OpenMesh – a generic and efficient polygon mesh data structure

M. Botsch S. Steinberg S. Bischoff L. Kobbelt
Lehrstuhl für Informatik VIII
Computergraphik und Multimedia
RWTH Aachen

Abstract

We describe the implementation of a half-edge data structure for the static representation and dynamic handling of arbitrary polygonal meshes. The particular design of the data structures and classes aims at maximum flexibility and high performance. We achieve this by using generative programming concepts which allow the compiler to resolve most of the special case handling decisions at compile time. We evaluate our data structure based on prototypic implementations of mesh processing applications such as decimation and smoothing.

1 Introduction

Polygonal meshes are the most appropriate geometry representation for interactive 3D graphics applications. They are flexible enough to approximate arbitrary shapes to any approximation tolerance and they can be processed efficiently by the current graphics hardware which is available even on today's low cost PCs.

One of the long-term goals of the OpenSGplus initiative is to design and implement a set of software tools to enhance the OpenSG API with high level functionality for handling complex geometric objects. In this context a properly designed geometry kernel plays an important role since most high level algorithms will be based on it. The nature of the planned target applications such as level-of-detail management and subdivision surfaces implies that this unified geometry representation should be based on polygonal meshes.

In this short paper we describe the design and implementation of our polygonal mesh data structure. We start by identifying the specific requirements for the data structure in the context of the particular algorithms that are planned within the OpenSGplus project. These requirements allow us to identify three major design goals which are *flexibility* (with respect to the underlying implementation and to the higher level algorithms), *efficiency* (in time and memory usage), and *ease-of-use* (the API has to hide the complexity of the underlying data structures).

2 Requirements

The main design goal for our library is *flexibility*. Since we do not want to overly restrict the set of applications, it should be able to provide randomly access to vertices, edges and faces, where faces can be arbitrary polygons and not just triangles. The user should be able to choose between arrays or lists as underlying container types and arbitrary scalar types.

The second goal is *time and space efficiency*. Especially algorithms like decimation or smoothing of meshes need to navigate through the one-ring-neighborhood of a vertex. So we must allow fast access to such information. For space efficiency, memory should be allocated only for elements actually used, e.g., polygons with no attributes do not allocate pointers to their elements.

The third goal is *ease-of-use*. The API should be easy to understand and to use for non-experts so that it can be integrated easily into any other library like, e.g., OpenSG. Our goal is a user interface that wraps and hides the complex underlying structure with a simple API.

3 Previous Work

A polygonal mesh consists of a set of vertices, edges, faces and topological relations between them. Based on these relations, a data structure defines how each element is stored and what references to its neighborhood it needs. In the following we give a short survey over such data structures.

We mainly distinguish between *face-based* and *edge-based* data structures. Face-based data structures store for each face pointers to its vertices and its neighboring faces. This makes it possible to navigate around each vertex by visiting all surrounding faces, which is frequently done in many algorithms. However, special case handling is needed, when navigation runs over items with variable valence. Suppose a mesh that consists of triangles as well as quadrangles, then navigation would not work without many time consuming case distinctions.

Edge-based data structures store for each edge pointers to both vertices and to the neighboring edges. Since an edge has always the same topological structure, it is possible to handle polygons with variable valence in one mesh.

There are several edge-based variants that differ only in the topological information they store. The *winged-edge* data structure [2, 12] stores for each edge references to its vertices, to both polygons sharing this edge, and to the four neighboring edges or *wings* (see Figure 1). Traversing the neighborhood requires one case distinction per step, because an edge does not encode its orientation explicitly.

The *half-edge* data structure solves this problem by splitting each edge into two halves, where each half-edge points to its opposite half-edge, an incident vertex and an incident polygon. For a detailed description see [7].

The Computational Geometry Algorithm Library, CGAL[1], is closely related to our mesh data structure. It is based on half-edges and consists of three main parts. The first part manages and organizes the geometric primitives (vertices, edges and faces), the second part handles the topological relations between the primitives and the third part provides additional functionality like circulators, iterators and I/O support.

