I N F O R M A T I K

On the Design of CGAL, the Computational Geometry Algorithms Library

Andreas Fabri   Geert-Jan Giezeman
Lutz Kettner   Stefan Schirra
Sven Schönherr

MPI–I–98–1–007   February 1998

FORSCHUNGSBERICHT   RESEARCH REPORT

M A X - P L A N C K - I N S T I T U T
F Ü R
I N F O R M A T I K

Im Stadtwald   66123 Saarbrücken   Germany

Authors' Addresses

Andreas Fabri
ABB Corporate Research Ltd.
CH-5405 Baden/Dättwil
Switzerland
andreas.fabri@chcrc.abb.ch

Geert-Jan Giezeman
Department of Computer Science
Utrecht University
N-3508 TB Utrecht
The Netherlands
geert@cs.ruu.nl

Lutz Kettner
Theoretical Computer Science
Eidgenössische Technische Hochschule Zürich
CH-8092 Zürich
Switzerland
kettner@inf.ethz.ch

Stefan Schirra
Max-Planck-Institut für Informatik
D-66123 Saarbrücken
Germany
stschirr@mpi-sb.mpg.de

Sven Schönherr
Fachbereich Mathematik und Informatik
Freie Universität Berlin
D-14195 Berlin
Germany
sven@inf.fu-berlin.de

Publication Notes

A revised version of this paper will appear in "TRENDS IN SOFTWARE", Volume on *Algorithm Engineering*, edited by Dorothea Wagner.

Acknowledgements

Work on this paper has been supported by ESPRIT LTR Project No. 21957 (CGAL).

Abstract

CGAL is a *Computational Geometry Algorithms Library* written in C++, which is developed in an ESPRIT LTR project. The goal is to make the large body of geometric algorithms developed in the field of computational geometry available for industrial application. In this chapter 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. Templates and the relatively new generic programming play 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 on the current structure of the library and consider software engineering aspects in the CGAL-project.

Keywords

Software library, C++, generic programming, computational geometry

1 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 best-known examples. Out of research on specific geometric problems in these areas the design and analysis of geometric algorithms has been investigated 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, mostly, because the correct implementation of even the simplest of these algorithms can be a notoriously difficult task [MN94]. This is mainly due to the degeneracy and precision problem [Sch98a]: Theoretical papers assume the input to be in general position and assume exact arithmetic with real numbers. Both assumptions hardly match the situation in practice. 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 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), Free University Berlin (Germany), Martin-Luther University Halle (Germany), INRIA Sophia-Antipolis (France), Max-Planck-Institute for Computer Science and University Saarbrücken (Germany), RISC Linz (Austria), and Tel-Aviv University (Israel). The participating sites are leading in the field of computational geometry in Europe and had ample experience with the implementation of geometric algorithms [Avn94, Gie94, MN95, MNU97, NSdL+91, Sch91]. Work on the CGAL-library is the central task of an ESPRIT IV LTR project which is called CGAL, too. It is the goal of the CGAL-project to

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

The CGAL-library is the key tool to reach this goal. It will be the basis for implementations of geometric algorithms in cooperation projects with industrial partners. These cooperations will be the test bed 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 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 were design goals of CGAL. Providing useful functionality is another design goal among a long list of many others. Design goals for CGAL are discussed in Section 3.

We decided to design CGAL as a C++-library because C++ is widely used and as it can easily be interfaced with existing C and Fortran code. 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 and ABACUS. We consider C++ as a compromise between aesthetic and efficiency. Eiffel or Smalltalk are more properly object oriented but lack acceptance. At present, Java was 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 the roots of CGAL. After discussing the design goals we consider the generic programming paradigm in Section 4. Section 5 discusses 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. In the subsequent sections we discuss the structure of CGAL and present the different layers of the library. Section 7 presents the kernel, which contains basic (constant-size) geometric objects and primitive operations on these objects. Section 8 presents the basic library, which contains standard geometric algorithms and (non constant-size) geometric structures. Besides the design of CGAL we look at engineering aspects addressed in the CGAL-project like manual writing, and separation between specification, implementation, and testing. These are discussed in Section 9. 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 and the basics of its Standard Template Library, see e.g. [Str97].

2 Related Work

Amenta [Ame97] 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, cf. [Lee96, Meh96, Ove96]. The design of the CGAL-kernel at that time is presented in [FGK+96] and the project goals in [Ove96]. A more recent overview can be found in [Vel97]. Precision and robustness aspects of a computational geometry library are discussed in [Sch96]. Further topics on designing combinatorial data structures in CGAL, such as polyhedrons, are described in [Ket97].

Many implementations of computational geometry algorithms exist in loosely coupled collections only. Use and combination of such algorithms usually requires some adaptation effort while components of a library are designed to seamlessly work together. First implementation efforts for computational geometry libraries have been started already end of the Eighties [EKK+94, dRJ93, NSdL+91, Sch91]. 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 less extent CGAL scavenged also implementation techniques from its precursors. These precursors are the XYZ library, developed at ETH Zürich, [NSdL+91, Sch91] PlaGeo/SpaGeo [Gie94], developed at Utrecht University, C++GAL [Avn94], developed at INRIA Sophia-Antipolis, and the geometric part of LEDA [MN95, MNU97], 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 started working on a computational geometry library called GeomLib [BTV97] implemented in Java.

3 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 expect different kind 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.

3.1 Flexibility

The different needs of the potential application areas lead to our design goal flexibility. In order to be useful in many different situations 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 on those modules that are actually of interest. In continuation, 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. Stressing 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. Similar 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 might want to extend the library. It should be possible to easily integrate new objects and algorithms into CGAL. For example, it should be possible to easily add new geometric objects to the library and to provide corresponding intersection functions similar to those existing for native CGAL objects.

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 besides the standard, and CGAL should be easily adaptable to them as well, in particular LEDA [MNU97] with its number types, combinatorial and graph algorithms, the Gnu Multiple Precision Arithmetic Library [Gra96] for a number type, and various visualization systems, some of them standardized.

3.2 Correctness

A library component is correct if it behaves according to its specification. Basically, correctness is therefore a matter of documentation and quality control that documentation and implementation coincides. 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 used modules 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 arithmetic demand of geometric computations has been studied for a few basic geometric problems [BP97, BMS94, LPT97], 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 approximation algorithms computing approximate solutions as long as they do what they pretend to do. 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 at the first hand.

In a modularized project structure it is important to test modules independently and as early as possible [Lak96]. One specific technique for quality assurance are assertions, assertions of invariants of an algorithm and the self-checking of functions at runtime [Mag93, MNS+96]. 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.

