ELSEVIER

Computational Geometry 13 (1999) 65–90

Computational Geometry
Theory and Applications

www. elsevier.nl/locate/comgeo

Using generic programming for designing a data structure for polyhedral surfaces

Lutz Kettner *,[1]

Institute of Theoretical Computer Science, ETH Zürich, CH-8092 Zürich, Switzerland

Communicated by J. Rossignac; submitted 22 October 1998; accepted 4 January 1999

Abstract

Software design solutions are presented for combinatorial data structures, such as polyhedral surfaces and planar maps, tailored for program libraries in computational geometry. Design issues considered are flexibility, time and space efficiency, and ease-of-use. We focus on topological aspects of polyhedral surfaces and evaluate edge-based representations with respect to our design goals. A design for polyhedral surfaces in a halfedge data structure is developed following the generic programming paradigm known from the Standard Template Library STL for C++. Connections are shown to planar maps and face-based structures.

Keywords: Library design; Generic programming; Combinatorial data structure; Polyhedral surface; Halfedge data structure

1. Introduction

Combinatorial structures, such as planar maps, are fundamental in computational geometry. In order to be useful in practice, a solid library for computational geometry must provide generic and flexible solutions as one of its fundamental cornerstones. We report a revised solution proposed for the Computational Geometry Algorithms Library CGAL,[2] which is a joint effort of nine academic institutes in Europe [6,7,28]. Our design criteria are flexibility, ease-of-use, time and space efficiency. Flexibility is an obvious goal for a library. Ease-of-use is considered to make the design easy to learn and accessible for non-experts. Efficiency is needed to qualify for industrial applications.

We focus on edge-based representations of three-dimensional polyhedral surfaces and illustrate connections to planar maps and face-based structures, which maintain polygons with holes for their

* E-mail: kettner@inf.ethz.ch

[1] Part of this research was carried out while the author was at the Freie Universität Berlin with a scholarship from the Graduiertenkolleg "Algorithmische Diskrete Mathematik", DFG We 1265/2-1. Support from the Swiss Federal Office for Education and Science (Projects ESPRIT IV LTR No. 21957 CGAL and No. 28155 GALIA) is also acknowledged.

[2] `http://www.cs.uu.nl/CGAL/`

0925-7721/99/$ – see front matter
PII: S0925-7721(99)00007-3

facets. We concentrate on the topological aspects and derive solutions applicable to other data structures as well. In particular, we want to vary the internal storage organization and the kind of incidences that are actually stored. Additional user data can be integrated easily. A top-level interface ensures ease-of-use and combinatorial integrity. On the other hand, a protected access to the internal representation is granted.

In the first part of the paper we define polyhedral surfaces and review known edge-based boundary representations. The second part we begin with a short introduction to the modern design principles available in C++ known as the generic programming paradigm [25,26] with the example STL, the Standard Template Library [15,24,31]. We derive design goals and evaluate previous work. We continue with an overview of our design, the separation of geometry and topology, the layered structure, the responsibilities of the layers, several example programs, and the realization in C++ using templates. To conclude the design, we present a technique for the protected access of the internal representation and the new concept of circulators.

The three main advantages of our design are the following. First, the flexibility is completely resolved at compile time, i.e., there is no runtime overhead due to the flexibility. Second, memory is only allocated for the features actually used. For example, a polyhedron with no information in facets does not allocate facet nodes and facet pointers at all. Third, these benefits have not sacrificed the ease-of-use of the data structure. Just on the contrary, the generic programming paradigm, which we used to achieve the flexibility, enabled us also to encapsulate the flexibility. For example, the top level interface of the data structure looks always the same, despite of the representation that has been chosen underneath. For completeness, we include the realization of this design in C++. Note, however, that it only has to be understood by the developer and not by the user.

2. Polyhedral surfaces

A boundary representation of a polyhedral surface consists of a set of vertices V, a set of edges E, a set of facets F, and an incidence relation on them. Introductions can be found in [13,20]. For a living example see Fig. 1.

The two types of boundary representations are 2-manifold and non-manifold surfaces. For each point on a 2-manifold surface there exists a neighborhood that is homeomorphic to the open disc. A non-manifold example would be two tetrahedra glued together at a single vertex or a common edge. The next distinction is between orientable and non-orientable 2-manifold surfaces. Without going into details, a surface is orientable if a consistent orientation can be assigned to each facet such that for each edge the two incident facets have opposite orientations at this edge. An example of a non-orientable 2-manifold is the Klein bottle. We consider only orientable 2-manifolds for our data structure.

The natural operations under which 2-manifolds are closed are Euler operations; four of them are shown in Fig. 2. The principal characteristic of an Euler operation is the invariance of the Euler–Poincaré formula. A sufficiency proof for a specific set of Euler operations can be found in [20]. Note that 2-manifolds are not closed under Boolean operations.

The class of representable surfaces is further restricted by the kind of geometry associated with vertices, edges and facets. Vertices map to points in $\mathbb{R}^3$. As far as polyhedra are concerned, the edges are typically the straight line segments between their two endpoints and the facets are simple, planar

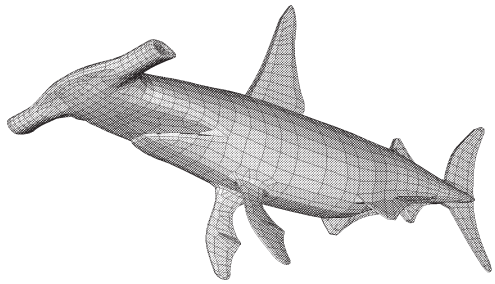

Fig. 1. Hammerhead, an orientable 2-manifold of 2560 vertices. This one is homeomorphic to a sphere.

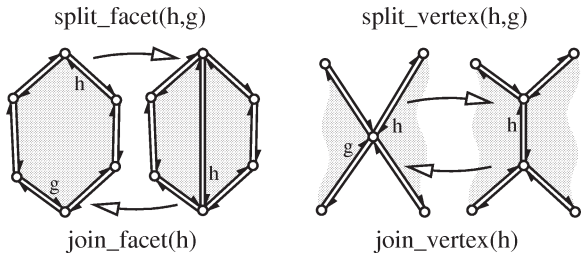

Fig. 2. Euler operator examples for polyhedral surfaces.

polygons. Other classes might allow curved surfaces as facets or might use vertices in projective space to model unbounded facets as well. However, they are currently not provided in CGAL.

We now present a definition for polyhedral surfaces following Steinitz [32]. This definition is of combinatorial nature, which makes reasoning about the data structure more convenient, for example, that the same facet cannot appear on both sides of an edge. And it leads directly to the integrity definition and related test function of the polyhedral surface data structure. The definition captures the same class of representable surfaces informally introduced so far, but may require a different combinatorial representation. For example, a facet that is incident to both sides of the same edge needs to be split into two facets.

Definition 1. A *structural complex* is a union $C = V \cup E \cup F$ of three disjoint sets together with an incidence relation. We call V the vertices, E the edges and F the facets of the structural complex. The incidence relation on C must be symmetric. No two elements from the same set V, E or F are incident. If $v \in V$ is incident to $e \in E$ and e is incident to $f \in F$ then v is incident to f.

Definition 2. A *polyhedral complex* is a structural complex with four additional conditions:
(1) Every edge is incident to two vertices.
(2) Every edge is incident to two facets.
(3) For every incident pair v, f, there are exactly two edges incident to both.
(4) Every vertex and every facet is incident to at least one other element.

The *neighborhood* of a vertex are the edges and facets incident to the vertex. If we restrict the incidence relation to this neighborhood then by condition (3) each facet is incident to exactly two edges and

by condition (2) each edge is incident to exactly two facets. Thus, the neighborhood decomposes into disjoint cycles, where each cycle is an alternating sequence of edges and facets. A polyhedral complex is a 2-manifold if and only if the neighborhood of each vertex decomposes into a single cycle. The definition of a polyhedral complex is symmetric for vertices and facets. A symmetrically defined neighborhood of a facet decomposes into cycles of incident edges and vertices. Assuming that the neighborhood of each facet is a single cycle (geometrically, the boundary of the facet is a single connected component, so the facet has no holes), we can define a polyhedral complex to be *oriented* if each cycle around a facet is oriented and if, for each edge, the two cycles of its two incident facets are oriented in opposite directions. A polyhedral complex is *orientable* if there exists such an orientation.

