# Applications of the Generic Programming Paradigm in the Design of CGAL *

Hervé Brönnimann[1], Lutz Kettner[2], Stefan Schirra[3], and Remco Veltkamp[4]

[1] INRIA Sophia-Antipolis, France
Herve.Bronnimann@sophia.inria.fr
[2] ETH Zürich, Switzerland
kettner@inf.ethz.ch
[3] Max-Planck-Institute for Computer Science, Saarbrücken, Germany
stschirr@mpi-sb.mpg.de
[4] Dept. Computing Science, Utrecht University, The Netherlands
Remco.Veltkamp@cs.uu.nl

**Abstract.** We report on the use of the generic programming paradigm in the computational geometry algorithms library CGAL. The parameterization of the geometric algorithms in CGAL enhances flexibility and adaptability and opens an easy way for abolishing precision and robustness problems by exact but nevertheless efficient computation. Furthermore we discuss circulators, which are an extension of the iterator concept to circular structures. Such structures arise frequently in geometric computing.

## 1 Introduction

CGAL is a C++ library of geometric algorithms and data structures. It is developed by several sites in Europe and Israel. The goal is to enhance the technology transfer of the algorithmic knowledge developed in the field of computational geometry to applications in industry and academia.

Computational geometry is the sub-area of algorithm design that deals with the design and analysis of algorithms for geometric problems involving objects like points, lines, polygons, and polyhedra. Over the past twenty years, the field has developed a rich collection of solutions to a huge variety of geometric problems including intersection problems, visibility problems, and proximity problems. See the textbooks [15, 23, 18, 21, 6, 1] and the handbooks [10, 24] for an overview. The standard approach taken in computational geometry is the development of provably good and efficient solutions to problems. However, implementing these algorithms is not easy. The most common problems 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

* This work is partially supported by the ESPRIT IV LTR Projects No. 21957 (CGAL) and 28155 (GALIA), and by the Swiss Federal Office for Education and Science (CGAL and GALIA).

of explicit handling of degenerate cases in these papers, and the inherent complexity of many efficient solutions. As a result, many useful geometric algorithms have not found their way into the many application domains of computational geometry yet. Therefore, a number of institutions from the computational geometry community itself has started to develop CGAL [22]. The availability of a software-library can make a tremendous difference. If there is no need anymore to implement the geometric algorithms from scratch, the effort of experimenting with the solutions developed in computational geometry is lowered.

The major design goals for CGAL include correctness, robustness, flexibility, efficiency and ease of use [7]. One aspect of flexibility is that CGAL algorithms can be easily adapted to work on data types in applications that already exist. The design goals, especially flexibility and efficient robust computation, led us to opt for the generic programming paradigm using templates in C++ [19], and to reject the object-oriented paradigm in C++ (as well as Java). In several appropriate places, however, we make use of object-oriented solutions and design patterns. Generic programming with templates in C++ provides us with the help of strong type checking at compile time, and also has the advantage that it doesn't cause runtime overhead. In the sequel we give examples of the use of the generic programming paradigm in CGAL.

## 2 Generic Programming in Geometric Computing

One of the hallmarks of geometry is the use of transformations. Indeed, geometric transformations link several geometric structures together [6, 1]. For example, duality relates the problem of computing the intersection of halfplanes containing the origin to that of computing the convex hull of their dual points. The Voronoi diagram of a set of points is also dually related to its Delaunay triangulation, and this triangulation can be computed as a lower convex hull of the points lifted onto a paraboloid in one dimension higher.

For this kind of problem, the relevance of generic programming is obvious. Much like the problem of finding a minimum element in a sequence, the use of a geometric algorithm such as computing the convex hull can have many applications via geometric transformations. In this setting, the algorithm does not operate on the original objects but on their transformed version, and the primitives used by the algorithm must also be applied through the same transformation. This is achieved in CGAL through the use of *traits classes*.[1]

Another hallmark of geometric programming is the multiple choice of representations of a geometric object. Naturally, some representations are more suited to a particular problem. In computing Delaunay triangulations or minimum enclosing circles, for example, a circle will likely be represented implicitly as the circumcircle of three points. Other representations would include a pair of cen-

[1] In CGAL, the term 'traits class' is used in a more general setting than associating related type information to built-in types, which is the original usage [20].

ter and squared radius[2] (this is the default representation in CGAL). In most problems, a point would be represented by its coordinates, either Cartesian or homogeneous. Another choice, more suited to boolean operations on polygons for instance, might be as an intersection of two lines: such a representation is stable when performing an arbitrary number of boolean operations.

