

On the design of CGAL, a computational geometry algorithms library

Andreas Fabri¹, Geert-Jan Giezeman², Lutz Kettner³, Stefan Schirra^{4*}, and Sven Schönherr⁵

¹*ABB Baden, Switzerland (email: andreas.fabri@chcrc.abb.ch)*

²*Utrecht University, The Netherlands (email: geert@cs.uu.nl)*

³*ETH Zürich, Switzerland (email: kettner@inf.ethz.ch)*

⁴*Max-Planck-Institut für Informatik, Saarbrücken, Germany (email: stschirr@mpi-sb.mpg.de)*

⁵*Freie Universität Berlin, Germany (email: sven@inf.fu-berlin.de)*

SUMMARY

CGAL is a *Computational Geometry Algorithms Library* written in C++, which is being developed by research groups in Europe and Israel. The goal is to make the large body of geometric algorithms developed in the field of computational geometry available for industrial application. We discuss the major design goals for CGAL, which are correctness, flexibility, ease-of-use, efficiency, and robustness, and present our approach to reach these goals. Generic programming using templates in C++ plays a central role in the architecture of CGAL. We give a short introduction to generic programming in C++, compare it to the object-oriented programming paradigm, and present examples where both paradigms are used effectively in CGAL. Moreover, we give an overview of the current structure of the CGAL-library and consider software engineering aspects in the CGAL-project. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: computational geometry; software library; C++; generic programming;

INTRODUCTION

Geometric algorithms arise in various areas of computer science. Computer graphics and virtual reality, computer aided design and manufacturing, solid modeling, robotics, geographical information systems, computer vision, shape reconstruction, molecular modeling, and circuit design are well-known examples. Research on specific geometric problems in these areas led to the more general study of geometric algorithms in the field of *Computational Geometry*. A lot of efficient geometric methods and data structures have been developed in this subfield of algorithm design over the past two decades. But many of these techniques have not found their way into practice yet. An important cause for this is

*Correspondence to: Stefan Schirra, Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany

the fact that the correct implementation of even the simplest of these algorithms can be a notoriously difficult task [1].

There are two particular aspects that need to be dealt with to close the gap between the theoretical results of computational geometry and practical implementations [2]. First, there is the precision problem. Theoretical papers assume exact arithmetic with real numbers. The correctness proof of the algorithms relies on the exact computations, thus replacing exact arithmetic by imprecise built-in floating-point arithmetic does not work in general.

Second, there is the degeneracy problem. Often, theoretical papers exclude degenerate configurations for the input to the algorithms they describe. Typically, these degeneracies are problem specific and would involve the treatment of special cases in the algorithm. Simple examples of configurations considered as degenerate are duplicate points in a point set or three lines intersecting in one point. For some problems, it is not difficult to handle the degeneracies, but for other problems the special case treatment distracts from the solution of the general problem and it can amount to a considerable fraction of the coding effort, especially if handling of degeneracies is treated as an afterthought [3]. In theory, this approach of excluding degeneracies from consideration is justified by the argument that degenerate cases are very rare in the set of all possible inputs over the real numbers, i.e. they are very unlikely if the input set is randomly chosen over the real numbers. Another argument is that it is first of all important to understand the general case before treating special cases. In practice, however, degenerate inputs do occur frequently. For instance, the coordinates of the geometric objects may not be randomly chosen over the real numbers, but lie on a grid, for example they may be created by clicking in a window in a graphical user interface. In some applications, what are called degeneracies are even high-valued design criterias, for example in architecture, where features of buildings do align on purpose. As a consequence, a library must address the handling of degeneracies.

Besides the precision problem and the degeneracy problem, advanced algorithms bring about the additional difficulty that they are frequently hard to understand and hard to code. For these reasons it is impractical for users to implement geometric algorithms from scratch. To remedy this situation a computational geometry library providing correct and efficient reusable implementations is needed. Such a library, called CGAL, *Computational Geometry Algorithms Library*, is being developed in a common project of several universities and research institutes in Europe and Israel. In this paper we present and discuss the design of this C++ software library.

The sites contributing to CGAL are Utrecht University (The Netherlands), ETH Zürich (Switzerland), Freie Universität Berlin (Germany), Martin-Luther Universität Halle (Germany), INRIA Sophia-Antipolis (France), Max-Planck-Institut für Informatik, Saarbrücken (Germany), RISC Linz (Austria), and Tel-Aviv University (Israel). The participating sites are leading in the field of computational geometry in Europe and have ample experience with the implementation of geometric algorithms [4, 5, 6, 7, 8, 9]. Work on the CGAL-library is the central task of two successive ESPRIT IV LTR projects with the names CGAL and GALIA. It is the goal of these projects to

make the large body of geometric algorithms developed in the field of computational geometry available for industrial application.

The CGAL-library is our key tool in reaching this goal. It will be the basis for implementations of geometric algorithms in cooperative projects with industrial partners. These projects will be the litmus test for the library. Feedback from these cooperations will ensure that CGAL serves industrial needs. Since in the CGAL-project we have to overcome the aforementioned problems arising in the implementation of geometric algorithms as well, implementation effort has to be accompanied by further research on these problems. To select the best solutions for practice, experimentation is needed as well.

Since computational geometry has so many potential application areas with different needs, flexibility of the library components, especially adaptability and modularity of the library, are important design issues for CGAL. Of course, correctness, ease-of-use, and efficiency are design goals of CGAL. Providing a decent amount of functionality that is common and important in computational geometry is another design goal on a long list of many others. Design goals for CGAL are discussed in the third section of this paper.

We decided to design CGAL as a C++-library. C++ supports the designs we wanted, is widely used and is standardized –and compilers are approaching the standard language now. Since CGAL can be seen as part of a more global European effort to provide algorithmic software to enhance the technology transfer to industry, the decision to use C++ was also partially motivated by corresponding decisions for related libraries, e.g. LEDA [6]. C++ is not a panacea. It can hardly be called a simple or elegant language. Eiffel or Smalltalk are more properly object oriented but lack acceptance. At present, Java is considered too slow for industrial strength code. We use the template mechanism of C++ and the generic programming paradigm known from the C++ Standard Template Library (STL) to design a generic and modular library. This approach is not supported by Java.

Through the use of templates and the generic programming paradigm, the code in the library gains a certain independence. The library algorithms and components work with a variety of implementations of predicates and subtasks and geometric objects. This allows one to easily interchange components as long as they have the same interface.

In the next section we regard previous and related work on computational geometry libraries and the roots of CGAL. After discussing the design goals we consider the generic programming paradigm. Then we discuss circulators, an extension of the iterator concept of the Standard Template Library to circular structures. They are useful in the implementation of geometric objects, where circular structures often arise. Subsequently we discuss the structure of CGAL and present the different layers of the library. First we present the kernel, which contains basic (constant-size) geometric objects and primitive operations on these objects. Next we present the basic library, which contains standard geometric algorithms and (non constant-size) geometric structures. Thereafter we look at engineering aspects addressed in the CGAL-project such as manual writing and the separation between specification, implementation, and testing. We conclude with an evaluation of the design.

In the more technical parts of the paper we assume that the reader is familiar with the C++ programming language [10] and the basics of its Standard Template Library.

RELATED WORK

Amenta [11] gives an overview on the state of the art of computational geometry software before CGAL and provides many references. Computational geometry software was intensively discussed at the First ACM Workshop on Applied Computational Geometry [12, 13, 14]. The design of the CGAL-kernel at that time is presented in [15] and the project goals in [14]. A more recent overview can be found in [16]. Precision and robustness aspects of a computational geometry library are discussed in [17]. Further topics on designing combinatorial data structures in CGAL, such as polyhedra, are described in [18].

Many implementations of computational geometry algorithms exist in loosely coupled collections only. Use and combination of such algorithms usually requires some adaptation effort. If well designed, the components of a library work seamlessly together. First implementation efforts for computational geometry libraries were started already at the end of the Eighties [19, 20, 8, 9]. These libraries were integrated into workbenches allowing animation and interaction, but were typically restricted to a particular platform.

To some extent, specifications of components of CGAL have their roots in CGAL's precursors developed by members of the CGAL consortium. To a much lesser extent, CGAL also scavenged implementation techniques from its precursors. These precursors are the XYZ library [8, 9], developed at ETH Zürich, PlaGeo/SpaGeo [5], developed at Utrecht University, C++GAL [4], developed at INRIA Sophia-Antipolis, and the geometric part of LEDA [6, 7], a library for combinatorial and geometric computing, developed at Max-Planck-Institut für Informatik, Saarbrücken.

In the US, an implementation effort with a goal similar to that of the CGAL-project has been started at the Center for Geometric Computing, located at Brown University, Duke University, and John Hopkins University. They state their goal as *an effective technology transfer from computational geometry to relevant applied fields*. Recently they have started working on a computational geometry library called GeomLib [21] implemented in Java.

DESIGN GOALS

Computational geometry has many potential application areas with different needs. As a foundation for application programs CGAL is supposed to be sufficiently generic to be usable in many different areas. We address different kinds of users, both in academia and industry. The users' knowledge of computational geometry or C++ programming will range from novice to expert. To capture the different requirements we have structured them in the following list of primary design goals for the project. There are further important design goals for such a project, such as maintainability, but we consider them as secondary for the project mission statement and do not discuss them here.

Correctness

A library component is correct if it behaves according to its specification. Basically,

correctness is therefore a matter of documentation and the verification that documentation and implementation coincide. However, this is easier said than done. In a modularized program the correctness of a module is determined by its own correctness and the correctness of all the modules it depends on. Clearly, in order to get correct results, correct algorithms and data structures must be used. Usually the correctness of a geometric algorithm has been proven in a theoretical context with simplifying assumptions, such as exact arithmetic or general position assumptions excluding degenerate configurations. See also the design goal *robustness* in the following subsection. If these assumptions, e.g., exact arithmetic, do not hold in practice, the correctness proof is not valid anymore. Accordingly, modules using other modules, e.g. arithmetic modules, do not necessarily yield correct results anymore, if the modules used do not behave according to their specification. Whether assumptions concerning exact computation hold for a concrete problem instance in practice depends on the demand of this instance on the arithmetic. Here, geometric computations impose subtle dependencies on modules that make the combinations of modules intrinsically harder. The demand of geometric computations on the arithmetic has been formalized [22, 23] and studied for a few basic geometric problems [22, 23, 24], but further research on the arithmetic demand as well as on an easy-to-use documentation of this demand is still needed. Ignoring the simplifying assumptions, such as relying on ‘sufficient exactness’ of the built-in arithmetic, would violate our understanding of correctness.

Exactness should not be confused with correctness in the sense of reliability. There is nothing wrong with algorithms computing approximate solutions instead of exact solutions, as long as their behaviour is clearly documented and they do behave as specified. Also, an algorithm handling only non-degenerate cases can be correct with respect to its specification, although in CGAL we would like to provide algorithms handling degeneracies.

In a modularized project structure it is important to test modules independently and as early as possible [25]. One specific technique for quality assurance is the use of assertions of invariants of an algorithm and the self-checking of functions at runtime [26, 27]. They are of great help in the implementation process and can reduce debugging efforts drastically. The user should be able to switch off the checking, e.g. when code goes in production mode.

