CGAL – THE COMPUTATIONAL GEOMETRY ALGORITHM LIBRARY

Andreas Fabri

INRIA
2004, Route des Lucioles
06903 Sophia-Antipolis, France
Andreas.Fabri@sophia.inria.fr

ABSTRACT

The CGAL project (www.cgal.org) is a collaborative effort of several research institutes in Europe. The mission of the project is to make the most important of the solutions and methods developed in computational geometry available to users in industry and academia.

Keywords: Computational geometry, C++, generic programming, exact computation paradigm

INTRODUCTION

Computational geometry is a research area in data structures and algorithm design which is a branch of theoretical computer science. In the seventies the term was coined, Preparata/Shamos wrote the first text book [20], the eighties brought theoretically optimal, but unimplementable algorithms, the nineties saw a renaissance of simple algorithms, partially due to new complexitiy analysis methods[1].

The CGAL project was started in 1995, by a group of European research institutes. The goal was to develop a computational geometry library, that is a homogeneous and coherent collection of algorithms. The design goals were robustness, efficiency, and flexibility.

We content ourselves with an extended abstract because several overview papers, library design papers, as well as papers about individual modules were published in the past. Instead we take excerpts from some of these publications and refer to further readings.

[1]Obviously, this short "history of computational geometry" is an oversimplification.

1. THE LIBRARY

The "product" of the CGAL project is CGAL, the Computational Geometry Algorithm Library, a highly modular C++ library of data structures and algorithms.

The data structures and algorithms were developed in academia, that is they are state of the art. The implementations are not only academic proofs of concept, but they are robust, that is they can deal with degenerate input, and they are time and space efficient.

The library contains 1000 classes, 300 kloc, documented on 1000 manual pages, that is it contains a critical mass of algorithms to be useful, and it is decomposed in reasonably sized classes that are well documented. CGAL is currently supported on (although not technically limited to) Solaris, Linux, Irix, and Windows, in combination with the appropriate C++ compiler from Microsoft, Borland, SGI, Kai and Gnu. We currently have release cycles of about 9 months, and about 1500 downloads per release.

The library consists of a kernel, the basic library, a collection of algorithms and data structures, and a support library.

1.1 Kernel

The geometric kernel contains simple (constant-size) geometric objects like points, lines, segments, triangles, tetrahedra. There are geometric predicates on those objects. Furthermore, there are operations such as computing intersection and distance of objects and affine transformations.

In fact CGAL offers several kernels. They differ in the representation of coordinates (Cartesian and homogeneous coordinates), in the memory management (reference counted or copying). There are even kernels that allow to execute two other kernels in parallel, and to check whether the geometric predicates give the same result. This is particularly useful to find out where an algorithm that uses floating point arithmetic has numerical problems, by executing it in parallel with an exact arithmetic.

1.2 Basic Library

The basic library contains more complex geometric objects and data structures: 2D/3D convex hulls, 2D/3D triangulations, boolean operations on polygons, polygon decomposition in monotone or convex parts, a generic half-edge data structure, geometric optimisation algorithms based on a quadratic solver for computing minimum enclosing sphere or ellipses in arbitrary dimension, arrangements of curves in the plane, multidimensional search structures for window queries, etc.

The basic library is independent from the kernel. Every algorithm defines in a very precise way which primitives it uses. For example, a 2D convex hull algorithm can take points as input and must be able to decide if one point lies to the left of another point, and to decide when you go from one point via a second point to a third point, if you make a left or a right turn. In this case the algorithm is parameterized by the point type and the two predicates that work on the point type. The algorithm is implemented in terms of the types and operations of the interface only. Generally, the CGAL kernel provides these types and predicates. For ease of use the algorithm has default arguments for these parameters, that is the user has not to worry about this, but has means of changing it, if necessary.

Among the data structures of the library, triangulations are probably the most relevant ones for mesh generation. All triangulation data structures have powerful APIs, and they are fully dynamic, that is they offer methods to insert and to remove points and constraints.

CGAL offers a Delaunay and a regular Delaunay triangulation. In 2D it offers additionally constrained and constrained Delaunay triangulation.

A Delaunay triangulation of a set of points fulfills the following empty circle property: the circumscribing circle of any facet of the triangulation contains no data point in its interior. For a point set with no subset of four cocircular points the Delaunay triangulation is unique, it is the dual of the Voronoi diagram of the points.

A constrained triangulation is a triangulation of a set of points that has to include among its edges a given set of segments joining the points. The corresponding edges are called constrained edges.

A constrained Delaunay triangulation is a triangulation with constrained edges which tries to be as much Delaunay as possible. As constrained edges are not necessarily Delaunay edges, the triangles of a constrained Delaunay triangulation do not necessarily fulfill the empty circle property but they fulfill a weaker constrained empty circle property. To state this property, it is convenient to think of constrained edges as blocking the view. Then, a triangulation is constrained Delaunay, iff the circumscribing circle of any facets encloses no vertex visible from the interior of the facet.

