Generic Programming and The CGAL Library

Efi Fogel, Monique Teillaud

Generic Programming and The CGAL Library

Efi Fogel* Monique Teillaud†

This was first published as Appendix of book
"Effective Computational Geometry for Curves and Surfaces",
Jean-Daniel Boissonnat and Monique Teillaud (Eds),
Springer-Verlag, Mathematics and Visualization, 2006

1 The CGAL Open Source Project

Several research groups in Europe had started to develop small geometry libraries on their own in the early 1990s. A consortium of eight sites in Europe and Israel was founded to cultivate the labour of these groups and gather their produce in a common library called CGAL — the Computational Geometry Algorithms Library [2].

The goal of CGAL was to promote the research in Computational Geometry and translate the results into useful, reliable, and efficient programs for industrial and academic applications, the very same goal that governs CGAL developers to date. In fact, CGAL meets two recommendations of the Computational Geometry Impact Task Force Report [5, 6], which was published roughly when CGAL came to existence: production and distribution of usable (and useful) geometric software was a key recommendation, which came with the need for creating a reward structure for implementations in academia.

An INRIA startup, GEOMETRY FACTORY,[1] was founded on January 2003. The company sells CGAL commercial licenses, support for CGAL, and customized developments based on CGAL.

In November 2003, when Version 3.0 was released, CGAL became an Open Source Project, allowing new contributions from various resources. Common parts of CGAL (i.e., the so-called *kernel* and *support* libraries) are now distributed under the GNU Lesser General Public License (or GNU LGPL for short) and the remaining part (i.e., the *basic* library) is distributed under the terms of the Q Public License (QPL).

The implementations of the CGAL software modules described in this book are complete and robust, as they handle all degenerate cases. They rigorously adapt the *generic programming paradigm*, briefly reviewed in the next section to

*School of Computer Science, Tel-Aviv University
†INRIA

[1] `http://www.geometryfactory.com/`.

overcome problems encountered when effective computational geometry software is implemented. *Geometric programming* is discussed in the succeeding section. Finally, a glimpse at the structure of CGAL is given in the concluding section.

2 Generic Programming

Several definitions of the term *generic programming* have been proposed since it was first coined about four decades ago along with the introduction of the LISP programming language. Since then several approaches have been put into trial through the introduction of new features in existing computer languages, or even new computer languages all together. Here we confine ourself to the classic notion first described by David Musser, Alexander Stepanov, Deepak Kapur, and collaborators, who considered generic programming as a discipline that consists of the gradual lifting of concrete algorithms abstracting over details, while retaining the algorithm semantics and efficiency [13].

One crucial abstraction supported by all contemporary computer languages is the subroutine (also known as procedure or function, depending on the programming language). Another abstraction supported by C++ is that of abstract data typing, where a new data type is defined together with its basic operations. C++ also supports object-oriented programming, which emphasizes on packaging data and functionality together into units within a running program, and is manifested in hierarchies of polymorphic data types related by inheritance. It allows referring to a value and manipulating it without needing to specify its exact type. As a consequence, one can write a single function that operates on a number of types within an inheritance hierarchy. Generic programming identifies a more powerful abstraction (perhaps less tangible than other abstractions), making extensive use of C++ class-templates and function-templates. It is a formal hierarchy of abstract requirements on data types referred to as *concepts*, and a set of classes that conform precisely to the specified requirements, referred to as *models*. Models that describe behaviours are referred to as *traits* classes [14]. Traits classes typically add a level of indirection in template instantiation to avoid accreting parameters to templates.

A generic algorithm has two parts: the actual instructions that describe the steps of the algorithm, and a set of requirements that specify which properties its argument types must satisfy. The following *swap* function is an example of the first part of a generic algorithm.

```
template <class T> void swap(T & a, T & b) {
  T tmp = a; a = b; b = tmp;
}
```

When the function call is compiled, it is instantiated with a data type that must have an assignment operator. A data type that fulfils this requirement is a model of a concept commonly called *Assignable* [4]. The instantiated data type

must also model the concept *CopyConstructible*. The *int* data type, for example, is a model of these two concepts associated with the template parameter T. Thus, it can be used to instantiate the function template [4].[2]

