Designing a Data Structure for Polyhedral Surfaces

Lutz Kettner*, ETH Zürich, Switzerland.

Abstract

Design solutions for a program library are presented for combinatorial data structures in computational geometry, such as planar maps and polyhedral surfaces. Design issues considered are genericity, flexibility, time and space efficiency, and ease-of-use. We focus on topological aspects of polyhedral surfaces. Edge-based representations for polyhedrons are evaluated with respect to the 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 managing holes in facets.

1 Introduction

Combinatorial structures, such as planar maps, are fundamental in computational geometry. In order to use computational geometry in practice, a solid library must provide generic and flexible solutions as one of its fundamental cornerstones. Other design criteria are time and space efficiency. Ease-of-use is necessary to make the power of a design accessible and to attract users. We report a solution proposed for the Computational Geometry Algorithms Library CGAL[1], which is a joint effort of seven academic institutes in Europe [7, 6, 27].

We focus on edge-based representations of three-dimensional polyhedral surfaces and illustrate connections to planar maps and face-based structures, which may have holes in their 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.

*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 ESPRIT IV LTR Project No. 21957 (CGAL) is also acknowledged.

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

SCG 98 Minneapolis Minnesota USA

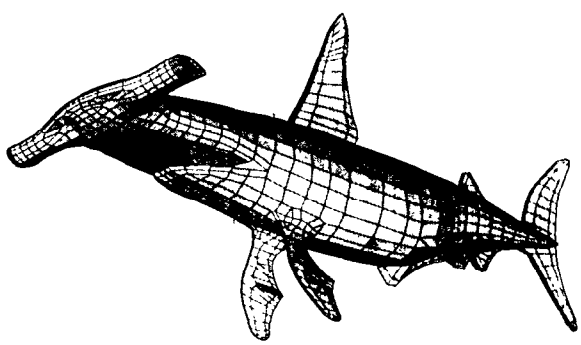

Figure 1: Hammerhead, an orientable 2-manifold of 2560 vertices. This one is homeomorphic to a sphere.

In the first part of the paper we define polyhedral surfaces and review known edge-based boundary representations. In the second part we start with a short introduction to the modern design principles available in C++ and known as the generic programming paradigm from the Standard Template Library, STL [4, 25, 29]. We derive design goals and evaluate previous work. We continue with an overview of our design, present several aspects in more detail and conclude with its evaluation. The two main advantages of our design are: The flexibility is completely handled at compile time, i.e. there is no runtime overhead due to the flexibility, and 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.

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, 21]. For a living example see Figure 1.

The two types of boundary representations are 2-manifold and non-manifold surfaces. A 2-manifold surface is a surface where for each point on the surface there exists a neighborhood that is homeomorphic to the open disc. Non-manifold examples are two tetrahedrons 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

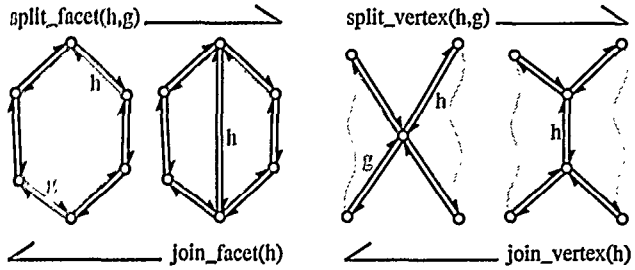

Figure 2: Euler operator examples for polyhedral surfaces.

the Klein bottle. We consider only orientable 2-manifolds.

The natural operations under which 2-manifolds are closed are Euler operations; Four of them are shown in Figure 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 [21]. Note that 2-manifolds are not closed under (regularized) boolean operations.

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

We now present a definition for polyhedral surfaces following Steinitz [30]. It is the basis for the combinatorial integrity definition of the polyhedral surface data structure and will lead to a stricter class of representable surfaces, which provides more insight in the combinatorial structure of the representation.