Robustness

A design goal particularly relevant for the implementation of geometric algorithms is robustness. Many implementations of geometric algorithms lack robustness because of precision problems. Design and correctness proofs of geometric algorithms usually assume exact arithmetic while many implementations simply replace it by imprecise arithmetic. Since imprecise calculations can cause wrong and mutually contradictory decisions in the control flow of an algorithm, many implementations crash or at best compute garbage for some inputs. For some applications the fraction of bad inputs compared to all possible inputs is small, but for other applications this fraction is large. No perfect solution to the precision problem is known, especially with respect to libraries. Primitives based on imprecise computations are hard to combine and therefore less useful as library components. Exact computation is possible for many geometric problems and saves the correctness proof given for a theoretical model of computation to the actual code, but it slows down the computation. CGAL allows

one to choose the underlying arithmetic and delegates the decision of how to balance the trade-off between efficiency and robustness to the user.

Flexibility

The different needs of the potential application areas demand flexibility in the library. Four sub-issues of flexibility can be identified.

Modularity

A clear structuring of CGAL into modules with as few dependencies as possible helps a user in learning and using CGAL, since the overall structure can be grasped more easily and the focus can be narrowed to those modules that are actually of interest. In addition, only those parts of the library could be isolated that are used in a particular situation, which keeps CGAL from being a monolithic library. Instead, CGAL has the flexibility to be used in smaller independent parts. Natural examples are the distinction between two-dimensional and three-dimensional geometry, or separate modules for convex-hull computation and point set triangulation.

Adaptability

CGAL might be used in an already established environment with geometric classes and algorithms. Most probably, the modules will need adaptation before they can be used. An example is the application of the convex-hull algorithm to a user defined point type, which differs from the CGAL point type. The idealistic situation would be like a theoretical paper on a convex-hull algorithm: The algorithm is described once and can be applied to virtually any programming language and point type. Continuing this analogy further, the ideal theoretical paper will typically declare the operations, which are assumed to be available somehow for the point type, and will express the algorithm in terms of these operations. Similarly in the library, the adaptation effort should only influence the declaration of the point type and operations used, not the convex-hull algorithm itself.

Extensibility

Not all wishes can be fulfilled with CGAL. So users may want to extend the library. It should be possible to integrate new classes and algorithms into CGAL. For example, it should be easy to add new geometric classes to the library and to provide corresponding intersection functions similar to those existing for native CGAL classes.

Openness

CGAL should be open to coexist with other libraries, or better, to work together with other libraries and programs. The C++ Standard defines with the C++ Standard Template Library a common foundation for all C++ platforms. So it is easy and natural to gain openness by following this standard. But there are important libraries outside the standard, and CGAL should be easily adaptable to them as well, in particular LEDA [7] with its number types, combinatorial and graph algorithms, the Gnu Multiple Precision Arithmetic Library [28] for a number type, and various visualization systems, some of them standardized.

Ease of Use

Many different qualities can contribute to the ease-of-use of a library and differ according to the experience of the user. The above mentioned correctness and robustness issues are among these qualities. Of general importance is the learning time and how quickly the library becomes useful. Another issue is the amount of new concepts and exceptions of general rules that must be learned and remembered. Ease-of-use tends to conflict with flexibility, but in many situations a solution can be found. The flexibility of CGAL should not distract a novice who takes the first steps with CGAL.

Smooth Learning Curve

One major point of the success story of C++ is its almost complete compatibility with C and the possible smooth transition from C to C++: from the new style of comments, to member functions and inheritance, up to full object-oriented programming. Each newly learned feature could be put into practice immediately. CGAL users are supposed to have a base knowledge of C++ and the STL. The reader of the paper should be aware that there is a tremendous difference between developing a library, such as CGAL, which this paper is about, and the use of such a library, which is usually much simpler. This has been successfully shown with LEDA, and can also be seen with the STL.

CGAL is based in many places on concepts borrowed from STL or the other parts of the C++ Standard Library. An example is the use of streams and stream operators in CGAL. Another example is the use of container classes and algorithms from the STL.

Uniformity

A uniform look-and-feel of the design in CGAL will help in learning and memorizing. A concept once learned should be applicable in all places where one would wish to apply it. A function name once learned for a specific class should not be named differently for another class.

Complete and Minimal Interfaces

Another goal with similar implications as uniformity is a design with complete and minimal interfaces, see for example Item 18 in [29]. An object or module should be complete in its functionality, but should not provide additional decorating functionality. Even if a certain function might look like ease-of-use for a certain class, in a more global picture it might hinder the understanding of similarities and differences among classes, and make it harder to learn and memorize.

Rich and Complete Functionality

We aim for a useful and rich collection of geometric classes, data structures and algorithms. CGAL is supposed to be a foundation for algorithmic research in computational geometry and therefore needs a certain breadth and depth. The standard techniques in the field are supposed to appear in CGAL. Completeness is related to uniformity. Examples are distance and intersection computations that should be available for all appropriate pairs of geometric classes, not only for a subset. However, for certain pairs, the return-type might not fit in the

framework currently available in CGAL, or solutions might not be known yet.

Completeness is also related to robustness. We aim for general purpose solutions that are for example not restricted by assumptions on general positions. Algorithms in CGAL should be able to handle special cases and degeneracies. In those cases where handling of degeneracies turns out to be inefficient, special variants that are more efficient but less general should be provided in the library in addition to the general algorithms handling all degeneracies. Of course, it needs to be clearly documented which degeneracies are handled and which are not.

Efficiency

For most geometric algorithms theoretical results for the time and space complexity are known. Also, the theoretic interest in efficiency for realistic inputs, as opposed to worst case situations, is growing [30]. For practical purposes, insight into the constant factors hidden in the O -notation is necessary, especially if there are several competing algorithms. Ideally, the implementations should be tested with realistic input. But, for a library, it is hard to tell which input is realistic. Therefore, different implementations are supplied if there is not one best solution, for instance, if there is a trade-off between time and space or if there is a more efficient implementation when there are no or few degeneracies. Also, the characteristics of a specific number type may be exploited.

Efficiency is a competing goal with respect to flexibility, robustness, and ease-of-use. One cannot expect a library to perform as well as tailored applications. In some places we sacrifice time and space efficiency in order to optimally meet the other goals, but in many places we were able to avoid compromises with respect to any of our primary design goals. In fact, the techniques used for flexibility in CGAL also enable us to achieve optimal efficiency.

GENERIC AND OBJECT-ORIENTED PROGRAMMING

Basically, two main techniques are available in C++ for realizing our design goal flexibility in CGAL: *Object-oriented programming*, using inheritance from base classes with virtual member functions, and *generic programming*, using class templates and function templates. Both paradigms are also available in other languages, but we stay with the notion used in C++, which is the choice made for CGAL.

The flexibility in the *object-oriented programming paradigm* is achieved with a base class, which defines an interface, and derived classes that implement this interface. Generic functionality can be programmed in terms of the base class and a user can select any of the derived classes wherever the base class is required. The classes actually used may even be determined at runtime and the generic functionality can be implemented without knowing all derived classes beforehand. In C++ so-called virtual member functions and runtime type information support this paradigm. The base class is usually a pure virtual base class.

The advantages are the explicit definition of the interface and the runtime flexibility. There are four main disadvantages: This paradigm cannot provide strong type checking at compile

time whenever dynamic casts are used, which is necessary in C++ to achieve flexibility. This paradigm enforces tight coupling through the inheritance relationship [25], it requires additional memory for each object (in C++ the so-called *virtual function table pointer*) and it adds for each call to a virtual member function an indirection through the virtual function table [31]. The latter is of particular interest when considering runtime performance since virtual member functions can usually not be made *inline* and are therefore not subject to code optimization within the calling function[†]. Modern microprocessor architectures can optimize at runtime, but, besides the difficulty of runtime predictions, these mechanisms are more likely to fail for virtual member functions. These effects are negligible for larger functions, but small functions will suffer a loss in runtime of one or two orders of magnitude. Significant examples are the access of point coordinates and arithmetic for low-dimensional geometric objects, see for example [32]. If the class hierarchy tends to be dense with long derivation chains and maybe even worse with multiple inheritance, the system will be hard to learn, to understand, to test and maintain [25].

The *generic programming paradigm* features what is known in C++ as *class templates* and *function templates*. Templates are incompletely specified components in which a few types are left open and represented by formal placeholders, the *template arguments*. The compiler generates a separate translation of the component with actual types replacing the formal placeholders wherever this template is used. This process is called *template instantiation*. The actual types for a function template are implicitly given by the types of the function arguments at instantiation time. An example is a swap function that exchanges the value of two variables of arbitrary types. The actual types for a class template are explicitly provided by the programmer. An example is a generic list class for arbitrary item types. The following definitions would enable us to use `list<int>`, with the actual type `int` given explicitly, for a list of integers and to swap two integer variables `x` and `y` with the expression `swap(x, y)`, where the actual type `int` is given implicitly.

```
template <class T> class list {
    // ... , placeholder T symbolically represents the item type.
};

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

The example of the swap function illustrates that a template usually requires certain properties of the template arguments, here that variables of type `T` are assignable. An actual type used in the template instantiation must comply with these assumptions in order to obtain a correct template instantiation. We can distinguish between *syntactical requirements*, the assignment operator is needed in our example, and *semantical requirements*, the operator should actually copy the value. If the syntactical requirements are not fulfilled, compilation simply fails.

[†]There are notable exceptions where the compiler can deduce for a virtual member function the actual member function that is called, which allows the compiler to optimize this call. The keyword `final` has been introduced in Java to support this intention. However, these techniques are not realized in C++ compilers so far and they cannot succeed in all cases, though it is arguable that typical uses of CGAL can be optimized. However, distributing the library in precompiled components will hinder their optimization, which must be done at link time.

Semantical requirements cannot be checked at compile time. However, it might be useful to connect a specific semantical requirement to an artificial, newly introduced syntactical requirement, e.g. a tag similar to iterator tags in [33]. This technique allows decisions at compile time based on the actual type of these tags.

The set of requirements that is needed to obtain a correct instantiation of a member function of a class template is usually only a fraction of all requirements for the template arguments of this class template. If only a subset of the member functions is used in an instantiation of a class template, it would be sufficient for the actual types to fulfill only the requirements needed for this subset of member functions. This is possible in C++ since as long as a C++ compiler is not explicitly forced, the compiler is not allowed to instantiate member functions that are not used, and therefore possible compilation errors due to missing functionality of the actual types cannot occur [10]. This enables us to design class templates with optional functionality, which can be used if and only if the actual types used in the template instantiation fulfill the additional requirements.

A good example illustrating generic programming is the Standard Template Library [33, 10, 34, 35]. Generality and flexibility is achieved with the carefully chosen set of *concepts*, where a concept is a well defined set of requirements. In our `swap`-function example, the appropriate concept is named ‘assignable’ and includes the requirement of an assignment operator [35]. If an actual type fulfills the requirements of a concept, it is a *model* for this concept. In our example, `int` is a model of the concept ‘assignable’.

Algorithmic abstraction is a key goal in generic programming [36]. One aspect is to reduce the interface to the data types used in the algorithm to a set of simple and general concepts. One of them is the *iterator* concept, which is an abstraction of pointers. Iterators serve two purposes: They refer to an item and they traverse over the sequence of items that are stored in a data structure, also known as *container class* in STL. Five different categories are defined for iterators: input, output, forward, bidirectional and random-access iterators, according to the different possibilities of accessing items in a container class. The usual C-pointer referring to a C-array is a model for a random-access iterator. *Generic algorithms* are not written for a particular container class in STL, they use iterators instead. For example, sequences of items are specified by two iterators forming a range `[first, beyond)`. This notion of a half-open interval denotes the sequence of all iterators obtained by starting with `first` and advancing `first` until `beyond` is reached, but does not include `beyond`. A container class is supposed to provide a local type, which is a model of an iterator, and two member functions: `begin()` returns the start iterator of the sequence and `end()` returns the iterator referring to the ‘past-the-end’-position of the sequence. For example, a generic `contains` function can be written to work for any model of an input iterator. It returns `true` iff the value is contained in the values of the range `[first, beyond)`.