CGAL is a very powerful library because of its flexibility and efficiency. But in the context of our applications some design decisions have to be reconsidered.

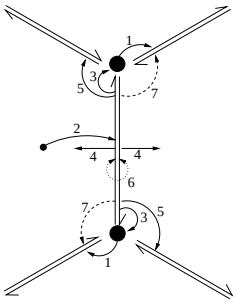
The CGAL programming interface does not support the inclusion of specialized mesh kernels such as, e.g., quad-tree or array-based representations for recursively refined subdivision surfaces which, however, are necessary for their efficient implementation. This restriction will also forbid later enhancements like more sophisticated kernels that are able to deal with non–manifold meshes, see Section 4.1.

Moreover, for array–based CGAL meshes the size has to be known a priori to the constructor. Therefore, applications that dynamically change the mesh complexity have to use the less space efficient list–based CGAL mesh. In the OpenMesh implementation we provide, in contrast, dynamic memory management also for array-based meshes.

[1] http://www.cgal.org

4 OpenMesh

This section will describe the basic design decisions leading to the current OpenMesh implementation. The programming interface and some of the implementation details are explained based on simple examples. A more detailed description can be found in the documentation [10].



1. Vertex ↦ one outgoing halfedge,
2. Face ↦ one halfedge,
3. Halfedge ↦ target vertex,
4. Halfedge ↦ its face,
5. Halfedge ↦ next halfedge,
6. Halfedge ↦ opposite halfedge (implicit),
7. Halfedge ↦ previous halfedge (optional).

Figure 1: Halfege data structure.

4.1 Data Structure

The first design decision for a mesh library is to choose one of the data structures introduced in the last section.

To be as flexible as possible we should represent each of the *mesh items* (vertices, edges and faces) explicitly, in order to be able to attach additional attributes and functionality to them. Since e.g. primal subdivision algorithms need to store new point positions per edge (see [13]), it is more efficient to provide an edge or halfedge type in addition to vertices and faces and to store this new point in the edge itself.

Although every general polygonal mesh can be tesselated to a mesh consisting of triangular faces only, we do not restrict the mesh library to pure triangle meshes. Instead we provide a general polygonal mesh on the one hand, and a more efficient triangle mesh on the other hand, so that the user can trade off between speed and generality.

When dealing with arbitrary polygonal meshes, halfedge–based representations also provide a much simpler algorithmic structure: Since halfedges store the main connectivity information and always have the same topology, vertices, (half–)edges and faces are types of constant size. In contrast, a face–based representation has to store a variable amount of vertex handles in each face.

In many real world applications one has to deal with non–manifold input data. These meshes may contain two types of non–manifold degeneracies:

- *Complex vertices*: The neighorhood of these vertices is not a topological disc (nor a halfdisc at the boundary).
- *Complex edges* are characterized by belonging to more than two faces.

Our current OpenMesh implementation can handle complex vertices, but does at the moment not support complex edges. This functionality could be added by providing a *mesh kernel* for non–manifold meshes (see the next section).

In order to enable efficient implementation of standard algorithms like e.g. smoothing, subdivision or decimation, the most frequent atomic mesh operations performed by these algorithms have to be taken into account when designing the data structure. In addition to the ususal connectivity information, all of these algorithms require access to the one–ring neighborhood of a vertex, possibly in terms of vertices or incident edges or faces.

Considering these special requirements implies the use of a halfedge–based data structure, where we have a natural representation of all types of mesh items and arbitrary polygonal faces. The halfedge structure also provides fast, constant–time access to the one–ring neighborhood of vertices. Starting at an outgoing halfedge of a certain vertex, the next halfedge in clockwise order can be reached by two indirections: go to the next halfedge of the face, then to its opposite halfedge. Face based structures have to perform an expensive search within the current face in order to find this next halfedge.