Definition 2.1. 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 2.2. 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 is the set of edges and facets incident to the vertex. If we restrict the incidence relation to this neighborhood then each facet is incident to exactly two edges and each edge is incident to exactly two facets. The neighborhood decomposes into disjoint cycles. As for the dual, the neighborhood of a facet is the set of incident edges and vertices and decomposes into cycles too. Assuming that the neighborhood of each facet is a single cycle (geometrically speaking: no holes in the facet), we can define a polyhedral complex as *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 2.3. The *boundary representation of a polyhedron* is a polyhedral complex with a mapping $V \to \mathbb{R}^3$. This extends to the edges by mapping them to the open, straight line segments between their two incident endpoints. The following additional conditions must hold.

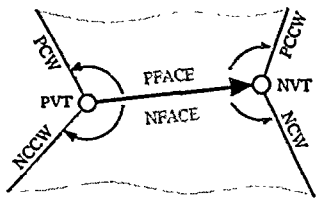

Figure 3: A winged-edge.

(5) The neighborhood of each vertex and each facet is a single cycle.

(6) The polyhedral complex is orientable.

(7) The mapping of the cycle of the neighborhood of each facet is the boundary of a simple, planar polygon. The mapping extends for F to the open region of these polygons.

(8) The images of V, E and F are pairwise disjoint.

The surface defined by such a boundary representation is an orientable 2-manifold where 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 where 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 by two changes in the definition: Condition (2) can be relaxed to allow edges that are incident to one facet; they are called *border edges*. This induces a modification of (5): The neighborhood of a vertex decomposes into either a 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 polyhedrons as well as for surfaces with boundaries with a simple extension denoting "empty" facets.

A suitable data structure based on the Definition 2.3 for polyhedral surfaces has been used successfully for three years 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 certain algorithms[2] [18]. 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.

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 primi-

[2] And holes are not represented in the file-formats that occur usually in visualization, for example VRML [12], Open Inventor [34] or the Object File Format OFF [28]. These consist of a list of vertices followed by a list of facets. Each facet is a list of indices denoting a subset of the points. Edges are not explicitly stored but can be derived from the vertices shared by facets. These formats are not strict enough for our purpose since they can represent non-manifold configurations where three or more facets are incident to a single edge, non-orientable 2-manifolds, and also violate condition (3) for polyhedral complexes. But they cannot represent holes in facets.

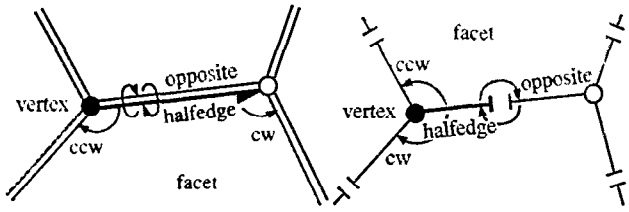

Figure 4: FE-structure (left) and VE-structure (right).

tive operations and storage costs. The representative example chosen is the traversal around a vertex to the next counterclockwise edge.

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 Figure 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 [33]. 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 loosing the traversal capabilities knowing the edge visited previously. However, all four edge pointers must remain if loops or multi-edges are allowed since otherwise the traversal around a vertex or facet is no longer uniquely defined [33]. The winged-edge data structure where the wings PCCW and NCCW are omitted has been called **Doubly Connected Edge List** (DCEL) by [24] 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 leads to cumbersome storage layouts and function interfaces.

Halfedge Data Structure. The orientation problem can be solved for the winged-edge data structure by splitting the edge into the two symmetric records, called *halfedges*, and adding mutual links to each other [33]. There are two ways of splitting the edge, which are actually dual to each other. In both situations the 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 [33] the source vertex and the facet to the right were chosen. The FE-structure in Figure 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 Figure 4

[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.

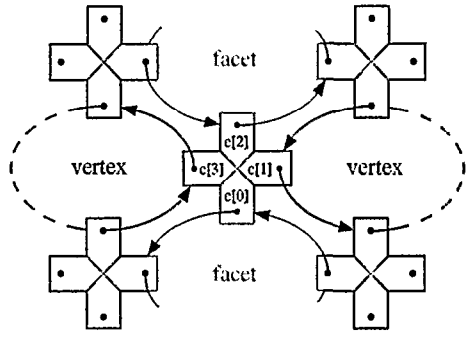

Figure 5: Quad-edge data structure.

on the right. 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 [33]. High-level operations maintaining integrity are again Euler operators. The solid modeling system GWB [21] 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.

