Triangulations in CGAL

Jean-Daniel Boissonnat, Olivier Devillers, Sylvain Pion, Monique Teillaud, Mariette Yvinec

published in Computational Geometry: Theory & Applications, 22:5-19, 2002. Special issue SoCG00

Triangulations in CGAL *

Jean-Daniel Boissonnat Olivier Devillers Sylvain Pion
Monique Teillaud Mariette Yvinec†

Abstract

This paper presents the main algorithmic and design choices that have been made to implement triangulations in the computational geometry algorithms library CGAL.

Keywords: Triangulation, Delaunay triangulation, implementation

1 Introduction

CGAL (Computational Geometry Algorithms Library) is a C++ library of geometric algorithms which is developed by a consortium of eight research teams in the framework of a European project. Geometric algorithms are known to be hard to code because they are highly sensitive to numerical rounding errors and also because the programmer has to take care of numerous degenerate cases. The goal of CGAL is to provide robust, efficient, flexible and easy to use implementations of geometric algorithms and data structures. This effort has been undertaken in order to make available to companies and application areas the algorithmic solutions developed in the field of computational geometry.

Triangulations are well known and ubiquitous geometric data structures which are used in numerous application areas like GIS, robotics, geometric modeling and meshing for numerical solution of partial differential equations. See eg. [6] or [4] for a survey on triangulations. This paper is intended to present the choices that have been made in the design of the CGAL triangulation package. These choices have been mainly guided by the general goals of CGAL which are robustness, efficiency, ease of use and flexibility. Some features in the design of CGAL triangulations arise from general decisions made for the whole library. For instance, every class in the basic library has two template parameters providing the geometric traits and the data structure implementation [21, 42]. Other features, such as the three layers design of CGAL triangulations (which is described

*This research was partially supported by the ESPRIT IV LTR Project No. 21957 (CGAL) and by the ESPRIT IV LTR Project No 28155 (GALIA)

†INRIA, BP93, 06902 Sophia Antipolis, France. Firstname.Lastname@sophia.inria.fr

in Section 4), are reminiscent of other CGAL packages [28, 22]. The choices concerning the data structure that represents the triangulations and the algorithms used to build them and answer queries are specific to the triangulation package.

In Section 2, called specifications, we describe the different kinds of triangulations implemented in CGAL and the main functionalities offered by these triangulations. We also state a set of requirements, concerning mainly robustness and flexibility issues, that we had in mind when designing the triangulation package. Section 3 describes the way triangulations are represented in CGAL. The representation we have chosen for two-dimensional triangulations is not a variant of the well known doubly connected edge list (*dcel*, for short). It is a representation that is specific for triangulations and has the advantage to be far less demanding with respect to memory space and to generalize nicely to higher dimensional triangulations. Section 4 is concerned with software engineering issues and shows how the design of the triangulation package provides a clear distinction between the combinatorial aspects of the triangulation and the geometric embedding. This distinction and the fact that, like everywhere in CGAL, the user can choose the arithmetic used to evaluate geometric predicates is a clear step towards robustness. The design of CGAL triangulations is also the fundamental tool through which flexibility is achieved. For example, this design allows the user to use his/her own point rather than those of CGAL kernel and to add his/her own information in the faces, edges or vertices of the triangulation. Section 5 gives some details about the algorithms used to build triangulations and answer location queries. This section also describes a checker for triangulations provided in the package. Section 6 is devoted to practical measurements of efficiency and presents some experimental results. These benches were intended to compare the Delaunay triangulations of CGAL with other equivalent software available on the web, and also to evaluate the cost of using exact arithmetic or exact filtered versions of the predicates. At last, we conclude with a few comments about applications.

2 Specifications

2.1 Triangulations

CGAL offers two- and three-dimensional triangulations. The two-dimensional triangulations are primarily intended to represent triangulations of a set of points in the plane, while three-dimensional triangulations represent tetrahedralizations of point sets in three-dimensional space. This calls for two remarks.

First, one of the data structures available to represent two-dimensional in CGAL to represent triangulations can be used to represent triangulated surfaces embedded in three-dimensional space and to handle them at the combinatorial level. However, except in the important special case of terrains, no functionality is provided to handle the geometric embedding of such a triangulation which thus remains under the responsibility of the user.

Second, at the geometric level, a triangulation in CGAL is a simplicial com-

plex whose domain covers the convex hull of its vertices. This does not imply that CGAL triangulations cannot be used to deal with triangulations of bounded polygonal regions. Indeed, CGAL offers constrained triangulations and constrained Delaunay triangulations which are intended to this purpose. In the case of triangulated polygonal regions, CGAL represents a simplicial complex whose domain is not restricted to the interior of regions but covers their entire convex hull. This feature is essential to have efficient point location algorithms and navigating tools.

