Generic Geometric Programming in the Computational Geometry Algorithms Library

Remco C. Veltkamp

Utrecht University, Dept. Computer Science, Padualaan 14, 3584 CH Utrecht, The Netherlands
Remco.Veltkamp@cs.uu.nl

Abstract

This paper describes a number of design issues and programming paradigms that affect the development of CGAL*, the computational geometry algorithms library. Genericity of the library is achieved by concepts such as templates, iterators, and traits classes. This allows the application programmer to plug in own types of containers and point types, for example. The paper gives an explanation of these concepts and examples of how they are used.*

1. Introduction

Geometric algorithms are used in many application domains. People in areas like computer graphics, robotics, geographic information systems and computer vision are more and more realizing that concepts and algorithms from computational geometry can be of importance for their work. Computational geometry is the subarea of algorithms design that deals with the design and analysis of algorithms for geometric problems involving objects like points, lines, polygons, and polyhedra. The field has, over the past twenty years, developed a rich collection of solutions to a huge variety of geometric problems including intersection problems, visibility problems, and proximity problems. See textbooks [1, 2, 3, 4, 5] or a handbook [6] for an overview. The standard approach taken in computational geometry is the development of exact, provably good and efficient solutions to problems.

However, implementing these algorithms is not easy. As a result, many useful geometric algorithms haven't found their way into practice yet. 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 of explicit handling of degenerate cases in these papers, and the inherent complexity of many efficient solutions. Therefore, the computational geometry community itself has started to develop a well-designed library: CGAL, the Computational Geometry Algorithms Library [7].

The CGAL library contains a number of different parts. The elementary part of the library (the kernel) consists of primitives, constant storage size geometric objects (such as points, lines, and spheres) and predicates on them (such as orientation test for points, intersection tests). The next part of the library contains a number of standard geometric algorithms and data structures such as convex hull, smallest enclosing circle, and triangulation. The last part of the library consists of a support library for example for I/O, visualization, and random generators.

2. Design issues

Rather than building a 'geometric gems' repository, where everybody can contribute to, we decided to design a library. Although the Graphics Gems Repository [8] is a big success, the code must often be changed in order to adapt it to the users' own needs. With CGAL we want to provide a foundation for application programs that is sufficiently generic to be usable in many different areas without the need to adapt the code. Developing such a foundation must be done in a consistent way in order to make it reliable, efficient, open, etc. To be able to cope with the complexity of the developing process, we have decided to build a consistent library by a limited number of institutes.

CGAL is developed for different groups of users, both in academia and in industry. There are the researchers working in computational geometry itself who want to use the library to more easily implement and test their own algorithms. There are users in related areas, with substantial knowledge of computational geometry who want to use geometric algorithms in their application areas. There are developers with little computational geometry knowledge, who want to use CGAL in, possibly commercial, applications. All these groups of users have rather different demands. To please all of them, we have made a number of design decisions, some of which are described below.

2.1. Robustness

Especially in the field of computational geometry, robustness of software is of vital importance. In geometric algorithms, many decisions are based on geometric predicates. If these predicates are not computed correctly (for example due to round-off errors), the algorithm may easily give incorrect results. There are different notions of robustness [9, 10]. For some algorithms, strategies exist to deal with inexact predicates. However, in general this is very difficult to achieve. One way to deal with robustness problems is to perform exact arithmetic, which implies exact geometric computation [11]. This completely removes any robustness problem.

2.2. Generality

The applications of the CGAL library will be very heterogeneous, with very different requirements. To make the library as general as possible, C++ templates (parameterized data types) are heavily used. This enables the user to choose an appropriate number type for doing computations and to choose the representation type of points and other geometric primitives. It is even possible to replace a CGAL data type with a user defined one.

2.3. Efficiency

A computational geometry library must be efficient to be really useful. Whenever possible, the most efficient version of an algorithm is used. Clearly, a library algorithm cannot be the best solution for every application. Therefore, sometimes multiple versions of an algorithm are supplied. For example, this will be the case if dealing with degenerate cases is expensive, or when for a specific number type a more efficient algorithm exists (in which case it will be implemented as a C++ specialization). Another (C++ level) decision that has been made in favor of efficiency is that geometric objects do not share a common base class with virtual methods.