Quad-Edge Data Structure. If we perform both halving steps for the halfedge data structure, we end up with the quad-edge data structure [11]. It provides a fully symmetric view on the primal and the dual graph, as can be seen in Figure 5. Instead of using opposite pointers, a two bit counter r is used to address a slot in an edge record of four quad-edges. With an additional bit f per edge for the flipped status the quad-edge data structure is able to model non-orientable 2-manifolds.

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

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

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

	Winged-Edge	Half-Edge	Quad-Edge
Modeling space	orientable 2-manifold		2-manifold
Operations	Euler operator		`Splice()`
Duality	at compile time (with adaptor)		at runtime (`Rot` operator)
Holes in facets	yes	yes	no
Basic traversal	case distinction	direct access	modulus 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

Table 1: Comparison of the edge-based data structures.

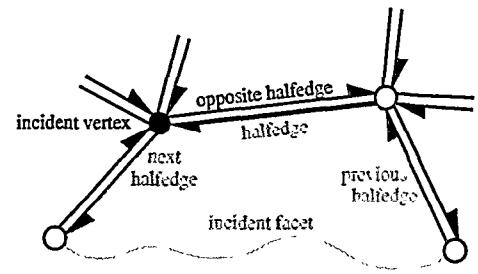

Figure 6: Halfedge data structure.

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 necessary. This reduces to 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.

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 an order of magnitude slower than computing. 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. We consider non-orientability as not so important since three-dimensional surfaces of solid objects are always orientable.

The choice for our design is a halfedge data structure like the FE-structure. The conventions used are depicted in Figure 6. 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 genericity, flexibility, time efficiency, space efficiency and ease-of-use. Two techniques are available in C++ for realizing generic and flexible designs: *Object-oriented programming*, using inheritance from base classes with virtual member functions, and *generic programming*, using class templates and function templates.

The flexibility in the *object-oriented programming paradigm* 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 – even at runtime. Generic 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 [19], 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 [20]. 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[4] 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 coordinate access and arithmetic for low-dimensional geometric objects and traversals of combinatorial structures. Vertices, edges and facets for polyhedrons are anticipated to be small objects with simple member functions. The space and runtime overhead introduced through virtual member functions would not be negligible.

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.
};
```

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

```
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++, 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. symbolic tags.

For class templates exist the specialty 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 [4]. 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 [4, 25, 29]. The main source of its generality and flexibility stems from the separation of *concepts* and *models* [29]. 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 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 [32]. 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 with the STL to promote the re-use of existing generic algorithms and container classes, but – more important – to unify the look-and-feel of the design of CGAL with the C++ Standard. It is therefore easy to learn and easy to use for users familiar with the STL. In a few places we make use of the object-oriented programming paradigm, for example the protected access to the internal representation, see Section 9.

5 Design Goals for Polyhedral Surfaces

We define the concept *polyhedron* similar to STL container classes to be responsible of managing the items of a polyhedral surface and their combinatorial structure. We have identified the following design issues:

1. The edge-based data structures discussed in the previous section have a natural notion of the edges around a vertex or 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 – what we call *circulator* – for this kind of structure.

2. STL containers base their interface on iterators. For polyhedral surfaces the order of the stored items is not well-defined in certain situations, 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.

3. 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 that 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, like managing chunks of memory or simple allocation on the heap without any iterators over all items could be anticipated too.

4. The necessary incidence information might depend on the application. The minimum needed for traversals are `next()` and `opposite()` pointers. The `prev()` pointer can be simulated with a search around the vertex or facet. For triangulations this is still a simple expression, i.e. `prev()` ≡ `next()->next()`, and for constant degree a constant time operation. If no information needs to be attached to vertices or facets, no storage should be allocated for them, including the referencing pointer in the edges. In its extreme the data structure reduces to an undirected graph.

5. It should be easy to add additional information to the different items, e.g. color to facets. Geometry will be attached using the same technique. Modifying 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.

6. 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 2.3. Advanced algorithms concerned with efficiency, e.g. a file format scanner, should be allowed to access the internal structure in a controlled fashion.