Definition 3. The *boundary representation of a polyhedron* is an orientable polyhedral complex, where the neighborhood of each vertex and each facet is a single cycle, together with a mapping $V \to \mathbb{R}^3$. This mapping extends for edges by mapping each edge to the open, straight line segment between its two endpoints. The following additional conditions must hold:

(5) The image of the neighborhood of each facet is the boundary of a simple, planar polygon. The mapping extends for facets to the open region of these polygons.
(6) For all elements in $C = V \cup E \cup F$, their images are pairwise disjoint.

The surface defined by such a boundary representation is an orientable 2-manifold. Some useful properties are for example that the neighborhoods of two vertices have at most one edge and two facets in common, the edge and vertex graphs are connected within each connected component of the surface, and each facet has at least three edges on its boundary. The smallest possible configuration is a tetrahedron.

The closed surfaces considered so far can be extended to surfaces with boundaries. We need to relax condition (2) to allow edges that are incident to one facet; they are called *border edges*. This induces a modification of Definition 3: the neighborhood of a vertex decomposes into either a single cycle or a collection of open paths going from border edge to border edge. Although the surface is no longer closed, the orientation still defines a "solid" side of the surface. The minimal configuration for surfaces with boundaries is a triangle. The data structures we will describe can be used for polyhedra as well as for surfaces with boundaries. Border edges are simply marked by the missing facet at one side, i.e., by a null pointer.

A suitable data structure based on Definition 3 for polyhedral surfaces has been successfully used in a project on contour-edge-based polyhedron visualization, where we take advantage of the strict properties imposed by the definitions: for example, the definition for contour-edges is based on the orientable 2-manifold property, and the lack of holes in facets simplifies our algorithms [17]. An initial implementation of the data structure made it easy to compute the silhouette for a polyhedral surface [14]. The extension of this data structure design and their advantages are presented in the following sections.

In visualization, a polyhedron is usually represented as a so-called indexed facet set, which is a sequence of points, followed by a sequence of facets. Each facet is a list of indices referring to the sequence of points. Shared edges must be derived implicitly from the points shared by facets. Examples are the internal representation as well as the file formats of VRML [12], Open Inventor [36] or the Object File Format OFF used by GeomView [29]. These formats were not strict enough for our purposes since they can represent non-manifolds, non-orientable 2-manifolds, and may also violate condition (3) of a polyhedral complex. However, they cannot represent holes in facets.

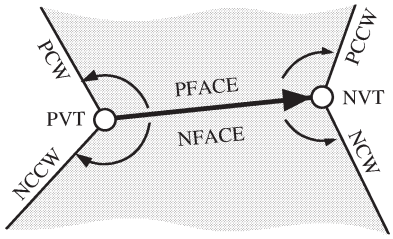

Fig. 3. Winged-edge.

3. Data structures for boundary representations

The following survey of edge-based data structures addresses their sufficiency for modeling topology and the efficiency of their primitive operations and storage costs. The representative example chosen is the traversal around a vertex from one edge to the next edge in counterclockwise direction.

3.1. Winged-edge data structure

The winged-edge data structure [1,10] stores, for each oriented edge, eight references: two vertices (PVT, NVT), two faces (PFACE, NFACE) and four incident edges that share the same faces and vertices (PCW, PCCW, NCW and NCCW), the so-called *wings*, see Fig. 3. An edge is oriented from the source vertex PVT to the target vertex NVT. The face PFACE is to the left of the oriented edge when the surface is seen from the outside.

This data structure is able to model orientable 2-manifolds. It is even sufficient for curved-surface environments where loops and multi-edges are allowed [35]. The basic operations include traversal around a vertex and around a facet. High-level operations maintaining integrity are Euler operators. The next edge counterclockwise around a vertex `v` for an edge `e` is equal to `e->PCW` if `e->PVT == v` and `e->NCW` otherwise.

Variants are possible where vertex and facet pointers can even be omitted without losing the traversal capabilities if the previously visited edge is known. However, if loops or multi-edges are allowed all four edge pointers must remain. Otherwise the traversal around a vertex or around a facet is no longer uniquely defined [35]. The winged-edge data structure where the wings PCCW and NCCW are omitted has been called *Doubly Connected Edge List* (DCEL) by [23], though this name is now more commonly used for the halfedge data structure [5]. [3]

The two symmetric parts in the winged-edge correspond to the two possible orientations of the edge. The inefficient case distinction in the traversal computation results from the fact that a pointer to an edge does not encode the orientation it is currently used with. One extension of the winged-edge maintains an additional bit with each edge-pointer to code the orientation, but this can lead to cumbersome storage layouts and function interfaces.

[3] In order to avoid confusion we will not use the name DCEL since it turned out to be ambiguous. In fact, the name is misleading when denoting halfedges and the possible variants of single linking.

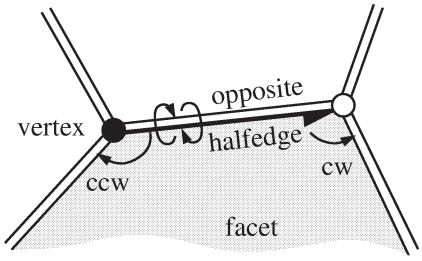

Fig. 4. FE-structure.

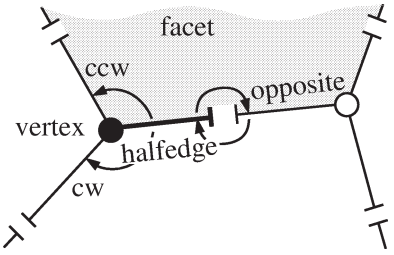

Fig. 5. VE-structure.

3.2. Halfedge data structure

The orientation problem can be solved for the winged-edge data structure by splitting the edge into two symmetric records, called *halfedges*, and adding mutual links to each other [35]. There are two ways of splitting the edge: either the edge is split along the facets, such that the oriented halfedges belong to the two facets incident to this edge, see Fig. 4, or it is split into two halfedges belonging to the two vertices incident to this edge, see Fig. 5. These two variants are actually dual to each other considering the usual notion of duality for graphs, where vertex and facet are dual to each other and the dual of an edge is an edge with the two incident facets as its endpoints.

In both splitting variants a halfedge contains a pointer to an incident vertex, an incident facet, and the opposite halfedge. It is a matter of convention whether the source or target vertex is the one chosen to be stored in a halfedge or whether the facet to the left or the right is stored. In [35] the source vertex and the facet to the right were chosen. The FE-structure in Fig. 4 additionally stores a pointer to the next clockwise halfedge and optionally a pointer to the previous counterclockwise halfedge around the facet. It is therefore biased towards traversals around the incident facet. The dual VE-structure is depicted in Fig. 5. Its next and optional previous pointer refer to halfedges counterclockwise and clockwise around the incident vertex. The traversal operation that is not directly accessible with a single pointer access is available through the opposite halfedge. For example, the next halfedge around the incident source vertex for the FE-structure is `opposite()->next()`. The different conventions are not independent. If the convention defines the halfedge order around a facet to be clockwise, the halfedge order around the vertex will be counterclockwise, and vice versa.

The halfedge data structure is able to model orientable 2-manifolds. It is sufficient for modeling topology even in the presence of loops and multi-edges, which can occur in curved-surface environments [35]. High-level operations maintaining integrity are again Euler operators. The solid modeling system GWB [20] is based on a halfedge-data structure, though it uses an additional edge record between two opposite halfedges, which makes this access less efficient. The Minimal Rendering Tool MRT [2] uses a halfedge data structure for polygonal surfaces.

3.3. Quad-edge data structure

Similar to the idea for halfedges, each edge is split into four quad-edges to obtain the quad-edge data structure [11]. It provides a fully symmetric view on the primal and the dual graph, as can be seen in Fig. 6. Instead of using opposite pointers, a two bit counter r is used to address a slot in an edge record e

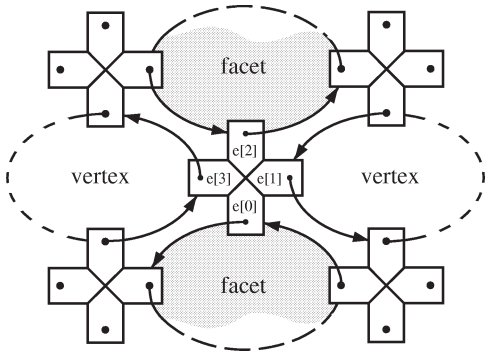

Fig. 6. Quad-edge data structure.

of four quad-edges. This data structure is able to model non-orientable 2-manifolds with an additional bit f denoting the flipped status per edge.