In the plane, CGAL offers basic, Delaunay and regular triangulations, as well as constrained triangulations and constrained Delaunay triangulations. A short description of these triangulations follows. Precise definitions can be found in [6, 1, 4].

Basic triangulations, also called lazy triangulations, are triangulations built incrementally without any control on the resulting shape of the triangular faces. If a new point lies in the interior of an existing triangle, the insertion splits this triangle into three new triangles. If the new point lies on an edge, the two adjacent triangles are split, each in two new triangles. If the new point lies outside the current convex hull, the convex hull is updated and the region between the new and the previous convex hull is triangulated by linking the new point to the edges that disappear from the convex hull boundary.

Delaunay triangulations are well known to be dual of Voronoi diagrams and to fulfill the so called *empty circle property* stating that the circumcircle of any face in the triangulation encloses no vertex.

Regular triangulations are generalized Delaunay triangulations. A regular triangulation is defined for a set of weighted points. Each weighted point can be considered as a sphere whose square radius is equal to the weight. Then the regular triangulation of a set of weighted points can be defined as the dual the power diagram of the associated spheres. Any regular triangulation of dimension d is the projection of the lower envelope of a polytope of dimension $d + 1$.

Constrained triangulations allow the user to enforce some edges in the triangulation. The enforced edges are part of the input. The vertices of the triangulations are the endpoints of enforced edges. Constrained triangulations are used e.g. to triangulate polygonal regions. In such case, the enforced edges are the edges of the polygons describing the boundary of the region.

Constrained Delaunay triangulations are constrained triangulations in which any triangle satisfies the *constrained empty circle property* saying that its circumscribing circle encloses no vertex visible from the interior of the triangle, where enforced edges are considered as visibility obstacles.

There is no need to advocate about the usefulness of Delaunay triangulations. Regular triangulations turn out to be very useful in shape reconstruction

when dealing with non-uniform samples and also in some meshing problems. Constrained Delaunay triangulations are the main tool used in meshing problems.

CGAL offers three-dimensional basic, Delaunay and regular triangulations. Three-dimensional constrained triangulations and constrained Delaunay triangulations are not straightforward extensions of their two-dimensional counterparts since, in three dimension, it is no longer true that every set of non intersecting constraints can be included in the set of faces of a triangulation. Recently some interesting condition guaranteeing the existence of constrained Delaunay triangulations has been reported (e.g. [41]). Another approach is to conform the input constraints, adding Steiner points so that each constraint is included in the Delaunay triangulation of the augmented set of vertices, see e.g [20] or [34]. Currently there is no three-dimensional constrained triangulation nor constrained Delaunay triangulation in CGAL. We plan to have three dimensional conforming and constrained triangulations available in future versions of the package.

2.2 Functionalities of Triangulations

The two-dimensional triangulation package in CGAL mainly provides point location, on-line insertion of new vertices, and deletion. There are also several functions to visit all or a subset of the faces, edges and vertices of the triangulation. For instance, CGAL provides iterators to visit all the faces of the triangulations, and circulators[1] to visit all the faces intersected by a line or all the faces incident to a given vertex.

There are also functionalities related to each special kind of triangulation. Delaunay triangulations allow to answer nearest neighbor queries asking for the vertex closest to a given query point. Constrained triangulations and constrained Delaunay triangulations support insertions and removals of constraints.

The three-dimensional triangulation package provides point location and on-line insertion. Deletions of vertices will be offered for Delaunay and regular triangulations. Deletions will not be offered for the basic triangulation. Indeed, it may happen that, for a given vertex, the region formed by its adjacent tetrahedra forms an instance of the famous Schönhardt's untetrahedralizable polyhedron [37]. Then, filling the hole created by the removal of this vertex is impossible. Deleting and rebuilding the whole triangulation might be the only way to update it. The three-dimensional triangulation also provides iterators to visit all the cells, facets, edges or vertices as well as circulators to visit the facets or cells incident to a given edge. Access is also provided to the subset of cells, facets or edges incident to a given vertex.

2.3 Requirements for the design

Robustness is one of the major goals of the CGAL library. Triangulation al-

[1] Circulators are a CGAL extension of the standard iterators of the STL, specially designed for circular sequences.

gorithms take as input a set of points and compute a purely combinatorial structure. The computed structure depends on the result of predicates which are numerical tests depending on the point coordinates. Robustness issues rely on the implementation of those predicates. In order to gain robustness, the design of CGAL allows a clear separation between combinatorial operations and predicate evaluations. As a result, the user can choose between different arithmetics to compute the predicates and can also easily change from one arithmetic to another. CGAL provides interface with different exact arithmetic packages and also provides number types for efficient exact evaluation of the predicates with filtering [35].

Flexibility. Triangulations in CGAL should be used in different contexts and for various purposes. Thus, flexibility was another major goal of the design.