2.4. Ease of use

Generality and ease of use are not always easy to combine. The abundant use of templates seems to make the library difficult to use for people who just want to do something simple with it. Through C++ typedefs, the use of templates can be effectively hidden to the novice user. It is not possible to guarantee that the user will never see templates at all (for example the templates will sometimes become visible in error messages of the compiler or during low level debugging). Developing computational geometry applications is in general very difficult because of problems with inaccuracies and degeneracies. To handle a number of these problems, the algorithms in the library contain many pre and postcondition checks. By setting a compiler flag, these checks will be performed, which can be a great help when debugging an application.

2.5. Visualization

Functionality for visualization is not part of the geometric objects themselves. Naturally, it is not an intrinsic property of, say, a sphere, that it can be converted to OpenGL format. In providing functionality for visualization, we aim for uniformity and openness. Uniformity means that all kinds of visualization is approached in the same way. This is achieved through expressing I/O in terms of C++ streams in a support library. Openness means that any user can supply other streams for other visualization tools than provided by the CGAL support library [12].

Apart from the above mentioned design issues, there are several others that apply to a library like CGAL [12, 13, 11]. In order to meet those different goals, CGAL is set up in a very generic way. Algorithms in CGAL are generic, they work with a variety of implementations of predicates and representations of geometric objects. This allows to easily interchange components as long as they have the same syntax and semantics. Genericity could have been achieved through inheritance and virtual functions. However, we opted for writing template code as it has the advantage that it doesn't cause runtime overhead. The next sections present the use of templates, operator overloading, the Standard Template Library, and traits classes in CGAL.

3. Templates

The template mechanism of C++ allows to write code that is parameterized by types. The template parameters are place-holders for types [14]. C++ allows to write class templates as well as function templates. Templates can be used to avoid duplication of code of classes and functions, for instance when code only differ by the underlying arithmetic.

An example of generic code in the CGAL kernel is the way different representations and arithmetics can be used by means of templates. One can choose between representations of the geometric objects based on representation of points by Cartesian coordinates and representations based on homogeneous coordinates. For both representations you can choose various number types. For example, an application programmer can define a Cartesian representation class instantiated by the C++ built-in type `double`:

```
typedef CGAL_Cartesian <double> RepClass;
```

Alternatively, you could define a homogeneous representation class instantiated by the LEDA type `integer`:

```
typedef CGAL_Homogeneous <integer> RepClass;
```

The number type `integer` provided by LEDA represent arbitrary large integers [15]. Homogeneous coordinates allows to reduce many computations to calculations over the integers, since divisions can be avoided.

Next, geometric primitives can be declared with the chosen representation class, for example a three-dimensional vector:

```
typedef CGAL_Vector_3 <RepClass> MyVector;
MyVector vec1;
```

Within the CGAL kernel, there exists an interface class for each of the primitives, such as for example the three-dimensional vector `CGAL_Vector_3`, and the three-dimensional affine transformation `CGAL_Aff_transformation_3`, which are template parameterized with a representation class `R`:

```
template <class R>
class CGAL_Vector_3 : public R::Vector_3
{ ... }

template <class R>
class CGAL_Aff_transformation_3 :
      public R::Aff_transformation_3
{ ... }
```

The representation class parameter `R` can be instantiated with the available classes `CGAL_Cartesian` or `CGAL_Homogeneous`. Within both representation classes, `Vector_3` is mapped onto an implementation class. For `CGAL_Cartesian`, this implementation class of the vector is called `CGAL_VectorC3`:

```
template<class NT>
class CGAL_Cartesian
{
  public:
      typedef CGAL_VectorC3<NT> Vector_3;
      typedef CGAL_Aff_transformationC3<NT>
                  Aff_transformation_3;
}
```

The implementation classes remain hidden for the application programmer, who just works with `MyVector`. The representation classes themselves are template parameterized by a number type `NT`, for example the C++ `double`, the LEDA type `integer`, or number types from the Gnu Multiple Precision Arithmetic Library [16].