A quad-edge data structure is formally defined as an edge algebra with three operations: `Onext()`, `Rot()` and `Flip()`. A quad-edge is represented as a triple (e, r, f) with e a record of four quad-edges $e[0]$ to $e[3]$ incident to the current edge, $r \in \{0, 1, 2, 3\}$ and $f \in \{0, 1\}$. The operations are implemented with a calculus modulus 4 for r and modulus 2 for f as follows:

$$\texttt{Rot}(e, r, f) = (e, r + 1 + 2f, f),$$
$$\texttt{Flip}(e, r, f) = (e, r, f + 1),$$
$$\texttt{Onext}(e, r, f) = \begin{cases} e[r] & \text{for } f = 0, \\ \texttt{Flip}(\texttt{Rot}(e[r + 1])) & \text{for } f = 1. \end{cases}$$

Four different orientations of an edge are considered: two orientations from vertex to vertex and two orientations for the dual edge from facet to facet. The `Rot` operator rotates the edge by 90 degrees, oscillating between the primal and the dual view of the structure. For non-orientable 2-manifolds an edge can additionally be seen from above or below the surface, which is encoded in the f bit. The `Flip` operation changes the view from above to below or vice versa. The `Onext` operation gives the next quad-edge in counterclockwise order around the source vertex (origin), or the next quad-edge in clockwise order if f is equal to one. The values for `Onext` are simply stored in the record for each edge (i.e., four pointers and four times three bits for r and f). The operations simplify considerably for orientable 2-manifolds. They can be further simplified if the dual graph is not needed. The result would be the winged-edge data structure enriched with a bit to encode orientation.

The single high-level operation that modifies a quad-edge data structure is the `Splice()` operation. It is its own dual. The usual Euler operators can be implemented in terms of `Splice()`. The quad-edge data structure provides a unified view for the primal and dual graph. This implies that vertices and facets cannot be distinguished with strong type checking at compile time. The definition used for duality implies, furthermore, that the facets must have a single connected boundary. Holes in facets are not allowed. If strong type checking is desired, the `Splice()` operation is needed twice, once for the primal view and once for the dual view. `Splice()` can also be provided for the halfedge data structure.

Table 1
Comparison of the edge-based data structures

	Winged-edge	Half-edge	Quad-edge
Modeling space	orientable 2-manifold		2-manifold
Operations	Euler operator		`Splice()`
Duality	adaptor at compile time		`Rot()` at runtime
Holes in facets	yes	yes	no
Basic traversal	case distinction	direct access	mod operation
Min size per edge	4 ptr	4 ptr	2 ptr + 2 bits
Max size per edge	8 ptr	10 ptr	8 ptr + 12 bits

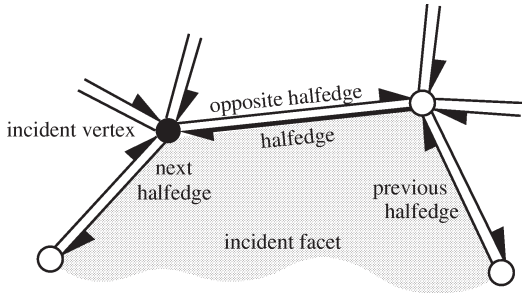

Fig. 7. Halfedge data structure.

3.4. Comparison of edge-based representations

The main differences of these edge representations are captured in Table 1. The differences in the basic traversal capabilities are not negligible, especially when considering modern microprocessor architectures where conditional branching can be slower than computing by an order of magnitude. The storage size requirements are quite similar. Our design will focus on the flexibility of trading runtime against storage costs. We are interested in the minimal and maximal configurations for the halfedge data structure and the space efficiency of the quad-edge data structure. Another issue is the preference for strong type checking at compile time. Polyhedral surfaces have different information stored in the vertices and facets, namely points and plane equations. These can be treated as duals of each other, but in a strongly-typed geometry kernel (like the one CGAL provides) they are different types and might even be represented differently. Additional information, like color, will finally destroy the typeless symmetry of the duality assumed by the quad-edges. Holes in facets can be modeled with the winged-edge and the halfedge data structure, but only with the implication that the vertex-edge graph can be disconnected for connected surfaces. Furthermore, the dual facet-edge graph is also a 2-manifold if there are no holes in facets. We neglect non-orientable 2-manifolds, for example the three-dimensional surface of a solid physical object is always orientable. With respect to a library design, we would refer to a more general non-manifold data structure to model non-orientable 2-manifolds.

We have chosen a halfedge data structure comparable to the FE-structure. The conventions used are depicted in Fig. 7. We have `next()`, `opposite()` and `prev()` pointers for the halfedges. The incident vertex is the target vertex of the oriented halfedge. The incident facet is to the left of the halfedge which implies a counterclockwise ordering of the halfedges around the facet and a clockwise ordering around the vertex when seen from the outside. This complies with the right-hand rule for out-facing normals of plane equations for facets.

4. Generic and object-oriented programming

The major design issues considered for polyhedral surfaces are flexibility, time efficiency, space efficiency and ease-of-use. To realize generic and flexible designs, basically two paradigms are available: *object-oriented programming* and *generic programming*. Both are available in C++: object-oriented programming, using inheritance from base classes with virtual member functions, and generic programming, using class templates and function templates. Object-oriented programming is also known in Smalltalk, Eiffel and Java, generic programming in Ada and Generic Java, to name a few examples. We stick with the notion used in C++, which is the choice made for CGAL.

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

The advantages are the explicit definition of the interface and the runtime flexibility. There are four main disadvantages: this paradigm cannot provide strong type checking at compile time whenever dynamic casts are used, which is necessary in C++ to achieve the flexibility; this paradigm enforces tight coupling through the inheritance relationship [18], it requires additional memory for each object (in C++ the so-called *virtual function table pointer*) and it adds for each call to a virtual member function an indirection through the virtual function table [19]. 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.[4] Modern microprocessor architectures 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 are the access of point coordinates and arithmetic for low-dimensional geometric objects, see for example the case study [30], and traversals of combinatorial structures. Vertices, edges and facets for polyhedra are anticipated to be small objects with simple member functions. The space and runtime overhead introduced through virtual member functions will not be negligible.

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

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

```
template <class T>  class list {
  // ... placeholder T represents the item type symbolically.
};
template <class T>  void swap(T& a, T& b) {
    T tmp = a; a = b; b = tmp;
}
```

The example of the swap function illustrates that a template usually requires certain properties of the template arguments, here that variables of type `T` are assignable. An actual type used in the template instantiation must comply with these assumptions in order to obtain a correct template instantiation. We can distinguish between *syntactical requirements*, the assignment operator is needed in our example, and *semantical requirements*, the operator should actually copy the value. If the syntactical requirements are not fulfilled, compilation simply fails. Semantical requirements are not checkable at compile time. However, it might be useful to connect a specific semantical requirement to an artificial, newly introduced syntactical requirement, e.g., a tag similar to iterator tags in [33]. This technique allows decisions at compile time based on the actual type of these tags.

The set of requirements that is needed to obtain a correct instantiation of a member function of a class template is usually only a fraction of all requirements for the template arguments of this class template. If only a subset of the member functions is used in an instantiation of a class template, it would be sufficient for the actual types to fulfill only the requirements needed for this subset of member functions. This is possible in C++ since as long as a C++ compiler is not explicitly forced, the compiler is not allowed to instantiate member functions that are not used, and therefore possible compilation errors due to missing functionality of the actual types cannot occur [15]. This enables us to design class templates with optional functionality, which can be used if and only if the actual types used in the template instantiation fulfill the additional requirements. An example would be the optional previous pointer in the halfedge of the halfedge data structure. The user provides an actual halfedge type as a template argument for the halfedge data structure, which is a class template. If the halfedge provides a previous pointer, a bidirectional access around a facet or around a vertex is available in the data structure, but if the previous pointer is not provided, the access will only be unidirectional.

A good example illustrating generic programming is the Standard Template Library [15,24,31]. Generality and flexibility is achieved with the carefully chosen set of *concepts*, where a concept is a well defined set of requirements. In our `swap`-function example, the appropriate concept is named ‘assignable’ and includes the requirement of an assignment operator [31]. If an actual type fulfills the

requirements of a concept, it is a *model* for this concept. In our example, `int` is a model of the concept 'assignable'.