There are many data types that model both concepts *Assignable* and *CopyConstructible* in conjunction, as they consist of only a single requirement each. Thus, it is hard to refer to any of its models as a traits. On the other hand, consider an imaginary generic implementation of a data structure that handles geometric arrangements. Its prototype is listed below. The *Arrangement_2* class must be instantiated with a class that must in turn define a type that represents a certain family of curves, and some functions that operate on curves of this family.

```
template <class Traits> class Arrangement_2 {
  // the code
};
```

It is natural to refer to a model of this concept as a traits class. One important objective is to minimize the set of requirements the traits concept imposes. A tight traits concept may save tremendously in analysis and programming of classes that model the concept. Another important reason for reaching the minimal requirements is to avoid computing the same algebraic entity in different ways. The importance of this is amplified in the context of computational geometry, as a non tight model that consists of duplicate, but slightly different, implementations of the same algebraic entity, can lead to superficial degenerate conditions, which in turn can drastically increase running times.

An algorithm implemented according to the standard object-oriented paradigm alone may resort to use dynamic cast to achieve flexibility, is enforced to have tight coupling through the inheritance relationship, may require additional memory for each object to accommodate the virtual-function table-pointer, and adds for each call to a virtual member function an indirection through the virtual function table. An algorithm implemented according to the generic programming paradigm does not suffer from these disadvantages. The set of requirements on data types is not tied to a specific C++ language feature. Therefore it might be more difficult to grasp. In return, a generic implementation gains stronger type checking at compile time and a higher level of flexibility, without loss of efficiency. In fact, it may expedite the computation. Many articles and a few books have been written on the subject. We refer the reader to [4] for a complete introduction.

The prime example of generic programming was STL, the C++ Standard Template Library, that became part of the C++ standard library in 1994. Since then a few other generic-programming libraries emerged. The most notable in our context were LEDA (Library of Efficient Data Types and Algorithms), a library of combinatorial and geometric data types and algorithms [3, 12], and

[2] See `http://www.sgi.com/tech/stl/` for a complete specification of the SGI STL.

CGAL [2, 8], the Computational Geometry Algorithms Library. Early development of LEDA started in 1988, ten years before the first public release of CGAL became available. While LEDA is mostly a large collection of fundamental graph related and general purpose data structures and algorithms, CGAL is a collection of large and complex data structures and algorithms focusing on geometry.

A noticeable influence on generic programming is conducted by the Boost online community, which encourages the development of free C++ software gathered in the Boost library collection [1]. It is a large set of portable and high quality C++ libraries that work well with, and are in the same spirit as, the C++ Standard Library. The Boost Graph Library (BGL), which consists of generic graph algorithms, serves a particularly important role in our context. It can be used for example to implement the underlying topological data structure of an arrangement instance, that is, a model of the concept *ArrangementDcel*; see [9] for more details. Using some generic programming techniques, an arrangement instance can be adapted as a BGL graph, and passed as input to generic algorithms already implemented in the BGL, such as the Dijkstra shortest path algorithm.

3 Geometric Programming and CGAL

Implementing geometric algorithms and data structures is notoriously difficult, as transforming such algorithms and data structures into effective computer programs is a process full of pitfalls. However, the last decade has seen significant progress in the development of software for computational geometry. The mission of such a task, which Kettner and Näher [11] call *geometric programming*, is to develop software that is correct, efficient, flexible (namely adaptable and extensible[3]), and easy to use.

The use of the *generic programming paradigm* enables a convenient separation between the topology and the geometry of data structures. This is a key aspect, for example, of the design of CGAL polyhedra, CGAL triangulations, and CGAL arrangements (explored in [9]) .

This way algorithms and data structures can be nicely abstracted in combinatorial and topological terms, regardless of the specific geometry and algebra of the objects at hand. This abstraction is realized through class and function templates that represent specific data structures and algorithmic frameworks, respectively. The main class or function template that implements a data structure or an algorithm is typically instantiated with yet another class, referred to as a *traits* class, that defines the set of geometric objects and operations on them required to handle a concrete type of objects.

