Designing a Computational Geometry Algorithms Library*

Stefan Schirra

1 Introduction

Geometric problems arise in many areas. Computer graphics, robotics, manufacturing, and geographic information systems are some examples. Often the same geometric subproblems are to be solved. Hence a library providing solutions for core problems in geometric computing has a wide range of applications and can be very useful.

The success of LEDA [16, 19], a library of efficient data types and algorithms, has shown that the existence of a library can make a tremendous difference for taking advanced techniques in data structures and algorithms from theory to practice. The field of computational geometry is now very close to a state where it can provide such a library of geometric algorithms. Over the past twenty years many algorithms for geometric problems have been developed by computational geometers. Many of these algorithms clearly have no direct impact for geometric computing in practice, because they are efficient compared to other solutions only for huge problem instances. They are mainly to be considered as contributions to the investigation of the complexity of a geometric problem. Many other algorithms are practical for reasonable problem sizes, but haven't found their way into practice yet. Among the reasons for this are the dissimilarity between fast floating-point arithmetic normally used in practice and exact arithmetic over the real numbers assumed in theoretical papers, the lack of explicit handling of degenerate cases in these papers, and the inherent complexity of many efficient solutions. For these reasons there is a definite need for correct and efficient implementations of geometric algorithms.

Although much progress has been made concerning the implementation of geometric algorithms, see e.g. [15], there is still a lot of theoretical and experimental research to be done to get a robust and efficient library of geometric algorithms. It is one of the goals of the CGAL-project, a common project of the sites Utrecht University (The Netherlands), ETH Zürich (Switzerland), Free University Berlin (Germany), INRIA Sophia-Antipolis (France), Max-Planck-Institute Saarbrücken (Germany), RISC Linz (Austria), and Tel-Aviv University (Israel), to successfully do this research and to provide reliable and efficient implementations in a library of computational geometry algorithms. The library will be constructed in cooperation with ten industrial companies (mainly) in Europe. Just as the project the library will be called CGAL.

In these notes some issues related to the design of a computational geometry algorithms library are discussed. Section 2 contains general remarks on libraries and geometric computing. In Section 3 it is argued that exact geometric computation is the most promising approach to ensure robustness in a geometric algorithms library. Sections 4 and 5 relate modularity to efficiency and generality and to generic code. Finally, the ease of use of a library is discussed in Section 6.

The view held in these notes is a personal view, not the official view of CGAL. However, many of the presented concepts have been developed jointly in the kernel design group of CGAL [9]. They are also influenced by discussions in the research group interested in exact geometric

*This report was originally written as part of lecture notes for *Advanced School on Algorithmic Foundations of Geographic Information Systems, CISM, Udine, Italy, September, 16 – 20, 1996*. Work on it was partially supported by the ESPRIT IV Long Term Research Project No. 21957 (CGAL)

computation at Max-Planck-Institute for Computer Science [3, 18], LEDA [16, 19], and CGAL's ancestors C++GAL [1], PlaGeo/SpaGeo [12], and XYZ-library [21].

2 Geometric Algorithms in a Library

Geometric algorithms consist of different layers, the bottommost layers being basic objects and predicates, and the arithmetic used to do the computations. A geometric algorithms library provides components for the different layers. The kernel of a geometric algorithms library provides geometric primitives. The next level are basic algorithms and data structures. Further layers convert data to different representations, to output, and from input. Application algorithms involving geometric computation can be built with these libraries and can be part of such a library as well.

The purpose of a library is to provide reusable software components. Reusability requires generality. The components must be usable in or adaptable to various applications. Generality is not sufficient. The components must not only be adaptable, they must also lead to efficient solutions.

Library components should come with a precise description what they compute and for which inputs they are guaranteed to work. Correctness means that a component behaves according to such a specification. Clearly, correctness in the sense of reliability should be beyond question for geometric algorithms and primitives in a library. However, by far not all implementations of geometric algorithms are correct. Many implementations of geometric algorithms pretend to solve a geometric problem, but for a not clearly specified set of problems instances they don't. Due to precision problems, missing or improper handling of special cases or just incorrect coding of complicated parts many implementations of geometric algorithms disappoint the user occasionally by unexpected failures, break downs, or computing garbage.