Algorithmic abstraction is a key goal in generic programming [25,26]. One aspect is to reduce the interface to the data types used in the algorithm to a set of simple and general concepts. One of them is the *iterator* concept, which is an abstraction of pointers. Iterators serve two purposes: they refer to an item and they traverse over the sequence of items that are stored in a data structure, also known as *container class* in STL. Five different categories are defined for iterators: input, output, forward, bidirectional and random-access iterators, according to the different possibilities of accessing items in a container class. The usual C-pointer referring to a C-array is a model for a random-access iterator. *Generic algorithms* are not written for a particular container class in STL, they use iterators instead. For example, sequences of items are specified by two iterators forming a range `[first,beyond)`. This notion of a half-open interval denotes the sequence of all iterators obtained by starting with `first` and advancing `first` until `beyond` is reached, but does not include `beyond`. A container class is supposed to provide a local type, which is a model of an iterator, and two member functions: `begin()` returns the start iterator of the sequence and `end()` returns the iterator referring to the 'past-the-end'-position of the sequence. For example, a generic `contains` function can be written to work for any model of an input iterator. It returns `true` iff the `value` is contained in the values of the range `[first,beyond)`.

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

The advantages of the generic programming paradigm are strong type checking at compile time during the template instantiation, no need for extra storage or additional indirections during function calls, and full support of inline member functions and code optimization at compile time [34]. One specific disadvantage of generic programming in C++ is the lack of a notation in C++ to declare the syntactical requirements for a template argument, the equivalent to the virtual base class in the object-oriented programming paradigm. The syntactical requirements are scattered throughout the implementation of the template. The concise collection of the requirements is left for the program documentation. In general, the flexibility is resolved at compile time, which gives us the advantages mentioned above, but it can be seen as a disadvantage if runtime flexibility is needed. However, the generic data structures and algorithms can be parameterized with the base class used in the object-oriented programming to achieve the runtime flexibility.

We applied mainly the generic programming paradigm to achieve flexibility and efficiency in CGAL. The compliance with STL is important for us to promote the re-use of existing generic algorithms and container classes, and – more important – to unify the look-and-feel of the design of CGAL with the C++ standard. CGAL is therefore easy to learn and easy to use for those who are familiar with STL. In a few places we made also use of the object-oriented programming paradigm, for example, the protected access to the internal representation of a polyhedral surface, see Section 10.

5. Design goals for polyhedral surfaces

A polyhedron can be viewed as a container class managing the vertices, halfedges and facets of a polyhedral surface and maintaining their combinatorial structure. The following design issues were taken into account for our design:

(1) The actual storage organization of the vertices, edges and facets influences the space and runtime efficiency. A doubly-connected list representation allows random insertion and removal while providing bidirectional iterators to enumerate all items. A more space efficient storage uses an STL vector, which allows only the efficient removal of items at the end of the vector, but provides random-access iterators. Other variants are possible, such as managing chunks of memory or simple allocation on the heap without any iterators.

(2) The necessary incidence relations might depend on the application. In order to keep the halfedge data structure connected, we need at least a `next()` and an `opposite()` pointer per halfedge. The other incidences are optional. For example, the `prev()` pointer can be simulated with a search around the vertex or around the facet in the opposite direction. For triangulations this is a simple expression, i.e., `prev()` $\equiv$ `next()->next()`. Assuming a constant degree at vertices or a constant number of edges for facets, it is still a constant time operation. If no information needs to be attached to vertices or facets, no storage should be allocated for them, even the referencing pointers in the halfedge should be omitted. In the limit the data structure reduces to an undirected graph structure.

(3) It should be easy for a user to add additional information to the different items, e.g., color to facets. Geometry will be attached using the same technique. Redefining one item should not hinder the re-use of the other items, for example, adding color to facets should not imply that a new vertex type must be declared.

(4) The data structure should provide an easy-to-use high-level interface. This interface should protect the internal combinatorial integrity of the data structure as given in Definition 3. Advanced algorithms concerned with efficiency, e.g., a file scanner, should be allowed to access the internal structure in a controlled manner.

(5) The management of connected components and containment relations, e.g., inner and outer boundaries for facets and nesting relationships of shells, is seen as an independent functionality that can be added separately, for example, in its own layer and with its own flexibility.

(6) The edge-based data structures discussed in Section 3 have a natural notion of the edges around a vertex or the edges around a facet. It would be costly to provide iterators for these kind of circular sequences since the notion of ranges and the 'past-the-end' value do not extend naturally. We propose a concept similar to iterators – which we call *circulator* – for this kind of structure.

(7) STL containers base their interfaces on iterators. For polyhedral surfaces the order in which the items are stored in the polyhedron is not always well-defined from the perspective of the user, e.g., after Euler operations. Here we fall back on the concept of *handles*, which is the item-denoting part of iterators, and ignore the traversal capabilities. In particular, any model of an iterator or circulator is a model of a handle. A handle is also known as trivial iterator.

We concentrate on the combinatorial aspects of the polyhedral surface. Additional issues appear when considering geometry, for example, flexibility in the point type and the geometric predicates. One technique explored for this is a variation of the traits classes known from the C++ standard library [27], see [3,7,9] for their use in CGAL.

6. Previous work

The design presented here is a major revision of the design described in [16]. The previous design is available in CGAL since Release 1.0 [3]. In the original design, the user provides the space for the incidence information in the vertices, halfedges and facets in terms of `void*` pointers. The halfedge data structure uses type-casts to provide correct type-safe pointers and the polyhedron converts the pointers to iterators and handles. The main drawback besides the tricky, nested implementation is that users cannot extend their vertices, halfedges and facets conveniently with further incidences. These new incidences are not captured in this process of type-casting. In the revised design presented here, all layers of the design share the same type informations. Vertices, halfedges and facets can use the correct handle types to provide the space for the various incidences although they are still decoupled from each other, e.g., the vertex type does not hard code the actual type of its related halfedge type. Internally, this new design saves us an implementation layer that was needed for the type-casting.

The Library of Efficient Datatypes and Algorithms (LEDA)[5] [21,22] provides a rich body of algorithms and data types. LEDA includes, for example, number types for exact arithmetic, dictionaries, priority queues, graph algorithms, and two-dimensional geometry. Its design is homogeneous and easy-to-use. LEDA has its own notion of iterators and it is generic and flexible within its own framework, which is incompatible to the approach taken by STL. But recent extensions make LEDA compliant with STL, in fact it replaces STL. LEDA contains no data structure specifically tailored for three-dimensional polyhedra, but it provides a general data structure for graphs and one for planar maps derived from the graph data structure. Additional information can be attached with node arrays and edge arrays or by using a parameterized graph. A parameterized graph is a class template with two template arguments: one for the auxiliary information in the vertices and one for the auxiliary information in the edges. The disadvantage is that if only one of the two types is needed a dummy type must be given to the other template argument, which wastes memory. Node and edge arrays are associative arrays based on hashing, which allow the easy addition of information even for temporary purposes. The disadvantages are the additional cost for the lookup operation and the additional memory consumption. A more subtle disadvantage is revealed when considering, for example, an iterator over graph nodes: a simple pointer to the node is in general not sufficient for the state of the iterator, since the iterator would not be able to give access to the associated attributes of the node. An additional reference to the associative arrays must be stored in the iterator (or alternatively in the node). The current memory requirements are equivalent to 13 pointers for a graph node and 11 pointers for a directed graph edge. There is no flexibility for obtaining smaller graph structures. LEDA does not reach the flexibility in tuning the runtime and space efficiency as achieved with the approach presented in this paper.

The Minimal Rendering Tool MRT uses a halfedge data structure to represent polyhedral surfaces [2]. It is implemented as a C++ class hierarchy and provides Euler operations to maintain combinatorial integrity. The internal representation is accessible at construction time and protected thereafter. No other access is granted. It separates geometry and topology except for vertices where a point is incorporated just at the combinatorial level for efficiency reasons. Flexibility is achieved with virtual member functions for geometric properties. No flexibility is available at the topological level. Facets are responsible of storing the ring of halfedges of their boundary. Summarizing, this approach leads to larger nodes for vertices, halfedges and facets and slower functions for geometric properties than the solution we propose.