Finally, geometric computing has been successful in abstracting several paradigms of its own. For example, the sweep paradigm is used to compute arrangements of lines of segments, triangulations of polygons, and Voronoi diagrams. Randomized incremental algorithms have been abstracted in a framework [5, 1] that uses few primitives. Geometric optimization problems have brought forth the generalized LP-type problems that have been so hugely successful in computing minimum enclosing circles and ellipses, distances between polytopes, linear programs, etc. In all three cases, one can write the skeleton of an algorithm that will work, given the appropriate primitives. The user then has to supply only those primitives via a traits class.

Algorithms in CGAL are generic, they work with a variety of implementations of predicates and representations of geometric objects. In the next subsection, we argue that generic programming is especially relevant to geometric computing.

## 3 Traits Classes and the CGAL Kernel

We briefly recall the main elements of the design [7] of the CGAL kernel [3]. The two basic representation classes in the CGAL kernel are `CGAL_Cartesian` and `CGAL_Homogeneous`. They provide basic geometric types like `Point_2` and `Segment_2`. Instead of a dense class hierarchy, these types are organized as a loosely coupled collection. For example, `PointC2` is a *model* for the abstract concept of a point (with internal representation as Cartesian coordinates). A model of a concept is an implementation that fulfills the requirements of this concept. A typedef declaration in the corresponding representation class `Cartesian` links `Cartesian::Point_2` with `PointC2`.

Genericity is extended by templating all these classes by the number type used to store the coordinates. This allows tailoring the kernel to the application, especially regarding robustness and precision concerns as described below in Section 6. The concept of a geometric kernel is currently extended to contain the predicates and constructions on the basic kernel objects. A kernel is a model of the concept of a geometric kernel as defined by the CGAL Reference Manual [3]. With this design, a model of a geometric kernel can be passed as a traits parameter to the algorithms of the CGAL basic library, as described in Section 5. This does not exclude users to provide an interface to their own library as a model of a CGAL kernel. For example, there are adaptations of the floating-point and the rational geometry kernel of LEDA [16].

[2] We use squared radius instead of radius, because the former is rational for a circle given by three points with rational coordinates, whereas the latter might be irrational.

## 4 Data Passing by Iterators and Circulators

A prominent example of the generic programming paradigm is the Standard Template Library (STL) accompanying the C++ standard. Algorithms interact with container classes through the concept of *iterators*, which are defined through a set of requirements. Each container class is supposed to provide a model for an iterator. The algorithms are parameterized with iterators, and can be instantiated with any model that fulfills the iterator requirements. So, containers and algorithms are kept independent from each other. New algorithms can be developed without knowing any container class, and vice versa.

For example, a range of points can be fed into an algorithm by providing two forward iterators `first` and `beyond`. They define a range `[first,beyond)`, if applying a finite number of times the operator `++` to `first` makes that `first == beyond`. The range refers to the points starting with `*first` up to but not including `*beyond`. The iterator `beyond` is said to point 'past the end' of the range. An application programmer need not copy the data from a favorite container into another container in order to feed it to the algorithm, as long as the container provides iterators. Inside the algorithm the points are addressed by advancing the iterator, independent of the particular container type.

In the following example, `min_encl_circle` is template-parameterized with a forward iterator:

```
template <class ForwardIterator>
Circle
min_encl_circle(ForwardIterator first, ForwardIterator beyond)
{ ... }
```

The concept of iterators in the STL is tailored for linear sequences. In contrast, circular sequences evolve naturally in many combinatorial and geometric structures. Examples are polyhedral surfaces and planar maps, where the edges emanating from a vertex or the edges around a facet form a circular sequence.