3.3 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 proof of geometric algorithms usually assume exact arithmetic while many implementations simply replace it by imprecise arithmetic. Since imprecise calculations can cause wrong and mutually contradicting 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. There is no perfect solution to the precision problem 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 thereby offers kind of a trade-off between efficiency and robustness.

3.4 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 fast the library gets useful. Another issue is the amount of new concepts and exceptions of the general rules that must be learned and remembered. Ease-of-use tends to get in conflict with flexibility, but in many situations a solution can be found to please them both. Especially the flexibility of CGAL should not distract a novice from the first steps with CGAL.

Smooth Learning Curve One major point of the success story of C++ was 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 to understand. This has been successfully shown with LEDA, and can also be seen with the STL.

CGAL is based in many places on concepts known 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 remembering. A concept once learned should be applicable in all places one would expect to. A function name once learned for a specific class should not be named differently for another class. Exceptions should be minimized in the design.

Complete and Minimal Interfaces Another goal with similar implications than uniformity is a design with complete and minimal interfaces, see for example Item 18 in [Mey92]. 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 makes it harder to learn and remember.

Rich and Complete Functionality We aim for a useful and rich collection of geometric objects, data structures and algorithms. CGAL is supposed to be a foundation for algorithmic research in computational geometry and needs therefore a certain breadth and depth. The standard techniques of 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 objects, not only for an arbitrary 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. If this is expensive, additional versions are possible, which are more efficient but less general.

3.5 Efficiency

We consider time and space efficiency. In situations, where a trade-off between them will be possible, we will provide the flexibility to do so. With efficiency we address the well studied, worst-case asymptotic complexity of an algorithm, and results from empirical studies to determine the constant factors hidden in the O-notation of theoretical results, as well as results on typical input sets that occur in practice. Whenever possible and known, the most efficient version of an algorithm is used. Sometimes multiple versions of an algorithm are supplied. For example if dealing with degeneracies is expensive, a faster but less general version might also be supplied. Another example is the exploitation of the characteristics of a specific number type within an algorithm.

Efficiency is a competing goal with respect to flexibility, robustness, and ease-of-use. As long as it is a small constant fraction, we are willing to sacrifice efficiency in favor of the other goals. One cannot expect a library with flexibility requirements as CGAL to provide hand-coded solutions for all purposes. The following sections will reveal that we have taken efficiency seriously. It is a primary design goal for CGAL. In fact, the techniques used for flexibility in CGAL enables us also to achieve optimal efficiency.

4 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.

In the *object-oriented programming paradigm* flexibility is achieved with a virtual base class, which defines an interface, and as many derived classes as different actual implementations of the interface are present in a system. The technique of so-called virtual member functions and runtime type information allows a user to select any of the derived classes wherever the base class is required and that even at runtime. Also, general functionality can be programmed in terms of the base class without knowing all possible derived implementations beforehand.

The advantages are the clear definition of the interface and the flexibility at runtime. There are four main disadvantages: This paradigm cannot provide strong type checking at compile time, enforces tight coupling through the inheritance relationship [Lak96], it adds additional memory to each object derived from the base class (the so-called *virtual function table pointer*) and it adds an indirection through the virtual function table for each call to a virtual member function [Lip96]. The latter one 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[1] can optimize at runtime, but, besides that runtime predictions are difficult, 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 for CGAL are coordinate access and arithmetic for low-dimensional geometric objects and traversals of combinatorial structures. 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 [Lak96].

The *generic programming paradigm* features what is known in C++ as *class templates* and *function templates*. Templates are program recipes where certain types are only given symbolically, the so called *template arguments*. The compiler replaces these arguments with actual types where the program recipe is actually used, at the place of the *template instantiation*. The recipe transforms to a normal part of a program. For function templates this can even be done automatically by the compiler, since the types of the function parameters are known to the compiler. Examples are a generic list class for arbitrary item types or a swap function exchanging variable values for all possible types. The following definitions would enable us to use `list<int>` as a list of integers or to swap two integer variables `x` and `y` with `swap(x,y)`.

```
template <class T>  class list {
  // ... , uses T as 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 assumes some properties to hold for the template arguments, here that variables of those type can be assigned to each other. These *requirements* are not expressed within C++, but only in the accompanying documentation. An actual type used in the template instantiation must fulfill the requirements of the template argument in order of the template to work properly. Requirements can be classified into syntactical ones, there must be an assignment operator, and semantical ones, the implementation of the operator must really do what it is supposed to do. Syntactical requirements will be checked by the compiler at instantiation time of the template. Semantical requirements cannot be checked. In certain situations it might be wishful to stress semantical requirements with additional syntactical, i.e. checkable, requirements, e.g. symbolical tags.

For class templates exist the special situation that different member functions might impose different requirements on the template arguments, but a certain instantiation of the class template uses only a subset of the member functions. Here, the arguments must only fulfill the requirements imposed by the member functions actually used. In particular, the compiler is only allowed to instantiate those member functions of an *implicit instantiation* of a class template that are actually used [C++96]. This enables us to design class templates with optional functionality that impose additional requirements on the template arguments if and only if this functionality is used.

A good example for the generic programming paradigm is the Standard Template Library [SL95, MS96, C++96, Sil97]. The main source of its generality and flexibility stems from the

[1] pipelining, branch prediction, speculative execution and reordering, global optimizers using runtime statistics and the interplay with the cache architecture.

separation of *concepts* and *models* [Sil97]. For example, an iterator is an abstract *concept* defined in terms of requirements. A certain class is said to be a *model* of the concept if it fulfills the requirements. The iterator concept is a generalization of a pointer and the usual C-pointer is a model of an iterator. Iterators serve two purposes: They refer to an item and they traverse over the sequence of items in a container class. Container classes manage collections of items. Different categories are defined for iterators: input, output, forward, bidirectional and random-access iterators. They differ mainly in their traversal capabilities. The usual C-pointer is a random-access iterator. *Generic algorithms* in the STL are not written for a particular *container class* but for a pair of iterators instead. The so called *range* `[first,beyond)` of two iterators 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 is supposed to provide a 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. A generic `contains` function could be written as follows and will work for any model of an input iterator.

```
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 the strong type checking at compile time during the template instantiation, no need for extra storage or additional indirections during function call, and full support of inline member functions and code optimization at compile time [Str97]. One disadvantage is the lack of a formal scheme in the language for expressing the requirements of template arguments, the equivalent to the virtual base class in the object-oriented programming paradigm. This is left to the program documentation. Another disadvantage is that the flexibility is only available at compile time. Polymorphic lists at runtime cannot be implemented in this way.

