An Adaptable and Extensible Geometry Kernel

Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, Michael Seel

An Adaptable and Extensible Geometry Kernel

Susan Hert[1], Michael Hoffmann[2], Lutz Kettner[3], Sylvain Pion[4], and
Michael Seel[1]

[1] Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
{hert,seel}@mpi-sb.mpg.de
[2] Institute for Theoretical Computer Science, ETH Zurich
CH-8092 Zurich, Switzerland
hoffmann@inf.ethz.ch
[3] University of North Carolina at Chapel Hill, USA
kettner@cs.unc.edu
[4] INRIA, Sophia Antipolis - France
Sylvain.Pion@sophia.inria.fr

Abstract. Geometric algorithms are based on geometric objects such
as points, lines and circles. The term *Kernel* refers to a collection of rep-
resentations for constant-size geometric objects and operations on these
representations. This paper describes how such a geometry kernel can be
designed and implemented in C++, having special emphasis on adapt-
ability, extensibility and efficiency. We achieve these goals following the
generic programming paradigm and using templates as our tools. These
ideas are realized and tested in CGAL [9], the Computational Geometry
Algorithms Library.

Keywords: Computational geometry, library design, generic program-
ming.

1 Introduction

Geometric algorithms that manipulate constant-size objects such as circles, lines,
and points are usually described independent of any particular representation of
the objects. It is assumed that these objects have certain operations defined on
them and that simple predicates exist that can be used, for example, to compare
two objects or to determine their relative position. Algorithms are described in
this way because all representations are equally valid as far as the correctness
of an algorithm is concerned. Also, algorithms can be more concisely described
and are more easily seen as being applicable in many settings when they are
described in this more generic way.

We illustrate here that one can achieve the same advantages when implement-
ing algorithms by encapsulating the representation of objects and the operations
and predicates for the objects into a geometry kernel. Algorithms interact with
geometric objects only through the operations defined in the kernel. This means
that the same implementation of an algorithm can be used with many different

representations for the geometric objects. Thus, the representation can be chosen to be the one most applicable (*e.g.*, the most robust or most efficient) for a particular setting.

Regardless of the representation chosen by a particular kernel, it cannot hope to satisfy the needs of every application. For example, for some applications one may wish to maintain additional information with each point during the execution of an algorithm or one may wish to apply a two-dimensional algorithm to a set of coplanar points in three dimensions. Both of these things are easily accomplished if the kernel is implemented to allow types and operations to be redefined, that is, if the kernel is easily adaptable. It is equally important that a kernel be extensible since some applications may require not simply modifications of existing objects and operations but addition of new ones.

Although adaptability and extensibility are important and worthwhile goals to strive for, one has to keep in mind that the elements of the kernel form the very basic and fundamental building blocks of a geometric algorithm built on top. Hence, we are not willing to accept *any* loss in efficiency on the kernel level. Indeed, using template programming techniques one can achieve genericity without sacrifying runtime-performance by resolving the arising overhead during compile-time.

After discussing previous work on the design of geometry kernels (Section 2), we give a general description of our new kernel concept (Section 3). We then describe how this concept can be realized in an adaptable and extensible way under the generic programming paradigm [21,22] (Sections 4 through 7). Section 8 illustrates the use of such a kernel and shows how the benefits described above are realized. Finally, we describe the models of this type of kernel that are provided in CGAL (Section 9).

As our implementation is in C++ [8], we assume the reader is familiar with this language; see [2,17,26] for good introductions.

2 Motivation and Previous Work

Over the past 10 years, a number of geometry libraries have been developed, each with its own notion of a geometry kernel. The C++ libraries PLAGEO and SPAGEO [15] provide kernels for 2- and 3-dimensional objects using floating point arithmetic, a class hierarchy, and a common base class. The C++ library LEDA [20] provides in its geometry part two kernels, one using exact rational arithmetic and the other floating point arithmetic. The Java library GEOMLIB [3] provides a kernel built in a hierarchical manner and designed around Java interfaces. None has addressed the questions of easily exchangeable and adaptable kernels.