First, triangulation algorithms are supposed to work not only with the points provided by the CGAL kernel but also with user defined points. For example, such a flexibility is useful in GIS to build a triangulated model of a terrain from a set of three-dimensional measured points. This is usually done following a three steps procedure: project the three-dimensional data points on the xy plane, compute the Delaunay triangulation of those projections, and lift up this triangulation by mapping its vertices back to the three-dimensional input points. CGAL allows the GIS user to compute triangulated models of terrains directly without explicitly projecting the data points and mapping back the two-dimensional projected points to the three-dimensional original points.

Then, the user can plug in a triangulation algorithm his own version of the predicates. For instance, he can compute a Delaunay triangulation algorithm for various distance functions. It is also possible and quite easy to attach additional information (like a color, a scalar value, a normal vector or anything else) to the different features (vertices, edges, facets or cells) of a triangulation. At last, the user can consider the triangulation in CGAL as a tool kit to experiment with his own triangulation algorithms.

3 Representation

The set of faces. CGAL triangulations are always assumed to cover the whole convex hull of their vertices. Therefore, regarded as a partition of the whole plane, a two-dimensional triangulation appears as a collection of triangular faces plus a single unbounded face which is the convex hull complement. As it is not very convenient to deal with this face of unbounded complexity, we have decided to add to any triangulation a fictitious vertex called the *infinite vertex*, with the convention that every edge of the convex hull forms an infinite face with this vertex. Thus any edge of a two-dimensional triangulation is incident to exactly two faces and the set of vertices, edges and faces of the triangulation is combinatorially equivalent to a triangulated two-dimensional sphere, see Figure 1. The infinite vertex has no coordinates, and no geometric predicate can be evaluated on this vertex. Likewise, the infinite faces do not form a partition of the complement of the convex hull.

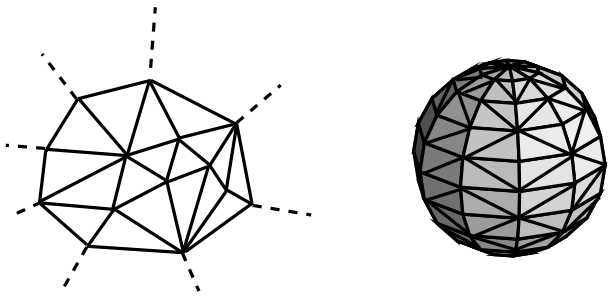

Figure 1: The set of faces of a two-dimensional triangulation.

Of course, the concept of *infinite vertex* applies as well to three-dimensional triangulations (where it forms infinite cells with the facets of the convex hull) and also to degenerate lower dimensional triangulations. The representation of triangulations in CGAL ensures that, whatever the actual dimension d of the triangulation may be, the set of represented faces is combinatorially equivalent to a d-dimensional triangulated sphere. Thus, the data structure handles in a unified way the degenerate cases occurring in the early steps of an incremental building or when vertices are removed.

A representation based on faces (cells in 3d) and vertices. A two-dimensional triangulation can be considered as a planar map and could be represented through one of the variants of the famous *dcel* data structure [10, pp. 31-33]. However, we have decided to use a representation based on vertices and faces rather than on edges. This leads to smaller space complexity and extends nicely in any dimension.

More precisely, a two-dimensional triangulation is represented as a collection of vertices and faces. Each face provides access to its three vertices and to its three neighboring faces in clockwise or counterclockwise order. Each face is equipped with three pointers to its three vertices and with three pointers to the three adjacent faces. These pointers are indexed by 0, 1 and 2 in counterclockwise order in such a way that, in each face, the vertex indexed by i is opposite to the adjacent face with the same index (see Figure 2). Each vertex provides access to one of its incident faces (from which any other incident face can then be accessed). The edges are only implicitly represented through the adjacency relations of faces. If information has to be attached to edges, it must be attached to each incident face. For instance, in constrained triangulations, the status (constrained or not constrained) of an edge is stored in both incident faces.

In a *dcel* data structure, the main object is the half-edge. An edge is represented by two half-edges, which are basically oriented edges. An half-edge has four pointers referring to
— the half-edge corresponding to the same edge with the opposite orientation

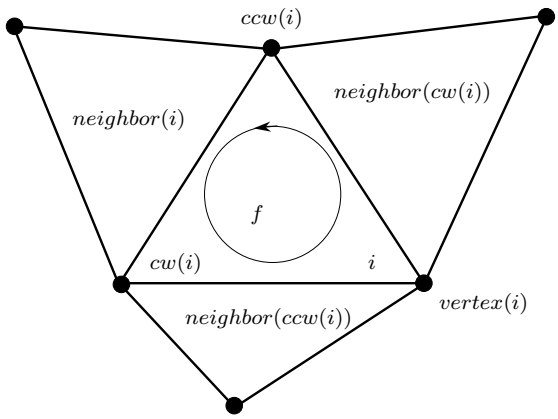

Figure 2: Representation of two-dimensional triangulations in CGAL

(twin),
— the triangle on its left (face),
— the next half-edge, in counterclockwise order, in the triangle on its left (next),
— and the vertex at the end of the oriented edge.
Faces and vertices have pointers to an incident half-edge. Edges are represented implicitly by two half-edges, and attaching information must be done twice as in CGAL. Some space can be saved by using an implicit representation for faces, but in that case, we obtained a less powerful structure.