Exactness should not be confused with correctness in the sense of reliability. There is nothing wrong with approximation algorithms or approximate solutions as long as they do what they pretend to do. Correctness can have unlike appearances. An algorithm handling only non-degenerate cases can be correct in the above sense. Also, an algorithm that guarantees to compute the exact result only if the numerical input data are integral and smaller than some given bound can be correct as well as an algorithm that computes an approximation to the exact result with a guaranteed error bound. Correctness in the sense of reliability is a must for (re-)usability and hence for a geometric algorithms library.

A good library is more than just a collection of reusable software. It provides reliable, reusable components that can be combined in a fairly seamless way.

A major design issue for a library is the choice of the programming language. C++ seems to be a good choice since it is widely available and supports library design [5]. Furthermore it allows to use other C++-libraries, e.g. LEDA [16, 19] and STL [26, 20] and gives access to functions coded in C.

3 Robustness of Library Components

In the computational geometry literature the term *robustness* is used with slightly different meanings. Sometimes an algorithm is already called robust if it does not fail. Some authors [7] use robustness to denote the ability of an algorithm to deal with inaccuracies and degenerate cases, whatever that means. Often an algorithm is called robust if the computed result is the exact result for some perturbation of the input. According to this definition a convex hull algorithm that outputs one of its input points as the "solution" would be robust. Therefore, in addition, one should require that the perturbation of the input is small. Robustness with small perturbation is called *stability*. Fortune's definition of robustness and stability [10] is a bit stronger. It requires in addition that the algorithm would compute the exact result if all computations would be precise.

According to this definition no algorithm that uses a symbolic perturbation scheme [24] could be robust.

Shewchuk [25] suggests to call algorithms *quasi-robust* if they compute some useful information which is not necessarily a correct output for any perturbation of the input.

Often it is not too difficult to get stable implementations of a basic geometric predicate. However, the composition of stable predicates can create problems. This makes even stable, imprecise predicates less useful as library components. The different evaluations of predicates in an execution of an implemented algorithm are not independent, because they operate on the same set of input data. Even if each implementation of a predicate used in an implementation of a geometric algorithm is stable, such that for each decision based on an imprecise evaluation of a predicate there is a perturbation of (some part of) the input data that justifies it, there might be no perturbation that justifies all decisions based on imprecise evaluations of predicates simultaneously. Using stable predicates does not automatically give a stable algorithm. Globally inconsistent decisions have to be avoided.

For a few basic geometric problems there are algorithms that reach (quasi-)robustness, cf. [23]. The techniques used in these algorithms are fairly special and it seems unlikely that they can be easily transferred to other geometric problems. A general theory how to implement geometric algorithms with imprecise predicates is still a distant goal.

Using imprecise geometric predicates in an implementation is not easy. It is much easier to built an implementation upon exact geometric predicates. With exact predicates all the algorithms developed under the real RAM model [22] can be implemented in a straightforward way. A redesign that deals with imprecision in the predicates is not necessary. Using exact predicates is the easiest and hence most promising way to get reliable implementations. Therefore exact components should be the first choice for a geometric algorithms library, at least at the lower levels of the library.

Note that this vote for exactness does not exclude approximate solutions at all! Exact basic predicates simplify the task of implementing approximation algorithms as well. In many applications the input data are known to be inaccurate, such that the exactness of the computed result is a minor issue. A stable, efficient algorithm computing an approximate solution is here sufficient. The specification of a library algorithm should clearly indicate what an algorithm computes. For the sake of ease of use of the library components names should be chosen appropriately and not suggest exactness if approximate solutions are computed. Correctness in the sense of reliability is the primary goal, not exactness, but exact computation paradigm [27] seems to be the safest way to reach it.

4 Efficiency, Generality, and Modularity

Users of a computational geometry algorithms library may have fairly different needs. For some users speed might be more important than exactness. Other users might need high accuracy or even the exact result. Sometimes an algorithm must work for a wide range of input data while in other applications the set of relevant input data is very restricted, e.g. all coordinates could be integers from a very small range. A library should enable efficient solutions in all situations.