In many places we follow in CGAL the generic programming paradigm to gain flexibility and efficiency. Important is the compliance of CGAL with the STL. This allows the reuse of existing generic algorithms and container classes, but – much more important – unifies the look-and-feel of the design of CGAL with the C++ Standard and is therefore easy to learn and easy to use for users familiar with the 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 retract to the concept of handles[2], which is the item denoting part of the iterator concept without traversal capabilities. Any model of an iterator is a model for a handle. Another refinement is the concept of *circulators*, a kind of iterators with slightly modified 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 make use of the object-oriented programming paradigm. For example the strategy pattern [GHJV95] has been applied to polyhedral surfaces to implement a protected access to the internal representation [Ket97], 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, a generic wrapper, as described in the Section 7. This keeps the influence of this design decision locally.

[2]Handles are already present in the STL where container classes invalidate iterators after insert or deletion operations, but they were not explicitly named as a concept.

5 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 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[3]. 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 similar 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 the circulator refers to. The operation `++c` advances the circulator by one item and `--c` steps a bidirectional circulator one item backwards. For random-access circulators `c+n` advances the circulator by `n` where `n` is a natural number. Two circulators can be compared for equality.

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 data structure this is always true if both circulators refer to items of the same data structure. In particular, `c` is always reachable from `c`. Given two circulators `c` and `d`, the range `[c,d)` denotes all circulators obtained by starting with `c` and advancing `c` until `d` is reached, but does not include `d` if $d \neq c$. 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 data structure, whereas for an iterator `i` the range `[i,i)` denotes the empty range. As long as `c != d` the range `[c,d)` behaves like an iterator range and could be used in STL algorithms. It is possible to write just as simple algorithms that work with iterators as well as with circulators, including the full range definition, see Chapter 3.9 in [Ket98]. An additional test `c == NULL` is now required that is true if and only if the data structure 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 [Ket98].

We repeat the example for the generic `contains` function from the previous Section 4 for a range of circulators. The main difference is the use of a `do-while` loop instead of a `while` loop.

```
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;
}
```

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

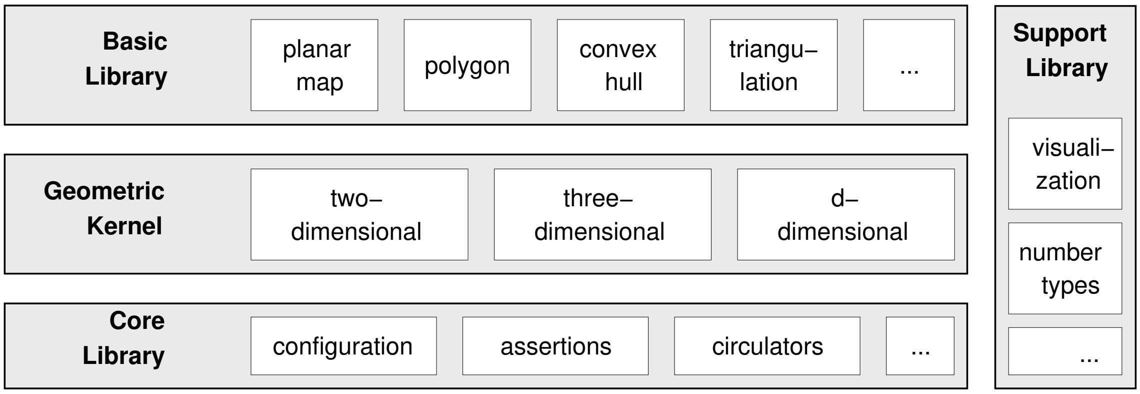

Figure 1: The structure of CGAL.

6 Library Overview

The CGAL-library is made of several modular units. In this modular structure, several bigger units can be distinguished: a core library with basic non geometric functionality, a geometric kernel, a basic library with more complicated geometric functionality, and a support library that offers supplementary functionality. The first three units can be seen as layers, built on top of each other, where the core library and the geometric kernel together are called the CGAL-kernel. The support library stands apart from the rest.

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 about 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 [Lak96]. This is discussed in more detail in Section 9.

The geometric kernel contains simple geometric objects, like points, lines, segments, triangles and tetrahedra. The criterion for simplicity is that those objects have 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, that 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 adds some support to enable the use of 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 that are needed in CGAL are quite basic, every library that supplies number types can easily be fitted in [CGAL98].

The basic library contains more complex geometric objects and data structures: polygons, planar maps, polyhedrons and such. 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 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 the 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.

7 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. With each type comes a set of functions which can be applied to an object of this type. The CGAL-kernel further contains basic operations, such as affine transformations, detection and computation of intersections, and distance computations.

The current CGAL-kernel provides two families of implementations of the geometric objects in the kernel: An implementation based on a representation of points by Cartesian coordinates and an implementation based on representation with homogeneous coordinates. The latter representation allows to reduce many computations involved in geometric algorithms to calculations over the integers, since divisions can be avoided with this representation. Both families are parameterized by the number type used to represent the (Cartesian or homogeneous) coordinates. The class templates implementing the two families are not directly visible to the user. The classes in the kernel, that are visible to a user, are class templates with a single template parameter, for example

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

Via this template argument one can choose the implementation of the geometric objects. If one chooses **CGAL_Cartesian< double >** one gets objects based on a representation with Cartesian coordinates of the double precision floating-point number type of the C++ programming language. For ease-of-use typedefs can introduce shorter names for the objects, e.g.

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

for points with homogeneous coordinates of type `long` integer.

The design goal robustness motivated the parameterization by a number type, which allows a user to choose the underlying arithmetic and influence the precision of the computations. It opens an easy way to overcome precision problems and resulting robustness problems by exact computation. For example, the number type `leda_real` of LEDA [BMS96, MNU97] can be used. This number types models a subset of algebraic numbers: All integers are `leda_real`s and `leda_real`s 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_real`s via `CGAL_Cartesian<leda_real>` guarantees exact decisions in the computations and hence guarantees exactly the same control flow in the execution of the implemented algorithm as in its theoretical counterpart. No robustness problems due to wrong and inconsistent decisions can arise with the number type `leda_real`. Furthermore, parameterization by a number type offers flexibility. Besides exact number types, e.g. arbitrary precision integer number types or rational number types based on the former, fast but potentially imprecise floating-point number types `float` and `double` can be used, if speed is more important than reliability. Floating-point number types, which allow 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 multiple precision sufficiently large for all arising results in arithmetic operations, but this requires some knowledge on the computations carried out. Thus this kind of adaptation is not easy to use. CGAL's requirements on a number type are kept small in order to make it easy to make a number type compliant with CGAL [CGAL98]. CGAL provides support for the number types of LEDA [MNU97] and for the Gnu Multiple Precision Arithmetic Library [Gra96].