Flexibility is one of the cornerstones of CGAL [9], the *Computational Geometry Algorithms Library*, which is being developed in a common project of several universities and research institutes in Europe and Israel. The recent overview [13] gives an extensive account of functionality, design, and implementation techniques in the library. Generic programming is one of the tools used to achieve this flexibility [6,21,22].

In the original design of the geometry kernel of CGAL [12], there was a representation class which encapsulates how geometric objects are represented. These

representation classes could be easily exchanged or extended, and they provided some limited adaptability. However, the design did not allow the representation classes to also include geometric operations. This extension was seen as desirable after the introduction of *geometric traits classes* into the library, which separate the combinatorial part of an algorithm or data structure from the underlying geometry. The term traits class was originally introduced by Myers [23]; we use it here to refer to a class that aggregates (geometric) types and operations. By supplying different traits classes, the same algorithm can be applied to different kinds of objects. The fact that the existing CGAL kernel did not present its functionality in a way that was immediately accessible for the use in traits classes was one motivation for this work. Factoring out common requirements from the traits classes of different algorithms into the kernel is very helpful in maintaining uniform interfaces across a library and maximizing code reuse.

While the new design described here is even more flexible and more powerful than the old design, it maintains backwards compatibility. The kernel concept now includes easily exchangeable functors in addition to the geometric types; the ideas of traits classes and kernel representations have been unified. The implementation is accomplished by using a template programming idiom similar to the Barton-Nackman technique [4,10] that uses a derived class as a template argument for a base class template. A similar idiom has been used in CGAL to solve cyclic template dependencies in the halfedge data structure and polyhedral surface design [19].

3   The Kernel Concept and Architecture

A geometry kernel K consists of types used to represent geometric objects and operations on these types. Since different kernels will have different notions of what basic types and operations are required, we do not concern ourselves here with listing the particular objects and operations to be included in the kernel. Rather, we describe the kernel concept in terms of the interface it provides for each object and operation.

Depending on one's perspective, the expected interface to these types and operations will look somewhat different. From the point of view of an imperative-style programmer, it is natural that the types appear as stand-alone classes and the operations as global functions or member functions of these classes.

```
K::Point_2   p(0,1), q(1,-4);
K::Line_2    line(p, q);
if (less_xy_2(p, q)) { ... }
```

However, from the point of view of someone implementing algorithms in a generic way, it is most natural, indeed most useful, if types and operations are both provided by the kernel. This encapsulation allows both types and operations to be adapted and exchanged in the same manner.

```
K k;
K::Construct_line_2  c_line   = k.construct_line_2_object();
K::Less_xy_2               less_xy  = k.less_xy_2_object();
K::Point_2                 p(0,1);
K::Point_2                 q(1,-4);
K::Line_2                  line = c_line(p, q);
if (less_xy(p, q)) { ... }
```

The concept of a kernel we introduce here includes both of these perspectives. That is, each operation is represented both as a type, an instance of which can be used like a function, and as a global function or a member function of one of the object classes. The techniques described in the following three sections allow both interfaces to coexist peacefully under one roof with a minimal maintenance overhead, and thus lead to a kernel that presents a good face to everyone.

Our kernel is constructed from three layers, illustrated in Figure 1. The bottom layer consists of basic numeric primitives such as the computation of matrix determinants and the construction of line equations from point coordinates. These numeric primitives are used in the geometric primitives that constitute the second layer of our structure. The top layer then aggregates and assimilates the geometric primitives. The scope of our kernel concept is representation-independent affine geometry. Thus the concept includes, for example, the construction of a point as the intersection of two lines but not its construction from x and y coordinates.
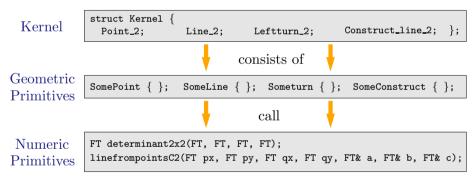

Fig. 1. The kernel architecture

4   An Adaptable Kernel

We present our techniques using a simplified example kernel. Consider the types `Point_2` and `Line_2` representing two-dimensional points and lines, respectively, and an operation `Construct_line_2` that constructs a `Line_2` from two `Point_2` arguments. In general, one probably needs more operations and possibly more types in order to be able to do something useful, but for the sake of simplicity we will stay with these four items for the time being.