One of the challenges in the construction of a geometric library is efficient exact computation. It should be clear that one has to pay for exactness. Recently, there has been much progress on exact geometric computations [27] and various tools for exact computation are already available [3, 8, 11]. How efficiently a predicate can be exactly evaluated depends on the numerical data that are involved. In applications the numerical input data are hardly arbitrary real numbers. The numerical input data are usually integers from a certain range, or fixed precision floating-point numbers, and sometimes algebraic numbers, given in some more or less symbolic representation. If a library places efficient implementations of predicates with different restrictions on the numerical input data at the user's disposal, modular geometric algorithms can be easily trimmed for an application. Since in a layered geometric algorithm calculations are wrapped in primitives the higher levels are combinatorial in nature. Therefore the code on the higher layers for an algorithm can be the same, no matter how the geometric predicates are realized. By choosing implementations of predicates that are adjusted to the numerical data of the problem instances to be solved, the performance of a geometric algorithm can be optimized and both efficiency and exactness (i.e. correctness) can be reached. Thereby, modularity of a library that allows to exchange (sub-)components can provide a bridge between the contradictory goals efficiency and generality.

The representation of geometric objects can have significant impact on the performance of geometric algorithms and predicates as well. Like LEDA [16, 19] disposes different implementations of a data structure, a computational geometry algorithms library could provide different implementations of geometric objects. For example, consider points and vectors. Representation with homogeneous coordinates allows to replace many computations involved in geometric algorithms by division-free computations. Hence they are very useful for exact geometric computation. On the other hand they are a waste if the calculations with Cartesian coordinates do not involve any divisions either. Thus a geometric algorithms library should offer points and vectors with homogeneous coordinates besides points and vectors with ordinary Cartesian coordinates. Homogeneous coordinates have also benefits for the representation of affine transformation [13].

5 Generic Code

For the correctness of an algorithm the correctness of its components is important, not their actual implementation. Layered geometric algorithms will work with a variety of implementations of predicates, subtasks, and representation of geometric objects, as long as they are correct. Representation-specific parts are hidden in lower layers.

The lowest layer in geometric computation is the underlying arithmetic. Exact geometric computation simply means that all decisions made by an algorithm are correct. For many evaluations of geometric predicates choosing an appropriate number type that guarantees exact comparison and hence correct decision is sufficient to do exact geometric computation. Which number type is sufficient highly depends on the application, more precisely on the numerical input data. If all numbers arising in the input or as intermediate results in the evaluation of a predicate are integers, a multiple precision integer arithmetic is sufficient. If all integers arising in the evaluation are sufficiently small, calculations can be done with limited precision number types like `int` and `double`. A very general and powerful number type is the number type `real` [4]. It guarantees exact comparisons for numbers given by arithmetic expressions with integral operands over the basic operations $+, -, *, /$ and $\sqrt[k]{\ }$-operations.

Operator overloading in C++ allows to write code for software provided number types in a natural way. The code for evaluating a predicate with number types guaranteeing exact comparisons differs only in the declarations where identifier of the number type are declared. Otherwise it is invariant, as long as the evaluation of a predicate is based on the same representation of objects and coordinates. The template mechanism of C++ allows to write code parametrized by types. The template parameters are place-holders for types [14]. C++ allows to write class templates and function templates. Function templates can be used to avoid duplication of code for geometric predicates that differ only by the underlying arithmetic. Not all predicates are global functions. Some of them are methods associated to classes. Class templates can be used to parametrize these predicates by number types.

The actual code for a certain number type is then automatically generated by the compiler. This process is called instantiation, the resulting code is called *implicit specialization*. For some number types the implicit specialization gives no optimal or even no correct code. For these number types alternative code can be defined. In C++ terminology this code is called *explicit specialization*. Instead of generating code automatically from the template the compiler will use the alternative code. For example, the implementation of a geometric predicate involving the computation of the sign of some determinant could deviate from the standard evaluation provided by the template code and use a special method for computing the sign, e.g. for determinants with entries given as integral `double`s [2] could be used.