```
template <class InputIterator, class T>
bool
contains( InputIterator first, InputIterator beyond, const T& value) {
    while ((first != beyond) && (*first != value))
        ++first;
    return (first != beyond);
}
```

The advantages of the generic programming paradigm are strong type checking at compile time during the template instantiation, no need for extra storage or additional indirections during function calls, and full support of inline member functions and code optimization at compile time [37]. One specific disadvantage of generic programming in C++ is the lack of a notation in C++ to declare the syntactical requirements for a template argument, the equivalent of the virtual base class in the object-oriented programming paradigm. The syntactical requirements are scattered throughout the implementation of the template. The concise collection of the requirements is left for the program documentation. In general, the flexibility is resolved at compile time, which gives us the advantages mentioned above, but it can be seen as a disadvantage if runtime flexibility is needed. However, the generic data structures and algorithms can be parameterized with the base class used in the object-oriented programming to achieve the runtime flexibility.

We applied mainly the generic programming paradigm to achieve flexibility and efficiency in CGAL. The compliance with STL is important for us to promote the re-use of existing generic algorithms and container classes, and – more important – to unify the look-and-feel of the design of CGAL with the C++ standard. CGAL is therefore easy to learn and easy to use for those who are familiar with STL. The abstract concepts used in the STL are so powerful that only a few additions and refinements are needed in CGAL. One refinement is the concept of *handles*. Combinatorial data structures might not necessarily possess a natural order on their items. Here, we restrict the concept of iterators to the concept of handles, which is the item denoting part of the iterator concept, and which ignores the traversal capabilities. Any model of an iterator is a model for a handle. A handle is also known as *trivial iterator*. Another refinement is the concept of *circulators*, a kind of iterators with slightly adapted requirements that suit the needs of circular sequences better as they occur naturally in several combinatorial data structures, such as the sequence of edges around a vertex in a triangulation. See the next section for more details on circulators.

In a few places we also made use of the object-oriented programming paradigm, for example the protected access to the internal representation of the polyhedral surface data structure [18], which is no time critical operation compared to the work that is supposed to be performed with the internal representation. Another example is the return-value of the intersection of two polygons, which might contain points, segments, or polygons in general. In CGAL, a polymorphic list is used to return the result of such intersection routines. Note that this does not necessarily imply a common base class for all CGAL classes. In fact, CGAL has no common base class for all objects, and its class hierarchy is very flat, if there is any derivation used at all. Instead, we applied an appropriate design pattern [38], a generic wrapper, as described in the section on the kernel layer. This keeps the influence of this design decision local.

CIRCULATORS

Our new concept of *circulators* reflects in CGAL the fact that combinatorial structures often lead to circular sequences, in contrast to the linear sequences supported with iterators and container classes in the STL. For example polyhedral surfaces and planar maps give rise to the circular sequence of edges around a vertex or around a facet. Implementing iterators for

circular sequences is possible, but not straightforward, since no natural past-the-end situation is available. An arbitrary sentinel in the cyclic order would break the natural symmetry in the configuration, which is in itself a bad idea, and will lead to cumbersome implementations. Another solution stores, within the iterator, a starting edge, a current edge, and a kind of winding-number that is zero for the `begin()`-iterator and one for the past-the-end iterator[‡]. No solution is known to us that would provide a light-weight iterator as it is supposed to be (in terms of space and efficiency). Therefore we introduced in CGAL the slightly different concept of *circulators*, which does allow light-weight implementations. The support library provides adaptor classes that convert between iterators and circulators, thus integrating this new concept into the framework of the STL.

Circulators share most of their requirements with iterators. Three circulator categories are defined: forward, bidirectional and random-access circulators. Given a circulator `c` the operation `*c` denotes the item to which the circulator refers. 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 by `n` where `n` is a natural number. Two circulators can be compared for equality, i.e., whether they refer to the same item.

Circulators develop different notions of reachability and ranges than iterators. A circulator `d` is called *reachable* from `c` if `c` can be made equal to `d` with finitely many applications of the operator `++c`. Due to the circularity of the sequence this is always true if both circulators refer to items of the same sequence. 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` if `d ≠ c`. So far it is the same range definition as for iterators. The difference lies in the use of `[c, c)` for denoting all items in the circular sequence, 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 can be used in STL algorithms. It is possible to write equally simple algorithms that work with iterators as well as with circulators, including the full range definition, see Chapter 3.9 in [39]. An additional test `c == NULL` is now required that is true if and only if the sequence is empty. In this case the circulator `c` is said to have a *singular value*. For the complete description of the requirements for circulators we refer to Chapter 3.7 in [39].

We repeat the example for the generic `contains` function from the previous section for a range of circulators. As usual for circular structures, a `do-while` loop is preferable, so that for the specific input `'c == d'` all elements in the sequence are reached.

```
template <class InputCirculator, class T>
bool
contains( InputCirculator c, InputCirculator d, const T& value) {
    if (c != NULL) {
        do {
            if (*c == value)
                return true;
        } while (++c != d);
    }
    return false;
}
```

[‡]This is currently implemented in CGAL as an adaptor class which provides a pair of iterators for a given circulator.

LIBRARY OVERVIEW

The CGAL-library is made of several modular units. Basically, the library is structured into three layers and a support library, which stands apart. The three layers are the core library with basic non-geometric functionality, the geometric kernel with basic geometric objects and operations, and the basic library with more complicated algorithms and data structures.

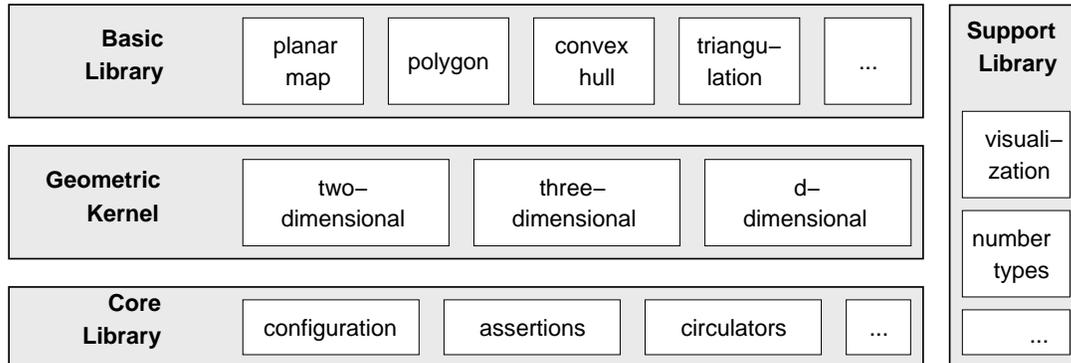


Figure 1. The structure of CGAL.

At a finer level, the layers and the support library are further subdivided into smaller modular units, see Fig. 1. The modular approach has several benefits. For the user, a modular design is easier to grasp because it is possible to understand a small part without having any knowledge of other parts. For building the library, the modules are a good way of organizing. They are used to divide work among the project partners and help to assemble those pieces in a convenient way when a release of the library is made. Testing is also easier when there is little or no coupling between parts [25]. This is discussed in more detail in the section on software engineering.

The geometric kernel contains simple geometric objects, like points, lines, segments, triangles and tetrahedra. The criterion for simplicity is that those objects have a constant size. There are geometric predicates on those objects. Furthermore, there are operations such as computing intersection and distance of objects and affine transformations.

The geometric kernel is split in three parts, which deal with two-dimensional objects, three-dimensional objects, and general-dimensional objects. Geometry in two and three dimensions is well studied and has lots of applications, which is the reason for their special status. For all dimensions there are Cartesian and homogeneous representations.

One thing that is not supplied by CGAL is number types. Deep down, all geometric objects are represented by numbers. The precise way in which computations with those numbers are done is very important. Especially for robustness issues, it is often preferable to use exact arithmetic instead of floating point arithmetic. In order to make it possible to choose a number type, the geometric kernel is parameterized by number types. CGAL does not provide an implementation of number types. Instead CGAL provides the necessary support to use number types from other sources, e.g. from LEDA. This is in line with the philosophy that libraries should be (re-)used where possible. Because the arithmetic operations needed in CGAL are

quite basic, every library that supplies number types can easily be supported in CGAL.

The basic library contains more complex geometric objects and data structures: polygons, planar maps, polyhedra and so on. It also contains algorithms, such as computing the convex hull of a set of points, triangulations, the union of two polygons and so on. The basic library is made of mostly independent parts, independent from each other, but even independent from the kernel.

The first kind of independence is the easiest to obtain and is striven for as much as possible. There are a few dependencies, for example, the algorithm that computes the union of two polygons depends on the part that defines polygons.

The independence from the kernel is harder to obtain, but from a design point of view quite interesting. Every algorithm defines in a very precise way which primitives it uses. This interface is a template parameter of the algorithm and is called a *traits* class. For example, a convex-hull algorithm can take points as input and must be able to decide if one point lies to the left of the other point and to decide when you go from one point via a second point to a third point, if you make a left or a right turn. In this case the algorithm is parameterized by a class that has a point type and the two predicates that work on this point type. The algorithm is implemented in terms of the types and operations of the interface only. As a consequence, no types and operations are hardwired into the basic library algorithms, and in this sense they are independent from the kernel.

The parameterization by traits classes offers great flexibility and modularity. In order to meet another design goal, ease of use, there is always a predefined traits class that uses types and operations of the kernel. Where possible, this traits class is chosen by default, so the user can totally ignore the existence of this mechanism. In this sense the basic library is a layer built on top of the kernel.

Both the core library and the support library deal with things that are not (purely) geometric in nature. The core library offers functionality that is needed in the geometric kernel or the basic library. There is some support for coping with different C++ compilers which all have their own limitations. Here is also basic support for dealing with assertions, preconditions and postconditions. Circulators and random number generators belong here, too.

The support library adds functionality that is not purely geometric and not vital for the rest of the library. Visualization is an important aspect of the support library. There are many languages (VRML, PostScript) and programs (GeomView, LEDA windows) that deal with 2D and 3D visualization. The support library interfaces CGAL objects with existing software. Because there is not a single standard way of doing visualization, it is important that this part is separate from the kernel and basic library. The adaptation of number types from other libraries is in the support library, too.

KERNEL

The geometric part of the CGAL-kernel contains objects of constant size, such as point, vector, direction, line, ray, segment, triangle, iso-oriented rectangle and tetrahedron. Each

type provides a set of member functions, for example affine transformation. Global functions are available for the detection and computation of intersections, as well as for distance computations.

The current CGAL-kernel provides two families of geometric objects in the kernel: One family is based on the representation of points using Cartesian coordinates. The other family is based on the representation of points using homogeneous coordinates. The homogeneous representation extends the representation with Cartesian coordinates by an additional coordinate, which is used as a common denominator. More formally, in d dimensional space, a point with homogeneous coordinates $(x_0, x_1, \dots, x_{d-1}, x_d)$, where $x_d \neq 0$, has Cartesian coordinates $(x_0/x_d, x_1/x_d, \dots, x_{d-1}/x_d)$. This allows one to avoid divisions and hence to reduce many computations involved in geometric algorithms to calculations over the integers. For example, a division of the Cartesian coordinates of a point by an integer n is replaced by multiplying the common denominator by n . Both families are parameterized by the number type used to represent the (Cartesian or homogeneous) coordinates. The class templates for these two families are not directly visible to the user. Instead, class templates with a single template parameter are used:

```
template <class R> CGAL_Point_2;
```

One can choose between the two alternatives using the template argument: For example, one can choose `CGAL_Cartesian<double>` to get objects with Cartesian coordinates of type `double`, the double precision floating-point number type of the C++ programming language, or `CGAL_Homogeneous<int>` to obtain objects with homogeneous coordinates of type `int`. Usually, typedefs are used to introduce conveniently short names for the types, e.g.:

```
typedef CGAL_Point_2< CGAL_Homogeneous<int> > Point_2;
```

The design goal robustness motivated the parameterization by a number type, which allows a user to choose the underlying arithmetic and to influence the precision of the computations. Algorithms in computational geometry are usually designed for a theoretical machine model based on exact arithmetic with arbitrary real numbers (in the mathematical sense). If computations are done with an imprecise number type instead, numerical errors in the evaluation of geometric predicates may lead to incorrect decisions. Hence the control flow of the algorithm may deviate from its theoretical counterpart. The interplay of discrete and numerical aspects in computational geometry allows the possibility that small errors in the evaluation of a predicate may lead to decisions contradicting basic laws of geometry and finally to catastrophic errors.

The parameterization by a number type opens an easy way to overcome such precision and robustness problems by exact computation. For example, the number type `leda_real` of LEDA [7, 40] can be used. This number type models a subset of algebraic numbers: All integers are `leda_reals` and `leda_reals` are closed under the operations $+$, $-$, \cdot , $/$, and $\sqrt[k]{}$. They use adaptive evaluation and guarantee that all comparison operations give the correct result. Thus the use of the `leda_reals` via `CGAL_Cartesian<leda_real>` guarantees exact decisions and therefore exactly the same control flow in the algorithm as in

its theoretical counterpart. No robustness problems can arise due to wrong and inconsistent decisions with the number type `leda_real`.

Furthermore, parameterization by a number type offers flexibility. Besides other exact number types, e.g., arbitrary precision integer number types or rational number types, the fast but potentially imprecise floating-point types `float` and `double` can be used if speed is more important than reliability. Floating-point number types, which allow one to choose the precision of the floating-point system, e.g., `leda_bigfloat` in LEDA, allow one to balance efficiency and accuracy. Small precision leads to faster computation, but might also lead to less accurate results. Especially with homogeneous coordinates, one could also use integer arithmetic with a fixed and sufficiently large precision, large enough for all intermediate results arising in the arithmetic operations, but this requires a certain knowledge of the computations carried out. Thus this kind of adaptation is not easy to use.

CGAL's requirements of a number type are kept small. It is easy to make a number type compliant with CGAL [41]. CGAL already supports the number types of LEDA [7] and the Gnu Multiple Precision Arithmetic Library [28].

The class templates parameterized with `CGAL_Cartesian` or `CGAL_Homogeneous` provide the user with a common interface to the underlying representation, which can be used in higher-level implementations that is independent of the actual coordinate representation. The list of requirements on the template parameter defines the concept of a *representation class* for the CGAL-kernel. A model for this concept of a representation class must essentially provide the names for the actual implementations. For example, the parameter used for the Cartesian types is implemented as follows:

```
template<class NumberType>
class CGAL_Cartesian {
public:
    typedef CGAL_PointC2<NumberType>      Point_2;
    typedef CGAL_VectorC2<NumberType>     Vector_2;
    typedef CGAL_DirectionC2<NumberType>  Direction_2;
    typedef CGAL_SegmentC2<NumberType>    Segment_2;
    typedef CGAL_LineC2<NumberType>       Line_2;
    typedef CGAL_RayC2<NumberType>        Ray_2;
    // ...
};
```

Type names ending in `C2` denote classes based on Cartesian representation with coordinates of number type `NumberType`. The number type is a template parameter of the class template `CGAL_Cartesian`. Each instantiation of `CGAL_Cartesian` with an argument, which fulfills the requirements for number types, is a proper model for a representation class.

The technique used here is known as the '*nested typedefs for name commonality*'-idiom [42, 43]. In particular, the representation class tells the name of an implementation class to the class template used as common interface. The class template inherits the implementation:

```
template <class R>
```

```
class CGAL_Point_2 : public R::Point_2 {
    // ...
};
```

The nested typedefs do provide not only the name of an actual implementation of a type, but also the names of related types. The representation class concept of the kernel and the traits class concepts used in the basic library of CGAL are closely related; both introduce name commonality for basic geometric types.

CGAL provides clean mathematical concepts to the user without sacrificing efficiency. For example, CGAL strictly distinguishes points and (mathematical) vectors, i.e., it distinguishes affine geometry from the underlying linear algebra. Points and vectors are not the same (see [44] for a discussion of illicit computations resulting from identification of points and vectors in geometric computations). In particular, points and vectors behave differently under affine transformations [45]. We do not even provide automatic conversion between points and vectors. The geometric concept of an origin is used instead. The symbolic constant `CGAL_ORIGIN` acts as a point and can be used to compute the locus vector as the difference between a point and the origin. Function overloading is used to implement this operation internally as a simple conversion without any overhead. Note that we do not provide the geometrically invalid addition of two points, since this might lead to ambiguous expressions: Assuming three points p , q , and r and an affine transformation A , one can write in CGAL the perfectly legal expression $A(p + (q - r))$. The slightly different expression $A((p + q) - r)$ contains the illegal addition of two points. But thinking in terms of coordinates, one might expect the same result as when the addition were allowed. However, this is not true, since the expression within the affine transformation would probably evaluate to a vector, not a point as in the previous expression. Vectors and points behave differently under affine transformations. For similar reasons, the automatic conversion is not provided.

A major design decision was to avoid derivation hierarchies of classes and virtual member functions with their tendency to couple classes tightly. This decision was made for the sake of efficiency and flexibility. Virtual member functions with their space and time performance penalties are largely avoided. Wherever necessary, we apply local solutions to get polymorphic behavior. For example, the intersection operations need a polymorphic return-value. The intersection of a line and a segment might be a point, the segment, or empty. It would be convenient to have a common base class for all these possible return-types. In CGAL the return-type of an intersection is a generic object that can contain an object of any type. One sooner or later needs to know what the result type is in order to take appropriate actions. Using the function `CGAL_assign()` one can try to assign the returned object of type `CGAL_Object` to potential return-types. If successful, `CGAL_assign()` returns `true`, otherwise `false`. The example function below calls `CGAL_intersection()` to compute the intersection of a segment and a line. The possible return-types are checked in turn using `CGAL_assign()` and actions corresponding to the detected return type are taken:

```
template <class R>
void foo(CGAL_Segment_2<R> seg, CGAL_Line_2<R> line) {
    CGAL_Object result;
    CGAL_Point_2<R> ipoint;
    CGAL_Segment_2<R> iseg;
```

```

result = CGAL_intersection(seg, line);
if (CGAL_assign(ipoint, result)) {
    // ipoint contains the intersection point.
} else if (CGAL_assign(iseg, result)) {
    // iseg contains the segment as intersection result.
} else {
    // the intersection is empty.
}
}

```

A class hierarchy is used which is not visible to the user. A `CGAL_Object` refers to an instance of a wrapper class containing the actual intersection result. All wrapper classes have a common base class with a virtual destructor. It is therefore possible for the `CGAL_Object` to use a pointer to the base class to refer to the actual instance of the wrapper class.

```

class CGAL_Base { // base class for wrapper classes
public:
    virtual ~CGAL_Base() {}
};
template <class T> // generic wrapper class
class CGAL_Wrapper : public CGAL_Base {
public:
    CGAL_Wrapper(const T& object) : _object(object) {}
    CGAL_Wrapper() {}
    operator T() { return _object; }
    virtual ~CGAL_Wrapper() {}
private:
    T _object;
};
class CGAL_Object { // polymorphic object
public:
    // ...
    CGAL_Base* base() const { return _base; }
private:
    CGAL_Base* _base;
};

```

The `CGAL_assign()` function uses runtime type information to check whether the passed object is of the appropriate type to get the object assigned which is stored in `CGAL_Object`.

```

template <class T>
bool CGAL_assign(T& t, const CGAL_Object& o) {
    CGAL_Wrapper<T>* wp = dynamic_cast<CGAL_Wrapper<T>*>(o.base());
    if (wp == 0)
        return false;
    t = *wp;
    return true;
}

```

The actual CGAL-code is slightly more complicated and simulates runtime type information for compilers not yet supporting it. In those cases, where the intersection of two objects

might consist of several parts of potentially different type, for example the intersection of two polygons, a `list<CGAL_Object>` is returned.

Note that the class hierarchy is only introduced locally. It provides a nice and extensible solution to the return-type problem for intersections while an overall class hierarchy with its performance penalties is still avoided. Class hierarchies are used whenever we felt that they provide a more appropriate solution, for example, in CGAL affine transformations maintain distinct internal representations using a hierarchy: The internal representations differ considerably in their space requirements and the efficiency of their member functions. For all but the most general representation we gain performance in terms of space and time. And for the most general representation, the performance penalty caused by the virtual functions is negligible, because the member functions are computationally expensive for this representation. Alternatively we could have used this most general representation for affine transformations only. But the use of a hierarchy is justified, since the specialized representations, namely translation, rotation and scaling, arise frequently in geometric computing.

Another design decision was to make the (constant-size) geometric objects in the kernel non-modifiable. For example, there are no member functions to set the Cartesian coordinates of a point. Points are viewed as atomic units (see also [46]) and no assumption is made on how these objects are represented. In particular, there is no assumption that points are represented with Cartesian coordinates. They might use polar coordinates or homogeneous coordinates instead, where for example member functions to set the Cartesian coordinates might be expensive and complicated. Nevertheless, in current CGAL the types based on the Cartesian representation as well as the types based on the homogeneous representation have both member functions returning Cartesian coordinates and member functions returning homogeneous coordinates. These access functions are provided to make implementing own predicates and operations more convenient. However, adding the existence of member functions returning Cartesian and homogeneous coordinates to the assumptions on a kernel would be a weakness concerning generality.

Like other libraries [7, 47, 48] we use reference counting for the kernel objects: Objects point to a shared representation. Each representation counts the number of objects pointing to it. Copying objects increments the counter of the shared representation, deleting an object decrements the counter of its representation. If the counter reaches zero by the decrement, the representation itself is deleted, see e.g. Item 29 in [49] for further information. The implementation of reference counting is simplified by the non-modifiability of the objects. However, the use of reference counting was not the reason for choosing non-modifiability. Using ‘copy on write’ (a new representation is created for an object whenever its value is changed by a modifying operation), reference counting with modifiable objects is possible and only slightly more involved.

BASIC LIBRARY

The basic library of CGAL contains more complex geometric objects and data structures, such as polygons, polyhedrons, triangulations (including Delaunay triangulations), planar maps, range and segment trees, and kd-trees. It also contains geometric algorithms, such as

convex hull, smallest enclosing circle, ellipse, and sphere, boolean operations, map overlay, and generators for geometric objects.

We describe in the sequel generic data structures, generic algorithms and traits classes in CGAL.

Generic Data Structures

Following the generic programming paradigm as introduced above, CGAL is made compliant with STL. The interfaces of geometric objects and data structures in the basic library make extensive use of iterators, circulators and handles, so that algorithms and data structures can be easily combined with each other and with those provided by STL and other compliant libraries.

Triangulations are an example of a container-like data structure in the basic library. The interface contains, among others, member functions to access the vertices of the triangulation, for example all vertices or, as another example, the vertices on the convex hull of the triangulation. One way to provide this functionality would be:

```
class Triangulation {
public:
    list<Vertex*>  vertices();
    list<Vertex*>  convex_hull();
    // ...
};
```

There are two main disadvantages. First, the whole sequence of vertex pointers has to be computed at once and copied into a list. Second, the vertex pointers are returned in a specific container, `list<Vertex*>`, but the user may need them in another container, for instance `vector<Vertex*>`, or in no container at all. The following approach avoids the disadvantages of the first sketch.