The reason to use representation classes this way, is to have all types and operations that depend on a particular mathematical representation defined within a single class. A user could define own representation classes, for example based on other coordinate systems (such as polar, complex, or Pleucker coordinates), or other primitive representation schemes (such as implicit functions, or NURBS).

4. Operator overloading

Points, vectors, and plane equations behave differently under affine transformations [17]. We use the strong typing and operator overloading mechanism of C++ to hide the details of different transformation properties in the transformation class. For example, a point can be translated, but a vector cannot. The three-dimensional affine transformation `CGAL_Aff_transformation_3<R>`, defines the operator () on a point and a vector separately:

```
template <class R>
class CGAL_Aff_transformation_3 :
      public R::Aff_transformation_3
{
  CGAL_Point_3<R>
  operator()(const CGAL_Point_3<R> &p) const
  {
    return R::Aff_transformation_3::transform(p);
  }

  CGAL_Vector_3<R>
  operator()(const CGAL_Vector_3<R> &v) const
  {
    return R::Aff_transformation_3::transform(v);
  }
}
```

For `CGAL_Cartesian<NT>` as the representation class `R`, `transform` is mapped to the `transform` of `CGAL_Aff_transformationC3<NT>`, which implements the transformation of a point as a full matrix multiplication, but the transformation of a vector ignores the translational part.

Additionally, type checking detects the incorrect use of operations such as the addition of two points, instead of the addition of a point and a vector. No type cast is provided to convert between points p and vectors v. Instead, a symbolic origin o is introduced, and the expressions $v = p - o$ and $p = o + v$ are legal. Internally, operator overloading maps these calls to hidden type casts for efficiency.

5. Standard Template Library

STL, the Standard Template Library [18], is part of the forthcoming C++ standard, and free implementations of it are available [19]. Its main components are containers, algorithms, iterators, and function objects. STL provides a framework and a programming paradigm which was adopted by the CGAL project.

An iterator is some kind of pointer to an object in a container (e.g. an array, or a list). Iterators must satisfy a number of requirements. Any object that satisfies these requirements is an iterator. It must be possible to go to the next element (advance the iterator) and to get to the object to which the iterator points (dereferencing). For C++ arrays the iterators are just pointers. For other types of containers, it should be possible to obtain iterators to the first and to one position beyond the last element. There is a limited number of different sets of requirements, defining different types of iterators: forward, bidirectional, random access, input, and output iterator. Because of the prescribed syntax and semantics that an iterator must obey, it is possible to write just one implementation of an algorithm that works for all iterators providing some minimal functionality. The algorithm then traverses through containers using iterators. This allows to write algorithms that operate on containers that are not yet defined but will provide the required iterators.

For example, a range of points can be fed into a convex hull algorithm by providing two forward iterators `first` and `beyond`. They must define a range `[first,beyond)`, which means that 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.

In the following example, the first and past-the-end iterator of two containers, an array and an STL vector, are fed into the CGAL convex hull algorithm:

```
typedef CGAL_Point_2 < RepClass > Point;

Point points1[num];          // array of Point-s
vector<Point> points2(num);  // vector of Point-s
Point ch1[num], ch2[num];    // output array

// fill points1 and points2
...

CGAL_convex_hull_points_2(
          points1, points1+num, ch1);
CGAL_convex_hull_points_2(
          points2.begin(), points2.end(), ch2);
```

An application programmer need not copy the data from his favorite container to another container in order to feed it to the algorithm, as long as his container provides iterators. Inside the algorithm `CGAL_convex_hull_points_2`, the points are addressed by advancing the iterator, independent of the particular container type:

```
template <class ForwardIterator,
          class OutputIterator>
OutputIterator CGAL_convex_hull_2(
                   ForwardIterator first,
                   ForwardIterator beyond,
                   OutputIterator  result)
{
  for ( ; first != beyond; ++first )
  { ... }

  return result;
}
```