Since circular sequences cannot provide efficient iterators (for reasons discussed later), we have introduced the new concept of *circulators* in CGAL. They share most of the requirements with iterators, while the main difference is the lack of a past-the-end position in the sequence. Appropriate adaptors are provided between iterators and circulators to seat circulators smoothly into the framework of the STL. We give a short introduction to circulators and discuss advantages and disadvantages thereafter.

The following example illustrates the use of a circulator in a generic function `contains` that tests whether a certain `value` exists in a circular sequence. As usual for circular structures, we use a `do-while` loop to reach all elements in the specific case `c == d`.

```
template <class Circulator, class T>
bool contains( Circulator c, Circulator d, const T& value) {
    if ( c != NULL) {
        do {
```

```
            if ( *c == value)
                return true;
        } while (++c != d);
    }
    return false;
}
```

Three circulator categories are defined: forward, bidirectional and random-access circulators. Given a circulator `c`, the operation `*c` denotes the item the circulator refers to. The operation `++c` advances the circulator by one item and `--c` steps a bidirectional circulator one item backwards. For random-access circulators `c+n` advances the circulator `n` times. Two circulators can be compared for equality.

Circulators have a different notion of reachability and ranges than iterators. A circulator `d` is called *reachable* from a circulator `c` if `c` can be made equal to `d` with finitely many applications of the operator `++`. Due to the circularity of the data structure this is always true if both circulators refer to items of the same data structure. In particular, `c` is always reachable from `c`. Given two circulators `c` and `d`, the range `[c,d)` denotes all circulators obtained by starting with `c` and advancing `c` until `d` is reached, but does not include `d` for $\texttt{d} \neq \texttt{c}$. So far it is the same range definition as for iterators. The difference lies in the use of `[c,c)` to denote all items in the circular data structure, whereas for an iterator `i` the range `[i,i)` denotes the empty range. As long as `c != d` the range `[c,d)` behaves like an iterator range and could be used in STL algorithms. For circulators however, an additional test `c == NULL` is required that returns true if and only if the data structure is empty. In this case the circulator `c` is said to have a *singular value*.

Supporting both iterators and circulators within the same generic algorithm is just as simple as supporting iterators only. This and the requirements for circulators are described in the CGAL Reference Manual [13].

The main reason for inventing a new concept with the same goals as iterators is efficiency. An iterator is supposed to be a light-weight object – merely a pointer and a single indirection to advance the iterator. Although iterators could be written for circular sequences, we do not know of an efficient solution. The missing past-the-end situation in circular sequences can be solved with an arbitrary sentinel in the cyclic order, but this would destroy the natural symmetry in the structure (which is in itself a bad idea) and additional bookkeeping in the items and checking in the iterator advance method reduces efficiency. Another solution may use more bookkeeping in the iterator, e.g. with a start item, a current item, and a kind of winding-number that is zero for the `begin()`-iterator and one for the past-the-end situation. Though the concept of circulators allows light-weight implementations, the CGAL support library provides adaptor classes that convert between iterators and circulators (with the corresponding penalty in efficiency), so as to integrate this new concept into the framework of the STL.

A serious design problem is the slight change of the semantic for circulator ranges as compared to iterator ranges. Since this semantic is defined by the intuitive operators `++` and `==`, which we would like to keep for circulators as well, circulator ranges can be used in STL algorithms. This is in itself a useful

feature, if there would not be the definition of a full range `[c, c)` that an STL algorithm will treat as an empty range. However, the likelihood of a mistake may be overestimated, since for a container `C` supporting circulators there is no `end()` member function, and an expression such as `sort( C.begin(), C.end())` will fail. It is easy to distinguish iterators and circulators at compile time, which allows for generic algorithms supporting both as arguments. It is also possible to protect algorithms against inappropriate arguments with the same technique, though it is beyond the scope of CGAL to extend STL algorithms.

## 5 Generic Geometric Algorithms using Traits classes

Also at a higher level, CGAL uses traits classes. In algorithms, primitives are encapsulated in traits classes, so that application specific primitives and specialized versions can be plugged in. To illustrate this, consider the problem of computing the minimum enclosing circle of a set of points. This problem is generally motivated as a facility location problem, or in robotics, as how to anchor a robot arm so as to minimize its range under the constraint that it must reach all specified locations, modeled as points.