```
class Triangulation {
public:
    Vertex*  vertex();
    Vertex*  successor( Vertex*);
    Vertex*  predecessor( Vertex*);

    Vertex*  convex_hull_vertex();
    Vertex*  convex_hull_successor( Vertex*);
    Vertex*  convex_hull_predecessor( Vertex*);

    // ...
};
```

Each vertex pointer is computed only when the user asks for it by calling the successor or predecessor function. The user is free to choose an appropriate container for the sequence or

to use the vertex pointers directly without storing them in a container. In contrast to the first approach, the functionality for storing the vertices in a container is provided, not the container itself.

However, this interface has the disadvantage that each algorithm working on either sequence of vertices has to know the names of the access functions, which makes it hard to implement such an algorithm generically. The following solution avoids this disadvantage:

```
class Triangulation {
public:
    Vertex_iterator      vertices_begin();
    Vertex_iterator      vertices_end();

    Convex_hull_iterator convex_hull_begin();
    Convex_hull_iterator convex_hull_end();

    // ...
};
```

Here, the whole functionality for accessing vertices is factored out in separate classes, which are models for the concept of iterators from STL. To demonstrate the genericity of the third approach, the following example shows the use of the generic `copy` function from STL to store the vertices in a C-array.

```
Triangulation t;
// ... insert less than 100 points in t.

Vertex vertices[100];
copy( t.vertices_begin(), t.vertices_end(), vertices);

Vertex convex_hull[100];
copy( t.convex_hull_begin(), t.convex_hull_end(), convex_hull);
```

This solution was chosen for CGAL, except that the vertices of the convex hull of the triangulation are accessed with a more efficient circulator, since the internal representation of this convex hull is cyclic.

As in the previous example, geometric data structures in the basic library often contain more than one sequence of interest, e.g., triangulations contain vertices, edges, and faces. Therefore the names of the member functions that return iterator ranges are prefixed with the name of the sequence, e.g., `vertices_begin()`, `edges_end()`. These names are the canonical extension of the corresponding names `begin()` and `end()` in STL. The iterator based interfaces together with the extended naming scheme assimilate the design of the container-like geometric data structures in the basic library with the C++ Standard. This guarantees a smooth learning curve for users having a base knowledge of STL, thus making CGAL easy to use.

Generic Algorithms

Instead of implementing an algorithm for a specific container, the geometric algorithms in the basic library are based on iterators, circulators, and handles. They are generic and comply with STL.

An example of a geometric algorithm is the convex hull. The algorithm takes a set of points and outputs the sequence of points on the convex hull. A possible declaration of the algorithms interface might fix a certain container class to pass the points around:

```
template < class Point >
list<Point> convex_hull( list<Point> points);
```

The input is read from the container `points` of type `list<Point>`, the output is written to another container of the same type. Again, a major drawback is that the user is forced to provide the input in a specific container. If the user wants to compute the convex hull of points stored in a vector, the user has to copy the points from the vector into a list before calling the algorithm. The same argument holds for the output written to the fixed container.

Our solution in CGAL, following the generic programming paradigm, uses iterator ranges instead of containers. The function declaration looks like this:

```
template < class InputIterator, class OutputIterator >
OutputIterator
convex_hull( InputIterator first, InputIterator beyond,
            OutputIterator result);
```

Here, the input is read from the iterator range `[first, beyond)` and the output is written to the output iterator `result`. Let the return-value be `result_beyond`, then the iterator range `[result, result_beyond)` contains the sequence of points on the convex hull. This design decouples the algorithm from the container and gives the user the flexibility to use any container, e.g., from STL, from other libraries or own implementations (provided they comply with STL), or to use no container at all, for example a sequence of points read from the standard input:

```
convex_hull( istream_iterator<Point>( cin), istream_iterator<Point>(),
            ostream_iterator<Point>( cout, "\n"));
```

Points are taken from the standard input and the resulting points on the convex hull are written to the standard output. Here so-called stream iterators [34] from STL are used. This example again demonstrates the flexibility gained from the STL-compliance of the geometric algorithms in the basic library.

The following example is a complete running program. It generates 100 points at random, uniformly distributed in a disc. The Delaunay triangulation and the convex hull of the point set are computed and displayed in two graphical windows. The output is shown in Fig. 2.

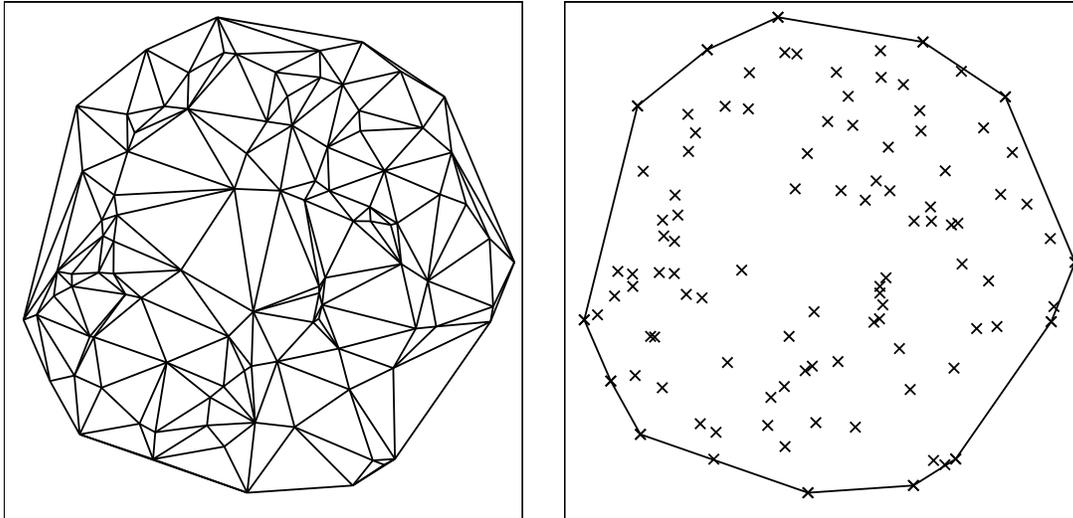


Figure 2. The output of `example_Delaunay_triangulation_Convex_hull.C`

```
#include "tutorial.h"
#include "tutorial_io.h"

int main() {
    // generate 100 random points, uniformly distributed in a disc
    Random          rnd( 2);
    Random_points_in_disc_2  rnd_pts( 250.0, rnd);
    list<Point_2>   pts;
    copy_n( rnd_pts, 100, back_inserter( pts));

    // compute and show the Delaunay triangulation
    Delaunay_triangulation_2 dt;
    dt.insert( pts.begin(), pts.end());
    Window_stream window1( 512, 512, 50, 50);
    window1.init( -256.0, 255.0, -256.0);
    window1 << dt;

    // compute and show the convex hull
    Polygon_2 ch;
    convex_hull_points_2( pts.begin(), pts.end(), back_inserter( ch));
    Window_stream window2( 512, 512, 600, 50);
    window2.init( -256.0, 255.0, -256.0);
    copy( pts.begin(), pts.end(), ostream_iterator_point_2( window2));
    window2 << ch;

    // wait for mouse click in window2
    Point_2 p;
    window2 >> p;
    return 0;
}
```

The file `tutorial.h` is used in the example programs in the CGAL-tutorial [50]. It includes some CGAL header files and uses the representation class `CGAL_Cartesian<double>` to

parameterize the geometric objects and algorithms. Via `typedefs` the template mechanism and the `CGAL_` prefix are hidden from the CGAL novice. The file `tutorial_io.h` includes CGAL header files related to graphical in- and output.

The class `Random_points_in_disc_2`, provided by CGAL, is a model for an input iterator. It generates two-dimensional points uniformly distributed in a disk. The function template `back_inserter()` from STL returns a model for an output iterator that appends items to the end of the given container. Note that it is applied twice, once to a list from STL, and once to a polygon from CGAL, showing the genericity of `back_inserter()` and the STL-compliance of the CGAL polygon. The function template `ostream_iterator_point_2()`, provided by CGAL, returns a model of an output iterator which writes points to the given output stream. The `ostream_iterator<type>` from STL cannot be used here, since it is tailored to C++ Standard Library streams, but the `Window_stream` from CGAL is not derived from these streams.

Traits Classes for Adaptability

In the section on design goals the analogy with an ideal theoretical paper on a geometric algorithm was introduced, which first declares geometric primitives and thereafter expresses the algorithm in terms of these primitives. Implementing an algorithm or data structure, we collect all necessary types and primitive operations in a single class, called *traits* class, which encapsulates details, such as the geometric representation. Collecting types in a single class is a template technique that is already intensively used in [42]. It is sometimes called ‘*nested typedefs for name commonality*’-idiom. The approach gains much additional value by the *traits technique* as used in the C++ Standard Library [51], where additional information is associated with already existing types or built-in types. An example is an iterator for which a user might want to know the value type to which the iterator refers. This can be easily encoded as a local type for all iterators that are implemented as classes.

```
struct iterator_to_int {
    typedef int value_type;
    // ...
};
```

Since a C-pointer is a valid iterator, this approach is not sufficient. The solution chosen for STL are iterator traits, i.e., class templates parameterized with an iterator.

```
template < class Iterator >
struct iterator_traits {
    typedef typename Iterator::value_type value_type;
    // ...
};
```

The value type of the iterator example class above can be expressed as `iterator_traits<iterator_to_int >::value_type`. For C-pointers a specialized version of iterator traits exists, i.e. a class template parameterized with a C-pointer.

