On the Design of CGAL, the Computational Geometry Algorithms Library

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

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

On the Design of CGAL, the Computational Geometry Algorithms Library

Andreas Fabri — Geert-Jan Giezeman — Lutz Kettner

Stefan Schirra — Sven Schönherr

N° 3407

April 1998

THÈME 2

ISSN 0249-6399

INRIA
SOPHIA ANTIPOLIS

On the Design of CGAL, the Computational Geometry Algorithms Library

Andreas Fabri* , Geert-Jan Giezeman† , Lutz Kettner‡
Stefan Schirra§ , Sven Schönherr¶

Thème 2 — Génie logiciel
et calcul symbolique

Projet Prisme

Rapport de recherche n° 3407 — April 1998 — 37 pages

Abstract: CGAL is a *Computational Geometry Algorithms Library* written in C++. The goal is to make the large body of geometric algorithms developed in the field of computational geometry available for industrial application. In this report 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.

Key-words: Computational geometry, software, library

A revised version of this paper will appear in "TRENDS IN SOFTWARE", Volume on *Algorithm Engineering*, edited by Dorothea Wagner. Work on this paper has been supported by ESPRIT LTR Project No. 21957 (CGAL). Work by Andreas Fabri was done while he was working at INRIA Sophia-Antipolis, in the Prisme project.

* ABB Corporate Research Ltd., CH-5405 Baden/Dättwil, Switzerland,
† Department of Computer Science, Utrecht University, N-3508 TB Utrecht, The Netherlands
‡ Institute for Theoretical Computer Science, ETH Zentrum, CH-8092 Zürich, Switzerland,
§ Max-Planck-Institut für Informatik, D-66123 Saarbrücken, Germany,
¶ Fachbereich Mathematik und Informatik, Freie Universität Berlin, D-14195 Berlin, Germany.
Email: andreas.fabri@chcrc.abb.ch, geert@cs.ruu.nl, kettner@inf.ethz.ch, stschirr@mpi-sb.mpg.de, sven@inf.fu-berlin.de.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Sur la conception de CGAL, une bibliothèque d'algorithmes géométriques

Résumé : CGAL (*Computational Geometry Algorithms Library*) est une bibliothèque d'algorithmes géométriques écrite en C++. Le but de cette recherche est de faciliter le transfert vers l'industrie des algorithmes géométriques conçus par la communauté internationale de géométrie algorithmique. Dans ce rapport, nous discutons les principales caractéristiques de CGAL, à savoir l'efficacité, la flexibilité, la facilité d'utilisation, et la robustesse. Nous présentons une approche pour atteindre ces buts basée sur la programmation générique et la paramétrisation qui jouent un rôle essentiel dans l'architecture de CGAL. Nous introduisons la programmation générique en C++, la comparons à l'approche orientée objets, et présentons des exemples d'utilisation effectives des deux approches dans CGAL. Nous présentons aussi une synthèse de la structure de CGAL, et discutons les aspects de génie logiciel dans le projet CGAL.

Mots-clés : Géométrie algorithmique, génie logiciel, bibliothèque de programmes

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

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

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 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 $\texttt{d} \neq \texttt{c}$. 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.