Geometric Computing with CGAL and LEDA

Kurt Mehlhorn and Stefan Schirra

Abstract. LEDA and CGAL are platforms for combinatorial and geometric computing. We discuss the use of LEDA and CGAL for geometric computing and show that they provide a unique framework for exact, efficient and convenient geometric computing.

§1. Introduction

LEDA (Library of Efficient Data Structures and Algorithms) [16,17] and CGAL (Computational Geometry Algorithms Library) [8,26] are platforms for combinatorial and geometric computing developed in the ESPRIT-projects ALCOM II, ALCOM-IT, CGAL, and GALIA. Concerning geometric computing, the systems provide number types, geometry kernels, geometric algorithms, and visualization. They by now provide a significant fraction of the algorithms and data structures described in the computational geometry literature, where in this context computational geometry subsumes the field covered by the annual ACM Symposia on Computational Geometry. The systems are designed such that it is easy to build programs on top of them. The computations in LEDA and CGAL are exact, i.e., behave according to their mathematical specifications. This is a strong point of both systems, distinguishing them form many other geometric software products.

Based on the insight that algorithm design must include implementation to have maximal impact, Kurt Mehlhorn and Stefan Näher started the development of the LEDA software library of efficient data structures and algorithms in Saarbrücken in '89 using C++ as programming language. LEDA is now developed at Max-Planck-Institut für Informatik, Saarbrücken (Germany), and Martin-Luther-Universität Halle-Wittenberg (Germany). The idea of CGAL was conceived in fall of '94, inspired by the success of LEDA and in order to bundle forces previously put into predecessors of CGAL [2,11,20]. Development of CGAL was started in fall '96 in the CGAL-project and is

Curve and Surface Design: Saint-Malo 1999
Pierre-Jean Laurent, Paul Sablonnière, and Larry L. Schumaker (eds.), pp. 277–286.

ISBN 0-8265-1356-5.

now continued in the GALIA-project. GALIA is carried out by Max-Planck-Institut für Informatik, Saarbrücken, ETH Zürich (Switzerland), Freie Universität Berlin (Germany), INRIA Sophia-Antipolis (France), Martin-Luther-Universität Halle-Wittenberg, Tel-Aviv University (Israel), and Utrecht University (The Netherlands). The goal is to make the most important of the solutions and methods developed in computational geometry available to users in industry and academia in a C++ library.

§2. The Need for a Geometry Software Library

Reusing code that already exists and is used and thereby tested rather than implementing everything from scratch saves development time and hence reduces cost [5]. It also eases maintenance of code. Software libraries also ease the transfer of state-of-the-art algorithmic knowledge into application areas. Since geometric computing is a wide area, many application areas can benefit from the availability of the re-usable code of a geometry software library. The importance of libraries of software components in subject area domains is clearly stated in a recent report of the information technology advisory committee of the president of the US [22].

In geometric computing, software libraries consisting of reliable components are particularly useful, since implementors of geometric algorithms are faced with notoriously difficult problems [18], especially the problems of robustness and degeneracies.

Robustness

Theory usually assumes exact computation with arbitrary real numbers, while the standard substitution for real numbers in scientific computing in practice, floating-point arithmetic, is inherently imprecise. In practice, implementations of geometric algorithms compute garbage or completely fail more or less occasionally, because rounding errors lead to wrong and contradictory decisions, see [14,25,27]. With floating-point arithmetic, basic laws of arithmetic, on which the correctness proof of geometric algorithms is based, of course, don't hold anymore. We invite the reader to carry out the following simple experiment: Compute the point of intersection of the two lines with built-in floating point arithmetic. Then, again using built-in floating point arithmetic, check whether the computed intersection point lies on the intersecting lines.