Suppose now that those points are at different heights and that the robot arm can slide along a vertical axis. To test whether a number of points in space can be reached by this robot arm, it is possible to compute the smallest enclosing vertical cylinder, and check whether the radius is not larger than the length of the robot arm. For the sake of simplicity we ignore possible collisions.

The smallest enclosing circle is calculated in the two-dimensional plane, but the points are three-dimensional. Of course we could make a version of the smallest enclosing circle algorithm that operates on three-dimensional points, but generates the two-dimensional circle. This is against the principle of code reuse, one of the hallmarks of modern programming. Alternatively, we could perform the projection explicitly, generate two-dimensional points, apply the two-dimensional algorithm, and maintain the correspondence between the three-dimensional and the two-dimensional points if needed. This requires extra bookkeeping and space, and is not so elegant.

In general, the smallest enclosing circle algorithm can treat points in an abstract way using predicates like `side_of_bounded_circle(p,q,r,test)` to test if the point `test` lies on the bounded side (inside), on the unbounded side, or on the boundary of the circle through `p`, `q` and `r` (assuming `p`, `q` and `r` are not collinear). So, the algorithm can be parameterized by point type and predicates. CGAL does this: the point type and predicates are put into what CGAL calls a traits class. In the following example, the algorithm `min_encl_circle` is template-parameterized with the ForwardIterator and the Traits class:

```
template <class ForwardIterator, class Traits>
Circle
min_encl_circle(ForwardIterator first, ForwardIterator beyond,
                Traits traits);
```

Rather than explicitly working with a CGAL point type, the algorithm internally works with `traits::Point_2`. Likewise, instead of using a global function for the `side_of_bounded_circle(p,q,r,test)` predicate, it uses a function object which is accessible through a member function of the traits class. This function object provides the `operator() (traits::Point_2 p, traits::Point_2 q, traits::Point_2 r, traits::Point_2 test)`.

```
template <class ForwardIterator, class Traits>
Circle
min_encl_circle(ForwardIterator first, ForwardIterator beyond,
                Traits traits) {
  typedef traits::Point_2  Point_2;  // local type for points
  for ( ; first != beyond; ++first ) {
      // ... using traits.side_of_bounded_circle()( p, q, r, test)
  }
  // return result
}
```

An application programmer can use any point type by defining an appropriate traits class `My_traits` with a proper `My_side_of_bounded_circle`:

```
struct My_side_of_bounded_circle {
  bool
  operator()(My_point *p, My_point *q, My_point *r, My_point *test)
  { ... }
};

struct My_traits {
  typedef My_point                   Point_2;
  typedef My_side_of_bounded_circle  Side_of_bounded_circle;
  Side_of_bounded_circle side_of_bounded_circle() const {
      return Side_of_bounded_circle();
  }
} my_traits;
```

The use of a member function to access the predicate allows one to pass additional data from a traits object to the predicate object in order to influence the predicate's behavior. The smallest enclosing circle algorithm can be invoked with this traits class:

```
vector<My_point> points(num);
Circle min_encl_circle(points.begin(), points.end(), My_traits());
```

In this way, specialized predicates can be used that work on 3D points, but evaluate the orientation on projections of those points, leaving the algorithm itself unchanged. CGAL uses traits classes for algorithms and for data structures throughout the basic library, and provides traits classes for common use, such

as the two CGAL kernels and projections of them, or the LEDA geometry kernels. A default argument even hides the mechanism of the traits classes from inexperienced users.