```
template< class T >
struct iterator_traits<T*> {
    typedef T value_type;
    // ...
};
```

Now the value type of a C-pointer, e.g., to `int`, can be expressed as `iterator_traits<int* >::value_type`. This technique of providing an additional, more specific definition for a class template is known as *partial specialization*.

Our approach using traits classes in the basic library does not attach information to built-in types, but to our data structures and algorithms. We use them as a modularization technique that allows a single implementation to be interfaced to different geometric representations and primitive operations. Our traits class is therefore a single template argument for algorithms and data structures in the basic library, for example triangulations:

```
template < class Traits >
class CGAL_Triangulation_2 {
    // ...
};
```

Note that each primitive could as well be provided as a template parameter for itself, but using traits classes simplifies the interface. All primitives are captured in a single argument, which makes it easier to apply already prepared implementations of traits classes.

Implementing (with) Traits Classes

Traits classes must provide the geometric primitives required by the geometric data structure or algorithm. Default implementations are provided for the geometric kernel of CGAL. They are class templates parameterized with a kernel representation class, for example `CGAL_Triangulation_euclidean_traits_2<CGAL_Cartesian<double> >`. A single traits class for triangulations is sufficient for all representations and number types possible with the kernel. Further traits classes are available in CGAL, for example for using the basic library with the geometry part of LEDA.

To give an example, the data structure `CGAL_Triangulation_2` needs, among other things, points, segments, triangles and an orientation predicate. Thus, a traits class for this data structure looks like:

```
template < class R >
class CGAL_Triangulation_euclidean_traits_2 {
public:
    typedef CGAL_Point_2<R>      Point;          // point type
    typedef CGAL_Segment_2<R>    Segment;        // segment type
    typedef CGAL_Triangle_2<R>   Triangle;      // triangle type
    // ...
};
```

```

CGAL_Orientation                                // orientation predicate
orientation( const Point& p, const Point& q, const Point& r) const {
    return CGAL_orientation( p, q, r);
}
// ...
};

```

For ease of use, typedefs can introduce shorter names, such as

```

typedef CGAL_Cartesian<long>                    R;
typedef CGAL_Triangulation_euclidean_traits_2<R> Traits;
typedef CGAL_Delaunay_triangulation_2<Traits>   Triangulation;

```

for Delaunay triangulations based on the Euclidean metric and using predicates and objects from the CGAL-kernel with Cartesian coordinates of type long integer.

Up to now, we have described the interface as it is presented to the user. The following code fragment of the triangulation illustrates the use of the traits classes in implementing a data structure:

```

template < class Traits >
class CGAL_Triangulation_2 {
public:
    typedef typename Traits::Point    Point;
    typedef typename Traits::Triangle Triangle;
    Traits traits;

    insert( const Point& p) {
        // ...
        if ( traits.orientation( p, q, r) != CGAL_COLLINEAR) {
            Triangle t( p, q, r);
            // ...
        }
        // ...
    }
    // ...
};

```

The insert member function inserts a given point p into the triangulation. The type of p is `Traits::Point`, which is the point type from the traits class. At some point in the insertion, the orientation of a point triple p, q, r is checked with a call to the `orientation` function provided by the traits class. If $p, q,$ and r are not collinear, a triangle t is created from the point triple. Again, t is of type `Triangle::Traits`, which is the triangle type from the traits class. Note that an instance `traits` of the traits class is stored in `CGAL_Triangulation_2`. This allows the user to provide additional information within this traits-class object, e.g., a direction for projecting three-dimensional points onto a two-dimensional plane. To allow access to the additional data, the `orientation` member function is not declared `static`.[§]

[§]If `orientation()` was declared `static`, the call would be `Traits::orientation(p, q, r)`, i.e., the call would not be on a specific instance of `Traits`.

Default Traits Classes

For algorithms implemented as functions, a default traits class is chosen automatically if there is none given explicitly in the function call. Thus the user can just ignore the traits class mechanism as in the following example:

```
typedef CGAL_Cartesian<double> R;
typedef CGAL_Point_2<R> Point_2;
typedef CGAL_Polygon_2<R> Polygon_2;

Polygon_2 hull;
CGAL_convex_hull_points_2( istream_iterator<Point_2>( cin),
                          istream_iterator<Point_2>( ),
                          back_inserter( hull));
```

In the call to the convex-hull algorithm no traits class is visible to the user. A default traits class is chosen automatically in the definition of the algorithm:

```
template < class InputIterator, class OutputIterator >
inline
OutputIterator
CGAL_convex_hull_points_2( InputIterator first, InputIterator beyond,
                          OutputIterator result ) {
    typedef typename iterator_traits<InputIterator>::value_type Point;
    typedef typename Point::R R;
    return CGAL_convex_hull_point_2( first, beyond, result,
                                     CGAL_convex_hull_traits_2<R>());
}
```

The value type of the iterator `InputIterator` is the point type used in the input. It is determined with the iterator traits described previously. Since the default traits class is supposed to use the geometric kernel of CGAL, we know that the point type must be a CGAL point type and it ‘knows’ its representation type `R` by means of a local type named `R`. Finally, another version of `CGAL_convex_hull_points_2` is called with four arguments. The additional fourth argument is set to the default traits class for the convex-hull algorithms, `CGAL_convex_hull_traits_2<R>()`. This second version of the function template is defined as follows:

```
template < class InputIterator, class OutputIterator, class Traits >
OutputIterator
CGAL_convex_hull_points_2( InputIterator first, InputIterator beyond,
                          OutputIterator result, const Traits& traits) {
    // compute the convex hull using only primitives from the traits class
}
```

This mechanism can be used in all functions that know directly or indirectly – via iterators or circulators – at least one object of the geometric kernel.

Examples for Adaptability through Traits Classes

The following examples demonstrate the adaptability of the basic library through the use of traits classes. We show

- how to plug a CGAL algorithm into an existing application which has its own point type,
- how to change the underlying metric for distance computations in a CGAL data structure, and
- how to use three-dimensional data in a two-dimensional data structure from CGAL.

Suppose a user already has a large application based on a specific point type, for example `leda_rat_point` from LEDA, and wants to compute the convex hull of a set of points with the CGAL algorithm `CGAL_ch_graham_andrew` [52]. The user only needs an appropriate traits class that provides a point type `Point_2` and predicates `Less_xy` and `Leftturn`:

```
#include <CGAL/ch_graham_andrew.h>
#include <LEDA/rat_point.h>
#include <LEDA/list.h>

struct Leda_traits {
    typedef leda_rat_point Point_2;

    struct Less_xy {
        bool operator() ( const Point_2& p, const Point_2& q) const {
            return( compare( p, q) < 0);
        }
    };
    struct Leftturn {
        bool operator() ( const Point_2& p,
                          const Point_2& q,
                          const Point_2& r) const {
            return( left_turn( p, q, r));
        }
    };
    Less_xy get_less_xy_object() const { return Less_xy(); }
    Leftturn get_leftturn_object() const { return Leftturn(); }
};

int main() {
    leda_list<leda_rat_point> pts;    // input points
    leda_list<leda_rat_point> ch;    // points on convex hull
    // ...
    CGAL_ch_graham_andrew( pts.begin(), pts.end(),
                           back_inserter( ch), Leda_traits());
    return 0;
}
```

The convex-hull algorithm from CGAL is adapted to the user's point type through the parameter `Leda_traits()` in the call to `CGAL_ch_graham_andrew`. The functions `compare` and `left_turn` used for the predicates are provided by LEDA and the list used from LEDA is STL compliant. CGAL already provides a traits class

for its convex-hull algorithms based on the point type `leda_rat_point`, namely `CGAL_convex_hull_rat_leda_traits_2`.

Another way of adapting CGAL is to change only a small part of a traits class. Only the necessary modifications are done, while most of the code from the library can be reused. We give an example showing the computation of the Delaunay triangulation of a point set using the maximum metric (L_∞) instead of the usual Euclidean metric (L_2).

```
#include <CGAL/Triangulation_euclidean_traits_2.h>
#include <CGAL/Delaunay_triangulation_2.h>

typedef CGAL_Cartesian<double>          R
typedef CGAL_Point_2<R>                 Point;
typedef CGAL_Triangulation_euclidean_traits_2<R> Euclidean_traits;

class L_infty_traits : public Euclidean_traits {
public:
    CGAL_Orientation
    extremal(const Point_2& p, const Point_2& q, const Point_2& test) const {
        // using L_infty metric
    }
    CGAL_Oriented_side
    side_of_oriented_circle( const Point_2& p, const Point_2& q,
                             const Point_2& r, const Point_2& test) const {
        // using L_infty metric
    }
};

int main() {
    typedef CGAL_Delaunay_triangulation_2< L_infty_traits > DT_l_infty;

    list<Point> pts;
    // ...
    DT_l_infty dt;
    dt.insert( pts.begin(), pts.end());

    return 0;
}
```

The adapted traits class `L_infty_traits` is derived from CGAL's default traits class for triangulations. Only two predicates, `extremal` and `side_of_oriented_circle`, have to be re-defined for the maximum metric. All other inherited primitives remain unchanged, since they do not depend on the chosen metric. CGAL already provides such a traits class, namely `CGAL_Triangulation_l_infty_traits_2`.

The third example deals with the problem of using a two-dimensional data structure with three-dimensional data. Suppose the user is given a terrain model by a set of terrain points, where a two-dimensional point represents the position in the plane and an additional number is the level. The user wants to compute the Delaunay triangulation for the vertical projection of the terrain model. The predicates which are defined in the following traits class ignore the additional number and use simply the two-dimensional points.

```

#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/Triangulation_euclidean_traits_2.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <vector.h>

template < class R, class Level >
class Terrain_point {
public:
    typedef CGAL_Point_2<R> Point;
    Terrain_point( ) { }
    Terrain_point( const Point& point, const Level& level)
        : m_point( point), m_level( level) { }
    const Point& point() const { return( m_point); }
    const Level& level() const { return( m_level); }
private:
    Point m_point;
    Level m_level;
};
// class Terrain_segment
// class Terrain_triangle

template < class R, class Level >
struct Terrain_traits {
    typedef Terrain_point< R, Level > Point;
    typedef Terrain_segment < Point > Segment;
    typedef Terrain_triangle< Point > Triangle;
    typedef CGAL_Triangulation_vertex< Point > Vertex;
    typedef CGAL_Triangulation_face< Vertex > Face;
    typedef CGAL_Triangulation_euclidean_traits_2<R> Traits;

    CGAL_Comparison_result
    compare_x( const Point& p, const Point& q) const {
        return( Traits::compare_x( p.point(), q.point()));
    }
    // compare_y()

    CGAL_Orientation
    orientation( const Point& p, const Point& q, const Point& r) const {
        return( Traits::orientation( p.point(), q.point(), r.point()));
    }
    // extremal()

    CGAL_Oriented_side
    side_of_oriented_circle( const Point& p, const Point& q,
        const Point& r, const Point& s) const {
        return( Traits::side_of_oriented_circle( p.point(), q.point(),
            r.point(), s.point()));
    }
};

int main() {
    typedef CGAL_Cartesian<double> R;
    typedef Terrain_point < R, int > TPoint;
    typedef Terrain_traits< R, int > Traits;
    typedef CGAL_Delaunay_triangulation_2< Traits > Dt;

```