Figure 1 shows an incorrect result of a computation due to rounding errors. The task is to compute the extreme points of intersection points of a set of line segments, where a point is called extreme with respect to a set of points if its removal from the set changes the convex hull of the point set. The line segments have randomly chosen endpoints lying on a circle. In a first step the intersection points of the line segments are computed, then a convex hull algorithm is run on the points computed in the first step. With floating-point arithmetic, some collinearities are not detected and too many extreme points are reported. Extreme points are shown as small disks in Figure 1. The points surrounded by a circle are actually not extreme. In the problem considered

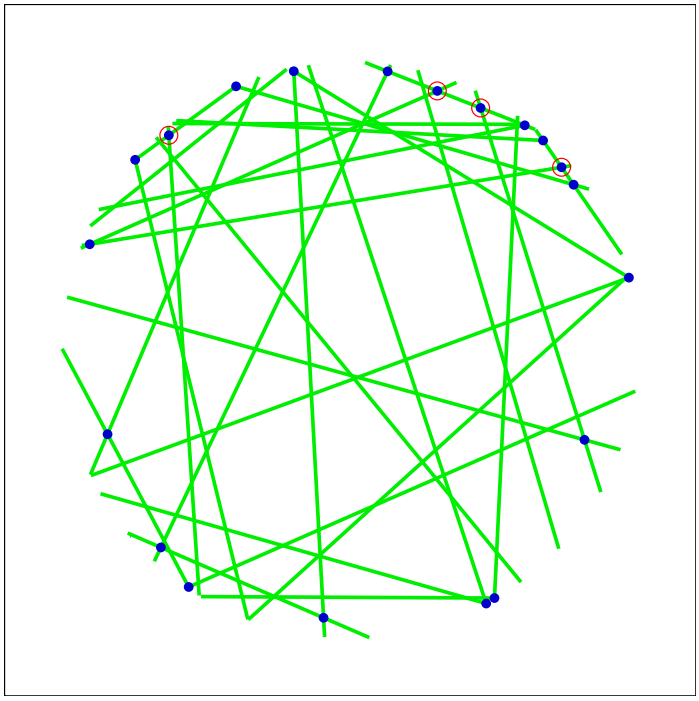

Fig. 1. Extreme points among intersection points of 30 line segments.

here, the computed output might still be useful; for many other geometric problems, however, failures of pure floating-point based implementations are much more drastically. They just crash.

Adding epsilons by trial and error to equality tests used to be common practice in implementations of geometric algorithms, but it in no way leads to a reliable correct implementations. Two main approaches to solving the precision-caused robustness problem can be identified. The first is re-designing algorithms such that they can deal with imprecision, e.g. compute a good approximate solution, but never crash. So far, this approach has been applied successfully to only very few basic problems in computational geometry, see [14,25,27]. The second approach is exact geometric computation [28], which means computing with such a precision that an implementation behaves like its theoretical counterpart, and therefore according to its mathematical specification. This is possible for many geometric problems, at least, theoretically. Note that in practice, the input does not involve arbitrary real numbers. Of course, exact geometric computation slows down computation, but thanks to clever adaptive computation using floating-point arithmetic whenever known to produce the correct result [10,15], it is now much closer to the speed of floating-point computation than it used to be a decade ago. Since libraries must be reliable in order to be usable in general, the exact geometric computation approach is taken in LEDA and in CGAL.

Degeneracies

Robustness problems caused by rounding errors are closely related to degeneracies, i.e. "exceptional" input configurations. Theory often neglects degen-

eracies for the sake of a cleaner exposition of an algorithm, and also because they are rare from a theoretical point of view: they have measure zero in the set of all possible inputs over the real numbers. In practice, however, they occur frequently. Since theory papers often leave handling degenerate cases as an exercise to the reader, implementors are often left alone with the burden of investigating the details of handling degeneracies. Furthermore, this leads to treating degeneracies as an afterthought, which is, according to our experience [3], not the most suitable way to think about them, since it leads to unnecessarily complicated and blown-up code. Considering degeneracies right from the beginning seems to be a much better approach.

Symbolic perturbation schemes have been proposed as a general approach to removing degeneracies, for an overview see [24]. With this approach, the input is perturbed symbolically such that no degeneracies arise anymore. The perturbed input can then be processed by an algorithm assuming general position. The computed output, however, does not correspond to the actual input, but to the perturbed input. Therefore, the complexity of the output might be much larger than the output for the actual input [3]. For some problems, the symbolic perturbation approach works out fine; for others, a postprocessing step is required to deduce the actual output from the output computed for the perturbed input. In many cases, this is a non-trivial task, as hard as dealing with degeneracies directly.