Generic programming is a key ingredient of flexibility. Changing only the traits class allows for instance to reuse the generic 2D class `Delaunay_triangulation_2` to compute a terrain in 3D: the traits class `Triangulation_euclidean_traits_xy_3` defines point sites as 3D points, and computes the elementary predicates on the

[3] *Adaptability* refers to the ability to incorporate existing user code, and *extendibility* refers to the ability to enhance the software with more code in the same style.

first two coordinates only.

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;

typedef CGAL::Delaunay_triangulation_2<Kernel> Delaunay;
// the kernel Kernel defines the orientation and the in_circle
// tests on 2D points

typedef CGAL::Triangulation_euclidean_traits_xy_3<Kernel> Traits;
typedef CGAL::Delaunay_triangulation_2<Traits> Terrain;
// the traits class Traits defines the orientation and the in_circle
// tests on the 2D projections of the 3D points
```

An immediate advantage of the separation between the topology and the geometry of data structures is that users with limited expertise in computational geometry can employ the data structure with their own special type of objects. They must however supply the relevant traits class, which mainly involve algebraic computations. A traits class also encapsulates the number types used to represent coordinates of geometric objects and to carry out algebraic operations on them. It encapsulates the type of coordinate system used (e.g., Cartesian, Homogeneous), and the geometric or algebraic computation methods themselves. Naturally, a prospective user of the package that develops a traits class would like to face as few requirements as possible in terms of traits development.

Another advantage gained by the use of generic programming is the convenient handling of numerical issues to expedite exact geometric computation. In the classic computational-geometry literature two assumptions are usually made to simplify the design and analysis of geometric algorithms: First, inputs are in "general position". That is, degenerate input (e.g., three curves intersecting at a common point) is precluded. Secondly, operations on real numbers yield accurate results (the "real RAM" model [15], which also assumes that each basic operation takes constant time). Unfortunately, these assumptions do not hold in practice, as numerical errors are inevitable. Thus, an algorithm implemented without keeping this in mind may yield incorrect results (see [10, 16] for examples).

In a geometric algorithm each computational step is either a construction step or a conditional step based on the result of a predicate. The former produces a new geometric object such as the intersection point of two segments. The latter typically computes the sign of an expression used by the program control. Different computational paths lead to results with different combinatorial characteristics. Although numerical errors can sometimes be tolerated and interpreted as small perturbations in the input, they may lead to invalid combinatorial structures or inconsistent state during a program execution. Thus, it suffices to ensure that all predicates are evaluated correctly to eliminate inconsistencies and guarantee combinatorially correct results.

Exact Geometric Computation (EGC), as summarized by Yap [17], simply amounts to ensuring that we never err in predicate evaluations. EGC represents a significant relaxation from the naive concept of numerical exactness. We only need to compute to sufficient precision to make the correct predicate evaluation. This has led to the development of several techniques such as precision-driven computation, lazy evaluation, adaptive computation, and floating point filters, some of which are implemented in CGAL, such as numerical filtering. Here, computation is carried out using a number type that supports only inexact arithmetic (e.g., double floating point), while applying a filter that indicates whether the result is exact. If the filter fails, the computation is re-done using exact arithmetic.

Switching between number types and exact computation techniques, and choosing the appropriate components that best suit the application needs, is conveniently enabled through the generic programming paradigm, as it typically requires only a minor code change reflected in the instantiating of just a few data types.

4 CGAL Contents

CGAL is written in C++ according to the *generic programming* paradigm described above. It has a common programming style, which is very similar to that of the STL. Its Application Programming Interface (API) is homogeneous, and allows for a convenient and consistent interfacing with other software packages and applications. .

The library consists of about 500,000 lines of code divided among approximately 150 classes. CGAL also comes with numerous examples and demos. The manual has about 3,000 pages. There are roughly 50 chapters that are grouped in several parts for a rough description.

The first part is the kernels [7], which consist of constant size non-modifiable geometric primitive objects and operations on these objects. The objects are represented both as stand-alone classes that are instantiated by a kernel class, and as members of the kernel classes. The latter option allows for more flexibility and adaptability of the kernel.