The connectivity information stored in our implementation is illustrated in Figure 1.

4.2 Interface

Depending on the application one often needs to store additional data for certain mesh items. The code in Listings 1 and 2, e.g., has to store one additional point for each vertex (the center of gravity of its neighbors). While an additional array could be used to hold these points, this approach will become cumbersome and error–prone if more and more external elements are to be stored and have to be synchronized with the mesh. The natural solution is to pack this data directly into the corresponding mesh items, like e.g. attaching the additional point coordinate to the vertex type. This approach will also result in more coherence in the memory footprint, thereby improving cache efficiency as well.

To provide this flexibility, meshes can be custom–tailored for the needs of specific algorithms by parameterizing them by a so–called *traits* class. The rest of this section will describe the customizing possibilities, their implementation details will be discussed in section 4.3. To fully specify a mesh, several parameters can be given:

Face Type: Specifies whether to use a general polygonal mesh or the (more efficient) triangle mesh.

Kernel: The mesh kernel is responsible for storing the mesh items internally. For dynamic meshes (e.g. for mesh decimation) a kernel based on doubly linked lists can be used, since inserting/deleting elements to/from lists is more efficient than for arrays. An array–based kernel in turn provides faster traversal and consumes less memory and should therefore by used for applications dealing with static meshes only.

Traits: In addition to the previous two points, the mesh is parameterized by a traits class, that allows to enhance mesh items by arbitrary functionality. These *user–defined* classes are added to the corresponding mesh items in terms of inheritance. In addition the user can choose from a set of pre–defined common *attributes* like e.g. normal vector or color and attach them to certain mesh items, too. The traits class also selects the *coordinate* type and the *scalar* type of the mesh, so that — depending on the application — 2D–, 3D– or n–D vectors and float, double, or even exact arithmetic can be used.

All combinations of face type and mesh kernels are conveniently accessible through pre–defined classes, that are further parameterized by the user traits (see Listing 1).

The mesh items are referred to by handle types, that are defined by the mesh kernel: an array–based kernel uses indices to address items, a list–based kernel uses pointers. Therefore most API functions are based on these abstract handle types.

In order to navigate through the mesh, the OpenMesh provides iterators and circulators, that are used in the same way as STL iterators. Iterators just enumerate all mesh items of one type: in Listing 2 a `VertexIter` is used to iterate through all the vertices, ranging from `mesh.vertices_begin()` to `mesh.vertices_end()`.

Convenient access to the one–ring neighborhood of vertices is provided by vertex circulators. Given a center vertex, all one–ring information (incoming/outgoing halfedges, incident faces and neighboring vertices) can be accessed using a similar syntax as for iterators. In Listing 2 the center of gravity of each vertex is computed

```
#include <ACG/Mesh/TriMesh_ArrayKernelT.hh>

struct MyTraits : public DefaultTraits
{
  template <class Base> class VertexT : public Base
  {
  public:
    const Vec3f& cog() const { return cog_; }
    void set_cog(const Vec3f& cog) { cog_ = cog; }
  private:
    Vec3f cog_;
  };
};

typedef TriMesh_ArrayKernelT<MyTraits> MyMesh;
```

Listing 1: Defining a custom–tailored mesh that stores an additional point per vertex.

```
#include <ACG/MeshIO/MeshReader.hh>
#include <ACG/MeshIO/MeshWriter.hh>

int main(int argc, char **argv)
{
  unsigned int          i(0), N(atoi(argv[1]));
  MyMesh                mesh;
  MyMesh::VertexIter    v_it;
  MyMesh::VertexIter    v_end(mesh.vertices_end());
  MyMesh::ConstVertexVertexIter c_it;

  // read mesh from argv[2]
  MeshIO::read_mesh(mesh, argv[2]);

  // smooth N iterations
  for (; i<N; ++i) {
    for ( v_it=mesh.vertices_begin();
          v_it!=v_end; ++v_it ) {
      Vec3f c(0,0,0);
      float valence(0.0);

      for ( c_it=MyMesh::ConstVertexVertexIter(mesh, v_it.handle());
            c_it; ++c_it ) {
        c += mesh.point(*c_it);   ++valence;
      }

      v_it->set_cog(c/valence);
    }

    for ( v_it=mesh.vertices_begin(); v_it!=v_end; ++v_it )
      if (!mesh.is_boundary(v_it.handle()))
        mesh.set_point(*v_it, v_it->cog());
  }

  // write result to argv[3]
  MeshIO::write_mesh(mesh, argv[3]);
}
```