[5] `http://www.mpi-sb.mpg.de/LEDA/leda.html`

7. Design overview

Fig. 8 illustrates the separation of topology and geometry in the design. Vertices, halfedges and facets store both. The container class `Halfedge_data_structure` manages these three items and their topological relations. The `Topological_map` adds to the halfedge data structure the management for holes in facets, which enumerates inner and outer boundaries for facets. It is usually classified as a face-based representation. The `Polyhedron` adds geometric operations to the `Halfedge_data_structure`. It is based on Definition 3 for polyhedral surfaces and guarantees a consistent representation (besides the costly self-intersection test and the polygon planarity test). The halfedge data structure is not restricted to these definitions, since it is also useful in implementing other data structures, but it will be used by the polyhedron to represent only surfaces following this restricted definition, for example, that an edge has always two distinct endpoints. The `Planar_map` is based on the topological map, since it maintains holes in facets.

The `Halfedge_data_structure` and the items `Vertex`, `Halfedge` and `Facet` are concepts. Currently three different models are provided for the `Halfedge_data_structure`: one is based on the STL vector, the other two are based on list representations to manage the items internally. Various models are provided for the items and they can be easily extended by the user, for example, with additional attributes.

Fig. 9 illustrates the three layers of the polyhedral surface design: `Items`, `Halfedge_data_structure` and `Polyhedron`. The items provide the space for the information that is actually stored, i.e., with member variables and access member functions in `Vertex`, `Halfedge` and `Facet`, respectively. Halfedges are required to provide a reference to the next halfedge and to the opposite halfedge. Optionally they may provide a reference to the previous halfedge, to the incident vertex and to the incident facet. Vertices and facets may be empty. Optionally they may provide a reference to the incident halfedge. The options mentioned are supported in the halfedge data structure and the polyhedron, for example, Euler operations update the optional references if they are present. Furthermore, the item classes can be extended with arbitrary attributes and member functions, which will be promoted by inheritance to the actual classes used for the polyhedral surfaces.

Implementations for vertices, halfedges and facets are provided that fulfill the mandatory part of the requirements. They can be used as base classes for extensions by the user. Richer implementations are

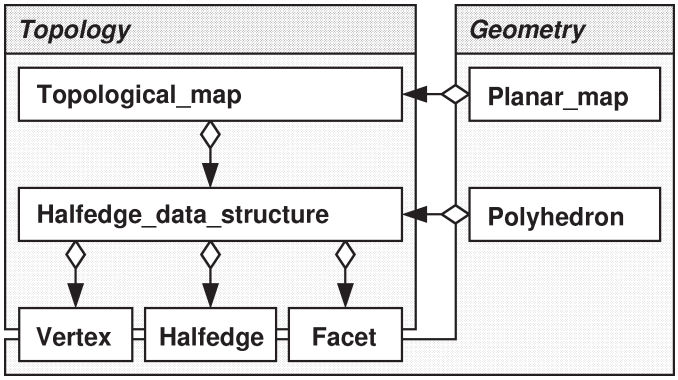

Fig. 8. Overview of the design with the separation of topology and geometry.

also provided to serve as defaults; for polyhedra they provide all optional incidences, a three-dimensional point in the vertex type and a plane equation in the facet type.

Vertices, halfedges and facets are passed as local types of the `Items` class to the halfedge data structure and polyhedron. This is an implementation detail explained in more depth in Section 9.

The `Halfedge_data_structure` is responsible of the storage organization of the items. Currently, implementations using internally a bidirectional list or an STL vector are provided. The `Halfedge_data_structure` defines the handles and iterators belonging to the items. These types are promoted to the declaration of the items themselves and are used there to provide the references to the incident items. This promotion of types is done with a template parameter of the item types, see Section 9. The halfedge data structure provides member functions to insert and delete items, to traverse all items, and it gives access to the items in order to manipulate the items.

There are already three different models for the `Halfedge_data_structure` available. Therefore we have kept their interface small. Functionality common to all these models is separated into a helper class `Halfedge_data_structure_decorator`, which is not shown in Fig. 9, but would be placed at the side of the `Halfedge_data_structure` since it broadens that interface but does not hide it. This helper class contains operations that are useful to implement the operations in the next layer, for example, the polyhedron. It adds, for example, the Euler operations and partial operations from which further Euler operations can be built, such as inserting an edge into the ring of edges at a vertex. Furthermore, the helper class contains adaptive functionality. For example, if the `prev()` member function is not provided for halfedges, the `find_prev()` member function of the helper class searches in the positive direction along the facet for the previous halfedge. But if the `prev()` member function is provided, the `find_prev()` member function simply calls it. This distinction can be resolved at compile time with a technique called *compile-time tags*, similar to iterator tags in [33], see Section 9.

The `Polyhedron` provides an easy-to-use interface of high-level functions and hides the flexibility provided underneath. The interface is designed to protect the integrity of the internal representation, handles are no longer mutable. The polyhedron adds the convenient and efficient circulators, see

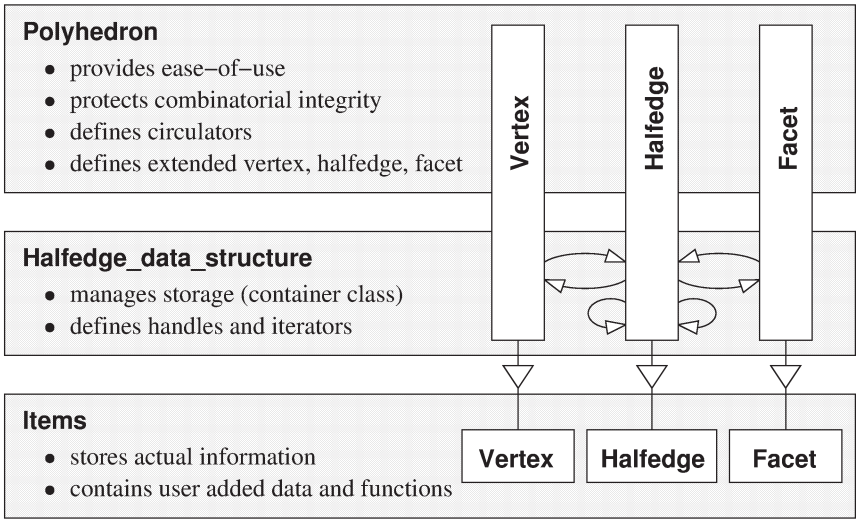

Fig. 9. Responsibilities of the different layers in the design.

Section 11, for accessing the circular sequence of edges around a vertex or around a facet. To achieve this, the `Polyhedron` derives new vertices, halfedges and facets from those provided in `Items`. These new items are those actually used in the `Halfedge_data_structure`, which gives us the coherent type structure in this design, especially if compared to our previous design [16]. The `Polyhedron` adds also the geometric interpretation. Therefore, the vertices used for polyhedron are required to store a geometric point.

8. Program examples

Default template arguments allow us to hide the flexibility provided for the halfedge data structure. Therefore, in the first example program the polyhedron only needs to be instantiated with the geometric kernel. In CGAL we use a traits class between the actual geometric kernel and the data structures and algorithms [3,7,9], which allows us to provide these adaptors easily for other kernels as well. A declaration of the default polyhedron based on the Cartesian kernel using doubles looks as follows:

```
typedef CGAL_Cartesian<double>                    Kernel;
typedef CGAL_Polyhedron_default_traits_3<Kernel> Traits;
typedef CGAL_Polyhedron_3<Traits>                 Polyhedron;
```

The following function applies an affine transformation to all points in the polyhedron. Note the use of `transform`, which is a generic algorithm from STL. Our design encourages such re-use instead of providing an own function for affine transformations: the polyhedron provides the iterator for accessing all points and the affine transformation of CGAL is a unary function object suitable for the `transform` function.

```
void
apply(Polyhedron& P, const CGAL_Aff_transformation_3<Kernel>& A) {
    transform(P.points_begin(), P.points_end(), P.points_begin(), A);
}
```

The default polyhedron provides already a plane equation for facets. In the following example we replace the default facet type with our own facet type providing only a normal vector instead of a full plane equation. For that we define our own facet type and replace the facet in the default items type we have used so far. We make use of a facet base class already available in CGAL. The template argument `Refs` will be explained in Section 9. It is used to make the handle and iterator types of the halfedge data structure available in the items. The `Traits` argument provides a local type that we can use for the normal vector type.