A first question might be: `Construct_line_2` has to construct a `Line_2` from two `Point_2`s; hence it has to know something about both types. How does it

get to know them? Since we are talking about adaptability, just hard-wiring the corresponding classnames is not what we would like to do.

A natural solution is to parameterize the geometric classes with the kernel. As soon as a class knows the kernel it resides in, it also knows all related classes and operations. A straightforward way to implement this parameterization is to supply the kernel as a template argument to the geometric classes.

```
template < class K > struct MyPoint { ... };
template < class K > struct MyLine { ... };
template < class K > struct MyConstruct { ... };
```

Then our kernel class looks as follows.

```
struct Kernel {
  typedef MyPoint< Kernel >      Point_2;
  typedef MyLine< Kernel >       Line_2;
  typedef MyConstruct< Kernel >  Construct_line_2;
};
```

At first, it might look a bit awkward; inserting a class into its own components seems to create cyclic references. Indeed, the technique we present here is about properly resolving such cyclic dependencies.

Let us come back to the main theme: adaptability. It should be easy to extend or adapt this kernel and indeed, all that needs to be done is to derive a new class from `Kernel` where new types can be added and existing ones can be exchanged.

```
struct New_kernel : public Kernel {
  typedef NewPoint< New_kernel >   Point_2;
  typedef MyLeftTurn< New_kernel > Left_turn_2;
};
```

The class `Point_2` is overwritten with a different type and a new operation `Left_turn_2` is defined. But there is a problem: the inherited class `MyConstruct` is still parameterized with `Kernel`, hence it operates on the old point class `MyPoint`. What can be done to tell `MyConstruct` that it should now consider itself being part of `New_kernel`?

An obvious solution would be to redefine `Construct_line_2` in `New_kernel` appropriately, *i.e.* by parameterizing `MyConstruct` with `New_kernel`. This is fine in our example where it amounts to just one more typedef, but considering a real kernel with dozens of types and hundreds of operations, it would be really tedious to have to repeat all these definitions.

Fortunately, there is a way out. If `Kernel` is meant as a base for building custom kernel classes, it is not wise to fix the parameterization (this process is called *instantiation*) of `MyPoint<>`, `MyLine<>` and `MyConstruct<>` at that point to `Kernel`, as this might not be the kernel in which these classes finally end up. We rather would like to defer the instantiation, until it is clear what the actual kernel will be. This can be done by introducing a class `Kernel_base` that serves as an "instantiation-engine." Actual kernel classes derive from `Kernel_base` and finally start the instantiation by injecting themselves into the base class.

```
template < class K >
struct Kernel_base {
  typedef MyPoint< K >        Point_2;
  typedef MyLine< K >         Line_2;
  typedef MyConstruct< K >  Construct_line_2;
};
struct Kernel : public Kernel_base< Kernel > {};;
```

In order to be able to extend New_kernel in the same way as Kernel, we can
defer instantiation once again. The construction is depicted in Figure 2.

```
template < class K >
struct New_kernel_base : public Kernel_base< K > {
  typedef NewPoint< K >   Point_2;
  typedef MyLeftTurn< K > Left_turn_2;
};
struct New_kernel : public New_kernel_base< New_kernel > {};;
```
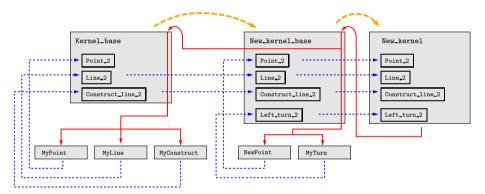

Fig. 2. Deferring instantiation. Boxes stand for classes, thick dashed arrows
denote derivation, solid arrows show (template) parameters, and thin dotted
arrows have to be read as "defines" (typedef or inheritance)

Thus we achieve our easily extensible and adaptable kernel through the use
of the kernel as a parameter at two different levels. The geometric object classes
are parameterized with the kernel such that they have a way of discovering the
types of the other objects and operations. And the kernel itself is derived from a
base class that is parameterized with the kernel, which assures that any modified
types or operations live in the same kernel as the ones inherited from the base
class and there is no problem in using the two together. Note again that this
design does not create any runtime overhead, as the lookup of the correct types
and operations is to be handled during compile time.