Listing 2: This example shows how to navigate through the mesh using iterators and circulators. It smoothes the mesh by iteratively moving each vertex into the barycenter of its neighbors.

using a `ConstVertexVertexIter`, i.e. an iterator circulating around a center vertex and enumerating all one–ring vertices. Since the one–ring vertices will not be changed, we use the `Const` version of the circulator. In the same way all vertices or (half–) edges of a given face can easily be traversed using face circulators.

4.3 Implementation

The low–level implementation of the halfedge data structure of a mesh is encapsulated into the *mesh kernel*. It is responsible for the storage of mesh items, for accessing them through their corresponding handles and for keeping the connectivity information consistent. The chosen mesh kernel acts as template parameter for the polygonal mesh, that inherits from the kernel and adds higher–level functionality, like e.g. topological operators used in mesh decimation.

The mesh can be further parameterized by a *traits* class, that holds member classes corresponding to the different mesh items. The final item types will be derived from these classes, so that any desired functionality can be plugged in by putting it into these classes. In Listing 1 the class `VertexT` is defined to store the point coordinate `cog_`.

These additional attributes are added to the system part of the vertex using some *generative* or *aspect oriented* programming techniques (see [1, 5]). This avoids problems with mulitple inheritance (ambiguity, no controlled overloading) by building a linear inheritance chain instead:

```
template <class Base>
class AddNormal : public Base
{
public:
  const Point& normal() const { return normal_; }
  void set_normal(const Point& _n) { normal_ = _n; }
private:
  Point normal_;
};

typedef AddNormal<SomeVertex> SomeVertexWithNormal;
```

Listing 3: Adding the attribute *normal vector* to vertices.

Since the handle types depend on the containers used by the mesh kernel (e.g. a `VertexHandle` may be a `Vertex*` or an `int`), the kernel itself depends on the mesh items (in order to construct the handle types), and the items require the handles (a vertex must store a halfedge handle), we have to use template forward declarations to get “safe” handle types (see [7]). Using this technique, the item types know each other and their respective handle types, thereby avoiding to use and cast `void` pointers. This also enables us to use the handles types in the traits classes, e.g. if the face type should contain a vertex handle, see Listing 4.

Here the class `ListKernelT` gets the final mesh items as template parameter and the class `FinalMeshItemsT` expects the kernel as template argument in turn. Therefore the class `FinalMeshItemsT` does a forward declaration of the mesh kernel, resulting in the class `Kernel`. This class now provides all types of items and their corresponding handles, and hence can be used as template argument for the internal vertex as well as for the user traits. The fully specified `FinalMeshItems` can then be used as the actual template argument of the `ListKernelT`.

As we have seen, we use a custom–tailored mesh for each application. All these meshes will be different C++ types. If we want to design algorithms operating on all of these mesh types, we either have to derive all meshes from a common virtual base class, or do it the STL way and use generic programming methods. Since virtual functions / classes lead to a certain overhead in space and time, we have chosen the generic approach: every algorithm gets the type `Mesh` in form of a template parameter.

This leads to the problem that certain algorithms have to know about some details of the mesh, e.g. whether the mesh provides face normals for surface lighting. Using generative programming methods (see [1, 5]), this information can be checked for at compile–time: if the mesh provides face normals, they will be used automatically.