An items class contains three template member classes representing a vertex type, a halfedge type and a facet type, respectively. Their names and template arguments are fixed, since they are used in the halfedge data structure to create objects of these types. We use inheritance from the default items class of the polyhedron to re-use the old vertex and halfedge types. Only the facet is replaced. The related template member class must provide a local type `Facet`, which is our redefined facet type. The new items class can be used as a second template argument to the polyhedron class, where it replaces the default items class. A third template argument can be used to replace the default halfedge data structure, though this is not illustrated here. The `Traits` argument is the same as above.

```
template <class Refs, class Traits>
struct My_facet : public CGAL_HalfedgeDS_facet_base<Refs> {
    typename Traits::Vector_3 normal;
};
struct My_items : public CGAL_Polyhedron_items_3 {
    template <class Refs, class Traits>
    struct Facet_wrapper {
        typedef My_facet<Refs, Traits> Facet;
    };
};
typedef CGAL_Polyhedron_3<Traits, My_items> Polyhedron;
```

Our next example computes the normal vectors of the facets using the modified facet type from our previous example. The generic `for_each` algorithm of STL can be used to apply a function object to each facet of the polyhedron. The function object assumes convex facets to compute the out-facing normal vector and it ignores numerical stability issues. We obtain three consecutive vertices on the boundary of the facet and compute the normal vector with a cross product. The function object uses a template member function, which gives us a generic implementation that can be used with any facet type with a compliant `normal` member variable and that keeps a reference to an incident halfedge. The facet knows the type for the vertex handle.

```
struct Normal_vector {
    template <class Facet>
    void operator()( Facet& f) {
        typedef typename Facet::Vertex_handle Vertex_handle;
        Vertex_handle p = f.halfedge()->vertex();
        Vertex_handle q = f.halfedge()->next()->vertex();
        Vertex_handle r = f.halfedge()->next()->next()->vertex();
        f.normal = CGAL_cross_product( q->point() - p->point(),
                                       r->point() - q->point());
    }
};
void compute_normals( Polyhedron& P) {
    for_each( P.facets_begin(), P.facets_end(), Normal_vector());
}
```

9. Implementation of the design using templates in C++

We present the necessary details of an implementation based on templates in C++ to prove the feasibility of our design. We begin with a simplified version presenting the technique we use to decouple the item types and the halfedge data structure. Thereafter we present the realization of the polyhedron class. We omit details such as const-correctness and the CGAL prefix. Note that only the developer but not the user of the data structure must know the following details of the design.

The key is that a certain item type, for example a vertex, does not know the actual types of the related items, for example the halfedge type. Instead, it knows a formal placeholder for this type. We put all formal placeholders together in a single template argument `Refs` as local types. We indicate only the use of `Halfedge_handle` in the vertex type, which provides the reference to the incident halfedge.

```
template <class Refs>
struct Vertex {
    typedef typename Refs::Halfedge_handle Halfedge_handle;
    Halfedge_handle halfedge() { return h; }
    void set_halfedge( Halfedge_handle g) { h = g; }
private:
    Halfedge_handle h;
};
```

The other item types are implemented similarly. The halfedge data structure is parameterized with the item types. Since the item types are already class templates, we need templates as template arguments for the halfedge data structure. The order of type declarations in the halfedge data structure is the vertex and halfedge type, the container classes for these item types, and the handle types known from the container classes. The type dependencies contain a cycle; the item types need a template argument that tells them the handle types. The halfedge data structure knows the handle types and can be used as the actual type for the `Refs` argument of the item types, even though the handles have not been declared at this point. However, it is the difference between declaration and definition that allows us to use the declared type of the halfedge data structure for the item type instantiation, even though it is not fully defined at this point. With respect to this instantiation our approach is similar to the template pattern described in [4], although we make no use of inheritance. We omit the facet type in this example.

```
template < template <class> Vertex,  template <class> Halfedge>
struct HalfedgeDS {
    typedef HalfedgeDS< Vertex, Halfedge> Self;
    typedef Vertex<Self>                  V;
    typedef Halfedge<Self>                H;
    typedef list<V>                       Vlist
    typedef list<H>                       Hlist
    typedef typename Vlist::iterator      Vertex_handle;
    typedef typename Hlist::iterator      Halfedge_handle;
    // ...
};
```

It remains to show the intermediate items class, the polyhedron class with its protected access to the items, the connection to the geometric kernel with the traits class, and the *compile time tags* to adapt functions to the flexibility provided in the items. An item type usually has two template arguments, `Refs` as explained above and a `Traits` argument. The traits contains the geometric types and the basic operations available for these types [7]. The optional functionality of the item types is indicated with tags. In the example of a vertex type from above we add the local type `Supports_vertex_halfedge`. It can be either of the two predefined classes, `Tag_true` or `Tag_false`. This tag indicates whether the reference to the incident halfedge is provided or not.

```
template <class Refs, class Traits>
struct Vertex {
    typedef Tag_true Supports_vertex_halfedge;
    // ...
};
```

Other functionality can make use of this tag to adapt at compile time. For example, the Euler operations are implemented using this *compile-time tag*. They update this reference automatically if it is provided. The following example uses function overloading in C++ to distinguish between two different implementations of a function `foo` depending on the tag. The single-argument function is called with a vertex as argument. The function call is forwarded to the corresponding two-parameter function that matches the actual type of the compile-time tag.

```
void foo( Vertex v) {
    foo( v, Vertex::Supports_vertex_halfedge());
}
void foo( Vertex v, Tag_true)  {
    // ... implementation making use of the v.halfedge() method.
}
void foo( Vertex v, Tag_false) {
    // ... implementation not making use of the v.halfedge() method.
}
```

The intermediate items class puts together the definition of the three item types: vertex, halfedge and facet. It uses member template classes, but is itself not a template class. It can be passed around without instantiating the item types. Furthermore, the halfedge data structure can be written without templates as template parameters. The names and template arguments within the items class are fixed: The member class templates are called wrapper and have two template arguments, `Refs` and `Traits`. These wrappers must provide a local type named after the corresponding item, `Vertex` for example, that refers to the actual class used. Besides technical reasons, it is convenient to pass the item types in a single parameter to the halfedge data structure. It is easier to use with items classes that are already defined in the library. On the other hand, a new items class needs to be derived if only a single item type is exchanged, see the example in the previous section. Another advantage is a possible separation of two kind of template parameters: user-specified parameters and those specified in the halfedge data structure, namely `Refs` and `Traits`. Further user-specified parameters cannot be passed as additional template arguments through the halfedge data structure, but the user can make the items class a template class for itself and can provide the actual types when instantiating the polyhedron. The items class for our example without additional user parameters looks as follows:

```
struct HalfedgeDS_items {
    template <class Refs, class Traits>
    struct Vertex_wrapper {
        typedef Vertex<Refs, Traits> Vertex;
    };
    // ... similar for halfedge and facet
};
```

The polyhedron derives new item types from the given item types to enhance functionality, e.g., circulators, and to protect the combinatorial structure. Member functions that allow changing the incidences of items are made private, while the remaining functionality and especially the user added functionality remains available to the user because of the public inheritance. Another solution would be to repeat explicitly the functionality that should remain public, but user added functionality could not be captured this way and would be lost. The solution chosen may sound weak with respect to protection, but since the user provides the bases, the user can always work around any protection mechanism. However, our solution prohibits accidental misuse, the main purpose of protection.