5   Functors

The question still remains how we provide the actual functionality needed by
the classes and functions that interact through the kernel. There are a number

of ways functions can be provided in a way that assures adaptability of the kernel. However, efficiency is also very important since many of the predicates and constructions are small, consisting of only a few lines of code. Therefore, the calling overhead has to be minimal.

The classic C-style approach would be to use *pointers to functions*, where adaptability is provided by the ability to change the pointer. *Virtual functions* are the Java-style means of achieving adaptability. In both cases though, there is an additional calling indirection involved; moreover, many compiler optimisations are not possible through virtual functions [27], as the actual types are not known at compile time. This overhead is considerable in our context [24].

The solution we propose is more in line with the standard C++ library [8], where many algorithms are parameterized with so-called *function objects*, or *functors*. A functor is an abstraction of a function; that is, it is anything that behaves as a function and thus, can be used as a function. It is something you can call by using parentheses and passing arguments [17]. Obviously, a function is a functor; but also objects of a class-type that define an appropriate `operator()` can be functors. There are some advantages that make this abstraction worthwhile.

Efficiency If the complete class definition is known at compile time, the `operator()` can be inlined. Handing this function object as a parameter to some other functor is like handing over a piece of code that can be inlined and optimized to the compiler's taste.

Functors Have State Functors also prove to be more flexible; a functor of class-type can carry local data. For example, the functor `Less_int` from above can easily be modified to count the number of comparisons done. Other examples of state in a functor are the binders `binder1st` and `binder2nd` in the STL. They use a local variable to store the value to which one of the two arguments of a binary adaptable functor gets bound.

Allowing local data for a functor adds a complication to the kernel. Clearly, a generic algorithm has to be oblivious to whether a functor carries local state or not. Hence, the algorithm cannot instantiate the functor itself. But we can assume that the kernel knows how to create functors. So we add access member functions to the kernel that allow a generic algorithm to obtain an object for a functor.

6   An Imperative Interface

Someone used to imperative-style programming might expect an interface based on member functions and global functions operating on the geometric classes rather than having to deal with functors and kernel objects. Due to the flexibility in our design, we can easily provide such an interface on top of the kernel with little overhead. For example, there is a global function

```
bool left_turn_2(Point_2 p, Point_2 q, Point_2 r) { ... }
```

which calls the corresponding functor `Left_turn_2` in the kernel where the points p, q and r originate from. Some care has to be taken, to define these functions in a proper way such that they operate on the kernel in a truly generic manner.

Similarly, one might also want to add some functionality to the geometric types; for example a constructor to the line class `MyLine` that takes two point arguments. Again it is important that MyLine does not make assumptions about the point type, but uses only the operations provided by the kernel. This way, the geometric types remain nicely separated, as their – sometimes close – relationships are encapsulated into appropriate operations.

7   A Function Toolbox

Our kernel concept nicely separates the representation of geometric objects from the operations on these objects. But when implementing a specific operation such as `Left_turn_2`, the representation of the corresponding point type `Point_2` will inevitably come into play; in the end, the predicate is evaluated using arithmetic operations on some number type. The nontrivial[1] algebraic computations needed in predicates and constructions are encapsulated in the bottom layer of our kernel architecture (Figure 1), the *number-type-based function toolbox*, which we describe in this section.

A *number type* refers to a numerical type that we use to store coordinates and to calculate results. Given that the coordinates we start with are rational numbers, it suffices to compute within the domain of rational numbers. For certain operations we will go beyond rational arithmetic and require roots. However, since the majority of our kernel requires only rational arithmetic we focus on this aspect here. Depending on the calculations required for certain operations, we distinguish between different concepts of number types that are taken from algebra. A *ring* supports addition, subtraction and multiplication. A *Euclidean ring* supports the three ring operations and an integral division with remainder, which allows the calculation of greatest common divisors used, *e.g.*, to cancel common factors in fractions. In contrast, a *field* type supports exact division instead of integral division.