Another example is the optional previous halfedge handle (cf. Fig. 1): if the mesh stores the previous halfedge in addition to the next–halfedge handle, asking for prev_halfedge just returns this handle. Otherwise, the face has to be traversed by iteratively jumping to the next halfedge until the original one is reached again. See Listing 5 for the implementation: the user calls prev_halfedge, this function in turn calls one of its helper functions, depending on the result of (Halfedge::Attributes & HasPrevHalfedge). If we add the previous halfedge to our vertex class, it will automatically register itself to the vertex, so that (Halfedge::Attributes & HasPrevHalfedge) will be true. This decision is made at compile time, so that we add no run–time overhead.

As we have seen, the OpenMesh is highly customizable: from the users point of view by plugging in different traits classes, and from

```
struct ListItems
{
  // internal vertex
  template <class Kernel> class VertexT
  {
  public:
    // use references provided by Kernel
    typedef typename Kernel::HalfedgeHandle HalfedgeHandle;
    ...
  };
};


struct ListKernel
{
  template <class FinalMeshItems> class KernelT
  {
  public:
    // use provided items & define handles
    typedef typename FinalMeshItems::Vertex  Vertex;
    typedef Vertex*                          VertexHandle;

    ... // define HalfedgeHandle, EdgeHandle, FaceHandle
  };
};


template <class MeshKernel, class SysItems, class Traits>
struct FinalMeshItemsT
{
  // forward declaration of mesh kernel
  typedef FinalMeshItemsT<MeshKernel, SysItems, Traits> This;
  typedef typename MEshKernel::template KernelT<This> Kernel;

  // final Vertex type
  typedef typename SysItems::template VertexT<Kernel>  SysVertex;
  typedef typename Traits::template VertexT<SysVertex> Vertex;

  ... // define Halfedge, Edge and Face
};


struct MyTraits : public DefaultTraits
{
  template <class Base> struct FaceT : public Base
  {
    typename Base::Kernel::VertexHandle my_vertex_handle;
  };
};


typedef FinalMeshItemsT<ListKernel, ListItems, MyTraits> FinalMeshItems;
typedef ListKernel::KernelT<FinalMeshItems>              MyMeshKernel;
typedef TriMeshT<MeshKernel>                             MyMesh;
```

Listing 4: Template forward declaration provides type–safe handles for the internal mesh items as well as for the user traits.

```
enum { HasPrevHalfedge = (1<<something) };

template <bool b> class Bool2Type { enum { my_bool = b }; };

HalfedgeHandle prev_halfedge_handle(HalfedgeHandle _heh) {
  return
    prev_helper(_heh,
                Bool2Type<(Halfedge::Attributes & HasPrevHalfedge)>());
}

HalfedgeHandle prev_helper(HalfedgeHandle _heh, Bool2Type<true>) {
  // halfedge handle is stored
  return halfedge(_heh).prev_halfedge_handle();
}


HalfedgeHandle prev_helper(HalfedgeHandle _heh, Bool2Type<false>) {
  // cycle through the face to find the previous halfedge
}
```

Listing 5: Checking for certain features at compile time

the developers point of view by implementing new mesh kernels. Since most of these options can be checked for at compile–time, we do not lose efficiency by providing this generality.

5 Examples and Applications

In the following section we present some example applications that we have implemented using the OpenMesh library. These applications take strong advantage of the OpenMesh library's generic conception, e.g. the possibility to associate arbitrary attributes with the mesh primitives at compile-time. This enables flexible, custom-tailored mesh types without the run-time overhead known from other programming techniques like virtual inheritance. Although the internal structure of the OpenMesh library is quite involved, the actual usage of the library is not: Well known design patterns like adaptors, iterators and traits facilitate the access to the mesh, hence application programming effort is kept at a minimum.