Algorithms and data structures in CGAL and LEDA handle all possible degenerate case by default. So a user need not to worry about all the degenerate cases. If an algorithm or data structure should not handle a degenerate case, this is clearly stated in the documentation and this precondition is checked in the implementation. However, mainly to support rapid prototyping, CGAL also provides tools for symbolic perturbation. A general randomized symbolic perturbation scheme is available for the CGAL kernels [6]. A new promising approach that has been started within the CGAL project, is controlled perturbation [23]. Here the input is perturbed numerically, such that general position is guaranteed.

§3. Number Types

The lowest level in geometric computing is the arithmetic level. LEDA and CGAL provide various number types to support exact geometric computation. LEDA provides `leda_integer`, a number type for arbitrary precision integer arithmetic and `leda_rational`, a number type for arbitrary precision rational arithmetic, based on `leda_integer`. Furthermore, it provides `leda_bigfloat`, a number type for floating point arithmetic with extended precision. A user can choose the mantissa length of the `leda_bigfloat`s or let the number type increase the mantissa length on demand. The most sophisticated number type in LEDA is the type `leda_real` [4]. This number types models a subset of algebraic numbers: All integers are `leda_real`s and `leda_real`s are closed under the operations $+$, $-$, $\cdot$, $/$, and $\sqrt[k]{}$. `leda_real`s record the computation history in an expression dag, and use adaptive evaluation to guarantee

that all comparison operations give the correct results. They use `bigfloats` internally. LEDA and CGAL also provide interval arithmetic. Furthermore, CGAL provides some fixed point arithmetic based on built-in `floats`, as well as wrappers for the gnu multiple precision integer arithmetic [12] and the number types provided by CLN [13].

§4. Geometry Kernels

The kernel of a geometry library contains the basic geometric objects and basic operations on them like points, lines, planes, sideness test, intersection and distance computations.

LEDA provides an exact geometry kernel for rational computations. Internally, it uses floating-point filters [10,15] to speed up exact computation. With floating-point filters, an expression whose sign has to be computed is first evaluated using floating-point arithmetic. Moreover, an upper bound on the error of the floating-point computation is computed as well. By comparison of the absolute computed floating-point value with this error bound, it is checked whether the floating-point computation is guaranteed to be reliable. If the sign cannot be deduced with floating-point arithmetic, the expression is re-evaluated with a more reliable arithmetic. In case of the rational geometry kernel in LEDA, arbitrary precision integer arithmetic is used. The rational geometry kernel of LEDA uses homogeneous coordinates and is coupled to the number types `leda_rational` and `leda_integer`.

CGAL provides two families of geometry kernels, one based on Cartesian coordinates and one based on homogeneous coordinates [9]. Both kernels are parameterized by a number type. All number types fulfilling a very small list of requirements can be used with the CGAL kernels. For example, the user might choose the Cartesian kernel with rational arithmetic or the homogeneous kernel with integer arithmetic. In particular, for computations involving k-th root operations, the number type `leda_real` can be used with the CGAL kernels. There are also number types that use floating-point filter techniques using interval arithmetic to speed up exact computation. These number types assume that the data passed to a test function are exact. Hence, this technique is not suited for cascaded computations. The parameterization allows a user to choose the arithmetic according to the actual needs. Using a CGAL kernel with `leda_real` is certainly the most convenient way to get reliable computation.

LEDA also provides a kernel that uses double precision floating-point arithmetic internally. Similarly, the CGAL kernels can be used with built-in floating point number types as well. This might be sufficient for some problems, but since correctness can not be guaranteed, the use of these kernels is not recommended in general.