In addition, CGAL offers a collection of basic geometric data structures and algorithms such as convex hull, polygons and polyhedra and operations on them (Boolean operations, polygon offsetting), 2D arrangements, 2D and 3D triangulations, Voronoi diagrams, surface meshing and surface subdivision, search structures, geometric optimization, interpolation, and kinetic data structures. These data structures and algorithms are parameterized by traits classes, that define the interface between them and the primitives they use. In many cases, the kernel can be used as a traits class, or the kernel classes provided in CGAL can be used as components of traits classes for these data structures and algorithms.

The third part of the library consists of non-geometric support facilities, such as circulators, random generators, I/O support for debugging and for interfacing

CGAL with various visualization tools. This part also provides the user with number type support.

CGAL kernel classes are parameterized by number types. Instantiating a kernel with a particular number type is a trade-off between efficiency and accuracy. The choice depends on the algorithm implementation and the expected input data to be handled. Number types must fulfil certain requirements, so that they can be successfully used by the kernel code. The list of requirements establishes a concept of a number type. A few number-type concepts have been introduced by CGAL, e.g., *RingNumberType* and *FieldNumberType*. Naturally, number types have evident semantic constraints. That is, they should be meaningful in the sense that they approximate some subfield of the real numbers. CGAL provides several models of its number-type concepts, some of them implement techniques to expedite exact computation mentioned in the previous paragraph. CGAL also provides a glue layer that adapts number-type classes implemented by external libraries as models of its number-type concepts.

The above describes the accessibility model of CGAL at the time this book was written. Constant and persistent improvement to the source code and the didactic manuals, review of packages by the Editorial board and exhaustive testing, through the years led to a state of excellent quality internationally recognized as an unrivalled tool in its field. At the time these lines are written, CGAL already has a foothold in many domains related to computational geometry and could be found in many academic and research institutes as well as commercial entities. Release 3.1 was downloaded more than 14.500 times, and the public discussion list counts more than 950 subscribed users.

Acknowledgements

The development of CGAL was supported by two European Projects CGAL and GALIA during three years in total (1996–1999). Several sites have kept on working on CGAL after the European support stopped.

The new European project ACS (Algorithms for Complex Shapes with certified topology and numerics)[4] provides again partial support for new research and developments in CGAL.

References

[1] BOOST, C++ libraries.
`http://www.boost.org`.

[2] CGAL, the Computational Geometry Algorithms Library.
`http://www.cgal.org`.

[4] `http://acs.cs.rug.nl/`.

[3] LEDA, Library for efficient data types and algorithms. `http://www.algorithmic-solutions.com/enleda.htm`.

[4] M. H. Austern. *Generic Programming and the STL*. Addison Wesley, 1999.

[5] B. Chazelle et al. Application challenges to computational geometry: CG impact task force report. Technical Report TR-521-96, Princeton Univ., Apr. 1996.

[6] B. Chazelle et al. Application challenges to computational geometry: CG impact task force report. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 407–463. American Mathematical Society, Providence, 1999.

[7] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 191–202. Springer-Verlag, 1996.

[8] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Softw. – Pract. Exp.*, 30(11):1167–1202, 2000.

[9] E. Fogel, D. Halperin, L. Kettner, M. Teillaud, R. Wein, and N. Wolpert. Arrangements. In J.-D. Boissonnat and M. Teillaud, editors, *Effective Computational Geometry for Curves and Surfaces*. Springer-Verlag, Mathematics and Visualization, 2006.

[10] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. In *Proc. 12th European Symposium on Algorithms*, volume 3221 of *Lecture Notes Comput. Sci.*, pages 702–713. Springer-Verlag, 2004.

[11] L. Kettner and S. Näher. Two computational geometry libraries: LEDA and CGAL. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 65, pages 1435–1463. CRC Press LLC, Boca Raton, FL, second edition, 2004.

[12] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[13] D. A. Musser and A. A. Stepanov. Generic programming. In *Proc. Intern. Symp. on Symbolic and Algebraic Computation, LNCS 358*, pages 13–25. Springer-Verlag, 1988.

[14] N. Myers. Traits: A new and useful template technique. *C++ Report*, 7(5):32–35, 1995.

[15] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[16] S. Schirra. Robustness and precision issues in geometric computation. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 14, pages 597–632. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[17] C. K. Yap. Robust geomtric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapman & Hall/CRC, 2nd edition, 2004.