The new item types are again collected in an items class, which is used in the halfedge data structure of the polyhedron. In consequence, the halfedge data structure uses the correct items defined by the polyhedron. The original items provided by the user are therefore parameterized with the handles to the derived item types, which are the correct types to achieve the coherent type system. But the consistent use of the derived item types prohibits also the to-be-allowed change of incidences by the polyhedron and the halfedge data structure. Therefore, we provide the type of the base class in the derived item types. The halfedge data structure and the polyhedron can change incidences by calling the member functions of the base class, similar to the implementation of the `set_halfedge` member function in the following example.

```
template <class Vertex_base>
struct Polyhedron_vertex : public Vertex_base {
    typedef Vertex_base Base;
private:
    void set_halfedge( typename Base::Halfedge_handle g) {
        Base::set_halfedge(g);
    }
};
template < class Items>
struct Polyhedron_items {
    template <class Refs, class Traits>
    struct Vertex_wrapper {
        typedef typename Items::Vertex_wrapper<Refs, Traits> Wrapper;
        typedef typename Wrapper::Vertex Vertex_base;
        typedef Polyhedron_vertex<Vertex_base> Vertex;
    };
    // ... similar for halfedge and facet
};
```

The polyhedron is a class template with three template parameters: `Traits`, `Items` and `HDS`, two of them have default arguments. It declares the derived item types and instantiates the halfedge data structure `HDS` to use it as internal representation. The `HDS` parameter is again a template as template parameter. If the halfedge data structure had used a template as template parameter for itself, we would have had a third level of templates, which is not allowed in C++. The items class avoids this, but other workarounds are possible as well.

```
template < class Traits,
           class Items = Items_default,
           template <class, class> class HDS = HalfedgeDS_default>
struct Polyhedron {
    typedef Polyhedron_items<Items> Derived_items;
    HDS <Traits, Derived_items> hds;
    // ...
};
```

The halfedge data structure from above remains basically unchanged. Instead of the separate item types we pass a traits parameter and an items class parameter. The item types are extracted from the wrappers in the items class.

Given a set of predefined implementations for these classes in a library, it is easy to combine and extend them. The flexibility we have demanded for our design in Section 5 is realized. The default list representation in the halfedge data structure can be replaced by other representations. Only those incidences we actually encode in the bases are stored, and the tags are used to implement adaptive functionality corresponding to the incidences provided, for example, vertices are not even allocated in the halfedge data structure if they are not referenced from the halfedges. Adding additional information can be easily done by deriving own item types and replacing their definitions in the items class. All this has been achieved without runtime nor storage overhead, nor any compromises for the ease-of-use at the top-level.

10. Granting access to the internal representation

Algorithms on polyhedra may have intermediate states that are invalid representations. To be efficient, they need access to the internal representation. We grant a protected access for all subclasses of `Modifier_base`. Our design was motivated by the strategy pattern [8], though it has a different intent.

Fig. 10 illustrates the class design for the example of a file scanner creating a polyhedron from a file. The approach is similar to a callback function, only encapsulated as a function object. The `Polyhedron` accepts a modifier object as an argument of its `delegate()` member function and calls the virtual `operator()` member function of this modifier object. Thereby, it passes the internal halfedge data structure as argument to the modifier object. The `Scanner` class from our example is derived from the `Modifier_base` and implements the `operator()` member function, where it can access the internal representation.

The achievement here is that the `delegate()` member function of the `Polyhedron` can verify the validity of the internal representation after the `operator()` member function returns from execution. The `Scanner` class is in charge of returning only with a valid representation, even in the case of a failure. This approach is also known from database systems as transactions; either the modifier succeeds or the modifier fails and is supposed to clean up. The special task the `Scanner` accomplishes (only creation of new items) enables us to implement the transaction scheme efficiently. A simple rollback function removes all items created so far in the case of a failure. In general, the rollback would be more costly.

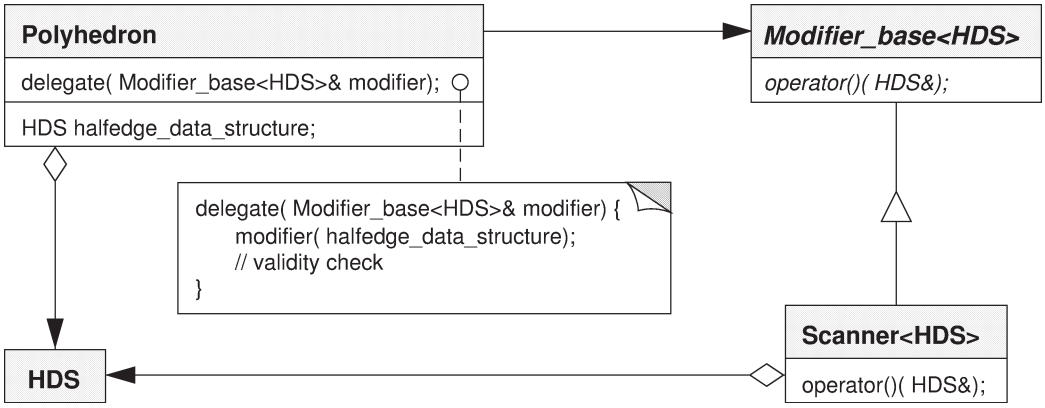

Fig. 10. Class diagram illustrating the safe access to the internal representation of a polyhedron.

11. Circulators

The concept of iterators in STL is tailored for linear sequences. In contrast, circular sequences occur naturally in many combinatorial and geometric structures. Examples are polyhedral surfaces and planar maps, where the edges emanating from a vertex or the edges around a facet form a circular sequence.

Since circular sequences cannot provide efficient iterators, we have introduced the new concept of *circulators* in CGAL. They share most of the requirements with iterators, while the main difference is the lack of a past-the-end position in the sequence. Appropriate adaptors are provided between iterators and circulators to integrate circulators smoothly into the framework of STL. We give a short introduction to circulators and discuss advantages and disadvantages thereafter. We rewrite the example of the generic `contains` function from above using circulators. As usual for circular structures, a `do-while` loop is preferable, such that for the specific input, `c == d`, all elements in the sequence are reached.

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

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 `n` steps. Two circulators can be compared for equality.

Circulators have a different notion of reachability and ranges than iterators. A circulator `d` is called *reachable* from a circulator `c` if `c` can be made equal to `d` with finitely many applications of the

operator ++. Due to the circularity of the sequence this is always true if both circulators refer to items of the same sequence. In particular, c is always reachable from c. Given two circulators c and d, the range [c,d) denotes all circulators obtained by starting with c and advancing c until d is reached, but does not include d, for d ≠ c. So far it is the same range definition as for iterators. The difference lies in the use of [c,c) to denote all items in the circular sequence, whereas for an iterator i the range [i,i) denotes the empty range. As long as c != d the range [c,d) behaves like an iterator range and could be used in STL algorithms. For circulators however, an additional test c == NULL is required that returns true if and only if the circular sequence is empty.

Supporting both, iterators and circulators, within the same generic algorithm is just as simple as supporting iterators only. This and the requirements for circulators are described in the CGAL Reference Manual [3].

Besides the conceptual clearness, the main reason for inventing a new concept with a similar intent as iterators is efficiency. An iterator is supposed to be a light-weight object – merely a pointer and a single indirection to advance the iterator. Although iterators could be written for circular sequences, we do not know of an efficient solution. The missing past-the-end situation in circular sequences can be solved with an arbitrary sentinel in the cyclic order, but this would destroy the natural symmetry in the structure (which is in itself a bad idea) and additional bookkeeping in the items and checking in the iterator advance method reduces efficiency. Another solution may use more bookkeeping in the iterator, e.g., with a start item, a current item, and a kind of winding-number that is zero for the begin()-iterator and one for the past-the-end situation. [6] Though we have introduced the concept of circulators that allows light-weight implementations and the CGAL support library provides adaptor classes that convert between iterators and circulators (with the corresponding penalty in efficiency), so as to integrate this new concept into the framework of STL.

A serious design problem is the slight change of the semantic for circulator ranges as compared to iterator ranges. Since this semantic is defined by the intuitive operators ++ and ==, which we would like to keep for circulators as well, circulator ranges can be used in STL algorithms. This is in itself a useful feature, if there would not be the definition of a full range $[c, c)$ that an STL algorithm will treat as an empty range. However, the likelihood of a mistake may be overestimated, since for a container C supporting circulators there is no end() member function, and an expression such as sort(C.begin(), C.end()) will fail. It is easy to distinguish iterators and circulators at compile time, which allows for generic algorithms supporting both as arguments. It is also possible to protect algorithms against inappropriate arguments using the same technique, though it is beyond our scope to extend STL algorithms.

12. Conclusion

We have presented a design framework for combinatorial data structures, such as polyhedral surfaces and planar-maps. It can be extended to model the topology of curved-surfaces and can be applied to other combinatorial data structures, such as triangle-based structures for triangulations.

A proper definition of the modeling space for polyhedral surfaces has been given whose strictness has been proven useful in our applications. Suitable edge-based data structures from the literature have been

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

discussed, wherefrom the halfedge data structure has been chosen for an implementation. The discussion has revealed several desirable options for the data structure, which have led to the demand for flexibility in the design, especially the tradeoffs between space and time. An example in the CGAL Reference Manual [3] uses a similar idea like the encoding with indices from the quad-edge data structure; it uses a bit instead of a pointer to encode the opposite halfedge.