7. The management of connected components and containment relations, e.g. holes in facets or shells, is seen as an independent functionality with its own layer. Different solutions can be envisioned.

We concentrate on the combinatorial aspects of the polyhedral surface. Additional issues will appear when considering geometry, for example flexibility with the point type and the geometric predicates. One technique explored for this in CGAL is an extension of the traits classes [26] known from the C++ standard library, see [7, 9, 3].

6 Previous Work

The Library of Efficient Datatypes and Algorithms (LEDA) [22, 23] contains no data structure tailored for three-dimensional polyhedrons, but it provides a general data structure for graphs and one for planar maps derived from graphs. Additional information can be attached either by parameterized graphs or by node arrays and edge arrays. These are associative arrays (hash tables) which allows the easy addition of information even for temporary purposes. The disadvantage of the parameterized graph is that one must always specify both parameters. The disadvantages of the node and edge arrays are the additional costs for the lookup operations and additional storage requirements. A more subtle disadvantage is that a reference to a node is not sufficient to retrieve its associated attributes. The array must be known too. The current size of graph nodes is equivalent to 13 pointers and for halfedges it is 11 pointers. There is no flexibility for obtaining smaller graph structures. LEDA has its own notion of iterators and a rich body of algorithms for working on graphs. At the moment it is not compliant to the STL. LEDA is very homogeneous and easy-to-use. Its own framework is generic and flexible but (currently) monolithic when combined with other libraries like the STL. It does not reach the flexibility in runtime and space efficiency tuning 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 and genericity are 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 developed.

7 Circulators

Our new concept of *circulators* reflects 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[5]. 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 CGAL 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 requirements with iterators. Three circulator categories are defined: forward, bidirectional and random-access circulators. Given a circulator `c` the operation `*c` denotes the item the circulator refers to. The operation `++c` advances the circulator by one item and `--c` steps a bidirectional circulator one item backwards. For random-access circulators `c+n` advances the circulator by `n` where `n` is a natural number. Two circulators can be compared for equality.

Circulators develop different notions of reachability and ranges than iterators. A circulator `d` is called *reachable* from `c` if `c` can be made equal to `d` with finitely many applications of the operator `++c`. Due to the circularity of the data structure this is always true if both circulators refer to items of the same data structure. In particular, `c` is always reachable from `c`. Given two circulators `c` and `d`, the range `[c,d)` denotes all circulators obtained by starting with `c` and advancing `c` until `d` is reached, but does not include `d` if $d \neq c$. So far it is the same range definition as for iterators. The difference lies in the use of `[c,c)` for denoting all items in the circular data structure, whereas for an iterator `i` the range `[i,i)` denotes the empty range. As long as `c != d` the range `[c,d)` behaves like an iterator range and could be used in STL algorithms. It is possible to write just as simple algorithms that work with iterators as well as with circulators, including the full range definition [16]. 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 [16].