The user-visible class templates give a common interface to the implementations based on homogeneous and Cartesian representation. They provide name commonality for different representations. This is used in higher-level template code. The list of requirements on the template parameter defines the concept of a *representation class* for the CGAL-kernel. A model for the concept representation class must essentially provide the names of the actual implementations. For example, the class template for models of the concept representation class for the Cartesian types looks like

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

Type names ending in `C2` denote classes based on Cartesian representation with coordinates of number type `NT`. `CGAL_Cartesian` is itself a class template and expects a model for a number type as an argument. Every template class obtained by 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 [BN94, KL96]. In particular, the representation class tells the name of an implementation class to the class template. The implementation is inherited:

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

The nested typedefs do not only provide the name of an actual implementation of a type, but also the names of related 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 geometry from the underlying linear algebra. Points and vectors are not the same, see [Gol85] 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 [Wal90]. We even do not 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 has been used to implement this operation internally as a simple conversion without any unnecessary operations. 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 results as if the addition would have been allowed. However, this is not true, since the expression within the affine transformation evaluates to a vector, not a point as in the previous expression. Vectors and points behave differently under affine transformations. For similar reasons, we do not provide automatic conversion between points and vectors.

A major design decision was to avoid a dense class hierarchy and keep the classes loosely coupled. This decision was made for the sake of efficiency and flexibility. Virtual functions and resulting space and time performance penalties are largely avoided. Wherever necessary, appropriate design pattern where applied to get polymorphic behavior. For example, intersection operations need a polymorphic return-value. The intersection of a line and a segment might be empty, a point, or the segment. So it would be convenient to have a common base class for all these possible return-types. In CGAL the return-type of an intersection routine 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:

```
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)) {
        // handle the point intersection case.
    } else if (CGAL_assign(iseg, result)) {
        // handle the segment intersection case.
    } else {
        // handle the no intersection case.
    }
}
```

Hidden to the user a class hierarchy is used. A `CGAL_Object` maintains a wrapped instance of another type. All wrapping classes have a common base:

```
class CGAL_Base {
  public:
    virtual  ~CGAL_Base() {}
};

template <class T>
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 {
  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 has the appropriate type to get assigned the object 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, a `list<CGAL_Object>` is returned in CGAL, for instance, in the case of intersection of two polygons.

Note that a class hierarchy is used here in a very localized way. 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 different internal representations using a hierarchy.

Another design decision was to make the (constant-size) geometric objects in the kernel non-modifiable. More precisely, there are no member functions to set the Cartesian coordinates of a point. Points are viewed as atomic units (see also [DeR89]) and no assumption is made on how these objects are represented. Especially there is no assumption that an implementation of points stores Cartesian coordinates. It might use polar coordinates, homogeneous coordinates, or something else. For such implementations handling member functions modifying Cartesian coordinates might be expensive and complicated. Nevertheless, for ease-of-use access functions to the Cartesian and homogeneous coordinates are available. Concerning generality this might be considered a weakness of the CGAL-kernel. However, the access functions are added to make implementing own predicates and operations more convenient. Like other libraries [BV96, Kef96, MNU97] we use a reference counting scheme for the kernel objects. The use of reference counting (copies of an object share a representation), see e.g. [Mey96] for motivation and use, is simplified by the non-modifiability, but not the reason for having non-modifiability. Using 'copy on write', reference counting with modifiable objects is possible and only slightly more involved.

8 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 and ellipse, boolean operations, and generators for geometric objects.

The two most important C++ programming techniques used in the design of the basic library are generic programming and traits classes, as described in the sequel.

8.1 Generic Data Structures

Following the generic programming paradigm as introduced in Section 4, CGAL is made compliant with the STL. The interfaces of geometric objects and data structures in the basic library make extensive use of iterators, circulators and handles, such that algorithms and data structures can be easily combined with each other and with those provided by the STL and other compliant libraries.

Triangulations are an example for a container-like data structure in the basic library. The interface contains, among others, member functions to access the vertices of the triangulation, e.g., all vertices or the vertices on the convex hull. 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 only computed 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 chosen in CGAL avoids this disadvantage:[4]

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

    Convex_hull_iterator  convex_hull_begin();
    Convex_hull_iterator  convex_hull_end();

    // ...
};
```

[4] The implementation in CGAL differs slightly, because the vertices on the convex hull are accessed using circulators (see Section 5) instead of iterators.

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

```
Triangulation  t;
// ...

Vertex  vertices[ /* ... */ ];     // large enough
copy( t.vertices_begin(), t.vertices_end(), vertices);

Vertex  convex_hull[ /* ... */ ];     // large enough
copy( t.convex_hull_begin(), t.convex_hull_end(), convex_hull);
```

Like in the previous examples, 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 the 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 the STL, thus making CGAL easy to use.

8.2 Generic Algorithms

Besides geometric data structures, the basic library also contains geometric algorithms. Instead of implementing an algorithm for a specific container, the geometric algorithms in the basic library are based on iterators, circulators, and handles. This makes them generic and compliant with the STL.

Convex hulls are an example for a geometric algorithm in the basic library. The algorithm takes a set of points as input and outputs the sequence of points on the convex hull. The following example gives a possible declaration, if we had fixed a certain container class in the algorithm's interface.

```
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 he wants to compute the convex hull from a vector of points, he has to copy the points from the vector into a list, before he can apply the algorithm. The same argument holds for the output to a specific container.

Our solution in CGAL, following the generic programming paradigm, uses iterator ranges instead of containers.

```
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 the STL, from other libraries or own implementations (provided they are compliant with the STL), or to use no container at all. The latter case is illustrated in the following example.

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

Points are taken from standard input and the points on the convex hull are written to standard output. Here so-called stream iterators [MS96] from the 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 Figure 8.2.

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

int main() {
    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));

    Delaunay_triangulation_2  dt;
    dt.insert( pts.begin(), pts.end());
    Polygon_2  ch;
    convex_hull_points_2( pts.begin(), pts.end(), back_inserter( ch));

    Window_stream window1( 512, 512, 50, 50);
    window1.init( -256.0, 255.0, -256.0);
    window1 << dt;
    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;

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

The file `tutorial.h` is used in the example programs from the CGAL-tutorial [GVW98]. It includes some CGAL header files and uses the representation class `CGAL_Cartesian<double>` to parameterize the geometric objects and algorithms via `typedef`s, in order to hide the template mechanism and the `CGAL_` prefix 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`, which is 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 the 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 the 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 `back_inserter()` cannot be used here, since it is specialized on Standard Library streams, but the `Window_stream` from CGAL is not derived from these streams.