CGAL has a triangulation class that efficiently answers point location queries. Internally, the data structure is a hierarchy of triangulations. The triangulation at the lowest level is the original triangulation where operations and point location are to be performed. Then at each succedding level, the data structure stores a triangulation of a small random sample of the vertices of the triangulation at the preceeding level. Point location is done through a top down nearest neighbor query. The nearest neighbor query is first performed naively in the top level triangulation. Then, at each following level, the nearest neighbor at that level is found through a linear walk performed from the nearest neighbor found at the preceeding level. Because the number of vertices in each triangulation is only a small fraction of the number of vertices of the preceeding triangulation, the data structure remains small and achieves fast point location queries on real data. This structure has an optimal behaviour when it is built for 2D/3D Delaunay triangulations [7]. Because in practice it also works well for other types of triangulations, it is parameterized with a triangulation class.

Efficiency is a must in order to be of practical relevance. For example the construction of a 3D Delaunay triangulation of 2 million points in a surface reconstruction application takes 340 sec and 650 MB of memory, on a Pentium III at 550 MHz.

We currently work on constrained 3D as well as on conformal Delaunay triangulation. The latter means that Steiner points are added, so that the Delaunay triangulation of the points automatically respects the constraints.

1.3 Support Library

The support library contains non-geometric data structures, and interfaces to other libraries providing visualization and numberetypes.

2. TECHNOLOGY

The CGAL library represents cutting edge technology. This holds for the geometric algorithms, as well as for the software design, where we did not reinvent the wheel, but followed best practice.

2.1 C++

CGAL is implemented in C++ [15]. There were similar efforts for making geometry libraries: the XYZ Geobench [21] was written in Pascal and the Geometry Workbench [17] was written in Smalltalk. Because these language disappeared or were never widely accepted these efforts were deemed to fail.

Being as mainstream as possible was not the only reason for choosing C++. It is object-oriented, that is it allows a clean separation of specification and implementation. It supports polymorphism, and most importantly for CGAL, it supports the generic programming paradigm.

2.2 Generic Programming

Generic programming [18] gives a tremendous flexibility during development, and efficient code at run time of a program. Its power became apparent with the STL, the Standard Template Library, which is now part of the ISO C++ standard, and shipped with every C++ compiler.

As the STL, CGAL makes use of the concept of iterators to decouple data structures from algorithms operating on them. Iterators are an abstraction of pointers, that is everything that implements a dereference and increment operator is an iterator[2]

For example, a set of points can be inserted into a triangulation with a function that has the same signature independently from the implementation of the set. This avoids copying to one canonical container, or an inflation of functions with "the most common containers" as argument.

```
typedef Cartesian<double> K;
typedef K::Point_2 Point;

typedef Istream_iterator<Point,
                         istream> Iter;

Triangulation_2<K> T;
vector<Point> v;

// points taken from a vector of points
T.insert(v.begin(), v.end());

// points taken from standard input
T.insert(Iter(cin), Iter());
```

Because a triangulation is a container of vertices and faces, it provides iterators that allow to enumerate them.

```
class Delaunay_triangulation_2 {
  Face_iterator faces_begin();
  Face_iterator faces_end();
};
```

Because the faces adjacent to a vertex are in a circular order with no natural beginning or end, CGAL introduces the concept of circulators.

```
class Delaunay_triangulation_2::Vertex {
  Face_circulator  incident_faces();
};
```

Similar to STL containers being parameterized with the type of objects they contain, the geometric kernels provided by CGAL are parameterised with a number type, e.g., floating point, rational, or real exact numbers. This offers a trade-off between robustness and efficiency. Best results are obtained by combining them, what we will explain in the next section.

[2]A simplification again. In fact there is a complete hierarchy of iterators with different requirements on the set of operations.

```
Cartesian<double>      // floating point
Homogeneous<Gmpz>      // rationals
Cartesian<leda_real>   // real numbers
Cartesian<Filtered_exact<double,
                         leda_real> >
```

This parameterization is only the tip of the iceberg concerning the adaptability and extensibility of the kernel [13].

As stated earlier the basic library is independent from the kernel. All data structures in the basic library are parameterised with a class that provides all the geometric primitives the data structure uses.

```
template < class Geometry >
class Delaunay_triangulation_2 {

  void insert(Geometry::Point_2 t) {
   if(Geometry::orientation(p,q,t)==..)
   if(Geometry::incircle(p,q,r,t))
  }
};
```

For most data structures any CGAL kernel can be chosen as template argument. Exceptions are data structures that need very particular predicates, which are not expected to be useful in other contexts.