We repeat the example for the generic `contains` function from 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;
}
```

8 Design Overview

The global picture of the design is given in Figure 7. The design strictly separates topology and geometry. Vertices, halfedges and facets carry both kinds of information. The `Halfedge_data_structure` is the container managing these three items and their topological relations. The `Topological_planar_map` is a facet-based representation. It maintains holes in facets and is able to enu-

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

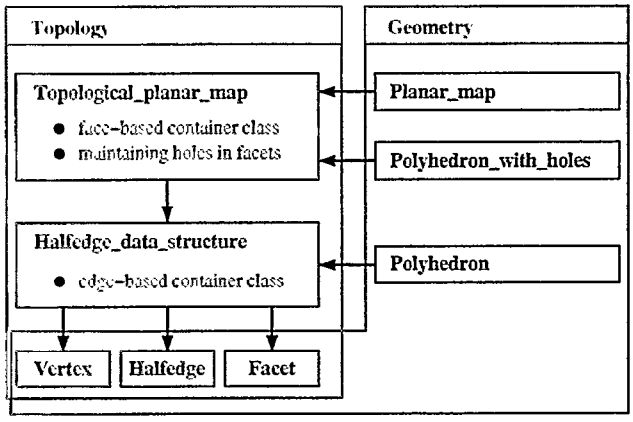

Figure 7: Design overview.

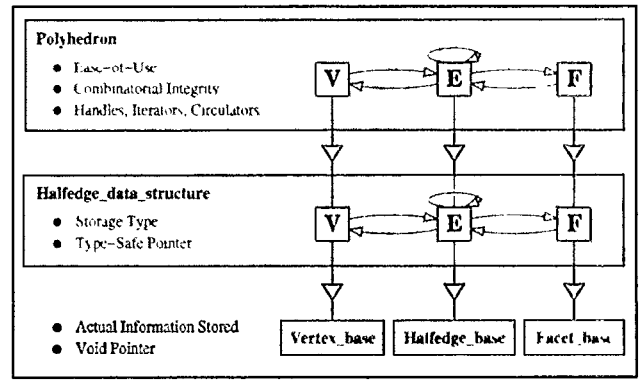

Figure 8: Responsibilities of the different layers in the design.

merate outer and inner boundaries of a facet. It uses the edge-based data structure. The `Polyhedron` uses the `Halfedge_data_structure` and adds geometric operations. It imposes further restrictions on the data structure as defined for the polyhedral complex above, for example, that an edge always has two distinct endpoints. The `Planar_map` and a possible `Polyhedron_with_holes` are based on the topological planar map since they will maintain holes in facets.

All entities in this picture are sets of requirements. Each can have multiple models. There are many different possibilities for vertices, edges and facets. Currently two different models are provided for the `Halfedge_data_structure`. Many combinations are possible and result in different polyhedron data structures. We make use of the implicit instantiation of template classes in C++ as described above. The requirement sets consist of a mandatory part that every model must comply with and certain optional parts a model must only comply with if the corresponding functionality is actually used. For example, a vertex is allowed to be empty. If we want to use it for the polyhedral surface then the normal vector computation imposes the additional requirements that the vertex must contain a three-dimensional point and must give access to it with the member function `point()`.

9 Design of the Polyhedron

A more refined picture of the design is shown in Figure 8. At the bottom we start with base classes for vertices, halfedges and facets. Their responsibilities are the actual storage of the incidences in terms of `void`-pointers, the geometry and other attributes. Especially the storage of incidences with `void`-pointers allows, for example, the facet class to be exchanged without changing the halfedge class. The advantage of strong type checking will be reestablished for the `void`-pointers in the next layer. Implementations for vertices, edges and facets are provided that fulfill the minimal set of requirements. They can be used as base classes for own extensions. Richer implementations are provided as defaults; for polyhedrons they provide a three-dimensional point in the vertices and a plane equation in the facets.

The `Halfedge_data_structure` is responsible of the storage organization of the vertices, halfedges and facets. Currently implementations are provided that use a bidirectional list or an STL vector internally. The `Halfedge_data_structure` derives new classes for vertices, halfedges and facets. They replace the `void`-pointer incidence information with type-safe pointers at the interface. Additional information besides the incidence information simply stays unaffected and will be inherited from the base classes.

For the `Halfedge_data_structure` different models are possible (two are already available). Thus the set of requirements for the `Halfedge_data_structure` is kept small. To support the implementation of high-level operations, a helper class `Halfedge_data_structure_decorator` is provided, which is not shown in Figure 8 but would be placed at the side of the `Halfedge_data_structure` since it broadens that interface but does not hide it. It adds Euler operations and adaptive functionality. For example, if the `prev()` function is not provided for halfedges, a `find_prev()` function searches the previous halfedge along the facet. If the `prev()` function is now implemented, the `find_prev()` function simply calls it. This distinction can be resolved at compile time with a technique called *compile-time tags*, similar to iterator tags in [31].

The `Polyhedron` layer adds ease-of-use in terms of high-level functions, high-level concepts for accessing the items, i.e. handles, iterators and circulators (pointers are no longer visible at this interface), and the protection of the combinatorial integrity. It derives new vertices, halfedges and facets to provide the handles and to hide the pointers.

Algorithms with invalid intermediate states need access to the internal representation. A protected access is granted for classes derived from `Modifier_base` following the strategy pattern [8]. The example in Figure 9 depicts the class design for a file format scanner for polyhedrons. The back-door provided here is a kind of callback-function embedded in a class object. The `Polyhedron` accepts a modifier class with the `delegate()` member function and calls its virtual `operator()` member function with the internal halfedge data structure. The `Scanner` class derives from the `Modifier_base` and implements the `operator()` function where it can access the internal representation. The achievement is here that the `delegate()` function of the `Polyhedron` can verify the validity of its own internal representation after the `operator()` function has done its work. The `Scanner` class is in charge of returning from execution only with a valid representation, even in the case of a failure. This approach is also known from database systems as transactions. The special task the `Scanner` accomplishes (only creation of new items) enables us to implement the transaction scheme efficiently with a simple rollback function that deletes all items created so far in the case of a failure. In general the rollback would be more costly.

10 Evaluation of the New Design

We will illustrate in the following that our design not only meets the design goals formulated in Section 5 but that it is also still easy to use. A certain familiarity with the look-and-feel of C or C++ will help in this section.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Halfedge_data_structure_polyhedron_default_3.h>
#include <CGAL/Polyhedron_default_traits_3.h>
#include <CGAL/Polyhedron_3.h>

typedef CGAL_Cartesian<double>                                R;
typedef CGAL_Halfedge_data_structure_polyhedron_default_3<R>  HDS;
typedef CGAL_Polyhedron_default_traits_3<R>                   Traits;
typedef CGAL_Polyhedron_3<Traits,HDS>                         Polyhedron;

int main() {
    Polyhedron P;
    P.make_tetrahedron();
    return 0;
}
```