8.3 Traits Classes for Adaptability

In Section 3 the analogy to an ideal theoretical paper on a geometric algorithm was introduced, which first declares geometric primitives and thereafter expresses the algorithm in terms of this

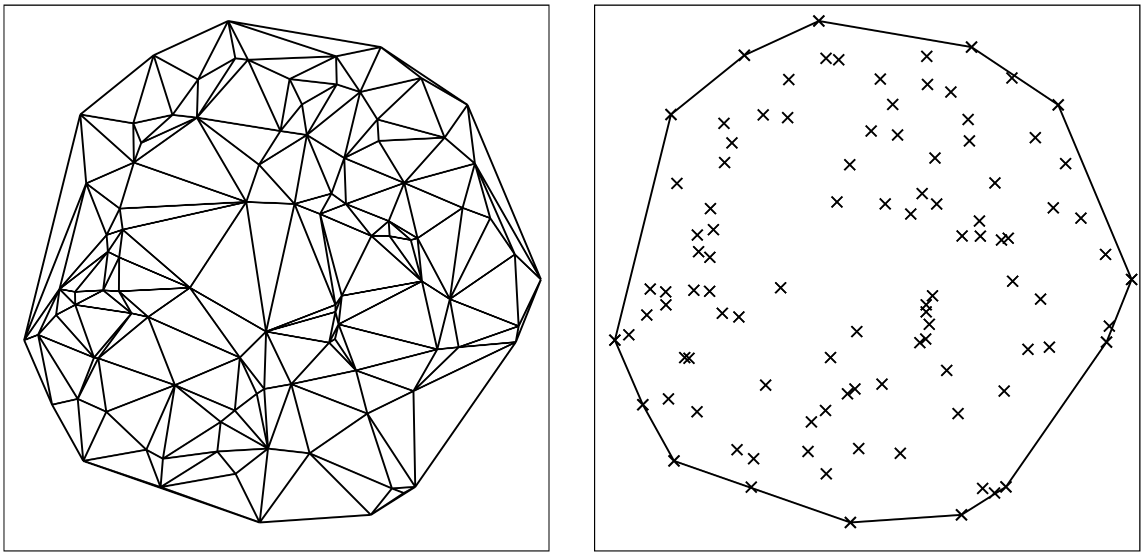

Figure 2: The output of `example_Delaunay_triangulation_Convex_hull.C`

primitives. Implementing an algorithm or data structure, we collect all necessary types and primitive operations in a single class, called *traits* class, which encapsulates such details like geometric representations. Collecting types in a single class is a template technique that is already intensively used in [BN94]. 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 [Mye95], where additional informations are associated to already existing types or built-in types. An example is the iterator concept for which a user might wish to know the value type that a model of an iterator refers to. This can be easily encoded as a local type for iterators implemented as classes.

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

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

```
template < class Iterator >
struct iterator_traits {
    typedef  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 *partial specialization* of the iterator traits class can be used.

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

Our approach with 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 get 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, e.g.,

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

Note, that each primitive could be provided by a single template parameter as well, but using traits classes simplifies the interface by putting all these primitives in one single argument and makes it easier to apply already prepared implementations of traits classes.

8.4 Implementing (with) Traits Classes

A model for the concept of traits classes must provide the geometric primitives required by the geometric data structure or algorithm. The basic library provides families of models based on the geometric objects and predicates from the CGAL-kernel. Each family in the basic library is realized as a class template parameterized by a representation class, (see Section 7), e.g. `CGAL_Triangulation_euclidean_traits_2< R >`, Remember that via the representation class the implementation of the objects from the kernel is chosen.

To give an example, the data structure `CGAL_Triangulation_2` needs, among others, points, segments, triangles and an orientation predicate, thus one family of models for the concept of traits classes 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 like

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

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

Up to now, we described the interface as it is presented to the user. The following code fragment shows some implementation details related to the concept of traits classes.[5]

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

[5] The actual CGAL-code differs slightly from the example, but the usage of the traits class is the same.

```
    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. During 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 access the additional data, the `orientation` member function is not declared `static`.[6]

8.5 Default Traits Classes

For algorithms implemented as functions, a default traits class is chosen automatically, if not provided in the function call. Thus the user can just ignore the traits class mechanism. The following example demonstrates this.

```
typedef  /* ... */          R;              // some representation type
typedef  CGAL_Point_2<R>    Point_2;
typedef  CGAL_Polygon_2<R>  Polygon_2;

const int  n = 100;
Point_2    pts[ n];
Polygon_2  hull;
// ...                                      // fill pts

CGAL_convex_hull_points_2( pts, pts+n, back_inserter( hull));
```

In the call to the convex-hull algorithm no traits class is visible to the user, but it is chosen silently by the compiler. How? Let us have a look at the definition of `CGAL_convex_hull_points_2`.

```
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 `Point` type of the input points is the value type of `InputIterator`. It is determined via iterator traits as described in Section 8.3. Since `Point` is a CGAL-point, it 'knows' its representation type `R`. Finally, another version of `CGAL_convex_hull_points_2` taking four arguments is called. The additional parameter is `CGAL_convex_hull_traits_2<R>()`, which is a model for the traits

[6] 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`.

class concept for convex-hull algorithms. The 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 works for iterator ranges with value type `CGAL_Point_2<R>` for any representation type `R`.

8.6 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 in 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 possibly large application based on an own point type, e.g., `leda_rat_point` from LEDA, and wants to compute the convex hull of a point set with the CGAL algorithm `CGAL_ch_graham_andrew` [Sch98b]. The only thing the user has to supply is a model for the concept of traits classes for the convex-hull algorithm. This model has to provide a point type `Point_2` and predicates `Less_xy` and `Leftturn`, as shown below.

```
#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));
        }
    };
};

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

The functions **compare** and **left_turn** used for the predicates are provided by LEDA. 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**.[7]

Another way of adapting CGAL is to change only a small part of a model for the concept of traits classes. This allows the user to do only the necessary modifications, while re-using most of the code from the library. We give an example showing the computation of the Delaunay triangulation of a point set using the maximum metric (L_∞) instead of the Euclidean metric (L_2).

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

typedef  /* ... */  R;                   // some representation type
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 model **L_infty_traits** of a traits class is derived from the standard model for triangulations provided by CGAL. Only the two predicates **extremal** and **side_of_oriented_circle** have to be re-defined for using the maximum metric. All other inherited primitives remain unchanged, since they do not depend on the chosen metric.[8]

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 model do the projection by simply ignoring the additional number and using only 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>
```

[7] CGAL already provides a traits-class model for the convex-hull algorithms using the point type **leda_rat_point**, namely the family **CGAL_convex_hull_rat_leda_traits_2**.

[8] CGAL already provides a family of such models for the traits-class concept, namely **CGAL_Triangulation_l_infty_traits_2**.