Many of the operations in our kernel boil down to determinant evaluations, *e.g.*, sidedness tests, in-circle tests, or segment intersection. For example, the left-turn predicate is evaluated by computing the sign of the determinant of a $2{\times}2$ matrix built from differences of the points' coordinates. Since the evaluation of such a determinant is needed in several other predicates as well, it makes sense to factor out this step into a separate function, which is parameterized by a number type to maintain flexibility even at this level of the kernel. This function can be shared by all predicates and constructions that need to evaluate a $2{\times}2$ determinant.

Code reuse is desirable not only because it reduces maintenance overhead but also from a robustness point of view, as it isolates potential problems in a small number of places. Furthermore, these basic numerical operations are equally as accessible to anyone providing additional or customized operations on top of our kernel in the future.

8   Adaptable Algorithms

In the previous sections, we have illustrated the techniques used to realize a kernel concept that includes functors as well as types in a way that makes both

---

[1] beyond a single addition or comparison

easily adaptable. Here we show how such a kernel can be put to good use in the implementation and adaptation of an algorithm.

Kernel as Traits Class In CGAL, the geometric requirements of an algorithm are collected in a geometric traits class which is a parameter of the algorithm's implementation. With the addition of functors to the kernel concept, it is now possible simply to supply a kernel as the argument for the geometric traits class of an algorithm. Consider as a simple example Andrew's variant of Graham's scan [1,11] for computing the convex hull of a set of points in two dimensions. Assuming the points are already sorted lexicographically, this algorithm requires only a point type and a left-turn predicate from its traits class. Hence, the simple example kernel from Section 4 would suffice.

In general, the requirements of many geometric traits classes are only a subset of the requirements of a kernel. Other geometric traits classes might have requirements that are not part of the kernel concept. They can be implemented as extensions on top, having easy access to the part of their functionality that is provided by the kernel.

Projection Traits As mentioned in Section 5, one benefit of using functors in the traits class and kernel class is the possible association of a state with the functor. This flexibility can be used, for example, to apply a two-dimensional algorithm to a set of coplanar points in three dimensions. Consider the problem of triangulating a set of points on a polyhedral surface. Each face of the surface can be triangulated separately using a two-dimensional triangulation algorithm and a kernel can be written whose two-dimensional part realizes the projection of the points onto the plane of the face in all functors while actually using the original three-dimensional data. The predicates must therefore know about the plane in which they are operating and this is maintained by the functors in a state variable.

Adapting a Predicate Assume, we want to compute the convex hull of a planar point set with a kernel that represents points by their Cartesian coordinates of type `double`[2]. The left-turn predicate amounts to evaluating the sign of a 2×2-determinant; if this is done in the straightforward way by calculations with `double`s, the result is not guaranteed to be correct due to roundoff errors caused by the limited precision.

By simply exchanging the left-turn predicate, a kernel can be adapted to use a so-called static filter (see also next section) in that predicate. Assume for example, we know that the coordinates of the input points are `double` values from $(-1, 1)$. It can be shown (cf. [25]) that in this case the correct sign can be determined from the `double` calculation, if the absolute value of the result exceeds $3 \cdot (2^{-50} + 2^{-102}) \approx 2.66 \cdot 10^{-15}$.

9   Kernel Models

The techniques described in the previous sections have been used to realize several models for the geometry kernel concept described in Section 3. In fact, we use class templates to create a whole *family* of models at once. The template

---

[2] A double precision floating point number type as defined in IEEE 754 [16].

parameter is usually the *number type* used for coordinates and arithmetic. We categorize our kernel families according to *coordinate representation*, *object reference and construction*, and *level of runtime optimization*. Furthermore, we have actually two kernel concepts in CGAL: a lower-dimensional kernel concept for the fixed dimensions 2 and 3, and a higher-dimensional kernel concept for arbitrary dimension d. For more details beyond what can be presented here, the reader is referred to the CGAL reference manuals [9].