Figure 10: Example program illustrating a default polyhedron instantiation in CGAL.

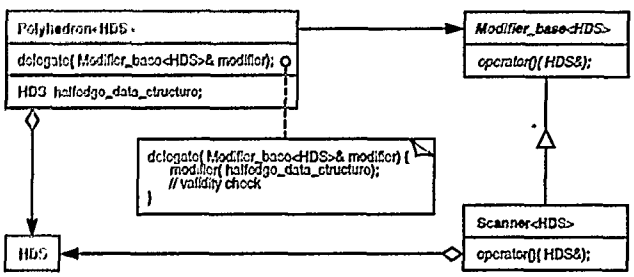

Figure 9: Class diagram of the polyhedron design illustrating the safe access to the internal representation using the strategy pattern.

We start with a complete program in Figure 10 using a default polyhedron instantiation in CGAL[6]. The `#include` directives provide the types used in the example. It is convenient to use `typedef`'s to create the nested type declarations one by one. CGAL supports different representation types; `CGAL_Cartesian` is one of them, parameterized with the coordinate type `double`. The default halfedge data structure for polyhedrons uses three-dimensional points for the vertices and a plane equation for the facets as determined by `R`. The `CGAL_Polyhedron_3` is parameterized with a traits class for the geometric parameterization and the halfedge data structure. The `main()` function declares a variable `P` for the polyhedron and creates a combinatorial tetrahedron in `P` where space for points and plane equations is reserved in the vertices and facets (even though the space is not used here). More information about representation classes and traits classes in CGAL can be found in [7].

The class `Polyhedron` provides the claimed ease-of-use with handles, iterators, circulators and high-level operations, thus addressing design goals (1) and (2). This default representation is actually equivalent to:

```
typedef  CGAL_Halfedge_data_structure_using_list<
    CGAL_Vertex_max_base< CGAL_Point_3<R> >,
    CGAL_Halfedge_max_base,
    CGAL_Polyhedron_facet_max_base<R> >     HDS;
```

The internal storage organization can be easily changed with the class `CGAL_Halfedge_data_structure_using_vector` to a vector-based one. This satisfies design goal (3). The requirements that a self-written class must fulfill to be a model of a halfedge data structure are documented in [17].

If we exchange the default base classes for the minimal base classes `CGAL_Vertex_min_base`, `CGAL_Halfedge_min_base` and `CGAL_Facet_min_base`, we get a data structure for undirected graphs; vertices and facets are empty (besides a few compile-time tags) and the minimal halfedge base class stores only a `next()` and an `opposite()` pointer. This makes four pointers per edge. See [17] for the actual short implementations of these base classes. In analogy to the quad-edge data structure we can replace the `opposite()` pointer internally by a single bit knowing that opposite halfedges are always stored in consecutive places by our halfedge data structures. Knowing C, this bit can be put into the least-significant bit of the `next()` pointer, which is always zero on todays systems. This yields an implementation with two pointers per edge! The easy realization of this idea in our design is shown in the manual [17].