A triangulation with n vertices (including the infinite vertex) has $2n-4$ faces and $3n-6$ edges, thus it requires $13n$ pointers to represent it in CGAL while a *dcel* data structure would require $27n$ pointers as shown in the following table.

	dcel	CGAL
Vertex	n	n
Edges	$4 \times 2 \times (3n-6)$	
Faces	$(2n-4)$	$6 \times (2n-4)$
Total	$27n-52$	$13n-4$

In the same way, the representation of three-dimensional triangulations in CGAL is based on the tetrahedral cells and vertices while the edges and facets of the triangulation are only implicitly represented. In such a tetrahedralization, each cell has pointers to its 4 vertices and to its 4 adjacent cells and each vertex has a pointer to one incident cell. The *dcel* extends to three dimensions, leading to a compact variant of 3-maps [5, 9, 7, 30]. In this structure, a tetrahedron is represented by a *dcel* with 12 half-edges; two neighboring tetrahedra are linked together by putting a pointer in each half-edge to its sibling in the other tetrahedron; thus the total number of pointers in each half-edge is 5, the four 4 pointers of the usual *dcel* plus this additional pointer. The total memory

requirements are described in the following table, where t is the number of tetrahedra and n the number of vertices.

	3-map	CGAL
Vertex	n	n
Edges	$12 \times 5 \times t$	
Faces		
Cells	t	$8t$
Total	$61t + n$	$8t + n$

4 Software design

4.1 Template parameters

The triangulation classes of CGAL are parameterized by two template parameters that provide a clear separation between the combinatorial aspects and the geometry of the triangulations. The set of requirements that has to be fulfilled by any class used to instantiate the first parameter defines the concept [2] of geometric traits class. The set of requirements to be fulfilled by any class used to instantiate the second parameter defines the concept of triangulation data structure. Both concepts are fully described in CGAL documentation.

The geometric traits class is assumed to provide the geometric objects (points, segments, triangles, tetrahedra etc..) the triangulation has to deal with and the geometric predicates on those objects. Essentially, triangulation algorithms used in CGAL rely on coordinate comparisons, the *orientation test* and the *in-circle* or *in-sphere* test for Delaunay triangulations (or the equivalent *power tests* for regular triangulations). In dimension two, the orientation test takes as input three points p, q, r and decides on which side of the oriented line pq, point r lies. The in-circle test takes as input four points p, q, r, s and decides on which side of the circle through p, q, r point s lies.

The geometric traits class parameter plays a great part in robustness. Indeed, numerical computations in the triangulation package are exclusively located in the implementation of the geometric predicates. The robustness of the triangulation algorithms will rely on the choice of the arithmetic used to evaluate those predicates. Objects of the CGAL kernel are parameterized by a number type that decides which arithmetic is used. CGAL offers different kinds of arithmetic allowing for exact evaluation of predicates or filtered-exact evaluation to combine robustness and efficiency.

The geometric traits class parameter also contributes to flexibility. Indeed, because the concept is documented, the user can plug his own model of geometric traits class into a triangulation. Thus the user can obtain a triangulation class using his own points and his own implementation of the predicates for those points. For instance, to build a triangulated terrain, the GIS user will

[2] Here and in the following, the term concept refers to the meaning it has in the C++ standard, i.e. it is the set of requirements to be fulfilled by any class (called a model of the concept) used to instantiate a given template parameter.

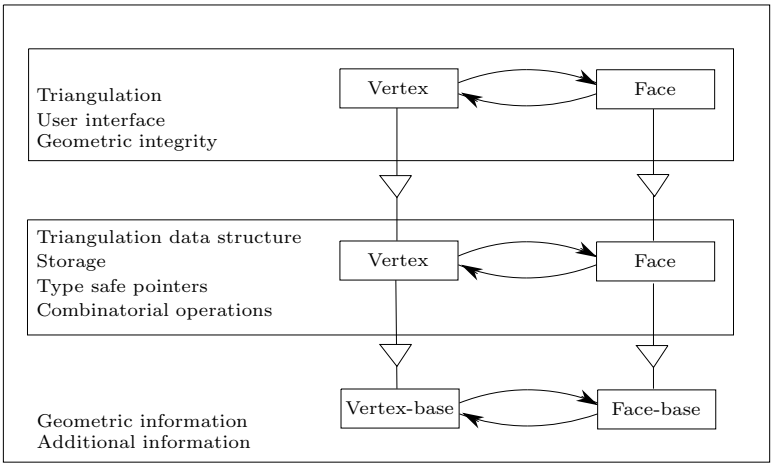

Figure 3: The three layers design used for CGAL triangulations.

use the Delaunay triangulation algorithm on three-dimensional points with the *orientation test* and the *in-circle test* computed on the x, y coordinates of the points. This application is so important that CGAL provides a model of geometric traits class specially tuned for terrains. Another way of using the flexibility offered by the geometric traits is to use different *in-circle* tests corresponding to various different metrics.