Coordinate Representation We distinguish between two coordinate representations; Cartesian and homogeneous. The corresponding kernel classes are called `Cartesian<FT>` and `Homogeneous<RT>` with the parameters `FT` and `RT` indicating the requirements for a *field type* and *ring type*, respectively. Homogeneous representation allows many operations to factor out divisions into a common denominator, thus avoiding divisions in the computation, which can sometimes improve efficiency and robustness greatly. The Cartesian representation, however, avoids the extra time and space overhead required to maintain the homogenizing coordinate and thus can also be more efficient for certain applications.

Memory Allocation and Construction The standard technique of *smart pointers* can be used to speed up copy constructions and assignments of objects with a reference-counted handle-representation scheme. Runtime experiments show that this scheme pays off for objects whose size is larger than a certain threshold (around 4 words depending on the machine architecture). To allow for an optimal choice CGAL offers for each representation a simple and a smart-pointer based version. In the Cartesian case, these models are called `Simple_cartesian<FT>` and `Cartesian<FT>`.

Filtered Models The established approach for robust geometric algorithms following the exact computation paradigm [28] requires the exact evaluation of geometric predicates, i.e., decisions derived from geometric computations have to be correct. While this can be achieved straightforwardly by relying on an exact number type [7,18], this is not the most efficient approach, and the idea of so-called *filters* has been developed to speed up the exact evaluation of predicates [5,14,25]. See also the example in Section 8. The idea of filtering is to do the calculations on a fast floating point type and maintain an error bound for this approximation. An exact number type is only used where the approximation is not known to give the correct result for the predicate and the hope is that this happens seldom.

CGAL provides an adaptor `Filter_predicate<>`, which makes it easy to use the filter technique for a given predicate, and also a full kernel `Filtered_kernel<>` with all predicates filtered using the scheme presented above.

Higher-Dimensional Kernel The higher-dimensional kernel defines a concept with the same type and functor technology, but is well separated from the lower-dimensional kernel concepts. Higher-dimensional affine geometry is strongly connected to its mathematical foundations in linear algebra and analytical geometry. Since the dimension is now a parameter of the interface and

since the solution of linear systems can be done in different ways, a linear algebra concept `LA` is part of the interface of the higher dimensional kernel models `Cartesian_d<FT,LA>` and `Homogeneous_d<RT,LA>`.

10   Conclusions

Many of the ideas presented here have already been realized in CGAL; parts of them still need to be implemented. Although standard compliance is still a big issue for C++ compilers, more and more compilers are able to accept template code such as ours.

We would like to remind the reader that in this paper we have lifted the curtain to how to implement a library, which is considerably more involved than using a library. A user of our design can be gradually introduced to the default use of one kernel, then exchanging one kernel with another kernel in an algorithm, exchanging individual pieces in a kernel, and finally – for experts – writing a new kernel. Only creators of a new library need to know all inner workings of a design, but we believe also interested users will benefit from studying the design.

Finally, note that many topics could be touched very briefly only within the scope of this article. The interested reader will find many more details and examples, in particular regarding the implementation, in the full paper.

Acknowledgments

References

1. ANDREW, A. M. Another efficient algorithm for convex hulls in two dimensions. *Inform. Process. Lett. 9*, 5 (1979), 216–219. 87

2. AUSTERN, M. H. *Generic Programming and the STL*. Addison-Wesley, 1998. 80

3. BAKER, J. E., TAMASSIA, R., AND VISMARA, L. GeomLib: Algorithm engineering for a geometric computing library, 1997. (Preliminary report). 80

4. BARTON, J. J., AND NACKMAN, L. R. *Scientific and Engineering C++*. Addison-Wesley, Reading, MA, 1997. 81

5. BRÖNNIMANN, H., BURNIKEL, C., AND PION, S. Interval arithmetic yields efficient dynamic filters for computational geometry. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.* (1998), pp. 165–174. 88

6. BRÖNNIMANN, H., KETTNER, L., SCHIRRA, S., AND VELTKAMP, R. Applications of the generic programming paradigm in the design of CGAL. In *Generic Programming—Proceedings of a Dagstuhl Seminar* (2000), M. Jazayeri, R. Loos, and D. Musser, Eds., LNCS 1766, Springer-Verlag. 80

7. BURNIKEL, C., MEHLHORN, K., AND SCHIRRA, S. The LEDA class real number. Technical Report MPI-I-96-1-001, Max-Planck Institut Inform., Saarbrücken, Germany, Jan. 1996. 88