We can add a color variable to the default facet:

```
template <class R>
struct Facet
: public CGAL_Polyhedron_facet_max_base<R> {
    CGAL_Color color;
};
```

This facet can be used instead of the default and we have a polyhedron with colored facets. This satisfies design goal (5). Similarly easy is the realization of data structure with incidences that lie between the minimal and maximal supported incidences. We add a previous pointer to the minimal halfedge in the following example. The type `Supports_halfedge_prev` indicates that the class now supports a previous pointer.

```
class Halfedge : public CGAL_Halfedge_min_base {
    void* prv;
public:
    typedef CGAL_Tag_true Supports_halfedge_prev;
    void*       prev()             { return prv;}
    const void* prev() const       { return prv;}
    void        set_prev( void* h) { prv = h;}
};
```

The whole spectrum of incidences is easily available in the design presented, fulfilling design goal (4). Design goals (6) and (7) are also met. The example for the easy use of the modifier mechanism and the transaction scheme for the scanner can be found in the manual [15].

Summing up, we have met all design goals presented in Section 5 without imposing runtime and storage overhead. Predefined implementations are easy to use and different solutions within the possible flexibility can be achieved with little effort.

[6] CGAL uses the prefix `CGAL_` for all global names, which will be replaced by a namespace, and the suffix `_3` for three-dimensional entities.

11 Conclusion

We have presented a design framework for combinatorial data structures, such as planar-maps and polyhedral surfaces. It extends to curved-surface environments and can also be applied to other combinatorial data structures, such as triangle-based structures for triangulations. We have identified important fundamentals for such a design: A proper definition of the modeling space, strong type checking, time and space efficiency.

The adaption of the generic programming paradigm used in the STL has led to an easy-to-use and flexible high-level interface for polyhedral surfaces featuring handles, iterators, the new concept circulators and Euler operators. The internal representation can be chosen from a wide range of different halfedge data structures exploiting many tradeoffs between time and storage efficiency, iterator categories and modifiability. Additional attributes are easy to add. Other solutions, such as dynamic type checking at runtime, generic attribute pointers or templates, can still be added within this design. The integrity of the internal representation is protected and a mechanism is available that grants safe access to it. We expect a continuation of this approach in CGAL.

Acknowledgments. The author wishes to thank Emo Welzl for the freedom and encouraging guidance throughout these studies, Nora Sleumer 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 three years ago. The CGAL project has grown and the author want 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.

References

[1] Bruce G. Baumgart. A Polyhedron Representation for Computer Vision. In *National Computer Conference*, pages 589–596, Anaheim, CA, 1975. AFIPS.

[2] Heinzgerd Bendels, Dieter W. Fellner, and Sven Havemann. Modellierung der Grundlagen: Erweiterbare Datenstrukturen zur Modellierung und Visualisierung polygonaler Welten. In D. W. Fellner, editor, *Modeling – Virtual Worlds – Distributed Graphics*, pages 149–157, Bad Honnef/Bonn, 27.–28. November 1995. `http://www.graphics.uni-bonn.de/CompGraph.ResearchProjects.MRT`.

[3] Hervé Brönnimann, Andreas Fabri, Stefan Schirra, and Remco Veltkamp, editors. *CGAL Reference Manual. Part 2: Basic Library*. 1998. CGAL R1.0. `http://www.cs.ruu.nl/CGAL`.

[4] Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++. ANSI X3, Information Processing Systems, December 1996. `http://www.maths.warwick.ac.uk/c++/pub/`.

[5] Mark de Berg, Marc van Krefeld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer Verlag, 1997.

[6] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. The CGAL Kernel: A Basis for Geometric Computation. In M. C. Lin and D. Manocha, editors, *ACM Workshop on Applied Computational Geometry*, pages 191–202, Philadelphia, Pennsylvenia, May, 27–28 1996. Lecture Notes in Computer Science 1148.