```
vector<TPoint> terrain;  
// ...  
Dt dt;  
dt.insert( terrain.begin(), terrain.end());  
return 0;  
}
```

Since the terrain points represent their positions with two-dimensional points from CGAL, all functions provided by the standard model of the traits-class concept for triangulations can be re-used. CGAL provides already traits classes based on `CGAL_Point_3`, where the triangulation works on the projection onto xy-, xz-, and yz-plane, respectively. For example the traits class for the xy-plane is called `CGAL_Triangulation_euclidean_traits_xy_3`.

SOFTWARE ENGINEERING IN THE PROJECT

The birth of the CGAL-library dates back to a meeting in Utrecht in January 1995. Shortly afterwards, the five authors started developing the kernel. The CGAL-project started officially in October 1996 and the team of developers has grown to circa 25 people, mostly research assistants, PhD students and postdocs in academia, who are professionals in the field of computational geometry and related areas. This amounts to a heterogeneous team of developers; some of them working part time for CGAL, some of them full time. The CGAL release 1.2 (January 1999) consists of approximately 110,000 lines of C++ source code[¶] for the library, plus 50,000 lines for accompanying sources, such as the test suite and example programs. In terms of the elder Constructive Cost Model (COCOMO) the line counts, people involved, and time schedule amount to a big project which is still growing, comparable to smaller operating systems or database management systems [53]. The need for software engineering and quality control is obvious.

The project structure of seven loosely coupled research groups needed to be taken into account for the management. The intensive use of the Internet with a project Web-server^{||} and email discussion groups are a matter of course, but major progress in the design was mostly made during implementers meetings. An early modularization of the library, with as few dependencies among the modules as possible, was crucial to the project in order to keep the communication needs between the project partners reasonable. The library layers, kernel and basic library, emphasized the obvious dependencies. The basic library was subdivided into mostly independent parts and assigned to different project partners. The kernel development started quite ahead so that the first internal release was ready by the official start of the project.

The idealized developing process in CGAL is structured into specification, implementation, test, integration and continuous regression testing. It is influenced by the spiral model [53], and successive iterations through this process are to be expected. The indivisible unit is the package. Four parties are involved in the developing process: The author, the editorial committee, the tester and the integration site. The author writes a specification for a package

[¶]C++ comments and empty lines are not counted.

^{||}<http://www.cs.uu.nl/CGAL/>

in the format of a reference manual page and submits it to the editorial committee for discussion and approval. The editorial committee considers the issues of interconnections between different packages and a unified look-and-feel of the design in CGAL. After approval of the specification the author implements and tests the package. Literate programming tools are recommended to structure the source code and accompanying documentation of the implementation (but not the specification) [54, 55, 56, 57]. However, the common format for source code distributions are plain C++ source files. The strict separation of the specification from the implementation is idealistic, but nonetheless important. It puts the main focus on the design qualities of a package and postpones the implementation-specific decisions and restrictions. A further discussion can be found in [58]. This separation is also supported by our tools used for the manual writing. In the next step, the author sends the package to the external tester at a distinct project site. Thereafter, the package is integrated in the CGAL-library, maintained and revision controlled at the integration site. Each package is supposed to provide a test suite. On integration the test suite needs to pass all runs on supported compiler/system combinations. Upon any internal and external releases of the library, the collected test suites will be automatically evaluated.

The developing process defines five places for quality control, although the heterogeneous, academic environment allows only recommendations: First, the design is reviewed by the editorial committee (or other reviewers commissioned by them). Typically this has been done at developer meetings. This is the most serious control, since errors or necessary changes not seen yet are the most costly ones. Second, the specification will mention pre- and postconditions for functions. They are supposed to be tested at runtime with assertions [37, 26] placed in the implementation**. Preconditions allow the early detection of usage errors by the function caller, postconditions catches implementation errors in the function. Further assertions can be placed anywhere between. The extension from assertions to program checkers with respect to geometric algorithms can be found in [27]. Third, the author is supposed to test the implementation thoroughly and to provide a test suite, which achieves at least code coverage of the package^{††}. This can be proven with code coverage tools, e.g., `gcov` of the Gnu tools. Other runtime tests are recommended, for example bounds checker and monitoring dynamic memory allocation. Fourth, the second, external tester again reviews specification and implementation. Fifth, the integration into the library uses the automatized test suite to check the compliance of the new package with the other parts of the library, especially when a revised version of a module is integrated.

Design and specifications are communicated with reference manual pages. Some reasons are given above with the developing process. Another reason is that the reference manual will be the main documentation for the users of CGAL. Its quality, along with that of the other manuals, such as the tutorial, will determine the acceptance of CGAL. A design is not a good design if we cannot enable others to understand it.

In order to provide appealing, high-quality manual pages we use \LaTeX . A self-written style file adds additional formatting capabilities as seen in Fig. 3, which displays an excerpt of the manual page for two-dimensional points in CGAL. The principal manual page layout

**For production code the assertions can be omitted from the code. Assertions can be independently switched on and off for major packages. A distinction into normal and expensive checks allows even the use of computationally non-trivial checks in these assertions without sacrificing too much speed.

^{††}Note that for C++ templates code coverage is even more important than ever, since otherwise the compiler is not even able to check for syntactical errors within templates.

2D Point (*CGAL_Point_2*<*R*>)

Definition

An object of the class *CGAL_Point_2* is a point in the two-dimensional Euclidean plane.

```
#include <CGAL/Point_2.h>
```

Creation

```
CGAL_Point_2<R> p( R::RT hx, R::RT hy, R::RT hw = R::RT(1));
```

introduces a point *p* initialized to (*hx/hw*, *hy/hw*). If the third argument is not explicitly given, it defaults to *R*::*RT*(1).

Operations

```
CGAL_Point_2<R> p.transform( CGAL_Aff_transformation_2<R> t)
```

returns the point obtained by applying *t* on *p*.

```
CGAL_Vector_2<R> p - q
```

returns the difference vector between *q* and *p*.

```
CGAL_Point_2<R> p + CGAL_Vector_2<R> v
```

returns a point obtained by translating *p* by the vector *v*.