Parametrization of geometric objects, data structures and algorithms by a class that maps basic geometric objects via typedefs to representations (and number types) allows to write generic code at a higher level, i.e. code working with a variety of implementations of components. A class defining such a mapping is called a representation class. Such a class looks like

```
class RepCls
{
  typedef   ...          Point_2 ;
  typedef   ...          Vector_2 ;
  typedef   ...          Line_2 ;
    ...     ...            ...
}
```

At a sufficiently high level code parametrized by a representation class uses components parametrized by representation classes only. At lower levels the code builts bridges to the actual representations of objects specified in the representation class, e.g. by converting to these representations and calling functions defined for these representations. Since all representations provided by the library for a geometric primitive have a common interface, the code is generic. For example, a convex hull algorithm

```
template < class R >
CGAL_Polygon_2<R>
CGAL_convex_hull( const list< CGAL_Point_2<R> & > )
```

can use components parametrized by representation classes like

```
bool
CGAL_leftturn( const CGAL_Point_2<R> & p,
               const CGAL_Point_2<R> & q,
               const CGAL_Point_2<R> & r)
```

which, on the other hand, simply forwards the call to a function operating on the actual representations by returning

```
CGAL_leftturn ( (R::Point_2 &)p, (R::Point_2 &)q, (R::Point_2 &)r) );
```

The proposal for the CGAL-kernel [9] suggest to provide representation classes that map to representations of predicates and basic objects based on homogeneous coordinate representation and Cartesian coordinate representation. Both representation classes are parametrized by the number type used to represent the coordinates. For basic geometric objects like points and vectors the classes parametrized by a representation class are derived from the implementation classes, e.g.

```
template < class R >
class CGAL_Point_2 : public R::Point_2
{ ...  }
```

By the use of parametrization by a representation class generic code working with both primitives based on homogeneous representation and primitives based on Cartesian representation can be written. Of course the correctness of the implementation depends on the correctness of the primitives that are used. In the rare cases where the overall algorithm is "parsimonious", i.e. never evaluates predicates whose outcome it should know by previous tests, even implicit specialization resulting in imprecise predicates gives robust algorithms. However, if the implementation makes dependent decisions, robustness with imprecise predicates is not guaranteed.

Templatization with number types has both advantages and disadvantages. If a number type is not adequate for the numerical input that has to be handled, the results will be incorrect. On the other hand it does not make sense to do all arithmetic operations with a powerful number type that slows down computation much more than necessary. Assume that there is a convex hull algorithm that works with arbitrary precision integer arithmetic. It would not work if it would be templatized and used with number type `double` or number type `int` and large integral input. On the other hand a more efficient number type for medium size integers could be used if all integers in the input were known to be small. The specification should tell the user how large the intermediate integral results can be. The convex hull code could also be used with more powerful number types like the LEDA number type `real` [4], if e.g. all input data are algebraic numbers given as arithmetic expressions involving square roots.

6 Ease of Use

Ease of use is an important aspect for the success of a library. User need easy access to the components of a library and don't want to have to learn a lot about the library before they can use it.

The use of representation classes and the possibility to choose number types and to write specializations offer many ways to optimize the performance of library components in an application and hence for combining efficiency and generality in a library. However, this flexibility should not deter a potential user from making use of a computational geometry algorithms library. Fortunately, these advanced features can be hidden to a novice user. To this end pre-instantiations (or explicit specializations) of the templated code are offered and the templatization is hidden through typedefs. Only the advanced user will see the whole power and profit from it.

A library should be a unified whole. The naming scheme should be uniform, for the classes describing geometric objects as well as for the member functions of these classes.

Another point where a library can make user's life easier is how it deals with a collections of objects. The iterator concept of the C++ Standard Template Library [26, 20], a part of the forthcoming C++ standard [6], can be used to make the feeding of an algorithm with a set of input data of the same type uniform. There is no need for a user to copy the data from his favorite container to a library container before they can be fed into an algorithm, if the favorite container supports iterators.

