation classes for each further space. Instead, we implement two space classes providing functionality for both triangulation types. The space classes correspond to the object in the above description of the visitor pattern. Now, we factor out the parts of the code that have to be implemented differently depending on the triangulation type. This functionality is coded in separate visitor classes.

An example for a visitor class is the conflict tester (Figure 7). The `find_conflicts` function needs to be able to decide whether or not a cell and a point are in conflict. In the Delaunay triangulation, this corresponds to an `in_sphere` test whereas in the regular triangulation, it corresponds to a `power_test` (cf. Section 2). Therefore, the `find_conflicts` function in the space class is templatized by a class of the model `Conflict_tester` and receives an object of this class. We only need to implement a class `Delaunay_conflict_tester` and a class `Regular_conflict_tester` instead of implementing the whole `find_conflicts` function twice.

Further visitors are needed:

- `[Delaunay|Regular]_point_hider`: In a regular triangulation it may happen that the insertion of a heavy-weighted point hides lighter-weighted points nearby. The hidden points disappear from the triangulation. But we need to store them in some data structure to reinsert them again in the case that the heavy-weight point that caused their disappearance is removed. The task of the point hider is to manage this data structure. The `Delaunay_point_hider` exists only for consistency reasons and does nothing.
- `[Delaunay|Regular]_point_remover`: The point remover extracts the hidden points from removed cells. Those vertices can be read by the remove function to be reinserted to the triangulation. This visitor is also empty for the Delaunay case.

4 The Periodic Space

In this section, we show how the design presented previously can be used for computing triangulations in the periodic space $\mathbb{T}^3$. We only sketch the basic ideas here. More details about this specific space will be developed in a forthcoming paper.

The periodic space $\mathbb{T}^3$ is represented here as $[0,1]^3$. It can be embedded in $\mathbb{R}^3$ using a rectangular tiling. We denote points in $\mathbb{R}^3$ and their coordinates by $p = (x, y, z)$ and points in $\mathbb{T}^3$ by $q = (u, v, w)$. Any point $q = (u, v, w) \in \mathbb{T}^3$ is mapped onto a regular point lattice in $\mathbb{R}^3$ given by:

$$g(q) := \{(u, v, w) + (i, j, k) \mid i, j, k \in \mathbb{Z}\},$$

In this definition, we map the periodic space into $\mathbb{R}^3$ by repeating it infinitely in all three directions of space.

Definition 4 (domain) *The* domain (i, j, k) *is defined as the set of points in* $\mathbb{R}^3$ *with the same constant* (i, j, k) *in the above defined mapping* g*.*

As mentioned above, a triangulation is defined as a simplicial complex (cf. Section 3.1). As long as the subdivision we compute is not a simplicial complex, it is not a valid triangulation. As an example, let us subdivide $\mathbb{T}^2$ using one vertex such that the subdivision meets the Delaunay property when we embed it into $\mathbb{R}^2$. It consists of 3 edges and 2 facets (see Figure 8). This subdivision is not a simplicial complex because the vertices of all simplices are equal. The corresponding subdivision of $\mathbb{T}^3$ with one vertex has 7 edges, 12 facets and 6 cells. This subdivision cannot be used as a starting point for an incremental construction of a Delaunay triangulation.

Coverings. We propose to construct the triangulation of the torus in two stages. First, we construct a Delaunay triangulation of a 3-sheeted covering of the underlying space [Zom05]. In

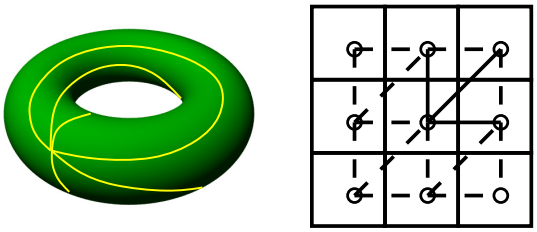

Figure 8: Left: invalid triangulation of $\mathbb{T}^2$ with a single vertex. Right: 9 periodic copies are shown.

other words, the domain is explicitly copied 3 times in each direction of space (see Figure 8 for $\mathbb{T}^2$). It can be shown that the subdivision computed as described in Sections 2.1 and 3.4 is always a valid triangulation in such a covering. However, for the three-dimensional space we now have to cope with 27 times as many points as the triangulation contains. It is clear that this becomes very inefficient when computing triangulations of large point sets. Once the triangulation contains enough points, it is likely that it is a simplicial complex in the 1-sheeted covering as well. We could prove that after checking some precise criteria, we can fall back to the 1-sheeted covering.

Offsets. As mentioned above, the triangulation is stored as a set of vertices and a set of cells, where a cell contains pointers to its four neighboring cells and to its four incident vertices. In a periodic space, we also need to store whether or not a cell is wrapped around the domain. To do so, we introduce offsets: Each vertex of a cell is endowed with an offset, which is a three-dimensional vector of non-negative[3] integer entries. To get the vertex coordinates of a cell, we take the point coordinates attached to the respective vertex and add the offset multiplied by the domain size (cf. Figure 9). To use the offsets in the implementation we specialize the cell class in the triangulation data structure. Additionally, the space class and the respective visitors need to be modified to handle offsets.

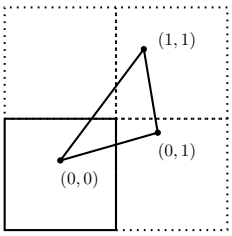

Figure 9: The concept of offsets in 2D space.