The triangulation data structure concept describes the data structure used to represent the triangulations. As explained above, the representation of triangulations in CGAL is based on faces and vertices. The triangulation data structure acts as a container for the set of faces and vertices of a triangulation. It also handles all the combinatorial operations on triangulations.

4.2 The three layers design

CGAL triangulations are implemented according to a three layers design (see Figure 3) analogous to the design used for the planar map and polyhedron packages of CGAL.

The bottom layer in this design consists of two base classes for faces and vertices that store geometric information such as the coordinates of vertices, and any other attribute (such as colors or information about constrained edges) needed by the application. Those base classes handle incidence and adjacency relations. Nevertheless they remain independent of each other's types (exchanging only `void*` pointers).

At the middle layer is a class that is a model for the triangulation data structure concept. This class is templated by the face base class and the vertex base class. It derives its own vertex and face classes from those base classes and restore type-safe pointer operations.

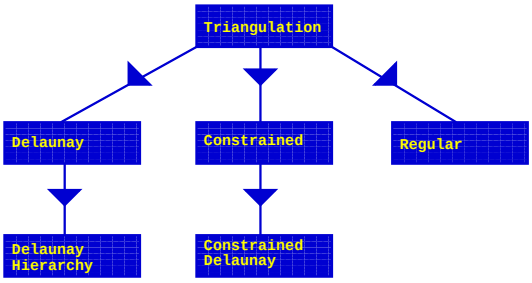

Figure 4: The derivation tree for the different two-dimensional triangulations in CGAL

At last, at the top level, the triangulation class itself handles the geometric embedding of the triangulation (i.e. deals with coordinate of vertices) and provides the user interface through high level functionalities. As explained above, the class is templated by two parameters, the geometric traits and the triangulation data structure. The geometric aspect of the triangulation is governed by the class plugged in as a model of geometric traits, while the actual representation of the triangulation relies on the class plugged in as a model of triangulation data structure. At this top level, different classes (Figure 4.2) are available to represent the different kinds of triangulations. The triangulation classes offer a user interface sufficient for most applications (see the description of functionalities in section 2.2.)

Some additional flexibility arises from this design. First, because the base classes for faces and vertices are independent from each other, the user can easily plug in the triangulation data structure his own vertex base class and/or his own face base class instead of the default ones. This allows the user to store alongside the vertices and/or faces some additional information required by his application. Second, because the triangulation data structure is a concept rather than a class, the user can instantiate the corresponding parameter with any model for this concept. Currently, CGAL provides two different models for the triangulation data structure concept of two dimensional triangulations. The first one is restricted to two-dimensional triangulations embedded in a plane. It does not use any additional pointers to handle the set of vertices and faces of the triangulation. The iterators on faces and vertices of the triangulation are coded using an implicit tree on the faces related to the planar embedding [11]. The second model of triangulation data structure offered in CGAL uses a doubly connected list to handle the set of faces. This model allows to represent any two-dimensional triangulation including triangulated surfaces embedded in three space. For three-dimensional triangulations, CGAL offers only one model of triangulated data structure which uses a doubly connected list to handle the set of cells.

5 Algorithms

Triangulations in CGAL are built through on-line insertion of vertices. Given this strategy, the main algorithmic issue is point location.

Point location. Different strategies are available in CGAL. The most naive strategy consists in walking along a line from an arbitrary start point to the query point. In the three-dimensional case, this line walk is replaced by a zigzag walk, where a traversed cell is simply left through any face whose affine hull separates the cell from the query point. In theory a zigzag walk may cycle. However a random choice among candidate faces ensures that the probability of cycling is zero. The zigzag walk has the advantage to avoid degenerate cases (such as traversal of an edge or a vertex) and has been widely used [24, pp. 38-40] [26]. This strategy is provably good in practice [15, 16].

In the special case of Delaunay triangulations, two alternative point location strategies are provided: the jump and walk strategy and the Delaunay hierarchy. The jump and walk strategy, proposed by Mücke *et al* [32] maintains a sample of $O(n^{1/3})$ of the n vertices. The vertex of the sample nearest to the query point is first found naively, then a line walk is performed from this vertex.

The Delaunay hierarchy class is a Delaunay triangulation equipped with a hierarchical data structure for point location. This structure [13] is based on random sampling and is analogous to the skip lists proposed by Mulmuley [33].