§5. Geometric Algorithms and Data Structures

CGAL and LEDA by now provide a significant fraction of the algorithms and data structures described in the computational geometry literature. They pro-

vide several algorithms to compute convex hulls in low and higher dimensions. They provide algorithms and data structures for triangulations, constrained triangulations, Delaunay triangulations, regular triangulations, and Voronoi diagrams in two-dimensional space and Delaunay triangulations and regular triangulations in three-dimensional space. Furthermore they provide several algorithms for line segment intersection and regularizing Boolean operations on polygons. CGAL and LEDA also provide a number of query data structures. For example, there are range- and segment trees, kd-trees, as well as a data structure for range and nearest neighbor queries based on Delaunay triangulations. CGAL provides data structures for polyhedral surfaces based on a half-edge data structure, it provides a topological map data type and a planar map data structure, and a data structure for arrangements. The libraries also contain algorithms for curve reconstruction in the plane.

CGAL and LEDA provide algorithms for a number of geometric optimization problems. There are algorithms computing smallest enclosing circles, smallest enclosing ellipses, and smallest enclosing spheres, the latter in any dimension. Furthermore, there are a number of algorithms based on matrix search like computing extremal polygons and rectangular p-centers. LEDA also contains algorithms for computing smallest enclosing annuli with respect to area and width, and algorithms to compute minimum spanning trees for a set of points.

The geometric algorithms of LEDA come in two versions, one using the exact rational geometry kernel and one using the unreliable floating-point kernel. CGAL's algorithms and data structures are even more flexible with respect to the geometry kernel used. All algorithms and data structures of CGAL are parameterized by a template parameter called traits class. This traits class provides an algorithm or data structure with all the type information it needs. It tells the algorithm on which types it should operate and which types it should use to do that.

The parameterization and the resulting genericity of CGAL's algorithms and data structures is best illustrated by a simple, but instructive example. Computing the convex hull of a set of points in the plane is an intensively studied problems in computational geometry. The input is a set of points in the plane, the output is the counterclockwise sequence of extreme points. Andrew's variant [1] of the Graham scan algorithm can be formulated in such a way that it needs only two primitive operations on the points, namely a primitive to compare two points in order to sort the points lexicographically by their Cartesian coordinates, and a primitive to check the order type of a triple of points, more precisely, to check whether a sequence of three points forms a left turn. The CGAL implementation of Andrew's algorithm is parameterized by a point type and the two required primitive operations. The latter two are passed as function object types and need to correspond to the point type. We call the parameter types of an algorithm or data structure the traits types. To avoid long lists of template parameters, the traits types are collected in the traits class. Note that the parameterization is on the level of data types, not on the level of objects. In order to use the CGAL implementation of

Andrew's algorithm with a point type from a CGAL kernel, no traits class needs to be specified. CGAL adds an appropriate one. If a user wants to run the algorithm on a different point type, for example, a point type from some other C++ library or system, for example from LEDA or from some Geographical Information System, an appropriate traits class for this point type must be passed to the function in order to tell it which operations it should use. That's it. Given such a traits class, the algorithm now works with non-CGAL types as well. CGAL provides traits classes for both LEDA kernels.

The parameterization by a traits class can be used to avoid explicit transformation of the data. Assume that we have points in three dimensional space. Using a traits class that provides a comparison primitive and an order type predicate that both operate on x and y coordinates of the points only, the CGAL implementation of Andrew's algorithm can be used to compute the sequence of three-dimensional points whose projections onto the xy-plane form the convex hull of the projections of all points onto that plane. There is no need to explicitly transform the points into two-dimensional points. With an appropriate traits class, the algorithm can directly operate on the three-dimensional points. This saves time and space.

This feature is most likely even more interesting for Delaunay triangulations. Assume we have a triangulated irregular network (TIN), and we want to make a TIN with the same set of vertices without long and skinny triangles. This is usually accomplished by computing the two-dimensional Delaunay triangulation of the projections of the vertices of the TIN and lifting the vertices and triangles again. CGAL allows you to do this without explicit projection using an appropriate traits class. There are further examples where traits classes can be used nicely in the context of geometric transformations.