[7] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the Design of CGAL, the Computational Geometry Algorithms Library. In Dorothea Wagner, editor, *Algorithm Engineering*, Trends in Software. John Wiley & Sons, New York, 1997. to appear.

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

[9] Geert-Jan Giezeman, Remco Veltkamp, and Wieger Wesselink. *Getting Started with CGAL*, 1998. CGAL R1.0.

[10] Andrew S. Glassner. Maintaining Winged-Edge Models. In James Arvo, editor, *Graphics Gems II*, pages 191–201. Academic Press, 1991.

[11] Leonidas Guibas and Jorge Stolfi. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. *ACM Transaction on Graphics*, 4(3):74–123, July 1985.

[12] Jed Hartman and Josie Wernecke. *The VRML 2.0 Handbook: Building Moving Worlds on the Web*. Addison-Wesley, 1996.

[13] Christoph M. Hoffmann. *Geometric and Solid Modeling – An Introduction*. Morgan Kaufmann, 1989.

[14] Michael Hoffmann. Line-Sweep auf einem Gitter, 1996. Diplomarbeit. Freie Univ. Berlin, Germany.

[15] Lutz Kettner. 3D-Polyhedral Surfaces. In Hervé Brönnimann, Andreas Fabri, Stefan Schirra, and Remco Veltkamp, editors, *CGAL Reference Manual. Part 2: Basic Library*. 1998. CGAL R1.0.

[16] Lutz Kettner. Circulators. In Hervé Brönnimann, Andreas Fabri, Stefan Schirra, and Remco Veltkamp, editors, *CGAL Reference Manual. Part 3: Support Library*. 1998. CGAL R1.0.

[17] Lutz Kettner. Halfedge Data Structure. In Hervé Brönnimann, Andreas Fabri, Stefan Schirra, and Remco Veltkamp, editors, *CGAL Reference Manual. Part 2: Basic Library*. 1998. CGAL R1.0.

[18] Lutz Kettner and Emo Welzl. Contour Edge Analysis for Polyhedron Projections. In Wolfgang Straßer, Reinhard Klein, and René Rau, editors, *Geometric Modeling: Theory and Practice*. Springer Verlag, 1997. (Proc. Int. Conf. Theory and Practice of Geometric Modeling in Blaubeuren, Germany, Oct. 1996).

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

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

[21] Martti Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, Maryland, 1988.

[22] Kurt Mehlhorn and Stefan Näher. LEDA: A Platform for Combinatorial and Geometric Computing. *Communications of the ACM*, 38(1):96–102, January 1995.

[23] Kurt Mehlhorn, Stefan Näher, and Christian Uhrig. *The LEDA User Manual, Version 3.5*. LEDA Software GmbH, 66123 Saarbrücken, Germany, 1997.

[24] D. E. Muller and F. P. Preparata. Finding the Intersection of two Convex Polyhedra. *Theoretical Computer Science*, 7:217–236, 1978.

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

[26] Nathan C. Myers. Traits: a New and Useful Template Technique. *C++ Report*, June 1995.

[27] Mark H. Overmars. Designing the Computational Geometry Algorithms Library CGAL. In M. C. Lin and D. Manocha, editors, *ACM Workshop on Applied Computational Geometry*, Philadelphia, Pennsylvenia, May, 27–28 1996. Lecture Notes in Computer Science 1148.

[28] Mark Phillips. *Geomview Manual: Geomview Version 1.5*. The Geometry Center, University of Minnesota, October 1994.

[29] Silicon Graphics Computer Systems, Inc. Standard Template Library Programmer's Guide. `http://www.sgi.com/Technology/STL/`, 1997.

[30] E. Steinitz and H. Rademacher. *Vorlesung über die Theorie der Polyeder (unter Einschluß der Elemente der Topologie)*. Springer, Berlin, 1934.

[31] Alexander Stepanov and Meng Lee. The Standard Template Library. `http://www.cs.rpi.edu/~musser/doc.ps`, October 1995.

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

[33] Kevin Weiler. Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments. *IEEE Computer Graphics and Application*, 5(1):21–40, January 1985.

[34] Josie Wernicke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2*. Addison-Wesley, 1994.