8. International standard ISO/IEC 14882: Programming languages – C++. American National Standards Institute, 11 West 42nd Street, New York 10036, 1998. 80, 85

9. CGAL, the Computational Geometry Algorithms Library. http://www.cgal.org/. 79, 80, 88

10. COPLIEN, J. O. Curiously recurring template patterns. C++ Report (Feb. 1995), 24–27. 81

11. DE BERG, M., VAN KREVELD, M., OVERMARS, M., AND SCHWARZKOPF, O. Computational Geometry: Algorithms and Applications. Springer-Verlag, Berlin, 1997. 87

12. FABRI, A., GIEZEMAN, G.-J., KETTNER, L., SCHIRRA, S., AND SCHÖNHERR, S. The CGAL kernel: A basis for geometric computation. In Proc. 1st ACM Workshop on Appl. Comput. Geom. (1996), M. C. Lin and D. Manocha, Eds., vol. 1148 of Lecture Notes Comput. Sci., Springer-Verlag, pp. 191–202. 80

13. FABRI, A., GIEZEMAN, G.-J., KETTNER, L., SCHIRRA, S., AND SCHÖNHERR, S. On the design of CGAL, the computational geometry algorithms library. Software – Practice and Experience 30 (2000), 1167–1202. 80

14. FORTUNE, S., AND VAN WYK, C. J. Static analysis yields efficient exact integer arithmetic for computational geometry. ACM Trans. Graph. 15, 3 (July 1996), 223–248. 88

15. GIEZEMAN, G.-J. PlaGeo, a library for planar geometry, and SpaGeo, a library for spatial geometry. Utrecht University, 1994. 80

16. IEEE Standard for binary floating point arithmetic, ANSI/IEEE Std 754 − 1985. New York, NY, 1985. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987. 87

17. JOSUTTIS, N. M. The C++ Standard Library, A Tutorial and Reference. Addison-Wesley, 1999. 80, 85

18. KARAMCHETI, V., LI, C., PECHTCHANSKI, I., AND YAP, C. The CORE Library Project, 1.2 ed., 1999. http://www.cs.nyu.edu/exact/core/. 88

19. KETTNER, L. Using generic programming for designing a data structure for polyhedral surfaces. Comput. Geom. Theory Appl. 13 (1999), 65–90. 81

20. MEHLHORN, K., AND NÄHER, S. LEDA: A Platform for Combinatorial and Geometric Computing. Cambridge University Press, Cambridge, UK, 2000. 80

21. MUSSER, D. R., AND STEPANOV, A. A. Generic programming. In 1st Intl. Joint Conf. of ISSAC-88 and AAEC-6 (1989), Springer LNCS 358, pp. 13–25. 80

22. MUSSER, D. R., AND STEPANOV, A. A. Algorithm-oriented generic libraries. Software – Practice and Experience 24, 7 (July 1994), 623–642. 80

23. MYERS, N. C. Traits: A new and useful template technique. C++ Report (June 1995). http://www.cantrip.org/traits.html. 81

24. SCHIRRA, S. A case study on the cost of geometric computing. In Proc. Workshop on Algorithm Engineering and Experimentation (1999), vol. 1619 of Lecture Notes Comput. Sci., Springer-Verlag, pp. 156–176. 85

25. SHEWCHUK, J. R. Adaptive precision floating-point arithmetic and fast robust geometric predicates. Discrete Comput. Geom. 18, 3 (1997), 305–363. 87, 88

26. STROUSTRUP, B. The C++ Programming Language, 3rd Edition. Addison-Wesley, 1997. 80

27. VELDHUIZEN, T. Techniques for scientific C++. Technical Report 542, Department of Computer Science, Indiana University, 2000. http://www.extreme.indiana.edu/~tveldhui/papers/techniques/. 85

28. YAP, C. K., AND DUBÉ, T. The exact computation paradigm. In Computing in Euclidean Geometry, D.-Z. Du and F. K. Hwang, Eds., 2nd ed., vol. 4 of Lecture Notes Series on Computing. World Scientific, Singapore, 1995, pp. 452–492. 88