6. Circulators

For inherently circular structures, such as the convex hull vertices of a triangulation, CGAL provides so-called circulators. Circulators are similar to iterators, but there is no past-the-end value, because of the circularity. A container providing circulators has no `end()`-method, only a `begin()`-method. For a circulator c, the range $[c, c)$ denotes the sequence of all elements in the data structure. By contrast, for iterators this range would be empty. A separate test for an empty sequence has been added to the requirements of a circulator: for a circulator `c`, `c==NULL` tests whether the data structure is empty or not. Just like iterators, there are different types of circulators, depending on the requirements they satisfy: forward, bidirectional, and random access circulators.

For example, the method `convex_hull()` of the class `CGAL_Delaunay_triangulation_2` returns a circulator to walk on the convex hull. If `delaunay` is a properly declared and initialized Delaunay triangulation, and `start` and `cur` are circulators of the right type, the following code demonstrates a typical use of circulators:

```
start = delaunay.convex_hull();

if (start != NULL) // convex hull not empty
{ cur = start;
  do
  {
    cout << *cur++;
  } while (cur != start);
}
```

Because $[c, c)$ denotes a full range if c is a circulator, but an empty range if it is an iterator, adaptors are provided to convert circulators to iterator. The forward and bidirectional iterator adaptors keep track of the number of rounds a circulator has done around

the circular data structure. It is zero for the begin-iterator and one for the end-iterator. It is incremented whenever a circulator passes the begin position. Two iterators are equal if their underlying circulators and number of rounds are equal. This is more general than necessary since the end-iterator is not supposed to advance further, which is still possible here in a well defined way. The random access iterator is not as flexible: it cannot compute in constant time the number of rounds that an addition operation implies, since it doesn't know the size of the circular data structure. The random access iterator adaptor therefore expects that indexing assume no modulo computation [20].

7. Traits classes

Suppose you are ray casting a three-dimensional polyhedron, and want to test if some bounding volume is intersected by a ray, in order to avoid unnecessary complex intersection tests with the whole polyhedron. Rather than using an axes parallel bounding box, you can use a tighter bounding volume in the following way. The vertices of the polyhedron are transformed by a viewing transformation that maps the perspective viewing volume to a parallel canonical viewing volume [21]. By the same transformation, each ray is mapped to a ray parallel to the z-axis, but with different origins, see Figure 1. The transformed vertices and the ray origin are projected onto the xy-plane, and it is tested if the projected ray origin falls inside the convex hull of the projected vertices. In this way, the three-dimensional intersection test is reduced to a two-dimensional inside-polygon test.

The convex hull is calculated in the two-dimensional plane, but the polyhedron vertices are three-dimensional. Of course we could make a version of the convex hull algorithm that operates on three-dimensional points, but generates the two-dimensional points. This is against the principle of code reuse, one of the hallmarks of generic programming. Alternatively, we could perform the projection explicitly, generate two-dimensional points, apply the two-dimensional convex hull 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.

However, the convex hull algorithm treats points in an abstract way, using predicates like $Less\_xy(p, q)$ to test if the Cartesian coordinates of p are lexicographically smaller than q, where the x-coordinates are compared first (and if they are equal, then the y-coordinates are compared). So, the algorithm can be parameterized by point type and predicates. CGAL does this: the point type and predicates are put into a so-called traits class `Traits`. The real signature of the convex hull algorithm is:

```
template <class ForwardIterator,
          class OutputIterator,
          class Traits>
OutputIterator CGAL_convex_hull_2(
                ForwardIterator first,
                ForwardIterator beyond,
                OutputIterator  result,
                const Traits&   ch_traits)
```