## 6 Generic Programming eases Efficient Robust Geometric Computing

Since the field of computational geometry has its roots in theoretical computer science, algorithms developed in computational geometry are designed for a theoretical machine model, the so-called real RAM model [23]. In a real RAM, exact computation with arbitrary real numbers is assumed (with constant cost per arithmetic operation). In practice, however, the most popular substitution for computation with real numbers in scientific computing is floating-point computation. Floating-point arithmetic is fast, but not necessarily exact. Depending on the type of geometric problem to be solved, implementations of theoretically correct algorithms frequently produce garbage, or crash, due to rounding errors in floating-point arithmetic. The reason for such crashes with floating-point arithmetic are inconsistent decisions leading the algorithms into states they never could get into with exact arithmetic [12, 25]. Exact computation has been proposed to resolve the problem [28]. This approach is appealing, since it lets an implementation behave exactly as its theoretical counterpart. No redesign for a machine model that takes imprecise arithmetic into account is necessary. Fortunately, exact computation in the sense of guaranteeing correct decisions is, at least in principle, possible for many geometric problems. It is, however, also known to be expensive in terms of performance. In this section we discuss how parameterization opens the way for affordable efficient exact computation.

In many cases, geometric algorithms are already described in layers. Elementary tasks are encapsulated in geometric primitives. As described above, implementations of geometric algorithms in CGAL are generic with respect to the concrete implementation of geometric primitives. A traits class specifies the concrete types that are actually used. Most interesting with respect to the precision and robustness issue are geometric predicates, e.g. checking whether three points form a left turn. Naturally, the correctness of the overall algorithm depends on the correctness of the primitive operations.

CGAL provides generic implementations of such primitive operations that allow one to exchange the arithmetic. Both currently available CGAL kernels, `CGAL_Cartesian` and `CGAL_Homogeneous`, are parameterized by a number type. The actual requirements on the number type parameter vary with the primitive. For example, division operations are avoided in most primitives for the homogeneous kernel. Thus, a number type need not have a division operation in order to be used in an instantiation of a template for such a geometric primitive. For the majority of the primitives in CGAL, the basic arithmetic operations $+, -, *$, and $/$ suffice. For some other primitives, a number type must also provide a square root operation. A number type model must not only provide the arithmetic operations in the syntactically correct form, the available arithmetic operations

must also have the correct semantics. Concerning the syntax of the operations on a number type, operator overloading is assumed. This makes the code highly readable.

Strictly speaking, a number type should be a model for the mathematical concept of a real number, just in accordance with the assumptions made in the abstract machine the geometric algorithm was developed for. However, there is no such valid model. Fortunately, in practice, the numerical input data for a geometric primitive are not arbitrary real numbers, but restricted to some subset of the real numbers, e.g. the set of real numbers representable by an `int` or a `double`. In such a situation, a concrete model for a number type is useful, if it provides the correct semantics for the relevant arithmetic operations restricted to the possible set of operands. Less strictly speaking, it suffices if the primitive computes the correct result.

C++ has several built-in number type models. There are signed and unsigned integral types and floating-point types. In general, they all fail to be valid models for the mathematical concepts of integers and real numbers, respectively. Due to rounding errors and over- or underflow, basic laws of arithmetic are violated. There are also various software packages [11, 2] and libraries [16, 14] that provide further models of number types, e.g. arbitrary precision integral types. A very useful model for geometric computations is the number type `leda_real` [4, 16]. This type models a subset of algebraic numbers. `leda_real`s subsume (arbitrary precision) integers and are closed under addition, subtraction, multiplication, division, and $k$-root operations. Invisible to the user, this number type uses adaptive computation to speed up exact comparison operations.

In order to be useful as a library component, there should be a handy description of the set of permissible inputs, for which an instantiation of the template code of a geometric primitive works. For example, a description like 'Instantiated with `double`, this primitive gives the correct result for all those inputs where rounding errors do not lead to incorrect decisions during the evaluation of the primitive' is by no means useful. For number types guaranteeing exact decisions, a useful description is easy, however. Admittedly, there is no satisfactory solution to this issue for potentially inexact number types yet.

Choosing a more powerful number type is only one way to get reliable primitive operations for CGAL algorithms. An alternative is to make special implementations for the primitives, which use evaluation strategies different from the default templates. A number of techniques for exact implementation of geometric primitives have been suggested, e.g. [8, 27]. Alternative implementations can be provided within the CGAL kernel framework as explicit specializations of the template primitives for certain number types. Furthermore, primitives can be implemented independently from the CGAL kernel and can be used directly in traits class models. Finally, existing geometry kernels can be easily used with CGAL algorithms. For example, there are adaptations of the floating-point and the rational geometry kernel of LEDA.