Since the correctness of an implementation depends on the correctness of the components that are used, the components in a library should be designed for checkability and (by default) self-checking [17]. For example, preconditions of routines can be checked and results of computations are "verified" by checking postconditions. Such checkings are of great help in the implementation process and can reduce debugging efforts drastically.

References

[1] F. Avnaim. *C++GAL: A C++ Library for Geometric Algorithms*, 1994.

[2] F. Avnaim, J.D. Boissonnat, O. Devillers, F.P. Preparata, and M. Yvinec. Evaluating signs of determinants using single precision arithmetic. Technical Report 2306, INRIA Sophia-Antipolis, 1994.

[3] C. Burnikel, J. Könemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proceedings of the 11th ACM Symposium on Computational Geometry*, pages C18–C19, 1995.

[4] C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA class `real` number. Technical Report MPI-I-96-1-001, Max-Planck-Institut für Informatik, 1996.

[5] M.D. Carrol and M. Ellis. *Designing and Coding Reusable C++*. Addison-Wesley, 1995.

[6] ANSI/ISO C++ Standards Committee. *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++*. Doc No: X3J16/95-0087, WG21/N0687. April 1995.

[7] T.K. Dey, K. Sugihara, and C.L. Bajaj. Delaunay triangulations in three dimensions with finite precision arithmetic. *Computer Aided Geometric Design*, 9:457–470, 1992.

[8] T. Dubé and C.K. Yap. A basis for implementing exact computational geometry. extended abstract, 1993.

[9] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel : a basis for geometric computation. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, pages 191–202. Springer LNCS 1148, 1996.

[10] S. Fortune. Stable maintenance of point-set triangulations in two dimensions. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Sience*, pages 494–499, 1989.

[11] S. Fortune and C. van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. of the 9th ACM Symp. on Computational Geometry*, pages 163–172, 1993.

[12] G.-J. Giezeman. *PlaGeo, a library for planar geometry, and SpaGeo, a library for spatial geometry*, 1994.

[13] D. Hearn and M.P. Baker. *Computer Graphics*. Prentice Hall, 1994. 2nd edition.

[14] S. Lippman. *C++ Primer*. Addison-Wesley, 1991. 2nd edition.

[15] K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *13th World Computer Congress IFIP94*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.

[16] K. Mehlhorn and S. Näher. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.

[17] K. Mehlhorn, S. Näher, T. Schilz, S. Schirra, R. Seidel, M. Seel, and C. Uhrig. Checking geometric programs or verification of geometric structures. In *Proceedings of 12th Annual ACM Symp. on Computational Geometry*, pages 159–165, 1996.

[18] K. Mehlhorn, S. Näher, S. Schirra, M. Seel, and C. Uhrig. A computational basis for higher-dimensional computational geometry. Technical report, Max-Planck-Institut für Informatik, 1996.

[19] K. Mehlhorn, S. Näher, and C. Uhrig. *The LEDA User manual*, 3.5 edition, 1997. cf. `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[20] D.R. Musser and A. Saini. *STL tutorial and reference guide*. Addison-Wesley, 1996.

[21] J. Nievergelt, P. Schorn, M. de Lorenzi, C. Ammann, and A. Brüngger. XYZ: Software for geometric computation. Technical Report 163, Institut für Theorische Informatik, ETH, Zürich, Switzerland, 1991.

[22] F. Preparata and M.I. Shamos. *Computational Geometry*. Springer Verlag, 1985.

[23] S. Schirra. Precision and robustness in geometric computations. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*. Springer LNCS, to appear.

[24] R. Seidel. The nature and meaning of perturbations in geometric computations. In *STACS94*, 1994.

[25] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. Technical Report CMU-CS-96-140, School of Computer Science, Carnegie Mellon University, 1996.

[26] A. Stepanov and M. Lee. The standard template library. July 1995. http://www.cs.rpi.edu/~musser/stl.html.

[27] C. K. Yap and T. Dubé. The exact computation paradigm. In D.Z. Du and F. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–492. World Scientific Press, 1995. 2nd edition.