```
#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)
        : _point( point), _level( level) { }
    const Point&  point() const { return( _point); }
    const Level&  level() const { return( _level); }
  private:
    Point _point;
    Level _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.[9]

9 Software Engineering in the Project

The birth of the CGAL-library dates back to a meeting in Utrecht in January 1995. Since then the five authors started developing the kernel. The CGAL-project started officially in October 1996 and the team of developers had grown to circa 25 people, mostly research assistants, PhD students and postdocs in academia, which 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 1.0 is in preparation and consists currently (January 1998) of approximately 70000 lines of C++ source code[10] for the library, plus 30000 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 on its way to a large project, comparable to smaller operating systems or database management systems [Fai85]. 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[11] and email discussion groups are a matter of course, but major progress in the design was mostly made on implementers meetings. An early modularization of the library, see Section 6, with as little dependencies among the modules as possible was crucial to the project in order to keep the communication needs reasonable between the project partners. 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 such that the first internal release was ready with 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 [Fai85] 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, Utrecht. The author writes a specification for a package in a format of a reference manual page and submits it to the editorial committee for discussion and approval. The editorial committee takes care about 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) [Wil92, Knu84, KL94, SS91]. 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 blinders one naturally evolves when a first implementation has been fixed. A further discussion can be found in [PC86]. This separation is also supported by our manual-writing tools described below. In the next step, the author sends the package to the external tester at a distinct project site. Thereafter, the package gets 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 on developer

[9] Projections of `CGAL_Point_3` onto the xy-, xz-, and yz-plane for using them in triangulations are available in CGAL, see `CGAL_Triangulation_euclidean_traits_xy_2`.

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

[11] `http://www.cs.ruu.nl/CGAL/`

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 [Mag93, Str97] placed in the implementation[12]. 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 [MNS+96]. 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[13]. 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 reviews again 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 gets integrated.

Design and specifications are communicated with reference manual pages. First 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 together with the other manuals, such as the tutorial, will determine the acceptance of CGAL. A design is not a good design if we cannot communicate it.

In order to provide appealing, high-quality manual pages we use LaTeX. A self-written style file adds additional formatting capabilities as can be seen in Figure 9, which displays an excerpt of the manual page for two-dimensional points in CGAL. The principal manual page layout is intentionally close to the LEDA user manual [MNU97], albeit 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 automatisms allow flexible layouts whenever an entry gets too long. Reduction rules remove `const` ... `&` declarations from function arguments (they are considered as implementation details), remove the current class name from function arguments and rewrite operator declarations in operator notation, see the operator examples in Figure 9. This is all done automatically using original C++ declarations within the LaTeX source. Summarizing, a short member function can be documented efficiently in a single line.

Our view on the developing process strictly separates specification and implementation. The tools provided for the manual writing share this view. A scenario how this tools can be applied together with literate programming tools for the implementation part is shown in Figure 9. The left side illustrates the formatting of the specification with LaTeX and the style file `cc_manual.sty` as well as the automatic conversion of the specification to an HTML online manual with the script `cc_manual_to_html`. The HTML online manual makes full use of the hyperlinks for cross-referencing. An index of all classes, functions, constants, enums etc. is thereby generated. The right side illustrates the use of literate programming tools with the two possible transformation steps, `tangle` and `weave`. In the middle the C++ source code gets extracted partially from the specification and partially from the literate programming source. The tool `cc_extract` extracts the C++ declarations written in the LaTeX-file of the specification to produce the public part of a C++ header file. The tool `cc_check` can be used in further refinements to check whether all declarations that are part of a specification are really present in the implementation.

Other approaches feature C++ comments to encode the reference manual within the C++ source code. Their advantage is that during further refinements it is easy to update the documentation, which is located nearby in the same file. Our approach uses C++ declarations within the reference manual and provide tools to maintain integrity. The benefit is the cleaner separation of the spec-

[12] 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.

[13] 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 $\mathbb{E}_2$.

Remember that *R::RT* and *R::FT* denote a ring type and a field type. For the representation class *CGAL_Cartesian*<*T*> the two types are equivalent. For the representation class *CGAL_Homogeneous*<*T*> the ring type is *R::RT* == *T*, and the field type is *R::FT* == *CGAL_Quotient*<*T*>.

#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_Bbox_2	*p.bbox()*	returns a bounding box containing p. Note that bounding boxes are not parameterized with whatsoever.
CGAL_Point_2<*R*>	*p.transform(CGAL_Aff_transformation_2*<*R*> *t)*	returns the point obtained by applying t on p.

The following operations can be applied on points:

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-R1.0. The equality test, the coordinate accessors, and the example subsection are omitted here.

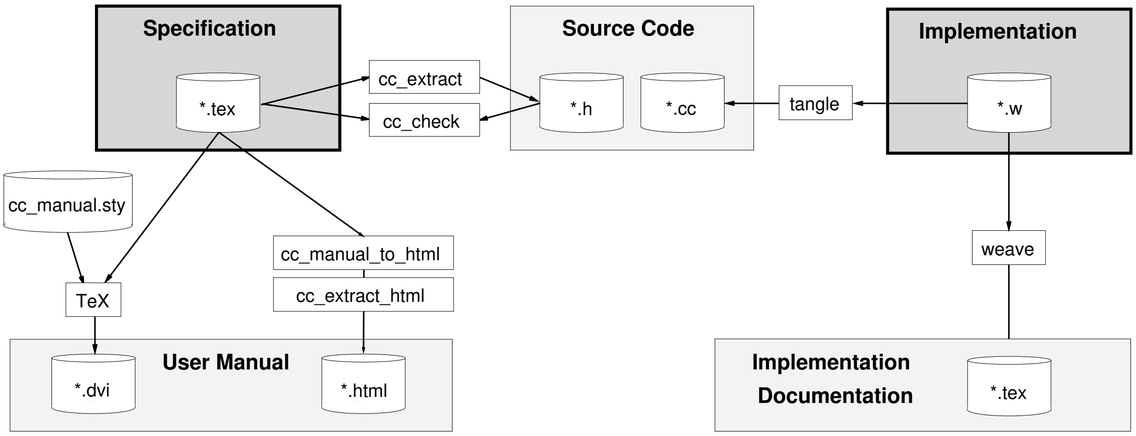

Figure 4: Overview of the developing process in CGAL. The files and tools involved are shown: Specification on the left, implementation on the right.

ification and the implementation task. Furthermore, it follows the conceptual idea of the literate programming tools to write programs embedded in their specification, not comments in programs. This slight conceptual shift puts the writing task of the specification at the first place, in accordance with the priority we gave it. The use of literate programming tools for the implementation documentation widens the gap even further. Specifications that are placed in C++ comments are now embedded in the implementation documentation, i.e. at nesting level three. Furtheron, additional manuals can be written with our scheme without cluttering the source code and checking is still possible.