The generic design offers a lot of flexibility. Figure 1 shows a bar chart with (typical running times for) various geometry kernels in a planar convex hull

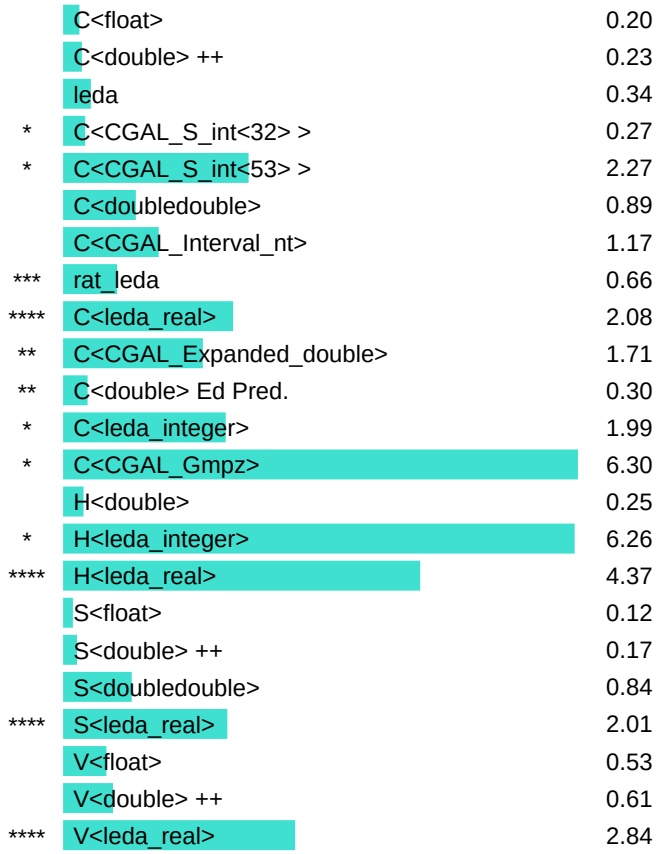


**Fig. 1.** Running times of a CGAL implementation of the Graham-scan algorithm with different geometry kernels. The kernels labelled `C< >` and `H< >` are instantiations of the Cartesian and homogeneous CGAL kernels resp., with the number type argument in angle-brackets. Both use reference counting [17]. The goal is to speed up copying operations at the cost of an indirection in accessing data. The kernels labelled `S< >` use Cartesian coordinates as well but no reference counting. The kernels labelled `V< >` are similar to the `S< >` kernels, but more Java-like: all access functions are virtual and not inlined. Finally, `leda` and `rat_leda` are adaptations of the floating-point and rational geometry kernels resp. of LEDA. In the `C<double>++` kernel, some primitives are specialized and use a slightly more robust computation in the primitives. Furthermore, the code for number type `CGAL_S_int<N>` is explicitly specialized. The primitives for `CGAL_S_int<N>` assume that the Cartesian coordinates, which are internally maintained as `doubles`, are integers with at most `N` bits. In the specializations, a static floating point filter [9] is used. The filter is based on the assumption of the integrality and the size of the numbers. If the filter fails, primitives are re-evaluated with arbitrary precision integer arithmetic. `C<double> Ed Pred` uses special primitives based on [27]. CGAL provides different inlining policies. Here, for all CGAL kernels, the level of inlining was increased with respect to the default in the current release.

computation. The geometry kernels differ not only in their efficiency, but also in their effectiveness with respect to exact geometric computation. The number of stars on the left in Fig. 1 is an indicator for the power of the kernels to produce correct results in the predicates. No star means potentially unreliable predicates. In the kernels with one star, the predicates are reliable, if the numerical data passed to the predicates are integers (of bounded size). The predicates in a kernel with two stars also make correct decisions for numerical data represented as double precision floating-point numbers.[3] The reliability of the kernels with one or two stars is based on the assumption that the numerical data passed to the predicates are exact. With cascaded computations, this assumption might not hold anymore. The `rat_leda` geometry kernel has three stars, because it gives correct result even with cascaded rational computations. Since the number type `leda_real` guarantees exact decisions for a superset of the rationals as described above, all kernels parameterized with number type `leda_real` have four stars. There are significant differences in the performance, ranging from fairly slow arbitrary precision integer arithmetic via fast, exact kernels based on adaptive evaluation to very fast, but potentially unreliable floating-point arithmetic. For further discussion of the various kernels and their performance we refer to [26].