This mechanism further allows to use projection classes. For example, 3D points, together with predicates that compute on the projection of the points, without explicitely constructing 2D points. This allows to triangulate the points of a GIS terrain model, or the face of a polyhedron, without changing a line of code in the triangulation data structure.

Finally, this allows to seamlessly integrate a CGAL triangulation in an already existing application, and to let it operate on the application point type.

In fact, the triangulation classes have a second template argument besides the geometry, namely the combinatorics. Triangulations can be represented by vertices and faces, where each face has a pointer to its three incident faces and three incident vertices, and where each vertex has a pointer to an incident face. Alternatively, it can be represented by vertices and halfedges, where each halfedge knows its successor, its reverse halfedge, and a vertex, and where each vertex knows an incident halfedge. Although a face based representation is more compact, it may be interesting for an application to use the halfedge data structure, as the triangulation may be a single step in an application pipeline, or in an application loop, so that converting forth and back between different representations is no option.

Stroustrup [26] provides a general introduction to C++ template programming. Austern [1] provides a good reference for generic programming and the STL, and a good reference for the C++ Standard Library is the book of Josuttis [14].

CGAL is not the only library which has adopted this paradigm. It is used by the Matrix Template Library [22], by the Boost Graph Library [23], by the Grid Algorithm Library [3], in the oonumerics [19], and the Blitz++ project [27].

Note that it is an explicit goal of the two scientific computing projects to offer solutions that are as fast as Fortran code. They know that their community cannot make any compromise on speed for only getting aesthetically or software engineering wise better code.

2.3 Exact Computing

The established approach for robust geometric algorithms following the exact computation paradigm [28] requires the exact evaluation of geometric predicates, i.e., decisions derived from geometric computations have to be correct. While this can be achieved straightforwardly by relying on an exact number type, this is not the most efficient approach, and the idea of so-called filters has been developed to speed up the exact evaluation of predicates [4, 11, 24].

The basic idea is to use a filtering step before the costly computation with an exact number type. The filter step evaluates quickly and approximately the result of the predicate, but is also able to decide if the answer it gives is certified to be true or if there is a risk for a false answer, in which case the exact number type is used to find the correct answer.

CGAL implements such a filtering technic using interval arithmetic, via the `Interval_nt` number type [4]. This number type stores an interval of values which bounds are `double`s, and propagates the round-off errors that occur during floating point computations. The comparison operators on this number type have the property that they throw a C++ exception in the case that the two intervals to be compared overlap. When this occurs, it means that the filter can not certify the exactness of the result using its approximate computation. Then we have to find a different method to evaluate exactly the predicate, by using an exact but slower number type. As this failure is supposed to happen rarely on average, the overall performance of the algorithm is about the same as the evaluation of the predicate over the intervals, which is pretty fast.

Note that CGAL offers only few exact number types. We concentrate on our core competence, namely geometric algorithms, including geometric predicates. The generic programming approach allows to plug in arbitrary precision number types for integers and rationals (GMP [12]), and approximations of reals (Core [16], and LEDA [5]).

3. APPLICATION AREAS

CGAL is enabling technology, that is it does not provide a vertical solution in one application area, but provides geometric primitives for many very different application areas.

Here are some examples for how CGAL data structures get used: 3D regular triangulation for transition mesh generation in geological modelling [2], 3D Delaunay triangulation for coarse grained molecular dynamics [10], and for surface reconstruction [8], 2D Delaunay triangulations for cell decomposition in air traffic control, polyhedral surfaces for surface extraction from MRIs, smallest enclosing spheres for fast collision detection in games, boolean operations on polygons for segmentation algorithms in imaging, arrangements of arcs of circles and polylines for controlling processing tools as laser and mill.

4. CONCLUSION AND OUTLOOK

In its first years the CGAL project was seen sceptically as we used very advanced C++ techniques, that nowadays are widely accepted as best practice, and supported by almost all C++ compilers.

A more philosophically discussion, entitled "gems vs. libraries", is about whether libraries force to make compromises leading to inefficiencies. Our experience is that only the design process is slower, as everything has to fit in the big picture. On the other hand it is often obvious how something has to be done, if one is familiar with the overall design ideas of the library. Also, by many the library is perceived as monolithic. However, this is not a technical, but a packaging and documentation problem, which we plan to overcome.

CGAL is used in teaching, in computational geometry research and by people that have a clear end-user perspective as they work outside of computational geometry and computer science.

Although CGAL is distributed as source code, it is not open source, which is currently under discussion. It makes sense to do it now, because the design of the library is stable, we start having a critical mass of algorithms and data structures to further build on, and the next step is broadening the base, a task that needs a community effort.

So far we have not reached the goal to be widely used by industry. This is partially due to the fact that no company offers support for it, something that is about to change as we work towards a CGAL company.

ACKNOWLEDGMENTS

This work has been supported by ESPRIT LTR projects No. 21957 (CGAL) and No. 28155 (GALIA).