10 Evaluation of the Design

The sections above illustrate the concepts and techniques we use in CGAL to accomplish the design goals from Section 3. 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: Neither the container must know the algorithms, nor the algorithm must know the container. 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 to write new algorithms or data structures that interface CGAL – similar as CGAL does interface with the STL. CGAL is open to the standards defined with the STL and allows the adaptation towards 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 for certain tools and the use of assertions and program checkers. The strong modularization and large independences help in testing and achieving correctness. Another strong point to achieve correctness is modularity and adaptability, which allow the combination of different modules according to the need of a particular solution. For example the adaptability for 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 result 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 through their similarity to the iterator concept. The concepts in the kernel behave similar: 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, such that the C++ novice sees only simple classes with a selected default representation and number type. When users gets familiar with the templates in the kernel, the representation class provides a uniform look on 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 with only a few new concepts. It 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 easy memorizable interfaces.

First practical teaching experiences have been made during a summer course at ETH Zurich, Switzerland, in 1997 with the use of the STL and the CGAL-kernel. Five students have used the geometric primitives, intersection computations, visualization, STL container classes, and iterators successfully within a week. Their previous knowledge had ranged from C++-novices up to a medium knowledge of the STL.

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 would assume points always implemented with Cartesian coordinates. However, as discussed in Section 7, this would restrict flexibility. An interesting issue concerning efficiency is reference counting, see e.g. [Mur93, Mey96]. Reference counting can be a source of efficiency, time efficiency as well as space efficiency due to fewer copies of an object. However, especially if the objects are small and copies made rarely, 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 Section 8 is about 40% faster with points not using reference counting, if the points coordinates are `double`s. If the coordinates are maintained 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.

Acknowledgments

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 days of implementing 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 (born Schwarzkopf), Jochen Comes, Olivier Devillers, Katrin Dobrindt, Eyal Flato, Wolfgang Freiseisen, Bernd Gärtner, Dan Halperin, Iddo Hanniel, Sariel 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, Frank Theiß, Christian Uhrig, Carl Van Geem, Remco Veltkamp, Emo Welzl, Wieger Wesselink, Peter Widmayer, Martin Will, Alexander Wolff, and Mariette Yvinec. We also want to thank Dietmar Kühl and Karsten Weihe for valuable comments and the polymorphism solution with the class `CGAL_Object`.

References

[Ame97] N. Amenta. Computational geometry software. In *Handbook of Discrete and Computational Geometry*, pages 951–960. CRC Press, 1997.

[Avn94] F. Avnaim. *C++GAL: A C++ Library for Geometric Algorithms.* INRIA Sophia-Antipolis, 1994.

[BMS94] C. Burnikel, K. Mehlhorn, and S. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimental results. In *Proc. 2nd Annu. European Sympos. Algorithms*, volume 855 of *Lecture Notes Comput. Sci.*, pages 227–239. Springer-Verlag, 1994.

[BMS96] 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.

[BN94] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++*. Addison-Wesley, Reading, MA, 1994.

[BP97] J.-D. Boissonnat and F. Preparata. Robust plane sweep for intersecting segments. Technical Report 3270, INRIA, Sophia-Antipolis, France, September 1997.

[BTV97] J. E. Baker, R. Tamassia, and L. Vismara. GeomLib: Algorithm engineering for a geometric computing library, 1997. (Preliminary report).

[BV96] G. Booch and M. Vilot. Simplifying the booch components. In S. Lippman, editor, *C++ Gems*, pages 59–89. SIGS publications, 1996.

[C++96] Working paper for draft proposed international standard for information systems – programming language C++. Doc. No. X3J16/96 - 0225 - WG21/N1043, December 1996. `http://www.maths.warwick.ac.uk/c++/pub/`.

[CGAL98] CGAL consortium. Number types. In Hervé Brönnimann, Stefan Schirra, and Remco Veltkamp, editors, *CGAL Reference Manual. Part 3: Support Library.* 1998. CGAL R1.0. `http://www.cs.ruu.nl/CGAL`. to appear.

[DeR89] A. D. DeRose. Geometric programming: A coordinate-free approach. In *Theory and Practice of Geometric Modeling*, Blaubeuren, FRG (Oct 1988), 1989. Springer-Verlag.

[dRJ93] P. de Rezende and W. Jacometti. Geolab: An environment for development of algorithms in computational geometry. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 175–180, Waterloo, Canada, 1993.

[EKK+94] P. Epstein, J. Kavanagh, A. Knight, J. May, T. Nguyen, and J.-R. Sack. A workbench for computational geometry. *Algorithmica*, 11:404–428, 1994.

[Fai85] Richard Fairley. *Software Engineering Concepts.* McGraw-Hill Series in Software Engineering and Technology. McGraw-Hill, 1985.

[FGK+96] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. The cgal kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *ACM Workshop on Applied Computational Geometry*, pages 191–202, Philadelphia, Pennsylvenia, May, 27–28 1996. Lecture Notes in Computer Science 1148.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gie94] G.-J. Giezeman. *PlaGeo, a library for planar geometry, and SpaGeo, a library for spatial geometry.* Utrecht University, 1994.

[Gol85] Ronald N. Goldman. Illicit expressions in vector algebra. *ACM Transaction on Graphics*, 4(3):223–243, July 1985.

[Gra96] T. Granlund. *GNU MP, The GNU Multiple Precision Arithmetic Library*, 2.0.2 edition, June 1996.

[GVW98] Geert-Jan Giezeman, Remco Veltkamp, and Wieger Wesselink. *Getting Started with CGAL*, 1998. CGAL R1.0. `http://www.cs.ruu.nl/CGAL`. to appear.

[Kef96] T. Keffer. The design and architecture of Tools.h++. In S. Lippman, editor, *C++ Gems*, pages 43–57. SIGS publications, 1996.

[Ket97] Lutz Kettner. Designing a data structure for polyhedral surfaces. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, 1998, to appear. Preliminary version available as Technical Report #278, Department Informatik, ETH Zürich, Switzerland, December 1997.

[Ket98] Lutz Kettner. Circulators. In Hervé Brönnimann, Stefan Schirra, and Remco Veltkamp, editors, *CGAL Reference Manual. Part 3: Support Library*. 1998. CGAL R1.0. `http://www.cs.ruu.nl/CGAL`. to appear.