Rather than explicitly working with `CGAL_Point_2` as the point type, the convex hull algorithm internally works with `Traits::Point_2`. Likewise, instead of using `CGAL_lexicographically_xy_smaller (CGAL_Point_2<R> p, CGAL_Point_2<R> q)` as the $Less\_xy(p, q)$ predicate, it uses a function object which is accessible through a member function `less_xy()` of the traits class. This `less_xy()` returns a function object, to which the ()-operator can be applied: `less_xy()(Traits::Point_2 p, Traits::Point_2 q)`.

```
template <class ForwardIterator,
          class OutputIterator,
          class Traits>
OutputIterator CGAL_convex_hull_2(
                ForwardIterator first,
                ForwardIterator beyond,
                OutputIterator  result,
                const Traits&   ch_traits)
{
  typedef Traits::Point_2  Point_2;

  for ( ; first != beyond; ++first )
  { ... } // using ch_traits.less_xy()(p,q)

  return result;
}
```

The use of a member function `less_xy()` to access the predicate allows to pass additional data from the traits object to the predicate, see below.

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

```
struct My_less_xy
{
   bool operator()(My_point *p, My_point *q) const
   {
      if (...) return TRUE;
      else if (...) return TRUE;
           else return FALSE;
   }
};
```

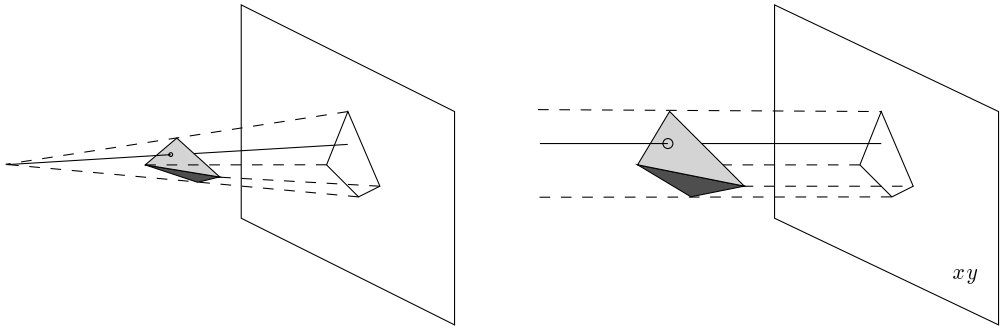

Figure 1: *Transformation from perspective viewing to a parallel viewing.*

```
struct My_traits
{
  typedef My_point    *Point_2;
  typedef My_less_xy  Less_xy;
  Less_xy less_xy() const
  {
    return Less_xy();
  }
} my_traits;
```

The member function `less_xy()` calls the constructor `Less_xy()`, which returns the function object of type `Less_xy`, which is type `My_less_xy`. Data could be passed to the constructor in order to influence the predicat's behavior.

The transformation of the points can be done implicitly in `Less_xy`. For computational efficiency, it could also be done explicitly once for every point, storing the result in a look-up table inside the traits class. The convex hull algorithm can be invoked with this traits class:

```
vector<My_point> points(num); // My-point vector
vector<My_point> ch(num);     // output vector

CGAL_convex_hull_points_2(
       points.begin(), points.end(),
       back_inserter(ch), my_traits);
```

There is no need to explicitly provide a traits class. In Section 5, the algorithm used a default traits class. In CGAL there are default traits classes for all template traits class parameters. Only when the programmer wants to do something special, a traits class needs to be explicitly provided. For a more extensive example of using traits classes in CGAL, see the CGAL getting started document [22].

8. Conclusions

Template parameterization of geometric objects, data structures, and algorithms by a class that maps basic geometric objects to representation (Cartesian, homogeneous) and number types (`double`, `integer`, etc.) allows to write generic code at a higher level. In this way, the application programmer can choose to use fast floating point arithmetic, or exact arithmetic with homogeneous coordinates, for example, changing just one line of code.

Algorithms and data structures are fully generic in the sense that they do not depend on the concrete physical representation of the underlying primitives. To enhance adaptability and reusability, CGAL algorithms and data structures are built on an abstract view of the primitives, through the use of templates, iterators, and traits classes. This allows the application programmer to plug in a container type or application point type that is not yet known at the time of developing the algorithm.