Thanks to all those colleagues from the CGAL project from whom I scavenged a paragraph or code sniplet.

REFERENCES

[1] M. H. AUSTERN Generic Programming and the STL. Addison-Wesley, 1998.

[2] S. BALAVEN, C. BENNIS, J-D. BOISSONNAT & S. SARDA. Generation of hybrid grids using power diagrams. In Proc. Numerical Grid Generation in Field Simulations, 2000.

[3] G. BERTI A Generic Toolbox for the Grid Craftsman. 17th GAMM-Seminar Leipzig on "Construction of Grid Generation Algorithms", 2001.

[4] H. BRÖNNIMANN, C. BURNIKEL & S. PION. Interval arithmetic yields efficient dynamic filters for computational geometry. Proc. 14th Annu. ACM Sympos. Comput. Geom. (1998), pp. 165–174.

[5] C. BURNIKEL, K. MEHLHORN & S. SCHIRRA The LEDA class real number. Technical Report MPI-I-96-1-001, Max-Planck Institut Inform., Saarbrücken, Germany, Jan. 1996.

[6] CGAL, the Computational Geometry Algorithms Library. `www.cgal.org/`.

[7] O. DEVILLERS Improved incremental randomized Delaunay triangulation. In Proc. 14th Annu. ACM Sympos. Comput. Geom., pages 106-115, 1998.

[8] T.K. DEY, J .GIESEN & J. HUDSON. Delaunay based shape reconstruction from large data. Proc. IEEE Symposium in Parallel and Large Data Visualization and Graphics, 2001.

[9] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra & S. Schönherr On the design of CGAL, the computational geometry algorithms library, Software - Practice and Experience, 2000, Vol. 30, 1167-1202.

[10] G. De Fabritiis, P.V. Coveney & E.G. Flekkoy Multiscale modelling of complex fluids, Proceedings of 5th European SGI/Cray MPP Workshop, Bologna (Italy) (1999)

[11] S. Fortune & C. J. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. ACM Trans. Graph. 15, 3 (July 1996), 223–248.

[12] Gmp – Arithmetic without limitations. www.swox.com/gmp/

[13] S. Hert, M. Hoffmann, L. Kettner, S. Pion & M. Seel An Adaptable and Extensible Geometry Kernel. 5th Workshop on Algorithm Engineering, BRICS, University of Aarhus, Denmark, August 28-30, 2001.

[14] N. M. Josuttis The C++ Standard Library, A Tutorial and Reference. Addison-Wesley, 1999.

[15] International standard ISO/IEC 14882: Programming languages – C++. American National Standards Institute, 11 West 42nd Street, New York 10036, 1998.

[16] V. Karamcheti, C. Li, I. Pechtchanski & C. Yap. The CORE Library Project, 1.2 ed., 1999. www.cs.nyu.edu/exact/core/.

[17] A. Knight, J. May, J. McAffer, T. Nguyen & J.-R. Sack A Computational Geometry Workbench. ACM Symposium on Comutational Geometry, 1990.

[18] D.R. Musser & A.A. Stepanov Generic programming. In 1st Intl. Joint Conf. of ISSAC-88 and AAEC-6 (1989), Springer LNCS 358, pp. 13–25.

[19] Scientific Computing in Object-Oriented Languages www.oonumerics.org

[20] F. Preparata, M.I. Shamos Computational Geometry – An Introduction. Springer Verlag, New York, 1985.

[21] P. Schorn The XYZ GeoBench for the experimental evaluation of geometric algorithms, Series in Discrete Mathematics and Theoretical Computer Science, Volume 15, 137-151, 1994.

[22] J. G. Siek & A. Lumsdaine The Matrix Template Library: Generic Components for High perfromance scientific computing. Computing in Science and Engineering, 1999. www.lsc.nd.edu/research/mtl/

[23] J. G. Siek, A. Lumsdaine & L.-Q. Lee The Boost Graph Library. www.boost.org/libs/libraries.htm

[24] Shewchuk, J. R. Adaptive precision floating-point arithmetic and fast robust geometric predicates. Discrete Comput. Geom. 18, 3 (1997), 305–363.

[25] Standard Template Library programmer's guide. www.sgi.com/tech/stl/.

[26] B. Stroustrup The C++ Programming Language, 3rd Edition. Addison-Wesley, 1997.

[27] T. Veldhuizen Techniques for scientific C++. Technical Report 542, Department of Computer Science, Indiana University, 2000. www.extreme.indiana.edu/~tveldhui/papers/techniques/.

[28] C.K. Yap & T. Dubé The exact computation paradigm. In *Computing in Euclidean Geometry*, D.-Z. Du and F. K. Hwang, Eds., 2nd ed., vol. 4 of *Lecture Notes Series on Computing*. World Scientific, Singapore, 1995, pp. 452–492.