Checking. As the triangulations in CGAL can be used as a tool kit to experiment other triangulation algorithms, a checker is provided to help debugging. This checker first checks the coherence of the triangulation data structure. This implies to check for each face (cell in 3d) that it is a neighbor of each of its neighbors, that it shares two (three in 3d) vertices with each of its neighbors and that the common vertices are consistently numbered in both faces (cells). The total number of faces, edges and vertices is also checked. Then the checker verifies the orientation of each face (each cell in 3d) and the correctness of the convex hull provided by the triangulation.

Let us show that these tests are sufficient to check the correctness of a triangulation. To this aim we will transform the triangulation in dimension d ($d \leq 3$) in a polytope in dimension $d+1$ by adding a point s lying outside the affine hull of the triangulation and creating new facets by linking s to every face on the convex hull boundary. Checking the validity of the triangulation reduces to checking the convexity of that polytope. The choice of s ensures that the silhouette of this polytope is just the boundary of the convex hull of the triangulation. The above tests guarantee the local convexity of each edge (facet in 3d) of the polyhedron and the convexity of its silhouette, which, relying on Theorems 10 and 13 of [14], proves the convexity of the polytope and hence, the correctness of the triangulation.

Times in seconds		**2d**			**3d**		
		square	**terrain**	**parabola**	**cube**	**object**	**moment**
walk	CGAL	18.6	20.7	453.	21.	27.8	27.5
	Gem IV	30.	loop	loop	not available	not available	not available
	Devillers	9.9	14.9	101.	17.	24.	18.
j&w	CGAL	5.4	9.0	15.	12.6	20.2	25.
	Shewchuk	6.7	11.3	18.	not available	not available	not available
	Devillers	2.9	4.4	5.2	11.2	17.	17.
Delaunay	CGAL	2.8	3.6	20.	12.1	18.2	24.
hierarchy	*Devillers*	1.7	2.3	2.2	10.9	15.9	17.
d&c	*Shewchuk*	1.3	2.2	1.1	not available	not available	not available
	LEDA	2.4	3.3	2.0	not available	not available	not available
	qhull	6.4	13.6	29.	27.	55.	swap
sweep	*Shewchuk*	2.3	3.5	2.4	not available	not available	not available
flip	LEDA	6.3	7.9	2.5	not available	not available	not available
CGAL j&w							
Arithmetic	`double`	4.	5.6	14.	10.6	16.6	20.8
	`GMP`	151.	199.	548.	860.	1346.	750.
	[ ]`+GMP`	15.2	20.7	64.	62.	97.	55.
	`Fixed`	4.2	5.8	16.	12.6	20.2	25.

Table 1: Running times

Times have been measured with the Unix command `clock` and do not take into account the time for generating the points or writing any output. We have used a PC-Linux with a 500MHz Pentium-III and 512MB of memory.

6 Benchmarks

In this section, we will compare CGAL code for Delaunay triangulation to various other codes; we also compare several variants provided within CGAL itself. The time performances are given in Table 1.

6.1 Data

We have tested our code on the following data sets
square: a set of 100,000 points in the plane, randomly distributed in a square.
terrain: a set of 120,658 points in the plane, obtained by projecting points from geographical data of Vancouver area.
parabola: a set of 100,000 points in the plane, randomly distributed along a parabola (in fact, due to rounding, the points are not exactly on the parabola).
cube: a set of 100,000 points in three-dimensional space, randomly distributed in a cube.
object: a set of 145,300 points in three-dimensional space belonging to the boundary of a 3D object. These points have been measured by a 3D laser scanner on a dental prothesis (courtesy of KREON Industrie).
moment: a set of 5,000 random points in three-dimensional space almost on the moment curve $y = x^2$, $z = x^3$.

6.2 Implementations comparison

We first compare CGAL with algorithms based on the same algorithmic principle as the algorithms in CGAL, i.e. incremental algorithms with similar point location strategies.

In this experiment, CGAL code is using 24-bit integers to store the point coordinates and efficient exact evaluation of predicates to insure robustness (package `Fixed_precision_nt` of CGAL).

Below is a brief description of the characteristics of the different softwares that have been used in the tests.

- WALK
  GemIV: Graphics Gem IV provides a walk algorithm in 2D using floating point arithmetic [31].
  Devillers: One of the authors provides a software [12] (independent from CGAL) for 2D and 3D triangulations. It uses exact evaluation of the predicates with efficient filtering for point coordinates represented as 24-bit integers, and symbolic perturbations to deal with degeneracies. It was primarily intended to implement the Delaunay hierarchy [13] but it can be parameterized to implement the walk strategy.

- JUMP & WALK [32]
  Shewchuk: Shewchuk provides an implementation of the jump and walk algorithm in two dimensions [38, 39]. This code uses exact arithmetic with efficient filtering [40].