In this way, the CGAL library meets design goals such as robustness, efficiency, and flexibility. The library is continuously being extended with additional algorithms and data structures that are designed in the same way, so as to maintain a uniform look and feel.

Acknowledgment

This work is supported by the ESPRIT IV LTR Projects No. 21957 (CGAL) and No. 28155 (GALIA).

The CGAL library is developed by a consortium of ETH Zürich (Switzerland), Freie Universität Berlin (Germany), INRIA Sophia-Antipolis (France), Martin-Luther-Universität Halle-Wittenberg (Germany), Max-Planck-Institut für Informatik Saarbrücken (Germany), RISC Linz (Austria), Tel Aviv University (Israel), and Utrecht University (The Netherlands). For a list of persons who contributed, see the CGAL manual pages.

An earlier version of this paper has been presented

at the sixth Eurographics workshop on Programming Paradigms in Graphics, Budapest, Hungary, 8 September 1997.

References

1. F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. Springer, (1985).
2. K. Mulmuley, *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, (1993).
3. J. O'Rourke, *Computational Geometry in C*. Cambridge University Press, (1994).
4. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry, Algorithms and Applications*. Springer, (1997).
5. J. D. Boissonnat and M. Yvinec, *Algorithmic Geometry*. Cambridge University Press, (1998).
6. J. E. Goodman and J. O'Rourke, *Handbook of Discrete and Computational Geometry*. CRC Press LLC, (1997).
7. "The Computational Geometry Algorithms Library." URL `http://www.cs.uu.nl/CGAL/`, email address `cgal@cs.uu.nl`.
8. "Graphics Gems repository." URL `http://www.acm.org/tog/GraphicsGems/`.
9. T. Dey, K. Sugihara, and C. Bajaj, "Delaunay triangulations in three dimensions with finite precision arithmetic", *Computer Aided Geometric Design*, **9**, pp. 457–470 (1992).
10. S. Fortune, "Stable maintenance of point-set triangulations in two dimensions", in *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pp. 494–499, (1989).
11. S. Schirra, "Designing a computational geometry algorithms library", in *Lecture Notes for Advanced School on Algorithmic Foundations of Geographic Information Systems, CISM, Udine, September 16-20*, (1996).
12. A. Fabri, G. J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr, "The CGAL kernel: A basis for geometric computation", in *Proceedings of the 1st ACM Workshop on Applied Computational Geometry*, vol. 1148 of *Lecture Notes in Computer Science*, Springer, (1996).
13. M. H. Overmars, "Designing the computational geometry algorithms library CGAL", in *Proceedings of the 1st ACM Workshop on Applied Computational Geometry*, vol. 1148 of *Lecture Notes in Computer Science*, Springer, (1996).
14. S. B. Lippman and J. Lajoie, *The C++ primer, 3rd ed.* Addison-Wesley, (1998).
15. K. Mehlhorn, S. Näher, and C. Uhrig, *The LEDA user manual*. URL `http://www.mpi-sb.mpg.de/LEDA`.
16. T. Granlund, *The Gnu Multiple Precision Arithmetics Library*. URL `http://www.gnu.org/software/gmp/gmp.html`.
17. K. Turkowski, "Properties of surface-normal transformations", in *Graphics Gems* (A. S. Glassner, ed.), Academic Press, (1990).
18. D. R. Musser and A. Saini, *STL Tutorial and Reference Guide: C++ programming with the standard template library*. Addison Wesley, (1996).
19. "Standard Template Library." URL `ftp://butler.hpl.hp.com/stl/`, URL `http://www.sgi.com/Technology/STL/`.
20. A. Fabri and L. Kettner, "The use of STL and STL extensions in CGAL", (January 1999). CGAL 1.2 distribution, URL `http://www.cs.uu.nl/CGAL/`.
21. J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics, Principles and Practice, 2nd ed.* Addison Wesley, (1990).
22. G. J. Giezeman, R. C. Veltkamp, and W. Wesselink, "Getting started with CGAL", (January 1999). CGAL 1.2 distribution, `http://www.cs.uu.nl/CGAL/`.