As attributes are only allocated when actually needed, no memory is wasted for unused attributes. For the mere connectivity information the OpenMesh library needs to allocate 4 bytes (= one pointer) per vertex, 12 bytes (= three pointers) per halfedge and 4 bytes (= one pointer) per face. Any user-specified attributes will increase the memory requirements accordingly.

Because all attributes of a mesh are statically defined at compile time, it is not possible to add further attributes to the mesh at run-time. This functionality, however, is desireable for some applications, in particular when the attribute in question is needed only temporarily in the application's life-time. Hence we assign a *unique identifier* to each primitive which can then be used as an index into an array of attributes that is kept outside of the actual mesh data structure.

5.1 Mesh Smoothing

Because of the limitations that are inherent to a physical sampling process, meshes that are acquired by a 3D scanner usually are noisy. To remove this noise, so-called *smoothing* algorithms have been developed (see e.g. [11]). A notedly simple one is derived from minimizing the membrane energy of the mesh and is implemented by repeatedly moving vertices to the center of gravity of their neighbors. Listing 2 shows how this algorithm can be implemented in a few lines of code using only the standard functions that are provided by the OpenMesh library. Note that only the necessary attributes are attached to each vertex (in this case the vertex' position) and that no further memory is allocated for unused attributes (like the vertex' normal). Because the OpenMesh library supports constant-time (w.r.t. the total number of vertices) iteration around a vertex, the smoothing algorithm is very fast. Figure 2 shows the result of applying the smoothing algorithm to a noisy mesh.

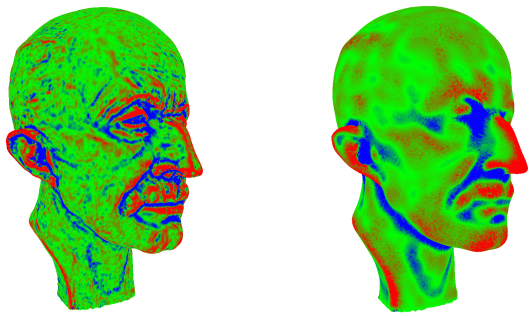


Figure 2: Left: Curvature plot of a noisy mesh acquired by a 3D scanner (400000 triangles). Right: Curvature plot of the mesh after applying 50 iterations of the smoothing algorithm. The complete smoothing took about 10 seconds on a standard PC.

5.2 Mesh Decimation

Meshes that are generated by scanning physical models typically consist of millions of triangles (see [9]). Because many downstream algorithms cannot handle this much data efficiently, *mesh decimation* techniques have been developed to reduce the number of faces of a polygonal mesh (see [8]). This is done by repeatedly removing vertices from a mesh, e.g. by collapsing edges or by removing the triangles adjacent to a vertex and retriangulating the resulting hole. Because the OpenMesh library always keeps track of the local connectivity information, these (Euler-)operations can efficiently be performed and are, in fact, already provided by the library. Using the OpenMesh's traits concept it is furthermore easy to consistently associate and update an a global approximation error, like e.g. the distance to the original mesh or the so-called error-quadric (see [6]), to each vertex. Light-weight vertex handles can then be stored in a priority-queue to schedule the decimation order. Our implementation based on the OpenMesh library uses error-quadrics as error measure and achieves a decimation rate of 20000 triangles per second (Figure 3).

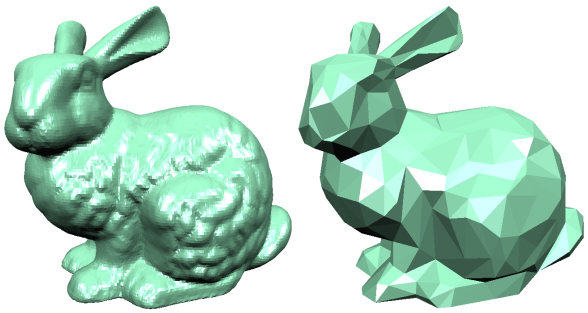


Figure 3: Left: Bunny consisting of 70000 triangles (model courtesy of Stanford University Computer Graphics Laboratory). Right: Bunny after mesh decimation using the error-quadric technique (700 triangles). The complete decimation process took less than 4 seconds.