Devillers: This software [12] also implements the jump and walk strategy with exact evaluation of the predicates.

- DELAUNAY HIERARCHY [13]
  Devillers: This software [12] implements the Delaunay hierarchy and uses exact predicates.

Two remarks can be raised. First, comparing different softwares may yield dubious and misleading results as the quality of the coding has a big influence on the running time whatever the actual efficiency of the algorithm could be. Second, it can be pointed out that providing flexibility in a big library has a price. Some optimizations that can be done in specialized softwares cannot be done in CGAL. Nevertheless, Table 1 shows that CGAL code performs quite well compared to other codes implementing similar algorithms.

6.3 Algorithms comparison

We have tested several algorithms based on other paradigms, namely divide and conquer, plane sweep and flip.

Shewchuk: Shewchuk's implementation of Dwyer's divide and conquer algorithm [17, 18] in two dimensions with exact predicates [38, 39].
LEDA: the Library of efficient data types provides an implementation of Dwyer's divide and conquer algorithm with floating point arithmetic in two dimensions [29].
qhull: Barber's `qhull` program reduces the construction of a Delaunay triangulation in a d-dimensional space to the computation of a convex hull in a $(d+1)$-dimensional space (for $d = 2$ or 3) [3, 2]. This is done through Edelsbrunner and Seidel lifting map [19]. Barber's implementation uses floating point arithmetic.
Shewchuk: Shewchuk's implementation of Fortune's sweep line algorithm [38, 39] with exact predicates in two dimensions.
LEDA: the Library of Efficient Data Types provides also an algorithm that constructs a Delaunay triangulation in two dimensions by flipping edges [29].

Table 1 shows that the best running time is obtained with the divide and conquer algorithm that alternatively divides the plane by horizontal and vertical lines. The running time of the best CGAL incremental algorithm is slower by a factor less than 2.5 except for the parabola case. We believe that this factor is quite acceptable and balanced by the fact that the obtained triangulation supports the insertions of additional points. This justifies the fact that CGAL offers only dynamic triangulation algorithms.

6.4 Choice of arithmetic

To manage robustness issues, CGAL allows the user to change the arithmetic used for the point coordinates in a very simple way (two lines must be changed in the user's source code). We have experimented the jump and walk algorithm

with different arithmetics:
`double`: the usual floating point arithmetic, which does not provide any guarantee on the result.
`GMP`: exact integer arithmetic using the Gnu Multi-Precision package [25].
[ ]`+GMP`: dynamic filtering provided in CGAL [36, 8]. Predicates are evaluated using interval arithmetic and GMP in case of uncertainty.
`Fixed`: Static filtering [35] provided in CGAL. Predicates are evaluated with `double` arithmetic and compared with a worst case error bound computed offline. In case of uncertainty, exact integer arithmetic is used (the exact computation is coded in similar way to Shewchuk's approach). `Fixed` needs to know an upper-bound on the coordinates in advance, and uses 24 bits of precision for the coordinates.

Computing with `double` may actually fail. Using exact arithmetic is very expensive but the extra-cost can be reduced a lot when using dynamic filtering. Dynamic filtering is a quite general technique that can be applied to various predicates. When an upper bound on the range of the data is known, one can use static filtering and write specialised on purpose predicates. This allows to compete with the floating point arithmetic while providing robustness.

7 Conclusions

The triangulation package of CGAL is already in use in a few application projects, including for example a meshing application for oil resource simulation at IFP (French Institute of Petroleum), a pattern matching application for identification of proteins in electrophoresis gel [27], the simulation of complex fluids through dissipative particle dynamics [23] and the study of various methods for shape reconstruction.

This paper should by no means be considered as a user or reference manual for CGAL triangulations. Such a manual, corresponding to the current CGAL version can be found at `http://www.cgal.org/`.

Acknowledgements

The authors would like to thank Carine Bonetto, Hervé Brönnimann, Frank Da, Andreas Fabri, Frdric Fichel and Franois Rebufat for contributing to the CGAL triangulation package.

References

[1] F. Aurenhammer. Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, September 1991.

[2] Brad Barber. Qhull. http://www.geom.umn.edu/locate/qhull, Version 2.3.

[3] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, December 1996.

[4] M. Bern and D. Eppstein. Mesh generation and optimal triangulation. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 1 of *Lecture Notes Series on Computing*, pages 23–90. World Scientific, Singapore, 1992.

[5] Y. Bertrand and J.-F. Dufourd. Algebraic specification of a 3D-modeler based on hypermaps. *CVGIP: Graph. Models Image Process.*, 56:29–60, 1994.

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

[7] E. Brisson. Representing geometric structures in d dimensions: Topology and order. *Discrete Comput. Geom.*, 9:387–426, 1993.

[8] Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 165–174, 1998.

[9] D. Cazier and J.-F. Dufourd. A formal specification of geometric refinements. *Visual Comput.*, 2000.