§6. Visualization

In LEDA, there is a data type `leda_window` which provides an interface for graphical input and output of basic geometric objects for both the X11 system on Unix platforms and Microsoft's Windows systems. This data type works with the basic geometric objects of both CGAL and LEDA. CGAL also provides preliminary support for graphical output via `OpenGL` and `geomview`.

A recent addition to CGAL and LEDA is the data type `GeoWin`. It provides an interface for the visualization of the result and progression of geometric algorithms using the window data type of LEDA. A `GeoWin` is an editor for sets of geometric objects. GeoWin manages the geometric objects in so called `scenes`. A scene contains a container storing geometric objects (the contents of the scene) and provides all operations that GeoWin needs to edit it. A `geo_scene` maintains a container with geometric objects. The `GeoWin` data type can be used for the construction and display of geometric objects and data structures, the visualization of geometric algorithms, writing interactive demos for geometric algorithms and debugging geometric algorithms.

§7. Conclusions

Reliability means that software behaves as specified. Unfortunately, there are many exceptions to this rule for geometric software, mainly due to the issues discussed in Section 2. Correctness and reliability are even more important for the components of a software library. You might be willing to accept shortcomings of a program designed for a special purpose, if problematic input instances never arise in your context. Since library component need to be generally applicable, any such shortcomings are not acceptable. CGAL (if used with a number type for exact geometric computation) and LEDA (with the rational kernel) provide geometric software that behaves according to its mathematical specification. That makes it easy to combine components form these libraries, and to build larger entities out of these components.

The use of exact computation alone cannot guarantee correctness. CGAL and LEDA also use program checking [19] to increase reliability of its components. A program checker need not compute the output for a given input. It already gets both input and output, and then has to verify that the output is the correct output for the given input. While a program gets x and has to compute $f(x)$, a checker gets x and y and must only check whether $y = f(x)$. The latter step should be computationally simpler, such that it is less likely that its implementation is buggy.

At present, LEDA and CGAL consists of more than 100,000 lines of C++ code each. Neither library provides class libraries in the sense of Smalltalk, but both provide fairly small class hierarchies if any. CGAL uses the generic programming paradigm that became known with the Standard Template Library (STL), which is now part of Standard C++. This makes CGAL very flexible, more flexible than LEDA. On the other hand, LEDA is a more complete, closed programming framework that also contains very useful components for combinatorial computing. Due to its generic design, CGAL is more open. It often relies on other sources for basic non-geometric data structures, mainly on the STL. Due to its generic design, it works well together with LEDA. Since CGAL has a more modern design and is developed by a larger group of people, the future will certainly be with CGAL. However, LEDA's components for geometric computing will continue to be useful, especially within CGAL. For more information and to download LEDA, see

`http://www.mpi-sb.mpg.de/LEDA`

For more information and to download CGAL, see

`http://www.cs.uu.nl/CGAL`

Acknowledgments. Work on this paper has been supported by ESPRIT-IV LTR project 28155 (GALIA).