[KL94] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation*, version 3.0 edition, 1994.

[KL96] K. Kreft and A. Langer. Iterators in the standard C++ library. *C++ Report*, 8(10):27–32, Nov.-Dec. 1996.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[Lak96] John Lakos. *Large Scale C++ Software Design*. Addison-Wesley, 1996.

[Lee96] D. T. Lee. Visualizing geometric algorithms – state of the art. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry (Proc. WACG '96)*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 45–50. Springer-Verlag, 1996.

[Lip96] Stanley B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996.

[LPT97] Giuseppe Liotta, Franco P. Preparata, and Roberto Tamassia. Robust proximity queries: an illustration of degree-driven algorithm design. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 156–165, 1997.

[Mag93] Steve Maguire. *Writing Solid Code*. Microsoft Press, 1993.

[Meh96] Kurt Mehlhorn. Position paper for panel discussion. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry (Proc. WACG '96)*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 51–52. Springer-Verlag, 1996.

[Mey92] Scott Meyers. *Effective C++*. Addison-Wesley, 1992.

[Mey96] Scott Meyers. *More Effective C++*. Addison-Wesley, 1996.

[MN94] K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *13th World Computer Congress IFIP94*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.

[MN95] K. Mehlhorn and S. Näher. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.

[MNS+96] Kurt Mehlhorn, Stefan Näher, Thomas Schilz, Stefan Schirra, Michael Seel, Raimund Seidel, and Christian Uhrig. Checking geometric programs or verification of geometric structures. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 159–165, 1996.

[MNU97] K. Mehlhorn, S. Näher, and C. Uhrig. *The LEDA User Manual, Version 3.5*. Max-Planck-Insitut für Informatik, 66123 Saarbrücken, Germany, 1997. `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.

[Mur93] Robert B. Murray. *C++ Strategies and Tactics*. Addison-Wesley, 1993.

[Mye95] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[NSdL+91] Jürg Nievergelt, Peter Schorn, Michele de Lorenzi, Christoph Ammann, and Adrian Brüngger. XYZ: A project in experimental geometric computation. In *Proc. Computational Geometry: Methods, Algorithms and Applications*, volume 553, pages 171–186. Springer-Verlag, 1991.

[Ove96] Mark H. Overmars. Designing the Computational Geometry Algorithms Library CGAL. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry (Proc. WACG '96)*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 53–58. Springer-Verlag, 1996.

[PC86] 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.

[Sch91] Peter Schorn. Implementing the XYZ GeoBench: A programming environment for geometric algorithms. In *Computational Geometry — Methods, Algorithms and Applications: Proc. Internat. Workshop Comput. Geom. CG '91*, volume 553 of *Lecture Notes Comput. Sci.*, pages 187–202. Springer-Verlag, 1991. `http://wwwjn.inf.ethz.ch/geobench/XYZGeoBench.html`.

[Sch96] 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.

[Sch98a] S. Schirra. Precision and robustness issues in geometric computation. In *Handbook on Computational Geometry.* Elsevier Science Publishers, Amsterdam, The Netherlands, 1998.

[Sch98b] 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.

[Sil97] Silicon Graphics Computer Systems, Inc. Standard template library programmer's guide. `http://www.sgi.com/Technology/STL/`, 1997.

[SL95] Alexander Stepanov and Meng Lee. The standard template library. `http://www.cs.rpi.edu/~musser/doc.ps`, October 1995.

[SS91] Lisa M. C. Smith and Mansur H. Samadzadeh. An annotated bibliography of literate programming. *ACM SIGPLAN Notices*, 26(1):14–20, January 1991.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3rd edition, 1997.

[Vel97] R. C. Veltkamp. Generic programming in cgal, the computational geometry algorithms library. In *Proceedings of the 6th Eurographics Workshop on Programming Paradigms in Graphics*, 1997.

[Wal90] Bob Wallis. Forms, vectors, and transforms. In Andrew S. Glassner, editor, *Graphics Gems*, pages 533–538. Academic Press, 1990.

[Wil92] Ross N. Williams. *FunnelWeb User's Manual*, V1.0 for FunnelWeb V3.0 edition, May 1992.

Contents

I N F O R M A T I K

Below you find a list of the most recent technical reports of the research group *Efficient Algorithms* at the Max-Planck-Institut für Informatik. They are available by anonymous ftp from our ftp server `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Birgit Hofmann
Im Stadtwald
D-66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-98-1-006	K. Jansen	A new characterization for parity graphs and a coloring problem with costs
MPI-I-98-1-005	K. Jansen	The mutual exclusion scheduling problem for permutation and comparability graphs
MPI-I-98-1-004	S. Schirra	Robustness and Precision Issues in Geometric Computation
MPI-I-98-1-003	S. Schirra	Parameterized Implementations of Classical Planar Convex Hull Algorithms and Extreme Point Compuations
MPI-I-98-1-002	G.S. Brodal, M.C. Pinotti	Comparator Networks for Binary Heap Construction
MPI-I-98-1-001	T. Hagerup	Simpler and Faster Static AC^0 Dictionaries
MPI-I-97-1-028	M. Lermen, K. Reinert	The Practical Use of the $\mathcal{A}^*$ Algorithm for Exact Multiple Sequence Alignment
MPI-I-97-1-027	N. Garg, G. Konjevod, R. Ravi	A polylogarithmic approximation algorithm for group Steiner tree problem
MPI-I-97-1-026	A. Fiat, S. Leonardi	On-line Network Routing - A Survey
MPI-I-97-1-025	N. Garg, J. Könemann	Faster and Simpler Algorithms for Multicommodity Flow and other Fractional Packing Problems
MPI-I-97-1-024	S. Albers, N. Garg, S. Leonardi	Minimizing Stall Time in Single and Parallel Disk Systems
MPI-I-97-1-023	S.A.M.A.P. Leonardi	Randomized on-line call control revisited
MPI-I-97-1-022	E. Althaus, K. Mehlhorn	Maximum Network Flow with Floating Point Arithmetic
MPI-I-97-1-021	J.F. Sibeyn	From Parallel to External List Ranking
MPI-I-97-1-020	G.S. Brodal	Finger Search Trees with Constant Insertion Time
MPI-I-97-1-019	D. Alberts, C. Gutwenger, P. Mutzel, S. Näher	AGD-Library: A Library of Algorithms for Graph Drawing
MPI-I-97-1-018	R. Fleischer	On the Bahncard Problem
MPI-I-97-1-017	S. Albers, M.R. Henzinger	Exploring Unknown Environments
MPI-I-97-1-016	M. Thorup	Faster deterministic sorting and priority queues in linear space