5.3 Geometry Transmission

We have developed a scheme for robustly transmitting 3D geometry over a network (see [3]). This scheme is *robust* in the sense that a certain percentage of vertex loss due to network jamming is tolerated by the client-side reconstruction algorithm (Figure 4). The implementation uses a number of techniques to speed up and facilitate the reconstruction process. For example, a space partitioning scheme is used to efficiently locate triangles and edges in the spatial vicinity of a transmitted vertex. Using the handle concept of the OpenMesh library, the space partitioning creates only minimal memory overhead. Furthermore attributes like feature information are easily attached to each vertex or edge using the OpenMesh traits concept.

6 Conclusions and Future Work

The OpenMesh library is an efficient and versatile implementation of a halfedge data structure. Because of its generic approach it allows for advanced optimizations at compile-time and hence usually results in very fast executables. This flexibility is achieved by using involved template concepts of the C++ programming language as defined in the ISO C++ standard. Unfortunately, at the time being some C++ compilers are not ISO C++ compliant and fail to compile the OpenMesh library. As a next development step we therefore plan to provide workarounds for these compilers.

Besides that, the OpenMesh library currently provides only the halfedge data structure pe se and a few supporting components. In analogy to the STL (Standard Template Library) we plan to also provide common algorithms working on that structure, like e.g. for mesh smoothing or mesh decimation. These algorithms will also be generic and hence will provide an optimal interaction with the mesh structure.

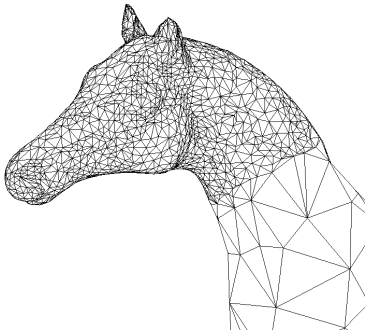


Figure 4: Local reconstruction of a horse model. Note the irregular mesh structure that requires a flexible halfedge data structure.

References

[1] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison–Wesley, 2001.

[2] B. G. Baumgart, *A Polyhedron Representation for Computer Vision*, National Computer Conference, Anaheim, CA, 1975, pp 589-596.

[3] S. Bischoff, L. Kobbelt, *Towards Robust Broadcasting of 3D Geometry Data*, to appear.

[4] S. Campagna, L. Kobbelt, H.-P. Seidel, *Directed Edges - A Scalable Representation For Triangle Meshes*, ACM Journal of Graphics Tools 3 (4), 1998.

[5] K. Czarnecki, U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison–Wesley, 2000.

[6] M. Garland, P. Heckbert, *Surface Simplification Using Quadric Error Metrics*, SIGGRAPH 97 Proceedings, 1997, pp 209–216.

[7] Lutz Kettner, *Using Generic Programming for Designing a Data Structure for Polyhedral Surfaces*, in Proc. 14th Annual ACM Symp. on Computational Geometry, 1998.

[8] L. Kobbelt, S. Campagna, H.-P. Seidel, *A general framework for mesh decimation*, Graphics Interface '98 Proceedings, 1998, pp 43–50.

[9] M. Levoy et al., *The Digital Michelangelo Project: 3D Scanning of Large Statues*, SIGGRAPH 00 Proceedings, 2000, pp 131–144.

[10] M. Botsch, S. Bischoff, *Online documentation of the OpenMesh package*, http://www-i8.informatik.rwth-aachen.de, research section.

[11] G. Taubin, *A Signal Processing Approach to Fair Surface Design*, SIGGRAPH 95 Proceedings, 1995, pp 351–358.

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

[13] D. Zorin et al., *Subdivision for Modeling and Animation*, SIGGRAPH 01 Course Notes, 2000.