```
CGAL_Point_2<R> p - CGAL_Vector_2<R> v
```

returns a point obtained by translating *p* by the vector $-v$.

Figure 3. The shortened reference manual page for two-dimensional points in CGAL-RI.2.

is intentionally close to the LEDA user manual [7], although the writing process and the supporting tools are different. The main goal of the layout is to provide a dense and compact presentation, which allows a fast overview and access to the information searched, without sacrificing correctness. A three column layout for the member functions displays the return-type in the first column, the remaining signature in the second column, and the documentation in the third column. Certain automatism allow flexible layouts whenever an entry gets too long. Reduction rules remove `const ... &` declarations from function arguments (they are usually an implementation detail, otherwise they will be stated explicitly), remove the current class name from function arguments and rewrite operator declarations in operator notation, see the operator examples in Fig. 3. This is all done automatically using original C++ declarations within the L^AT_EX source. In summary, a short member function can be documented efficiently in a single line.

EVALUATION OF THE DESIGN

The sections above illustrate the concepts and techniques we use in CGAL to accomplish our design goals. Most of the techniques described are dedicated to our design goal *flexibility*. Modularity has been achieved with the structuring of the library in layers and packages. The approach of the generic programming paradigm to specify an interface for a template

argument in terms of a concept has led to a strongly decoupled collection of modules. The example STL illustrates the effect with algorithms and container classes: The container classes do not know any algorithms, nor must the algorithms know any container classes. The connection is established with the concept of the iterator. Various file formats and visualization tools exist. Thus, I/O functions and visualization are not part of the geometric objects themselves, but of separate modules. *Adaptability* has been addressed with the free choice of an arithmetic in the kernel, the traits classes in the basic library, and the modularization. Kernel predicates, distance and intersection functions are so far not adaptable to other geometric objects than those of the kernel. This will be addressed in future work on the kernel. *Extensibility* is primarily based on function overloading in C++ for the global functions and the independence of the modules. The generic object `CGAL_Object` can cope with any new object which solves the extensibility of the intersection function with its polymorphic return-value. The decoupling of the modules through the generic programming paradigm allows one to write new algorithms or data structures that interface CGAL – as CGAL interfaces with the STL. CGAL is open to the C++ standard, especially the STL part of it, and allows the adaptation to other libraries. An example is the already provided support for the LEDA number types or the wrapper class for the Gnu Multiple Precision Arithmetic Library. Support for other number types can be added easily. The modularization of I/O functions opens CGAL for any file format or visualization tool.

Correctness is addressed by the quality control in the project structure. It includes the recommendation of certain tools and the use of assertions and program checkers. The strong modularization and large independence help in testing and achieving correctness. Another strong point in achieving correctness is modularity and adaptability, which allow the combination of different modules according to the need of a particular solution. For example, the adaptability with respect to number types can be utilized for a convex-hull algorithm if the input points are known to have integer coordinates within a certain range; an optimized arithmetic of fixed but sufficient bit precision instead of a general purpose arithmetic still results in a correct algorithm.

Robustness is partially coupled with correctness. The choice of an exact arithmetic and homogeneous representation class in the kernel lead to exact and efficient primitives which are easier to combine to form robust algorithms.

Ease-of-use is achieved through a strong modularization and a *smooth learning curve*. Users who are familiar with the iterator concept are immediately familiar with its use in many algorithms and data structures in CGAL. The new concepts, handle and circulator, are easy to learn due to their similarity to the iterator concept. The concepts in the kernel behave similarly: Once users are used to the parameterization with a representation class and arithmetic type, the complete CGAL kernel can be used. Even easier, the header file from the CGAL-tutorial uses typedefs to hide the template instantiations, so that the C++ novice sees only simple classes with a selected default representation and number type. When users get familiar with the templates in the kernel, the representation class provides a uniform look at all geometric objects in the kernel. In the basic library the flexibility of CGAL is usually hidden behind a single template argument, the traits class, for which a default class exists. For functions, even this argument can be hidden with a default parameter. The generic programming paradigm has been applied successfully to CGAL. Only a few new concepts were introduced. Generic programming has contributed considerably to the uniform

look-and-feel of the design. A naming convention and the choice of expressive names, with abbreviations limited to standard abbreviations in geometry, result in readable and easily memorizable interfaces.

Efficiency has been tackled with the extensive use of templates and inline functions. Relying on compiler optimization capabilities, the traits class technique used in the basic library should allow optimal results. Compromises concerning efficiency have been made for the benefit of other design goals in some places. For instance, for ease-of-use the intersection computation of kernel objects uses a unified interface with the generic object as return value even for relative elementary intersection computations, such as line/line intersections. Another example is the non-modifiability of the kernel objects. Modifiability could lead to more efficient computation if we assume that points are always implemented with Cartesian coordinates. However, as discussed in the section on the kernel layer, this would restrict flexibility. An interesting issue concerning efficiency is reference counting, see, e.g., References [49] and [59]. Reference counting can be a source of efficiency, time efficiency as well as space efficiency, since fewer copies of an object are necessary. However, especially if the objects are small and copies are rarely made, there are scenarios where reference counting slows down the computation, especially if no memory management is used to support fast allocation of reference-counted representation objects on the heap. For example, convex-hull computation, as illustrated in the section on the basic library layer, is about 40% faster with points not using reference counting, if the points coordinates are `doubles`. If the coordinates are represented as `leda_integer`, which use reference counting as well, but take more space than a `double`, reference-counted points are already slightly faster. In both experiments, LEDA's memory management was used to speed up heap allocation of reference-counted objects. Currently the kernel does not offer an alternative to reference-counted objects, but it will do so in the future. Moreover, the flexibility of the basic library allows the use of other kernels, if the efficiency is not sufficient.

ACKNOWLEDGEMENTS

The authors of this paper designed and implemented the CGAL-kernel right before the start of the CGAL-project in order to enable an immediate start of the library implementation. They were also the main architects of the design of the whole library as it is presented here, but many more people were involved in starting the project and since the initial implementation of the kernel many more people have joined the CGAL team and contributed to the development of CGAL. We want to thank all of them, especially Helmut Alt, Mark de Berg, Jean-Daniel Boissonnat, Hervé Brönnimann, Christoph Burnikel, Paul Callahan, Otfried Cheong (née Schwarzkopf), Jochen Comes, Olivier Devillers, Katrin Dobrindt, Eyal Flato, Wolfgang Freiseisen, Stefan Funke, Bernd Gärtner, Dan Halperin, Iddo Hanniel, Sarel Har-Peled, Michael Hoffmann, Marc van Kreveld, Doron Jacoby, Markus Lohninger, Kurt Mehlhorn, Stefan Näher, Gabriele Neyer, Jürg Nievergelt, Marco Nissen, Mark Overmars, Michal Ozery, Petru Pau, Sylvain Pion, Sigal Raab, Ralf Rickenbach, Holger Sabo, Michael Seel, Raimund Seidel, Micha Sharir, Nora Sleumer, Sabine Stifter, Monique Teillaud, Carl Van Geem, Remco Veltkamp, Emo Welzl, Wieger Wesselink, Peter Widmayer, Martin Will, Alexander Wolff, Mariette Yvinec, and Mark Ziegelmann. We also want to thank Dietmar Köhl and Karsten Weihe for valuable comments and the polymorphism solution with the

class `CGAL_Object`. Finally, we would like to thank the anonymous referees for their helpful comments.

Work on this paper has been 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).

REFERENCES

1. K. Mehlhorn and S. Näher, 'The implementation of geometric algorithms', *13th World Computer Congress IFIP94*, volume 1. Elsevier Science B.V. North-Holland, Amsterdam, 1994, pp. 223–231.
2. S. Schirra, 'Precision and robustness issues in geometric computation', *Handbook on Computational Geometry*, Elsevier Science Publishers, Amsterdam, The Netherlands, 1999.
3. C. Burnikel, K. Mehlhorn, and S. Schirra, 'On degeneracy in geometric computations', *Proc. of the 5th ACM-SIAM Symp. on Discrete Algorithms*, 1994, pp. 16–23.
4. F. Avnaim, *C++GAL: A C++ Library for Geometric Algorithms*, INRIA Sophia-Antipolis, 1994.
5. G.-J. Giezeman, *PlaGeo, a library for planar geometry, and SpaGeo, a library for spatial geometry*, Utrecht University, 1994.
6. K. Mehlhorn and S. Näher, 'The LEDA Platform for Combinatorial and Geometric Computing'. Cambridge University Press, forthcoming, 1999.
7. K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig, *The LEDA User manual*, 3.7 edition, 1998. see <http://www.mpi-sb.mpg.de/LEDA/leda.html>.
8. Jürg Nievergelt, Peter Schorn, Michele de Lorenzi, Christoph Ammann, and Adrian Brünger, 'XYZ: A project in experimental geometric computation', *Proc. Computational Geometry: Methods, Algorithms and Applications*, volume 553. Springer-Verlag, 1991, pp. 171–186.
9. Peter Schorn, 'Implementing the XYZ GeoBench: A programming environment for geometric algorithms', *Computational Geometry — Methods, Algorithms and Applications: Proc. Internat. Workshop Comput. Geom. CG '91*, volume 553 of *Lecture Notes Comput. Sci.*, Springer-Verlag, 1991, pp. 187–202. <http://wwwjn.inf.ethz.ch/geobench/XYZGeoBench.html>.
10. International standard ISO/IEC 14882: Programming languages – C++. American National Standards Institute, 11 West 42nd Street, New York 10036, 1998.
11. N. Amenta, 'Computational geometry software', *Handbook of Discrete and Computational Geometry*, CRC Press, 1997, pp. 951–960.
12. D. T. Lee, 'Visualizing geometric algorithms – state of the art', M. C. Lin and D. Manocha (eds.), *Applied Computational Geometry (Proc. WACG '96)*, volume 1148 of *Lecture Notes Comput. Sci.* Springer-Verlag, 1996, pp. 45–50.
13. Kurt Mehlhorn, 'Position paper for panel discussion', M. C. Lin and D. Manocha (eds.), *Applied Computational Geometry (Proc. WACG '96)*, volume 1148 of *Lecture Notes Comput. Sci.* Springer-Verlag, 1996, pp. 51–52.
14. Mark H. Overmars, 'Designing the Computational Geometry Algorithms Library CGAL', M. C. Lin and D. Manocha (eds.), *Applied Computational Geometry (Proc. WACG '96)*, volume 1148 of *Lecture Notes Comput. Sci.* Springer-Verlag, 1996, pp. 53–58.
15. Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr, 'The CGAL kernel: A basis for geometric computation', M. C. Lin and D. Manocha (eds.), *ACM Workshop on Applied Computational Geometry*, Philadelphia, Pennsylvania, May, 27–28 1996, pp. 191–202. *Lecture Notes in Computer Science* 1148.
16. R. C. Veltkamp, 'Generic programming in CGAL, the computational geometry algorithms library', *Proceedings of the 6th Eurographics Workshop on Programming Paradigms in Graphics*, 1997.
17. S. Schirra. Designing a computational geometry algorithms library. *Lecture Notes for Advanced School on Algorithmic Foundations of Geographic Information Systems*, CISM, Udine, September 16-20 1996.
18. Lutz Kettner, 'Using generic programming for designing a data structure for polyhedral surfaces', *Computational Geometry: Theory and Applications* (1999). to appear.
19. P. Epstein, J. Kavanagh, A. Knight, J. May, T. Nguyen, and J.-R. Sack, 'A workbench for computational geometry', *Algorithmica*, **11**, 404–428 (1994).

20. P. de Rezende and W. Jacometti, 'Geolab: An environment for development of algorithms in computational geometry', *Proc. 5th Canad. Conf. Comput. Geom.*, Waterloo, Canada, 1993, pp. 175–180.
21. J. E. Baker, R. Tamassia, and L. Vismara. GeomLib: Algorithm engineering for a geometric computing library, 1997. (Preliminary report).
22. C. Burnikel, K. Mehlhorn, and S. Schirra, 'How to compute the Voronoi diagram of line segments: Theoretical and experimental results', *Proc. 2nd Annu. European Sympos. Algorithms*, volume 855 of *Lecture Notes Comput. Sci.* Springer-Verlag, 1994, pp. 227–239.
23. Giuseppe Liotta, Franco P. Preparata, and Roberto Tamassia, 'Robust proximity queries: an illustration of degree-driven algorithm design', *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, 1997, pp. 156–165.
24. J.-D. Boissonnat and F. Preparata, 'Robust plane sweep for intersecting segments', *Technical Report 3270*, INRIA, Sophia-Antipolis, France, September 1997.
25. John Lakos, *Large Scale C++ Software Design*, Addison-Wesley, 1996.
26. Steve Maguire, *Writing Solid Code*, Microsoft Press, 1993.
27. Kurt Mehlhorn, Stefan Näher, Thomas Schilz, Stefan Schirra, Michael Seel, Raimund Seidel, and Christian Uhrig, 'Checking geometric programs or verification of geometric structures', *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, 1996, pp. 159–165.
28. T. Granlund, *GNU MP, The GNU Multiple Precision Arithmetic Library*, 2.0.2 edition, June 1996.
29. Scott Meyers, *Effective C++*, Addison-Wesley, 1992.
30. J. Vleugels, 'On fatness and fitness — realistic input models for geometric algorithms', *Ph.D. thesis*, Dept. Comput. Sci., Univ. Utrecht, Utrecht, The Netherlands, 1997.
31. Stanley B. Lippman, *Inside the C++ Object Model*, Addison-Wesley, 1996.
32. S. Schirra, 'A case study on the cost of geometric computing', *Proc. of ALENEX'99*, 1999. to appear.
33. Alexander Stepanov and Meng Lee. The standard template library. <http://www.cs.rpi.edu/~musser/doc.ps>, October 1995.
34. David R. Musser and Atul Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Addison-Wesley, 1996.
35. Silicon Graphics Computer Systems, Inc. Standard template library programmer's guide. <http://www.sgi.com/Technology/STL/>, 1997.
36. David R. Musser and Alexander A. Stepanov, 'Algorithm-oriented generic libraries', *Software – Practice and Experience*, **24**(7), 623–642 (1994).
37. Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 3rd edition, 1997.
38. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
39. Lutz Kettner, 'Circulators', in Hervé Brönnimann, Stefan Schirra, Remco Veltkamp, and Mariette Yvinec (eds.), *CGAL Reference Manual. Part 3: Support Library*, 1999. CGAL R1.2. <http://www.cs.uu.nl/CGAL>.
40. 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.
41. CGAL consortium, 'Number types', in Hervé Brönnimann, Stefan Schirra, Remco Veltkamp, and Mariette Yvinec (eds.), *CGAL Reference Manual. Part 3: Support Library*, 1999. CGAL R1.2. <http://www.cs.uu.nl/CGAL>.
42. J. J. Barton and L. R. Nackman, *Scientific and Engineering C++*, Addison-Wesley, Reading, MA, 1994.
43. K. Kreft and A. Langer, 'Iterators in the standard C++ library', *C++ Report*, **8**(10), 27–32 (1996).
44. Ronald N. Goldman, 'Illicit expressions in vector algebra', *ACM Transaction on Graphics*, **4**(3), 223–243 (1985).
45. Bob Wallis, 'Forms, vectors, and transforms', in Andrew S. Glassner (ed.), *Graphics Gems*, Academic Press, 1990, pp. 533–538.
46. A. D. DeRose, 'Geometric programming: A coordinate-free approach', *Theory and Practice of Geometric Modeling*, Blaubeuren, FRG (Oct 1988), 1989. Springer-Verlag.
47. G. Booch and M. Vilot, 'Simplifying the booch components', in S. Lippman (ed.), *C++ Gems*, SIGS publications, 1996, pp. 59–89.
48. T. Keffer, 'The design and architecture of Tools.h++', in S. Lippman (ed.), *C++ Gems*, SIGS publications, 1996, pp. 43–57.
49. Scott Meyers, *More Effective C++*, Addison-Wesley, 1996.
50. Geert-Jan Giezeman, Remco Veltkamp, and Wieger Wesselink, *Getting Started with CGAL*, 1999. CGAL R1.2. <http://www.cs.uu.nl/CGAL>.
51. Nathan C. Myers, 'Traits: a new and useful template technique', *C++ Report* (1995).

52. Stefan Schirra, 'Parameterized implementations of classical planar convex hull algorithms and extreme point computations', *Research Report MPI-I-98-1-003*, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, January 1998.
53. Richard Fairley, *Software Engineering Concepts*, McGraw-Hill Series in Software Engineering and Technology, McGraw-Hill, 1985.
54. Ross N. Williams, *FunnelWeb User's Manual*, V1.0 for FunnelWeb V3.0 edition, May 1992.
55. Donald E. Knuth, 'Literate programming', *The Computer Journal*, **27**(2), 97–111 (1984).
56. Donald E. Knuth and Silvio Levy, *The CWEB System of Structured Documentation*, version 3.0 edition, 1994.
57. Lisa M. C. Smith and Mansur H. Samadzadeh, 'An annotated bibliography of literate programming', *ACM SIGPLAN Notices*, **26**(1), 14–20 (1991).
58. David L. Parnas and Paul C. Clements, 'A rational design process: How and why to fake it', *IEEE Transactions on Software Engineering*, **12**(2), 251–257 (1986).
59. Robert B. Murray, *C++ Strategies and Tactics*, Addison-Wesley, 1993.