The generic programming paradigm has led to an easy-to-use and flexible design. We can explore many tradeoffs between time and storage efficiency, iterator categories, and modifiability. The solutions given for the design goals from Section 5 are in the same order:

(1) The actual storage organization of the item types can be easily changed by selecting an appropriate halfedge data structure provided in the library.
(2) The actually provided incidence relations are selected by the user with the kind of item types used. If the predefined item types from the library are not sufficient, new incidences are easily added, similar to the addition of other attributes. The optional incidences are supported by the halfedge data structure and the polyhedron. Compile-time tags are used to implement this support.
(3) Further data and functions can be easily added to the item types by derivation of predefined item types and replacing them in the items class. The coherent type system with its use of templates decouples the item types, such that, for example, adding color to a facet does not imply that the vertex type or the halfedge type needs to be changed as well.
(4) The data structure provides an easy-to-use high-level interface based on appropriate concepts, e.g., Euler operators, iterators, and circulators. These concepts support the optional incidences of the item types based on the compile-time tags. Furthermore, the integrity of the representation is maintained, but a protected access to the internal representation is granted for special function objects derived from the base class `Modifier_base`.
(5) The management of connected components and containment relations is separated into its own layer, for example the topological map.
(6) The concept of circulators supports circular sequences of edges around a vertex or around a facet. The concept is well integrated into the framework of STL.
(7) The concept of handles has been introduced. Internally the handles are defined as the iterators of the container classes. Wherever handles are required as arguments, iterators or circulators can be used as well.

The design is still open to incorporate other techniques as well, such as runtime flexibility where appropriate or additional template parameters. We expect a continuation of this approach in CGAL.

Acknowledgements

The author wishes to thank Emo Welzl for the freedom and encouraging guidance throughout these studies, Michael Hoffmann, Nora Sleumer, Falk Tschirschnitz and Martin Will for reading and commenting on previous versions, and Andreas Fabri, Geert-Jan Giezeman, Stefan Schirra and Sven Schönherr for the intensive discussions on C++ design and software engineering since we started with the CGAL kernel almost four years ago. The CGAL project has grown and the author wants to thank all members of the project for the inspiring environment, especially the people attending the CGAL meeting at INRIA, Sophia Antipolis, where the connection between the planar map and the polyhedral

surface design has been discussed. Thanks also to Jean-Daniel Boissonnat for encouragement and to the anonymous referees for valuable comments.

References

[1] B.G. Baumgart, A polyhedron representation for computer vision, in: National Computer Conference, AFIPS, Anaheim, CA, 1975, pp. 589–596.
[2] H. Bendels, D.W. Fellner, S. Havemann, Modellierung der Grundlagen: Erweiterbare Datenstrukturen zur Modellierung und Visualisierung polygonaler Welten, in: D.W. Fellner (Ed.), Modeling – Virtual Worlds – Distributed Graphics, Bad Honnef/Bonn, November 1995, pp. 149–157.
[3] H. Brönnimann, S. Schirra, R. Veltkamp, M. Yvinec (Eds.), CGAL Reference Manual, CGAL R1.2, January 1999.
[4] J.O. Coplien, Curiously recurring template patterns, C++ Report, February 1995, pp. 24–27.
[5] M. de Berg, M. van Krefeld, M. Overmars, O. Schwarzkopf, Computational Geometry: Algorithms and Applications, Springer, 1997.
[6] A. Fabri, G. Giezeman, L. Kettner, S. Schirra, S. Schönherr, The CGAL kernel: A basis for geometric computation, in: M.C. Lin and D. Manocha (Eds.), ACM Workshop on Applied Computational Geometry, Philadelphia, PA, 27–28 May 1996, Lecture Notes in Computer Science, Vol. 1148, pp. 191–202.
[7] A. Fabri, G. Giezeman, L. Kettner, S. Schirra, S. Schönherr, On the design of CGAL, the computational geometry algorithms library, Software – Practice and Experience, 1999, to appear.
[8] E. Gamma, R. Helm, R. Johnson, J. Vlissidis, Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.
[9] G. Giezeman, R. Veltkamp, W. Wesselink, Getting Started with CGAL, CGAL R1.2, January 1999.
[10] A.S. Glassner, Maintaining winged-edge models, in: J. Arvo (Ed.), Graphics Gems II, Academic Press, New York, 1991, pp. 191–201.
[11] L. Guibas, J. Stolfi, Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams, ACM Trans. Graphics 4 (3) (1985) 74–123.
[12] J. Hartman, J. Wernecke, The VRML 2.0 Handbook: Building Moving Worlds on the Web, Addison-Wesley, Reading, MA, 1996.
[13] C.M. Hoffmann, Geometric and Solid Modeling– An Introduction, Morgan Kaufmann, San Mateo, CA, 1989.
[14] M. Hoffmann, Line-Sweep auf einem Gitter, Diplomarbeit, Freie Universität Berlin, Germany, 1996.
[15] International Standard ISO/IEC 14882: Programming Languages – C++. American National Standards Institute, New York, 1998.
[16] L. Kettner, Designing a data structure for polyhedral surfaces, in: Proc. 14th Annu. ACM Sympos. Comput. Geom., 1998, pp. 146–154.
[17] L. Kettner, E. Welzl, Contour edge analysis for polyhedron projections, in: W. Straßer, R. Klein, R. Rau (Eds.), Geometric Modeling: Theory and Practice, Springer, 1997. (Proc. Internat. Conf. Theory and Practice of Geometric Modeling in Blaubeuren, Germany, October 1996.)
[18] J. Lakos, Large Scale C++ Software Design, Addison-Wesley, Reading, MA, 1996.
[19] S.B. Lippman, Inside the C++ Object Model, Addison-Wesley, Reading, MA, 1996.
[20] M. Mäntylä, An Introduction to Solid Modeling, Computer Science Press, Rockville, MD, 1988.
[21] K. Mehlhorn, S. Näher, LEDA: A platform for combinatorial and geometric computing, Comm. ACM 38 (1) (1995) 96–102.
[22] K. Mehlhorn, S. Näher, C. Uhrig, The LEDA User Manual, Version 3.5, LEDA Software GmbH, Saarbrücken, Germany, 1997.
[23] D.E. Muller, F.P. Preparata, Finding the intersection of two convex polyhedra, Theor. Comput. Sci. 7 (1978) 217–236.

[24] D.R. Musser, A. Saini, STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Addison-Wesley, Reading, MA, 1996.
[25] D.R. Musser, A.A. Stepanov, Generic programming, in: Proc. 1st Internat. Joint Conf. of ISSAC-88 and AAEC-6, Lecture Notes in Computer Science, Vol. 358, Springer, 1989, pp. 13–25.
[26] D.R. Musser, A.A. Stepanov, Algorithm-oriented generic libraries, Software – Practice and Experience 24 (7) (1994) 623–642.
[27] N.C. Myers, Traits: a new and useful template technique, C++ Report, June 1995.
[28] M.H. Overmars, Designing the computational geometry algorithms library CGAL, in: M.C. Lin and D. Manocha (Eds.), ACM Workshop on Applied Computational Geometry, Philadelphia, PA, 27–28 May 1996, Lecture Notes in Computer Science, Vol. 1148.
[29] M. Phillips, Geomview Manual: Geomview Version 1.5 for Silicon Graphics Workstations, The Geometry Center, University of Minnesota, October 1994.
[30] S. Schirra, A case study on the cost of geometric computing, in: Proc. of ALENEX'99, 1999, to appear.
[31] Silicon Graphics Computer Systems, Inc., Standard template library programmer's guide, `http://www.sgi.com/Technology/STL/`, 1997.
[32] E. Steinitz, H. Rademacher, Vorlesung über die Theorie der Polyeder (unter Einschluß der Elemente der Topologie), Springer, 1934.
[33] A.A. Stepanov, M. Lee, The standard template library, `http://www.cs.rpi.edu/~musser/doc.ps`, October 1995.
[34] B. Stroustrup, The C++ Programming Language, 3rd Edition, Addison-Wesley, Reading, MA, 1997.
[35] K. Weiler, Edge-based data structures for solid modeling in curved-surface environments, IEEE Comput. Graphics Appl. 5 (1) (1985) 21–40.
[36] J. Wernicke, The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2, Addison-Wesley, Reading, MA, 1994.