## 7 Conclusions

The use of the generic programming paradigm is the main source of the flexibility of CGAL. Whereas in other geometry libraries there is a dovetailing between the geometric algorithms and the types on which they operate (including container types), in CGAL, algorithms are parameterized by the types on which they operate (not only by container types) such that these types can be exchanged. We have shown that the parameterization allows one to easily combine a CGAL algorithm with existing non-CGAL types, e.g. geometric types provided by a GIS or a CAD system. Furthermore, it allows one to apply a CGAL algorithm to a related geometric problem via geometric transformation. Finally, it allows an advanced user to tailor a CGAL algorithm to a particular practical context by exploiting features that do not hold in general, but hold in the given context. For example, this allows one to gain efficiency, especially for exact computation. The number of geometric algorithms in CGAL is still limited, so a major future task is to add generic implementations of further geometric algorithms. This includes finding suitable abstractions for the primitives they use.

## References


1. J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry*. Cambridge University Press, UK, 1998. translated by H. Brönnimann.
2. K. Briggs. The doubledouble home page. `http://epidem13.plantsci.cam.ac.uk/~kbriggs/doubledouble.html`.


[3] provided that neither overflow nor underflow occurs in the internal computation.

3. H. Brönnimann, S. Schirra, and R. Veltkamp, editors. *CGAL Reference Manuals.* CGAL consortium, 1998. `http://www.cs.uu.nl/CGAL`.
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. K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.
6. M. de Berg, M. van Kreveld, M. Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, 1997.
7. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the computational geometry algorithms library. Research Report MPI-I-98-1-007, Max-Planck-Institut für Informatik, 1998.
8. S. Fortune. Numerical stability of algorithms for 2D Delaunay triangulations and Voronoi diagrams. *Int. J. Computational Geometry and Appl.*, 5:193–213, 1995.
9. S. Fortune and C. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, 1996.
10. J. E. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry.* CRC Press LLC, Boca Raton, FL, 1997.
11. T. Granlund. *GNU MP, The GNU Multiple Precision Arithmetic Library*, 2.0.2 edition, June 1996.
12. C. M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 22(3):31–41, March 1989.
13. L. Kettner. Circulators. In H. Brönnimann, S. Schirra, and R. Veltkamp, editors, *CGAL Reference Manual. Part 3: Support Library.* 1998.
14. LiDIA-Group, Fachbereich Informatik, TH Darmstadt. *LiDIA Manual A library for computational number theory*, 1.3 edition, April 1997.
15. K. Mehlhorn. *Multi-dimensional Searching and Computational Geometry*, volume 3 of *Data Structures and Algorithms.* Springer-Verlag, Heidelberg, Germany, 1984.
16. K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA User manual*, 3.7 edition, 1998. `http://www.mpi-sb.mpg.de/LEDA/leda.html`.
17. S. Meyers. *More Effective C++.* Addison-Wesley, 1996.
18. K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1994.
19. D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, 1996.
20. N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.
21. J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.
22. M. H. Overmars. Designing the computational geometry algorithms library CGAL. *Applied Computational Geometry*, Lect. Notes in Comp. Science Vol. 1148, 1996, pages 53–58.
23. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.
24. J. R. Sack and J. Urrutia, editors. *Handbook on Computational Geometry.* Elsevier Science Publishers, Amsterdam, The Netherlands, 1999.
25. S. Schirra. Precision and robustness issues in geometric computation. In *Handbook on Computational Geometry.* Elsevier, Amsterdam, The Netherlands, 1999.
26. S. Schirra. A case study on the cost of geometric computing. *Algorithm Engineering and Experimentation.* Lect. Notes in Comp. Science Vol. 1619, pages 156–176.
27. J. R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.
28. C. K. Yap. Towards exact geometric computation. *Computational Geometry: Theory and Applications*, 7(1-2):3–23, 1997.