References

1. Andrew, A. M., Another efficient algorithm for convex hulls in two dimensions, Inform. Process. Lett. **9** (1979), 216–219.
2. Avnaim, F., C++GAL: A C++ Library for geometric algorithms, INRIA Sophia-Antipolis, 1994.
3. Burnikel, C., K. Mehlhorn, and S. Schirra, On degeneracy in geometric computations, Proc. of the 5th ACM-SIAM Symp. on Discrete Algorithms, 1994, 16–23.
4. Burnikel, C., R. Fleischer, K. Mehlhorn, and S. Schirra, Efficient exact geometric computation made easy, in Proc. 15th Annu. ACM Sympos. Comput. Geom., 1999, 341–350.
5. Carroll, M. D., M. A. Ellis, *Designing and Coding Reusable C++*, Addison-Wesley, 1995.
6. Comes, J., and M. Ziegelmann, An easy to use implementation of linear perturbations within CGAL, in Algorithm Engineering, WAE99, Lect. Notes in Comp. Science vol. 1668, Springer Verlag, 1999, 169–182.
7. Epstein, P., J. Kavanagh, A. Knight, J. May, T. Nguyen, and J.-R. Sack, A workbench for computational geometry, Algorithmica **11** (1994), 404–428.
8. Fabri, A., G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr, On the design of CGAL, a computational geometry algorithms library. Software – Practice & Experience, special issue on Algorithm Engineering
9. Fabri, A., G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr, The CGAL Kernel: A basis for geometric computation, in Workshop on Applied Computational Geometry (WACG'96), Lect. Notes in Comp. Science, vol. 1148, 1996, 191–202.
10. Fortune, S., and C. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry, ACM Transactions on Graphics, **15** (1996), 223–248.
11. Giezeman, G. J., PlaGeo, a library for planar geometry, and SpaGeo, a library for spatial geometry, Utrecht University, 1994.
12. Granlund, T., GNU MP, The GNU multiple precision arithmetic library, 2.0.2 edition, June 1996.
13. Haible, B., CLN, The class library for numbers, 1.0.1 edition, June 1999. `http://clisp.cons.org/~haible/packages-cln.html`.
14. Hoffmann, C., The problem of accuracy and robustness in geometric computation, IEEE Computer, March 1989, 31–41.
15. Karasick, M., D. Lieber, and L. Nackman, Efficient Delaunay triangulation using rational arithmetic, ACM Transactions on Graphics, **10** (1991), 71–91.
16. Mehlhorn, K., S. Näher, M. Seel, and C. Uhrig, The LEDA user manual, version 4.0, 1999.

17. Mehlhorn, K., and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
18. Mehlhorn, K., and S. Näher, The implementation of geometric algorithms, 13th World Computer Congress IFIP94, volume 1. Elsevier Science B.V. North-Holland, Amsterdam, 1994, 223–231.
19. Mehlhorn, K., S. Näher, T. Schilz, S. Schirra, M. Seel, R. Seidel, and C. Uhrig, Checking Geometric Programs or Verification of Geometric Structures Computational Geometry: Theory and Applications **12** (1999), 85-103.
20. Nievergelt, J., P. Schorn, M. de Lorenzi, C. Ammann, and A. Brüngger, XYZ: A project in experimental geometric computation, Computational Geometry: Methods, Algorithms and Applications, Lect. Notes in Comp. Science vol. 553. Springer-Verlag, 1991, 171–186.
21. Overmars, M., Designing the computational geometry algorithms library CGAL, in *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, M. C. Lin and D. Manocha (eds.), Lect. Notes in Comp. Science vol. 1148, 1996, 53–58.
22. Presidents Information Technology Advisory Committee, Interim Report to the President, sect. 3.1.2, `http://www.ccic.gov/ac/interim/`, 1998.
23. Raab, S., Controlled perturbation for arrangements of polyhedral surfaces with application to swept volumes, in Proc. 15th ACM symposium on Computational Geometry, 1999, 163–172.
24. Seidel, R., The nature and meaning of perturbations in geometric computing, Proc. 11th Sympos. Theoret. Aspects Comput. Sci., Lect. Notes Comp. Science vol. 775, Springer Verlag, 1994.
25. Schirra, S., Precision and robustness issues in geometric computation, Handbook on Computational Geometry, Elsevier Science Publishers, Amsterdam, The Netherlands, 1999.
26. Schirra, S., R. Veltkamp, M. Yvinec (Eds.), CGAL reference manuals, version 2.1, 1999.
27. Yap, C. K., Robust geometric computation, in *CRC Handbook on Discrete and Computational Geometry*, J. E. Goodman and J. O'Rourke (eds.), CRC Press, 1997, 653–668.
28. Yap, C. K., and T. Dubé, The exact computation paradigm, in *Computing in Euclidean Geometry*, 2nd edition, D. Du and F. Hwang (eds.), World Scientific Press, 1995, 452–492.

Kurt Mehlhorn and Stefan Schirra
Max-Planck-Institut für Informatik
66123 Saarbrücken
Germany
`mehlhorn@mpi-sb.mpg.de`
`stschirr@mpi-sb.mpg.de`