[10] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[11] Mark de Berg, René van Oostrum, and Mark Overmars. Simple traversal of a subdivision without extra storage. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages C5–C6, 1996.

[12] Olivier Devillers. The Delaunay hierarchy. http://www-sop.inria.fr/prisme/logiciel/del-hierarchy/.

[13] Olivier Devillers. Improved incremental randomized Delaunay triangulation. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 106–115, 1998.

[14] Olivier Devillers, Giuseppe Liotta, Franco P. Preparata, and Roberto Tamassia. Checking the convexity of polytopes and the planarity of subdivisions. *Comput. Geom. Theory Appl.*, 11:187–208, 1998.

[15] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulation. In *Proc. 17th Annual ACM Symposium on Computational Geometry*, pages 106–114, 2001.

[16] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulation. Research Report 4120, INRIA, 2001.

[17] R. A. Dwyer. A simple divide-and-conquer algorithm for computing Delaunay triangulations in $O(n \log \log n)$ expected time. In *Proc. 2nd Annu. ACM Sympos. Comput. Geom.*, pages 276–284, 1986.

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

[19] H. Edelsbrunner and R. Seidel. Voronoi diagrams and arrangements. *Discrete Comput. Geom.*, 1:25–44, 1986.

[20] H. Edelsbrunner and T. S. Tan. An upper bound for conforming Delaunay triangulations. *Discrete Comput. Geom.*, 10(2):197–213, 1993.

[21] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the Computational Geometry Algorithms Library. Research Report 3407, INRIA, 1998.

[22] Eyal Flato, Dan Halperin, Iddo Hanniel, and Oren Nechushtan. The design and implementation of planar maps in CGAL. In *Abstracts 15th European Workshop Comput. Geom.*, pages 169–172. INRIA Sophia-Antipolis, 1999.

[23] E. G. Flekky, P. V. Coveney, and G. De Fabritiis. Foundations of dissipative particle dynamics. *Phys. Rev. E*, 62:2140, 2000.

[24] Paul-Louis George and Houman Borouchaki. *Triangulation de Delaunay et maillage. Applications aux éléments finis.* Hermes, Paris, France, 1997.

[25] Torbjörn Granlund. *GMP, The GNU Multiple Precision Arithmetic Library*, 2.0.2 edition, 1996. http://www.swox.com/gmp/.

[26] Leonidas J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4(2):74–123, April 1985.

[27] Frank Hoffmann, Klaus Kriegel, and Carola Wenk. A geometric approach to protein identification in 2D electrophoretic gel images. In *Abstracts 15th European Workshop Comput. Geom.*, pages 173–174. INRIA Sophia-Antipolis, 1999.

[28] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Comput. Geom. Theory Appl.*, 13:65–90, 1999.

[29] LEDA. http://www.mpi-sb.mpg.de/LEDA/, Version 4.0.

[30] P. Lienhardt. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *Internat. J. Comput. Geom. Appl.*, 4(3):275–324, 1994.

[31] Dani Lischinski. Graphics gems IV. ftp://wuarchive.wustl.edu/graphics/graphics/books/graphics-gems/.

[32] Ernst P. Mücke, Isaac Saias, and Binhai Zhu. Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 274–283, 1996.

[33] K. Mulmuley. Randomized multidimensional search trees: Dynamic sampling. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 121–131, 1991.

[34] Michael Murphy, David M. Mount, and Carl W. Gable. A point-placement strategy for conforming Delaunay tetrahedralization. In *Proc. 11th ACM-SIAM Sympos. Discrete Algorithms*, pages 67–74, 2000.

[35] Sylvain Pion. *De la géométrie algorithmique au calcul géométrique*. Thèse de doctorat en sciences, université de Nice-Sophia Antipolis, France, 1999.

[36] Sylvain Pion. Interval arithmetic: An efficient implementation and an application to computational geometry. In *Workshop on Applications of Interval Analysis to systems and Control*, pages 99–110, 1999.

[37] E. Schnhardt. ber die zerlegung von dreieckspolyedern in tetraeder. *Mathematische Annalen*, 98:309–312, 1928.

[38] 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.

[39] Jonathan Shewchuk. Triangle. A two-dimensional quality mesh generator and Delaunay triangulator. http://www.cs.cmu.edu/~quake/triangle.html, Version 1.3.

[40] Jonathan R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.

[41] Jonathan R. Shewchuk. A condition guaranteeing the existence of higher-dimensional constrained Delaunay triangulations. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 76–85, 1998.

[42] Remco C. Veltkamp. Generic programming in CGAL, the Computational Geometry Algorithms Library. In F. Arbab and Ph. Slusallek, editors, *Proceedings of the 6th Eurographics Workshop on Programming Paradigms in Graphics, Budapest, Hungary, 8 September 1997*, pages 127–138, 1997.