5 Benchmarks

The main constraint on our work is that the implementation should not lose efficiency after the redesign of the triangulation package. To ensure this, we tested the performance of both current and new implementations for Delaunay triangulations as well as regular triangulations in $\mathbb{R}^3$.

All benchmarks have been run with the CGAL internal release CGAL-3.4-I-85 on an Intel Pentium 4 CPU clocked at 3.6 GHz. The used operating system is Linux Fedora Core 5 and `gcc` version 4.1.1 with the optimization option `-O2`. The given results are obtained by using `CGAL::Timer` and computing the average of the run time of three runs rounded to three significant digits. All computations have been performed on a random point set, uniformly distributed in a half-open unit cube. The results are given in seconds.

[3] We can always move the reference coordinate system such that no negative entries are needed.

No. of points	current design	new design
1000	0.0190	0.0190
10000	0.204	0.205
100000	2.11	2.10
1000000	21.5	21.4

No. of points	current design	new design
1000	0.0880	0.0907
10000	1.00	1.01
100000	10.3	10.5
1000000	104	106

Table 1: Benchmarks for Delaunay triangulation (left) and regular triangulation (right).

Table 1 allows us to compare the two designs. As we see, no significant loss in efficiency is identifiable. The point insertion uses spatial sorting[4] [Del07], which accelerates the point location considerably. We can observe that the asymptotic behavior is almost linear.

The last benchmark shows a preliminary comparison of the computation of triangulations in $\mathbb{R}^3$ and in $\mathbb{T}^3$, made possible by the new design. To be able to compare the two triangulations, we precompute a triangulation of the first 1000 points. This is enough for the torus triangulation to switch back to the 1-sheeted covering (cf. Section 4). It does not make much sense to compare with computing in the 3-sheeted covering. Table 2 shows that with the first version of the space class for $\mathbb{T}^3$, the computation is clearly slower than in $\mathbb{R}^3$, but the difference will be reduced after improvements in the implementation.

No. of points	$\mathbb{R}^3$	$\mathbb{T}^3$
1000	0.0190	0.0500
10000	0.204	0.498
100000	2.11	4.93
1000000	21.5	49.5

Table 2: Benchmarks for Delaunay triangulation in $\mathbb{R}^3$ and $\mathbb{T}^3$.

References

[Aur87] F. Aurenhammer. Power diagrams: properties, algorithms and applications. *SIAM J. Comput.*, 16:78–96, 1987.

[BKSV00] Hervé Brönnimann, Lutz Kettner, Stefan Schirra, and Remco Veltkamp. *Applications of the Generic Programming Paradigm in the Design of* CGAL, volume 1766 of *Lecture Notes in Computer Science*, pages 206–216. Springer, Berlin, Germany, January 2000.

[BY98] Jean-Daniel Boissonnat and Mariette Yvinec. *Algorithmic Geometry*. Cambridge University Press, UK, 1998. Translated by Hervé Brönnimann.

[cga] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[dBvKOS00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

[Del07] Christophe Delage. Spatial sorting. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. 3.3 edition, 2007.

[Dev02] Olivier Devillers. The Delaunay hierarchy. *Internat. J. Found. Comput. Sci.*, 13:163–180, 2002.

[DPT02] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulation. *Internat. J. Found. Comput. Sci.*, 13:181–199, 2002.

[4] The idea is to sort the points so that geometrically close points will be close in the insertion order with high probability.

[DT03] Olivier Devillers and Monique Teillaud. Perturbations and vertex removal in a 3D Delaunay triangulation. In *Proc. 14th ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 313–319, 2003.

[DT06] Olivier Devillers and Monique Teillaud. Perturbations and vertex removal in delaunay and regular 3d triangulations. Research Report 5968, INRIA, Sophia-Antipolis, 2006.

[ES96] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15:223–241, 1996.

[FT06] Efi Fogel and Monique Teillaud. Generic programming and the CGAL library. In Jean-Daniel Boissonnat and Monique Teillaud, editors, *Effective Computational Geometry for Curves and Surfaces*. Springer-Verlag, Mathematics and Visualization, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[Hel01] M. Held. Vroni: An engineering approach to the reliable and efficient computation of Voronoi diagrams of points and line segments. *Comput. Geom. Theory Appl.*, 18:95–123, 2001.

[PT07a] Sylvain Pion and Monique Teillaud. 3d triangulation data structure. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. 3.3 edition, 2007.

[PT07b] Sylvain Pion and Monique Teillaud. 3d triangulations. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. 3.3 edition, 2007.

[qhu] Qhull. http://www.qhull.org.

[RV06] Günter Rote and Gert Vegter. Computational topology: An introduction. In Jean-Daniel Boissonnat and Monique Teillaud, editors, *Effective Computational Geometry for Curves and Surfaces*. Springer-Verlag, Mathematics and Visualization, 2006.

[She96] J. R. Shewchuk. Triangle: Engineering a 2d quality mesh generator and Delaunay triangulator. In *First Workshop on Applied Computational Geometry*. Association for Computing Machinery, May 1996.

[WFZH07] Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin. Advanced programming techniques applied to CGAL's arrangement package. *Computational Geometry: Theory and Applications*, 38:37–63, 2007. Special issue on CGAL.

[Zom05] Afra Zomorodian. *Topology for Computing*. Cambridge University Press, Cambridge, 2005.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399