Design, Implementation, and Evaluation of the `Surface_mesh` Data Structure

Daniel Sieger and Mario Botsch

Computer Graphics & Geometry Processing Group,
Bielefeld University, Germany
{dsieger,botsch}@techfak.uni-bielefeld.de

Summary. We present the design, implementation, and evaluation of an efficient and easy to use data structure for polygon surface meshes. The design choices that arise during development are systematically investigated and detailed reasons for choosing one alternative over another are given. We describe our implementation and compare it to other contemporary mesh data structures in terms of usability, computational performance, and memory consumption. Our evaluation demonstrates that our new `Surface_mesh` data structure is easier to use, offers higher performance, and consumes less memory than several state-of-the-art mesh data structures.

1 Introduction

Polygon meshes, or the more specialized triangle or quad meshes, are the standard discretization for two-manifold surfaces in 3D or solid structures in 2D. The design and implementation of mesh data structures therefore is of fundamental importance for research and development in as diverse fields as mesh generation and optimization, finite element analysis, computational geometry, computer graphics, and geometry processing.

Although the requirements on the mesh data structure vary from application to application, a generally useful and hence widely applicable data structure should be able to (i) represent vertices, edges, and triangular/quadrangular/polygonal faces, (ii) provide access to all incidence relations of these simplices, (iii) allow for modification of geometry (vertex positions) and topology (mesh connectivity), and (iv) allow to store any custom data with vertices, edges, and faces. In addition, the data structure should be easy to use, be computationally efficient, and have a low memory footprint.

Since it is hard to implement a mesh data structure that meets all these goals, many researchers and developers in both academia and industry rely on publicly available `C++` libraries like CGAL [8] (computational geometry), Mesquite [6] (mesh optimization), and OpenMesh [5] (computer graphics).

However, even these highly successful data structures have their individual deficits and limitations, as we experienced during several years of research and teaching in geometry processing. In this paper we systematically derive the design choices for our new `Surface_mesh` data structure and provide an analysis and comparison to the widely used mesh data structures of CGAL, Mesquite, and OpenMesh. These comparisons demonstrate that `Surface_mesh` is easier to use than these implementations, while at the same time being superior in terms of computational performance and memory consumption.

2 Related Work

Due to their fundamental nature, a wide variety of data structures to represent polygon meshes have been proposed. Some are highly specialized to only represent a certain type of polygons, such as triangles or quadrilateral elements. Others are designed for specific applications, e.g. parallel processing of huge data sets. In general, mesh data structures can be classified as being either *face-based* or *edge-based*. We refer the reader to [18, 4] for a more comprehensive overview of mesh data structures for geometry processing.

In its most basic form a face-based data structure consists of a list of vertices and faces, where each face stores references to its defining vertices. However, such a simple representation does not provide efficient access to adjacency information of vertices or faces. Hence, many face-based approaches additionally store the neighboring faces of each face and/or the incident faces for each vertex. Examples for face-based mesh data structures include CGAL's 2D triangulation data structure [8], Shewchuck's Triangle [23], Mesquite [6], and VCGLib [29].

In contrast to face-based approaches, edge-based data structures store the main connectivity information in edges or halfedges [2, 13, 7, 20]. In general, edges store references to incident vertices/faces as well as neighboring edges. Kettner [18] gives a comparison of edge-based data structures and describes the design of CGAL's halfedge data structure. Botsch et al. [5] introduce OpenMesh, a halfedge-based data structure widely used in computer graphics. Alumbaugh and Jiao [1] describe a compact data structure for representing surface and volume meshes by halfedges and half-faces.

Furthermore, a fairly large number of publications describe more specialized mesh representations. For instance, Blandford et al. [3] introduce a compact and efficient representation of simplicial meshes containing triangles or tetrahedra. Other works focus on data structures for non-manifold meshes [10, 11], highly compact representations of static triangle meshes [14, 15], or mesh representations and databases for numerical simulation [12, 27, 22, 9].

3 Design Decisions

While virtually all of the publications cited above describe the specific design decisions made for a particular implementation, a comprehensive and systematic investigation of the design choices available is currently lacking. We therefore try to provide such an analysis in this section.

As mentioned in the introduction, the typical design goals for mesh data structures are computational performance, low memory consumption, high flexibility and genericity, as well as ease of use. Since these criteria are partly contradicting, one has to set priorities and make certain compromises.

Based on our experience in academic research and teaching as well as in industrial cooperations, our primary design goal is *ease of use*. An easy-to-use data structure is learned faster, allows to focus on the main problem (instead of on the details of the data structure), and fosters code exchange between academic or industrial research partners. The data structure should therefore be just as flexible and generic as needed, but should otherwise be free of unnecessary switches and parameters. At the same time, however, we have to make sure not to compromise computational performance and memory consumption. Otherwise the data structure would be easy to use, but not useful, and hence would probably not be used at all.

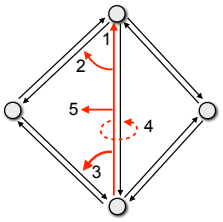
In the following we systematically analyze the typical design choices one is faced with when designing a mesh data structure. Driven by our design goals we argue for choosing one alternative over another for each individual design criterion. We start with high-level design choices and successively focus on more detailed questions.

3.1 Element Types

The most fundamental question is which types of elements or faces to support. While in computer graphics and geometry processing triangle meshes still are the predominant surface discretization [4], quad meshes are at least as important as triangle meshes for structural mechanics. For many applications, restricting to pure triangle or quad meshes is not an option, though. Polygonal finite element methods [26] decompose their simulation domain into arbitrary polygons. In discrete exterior calculus many computations are performed on the dual mesh [16]. In computational geometry, computations on Voronoi diagrams also need arbitrary polygon meshes [8]. Since we want our data structure to be suitable for an as wide as possible range of applications we choose to support *arbitrary polygonal elements*.

3.2 Connectivity Representation

As discussed in Section 2 there are two ways to represent the connectivity of a polygon mesh: a face-based or an edge-based representation.



1. Target vertex
2. Next halfedge
3. Previous halfedge
4. Opposite halfedge
5. Adjacent face

Fig. 1. Connectivity relations within a halfedge data structure.

Face-based data structures store for each face the references to its defining vertices. While this is sufficient for, e.g., visualization or setting up a stiffness matrix, it is inefficient for mesh optimization, since vertex neighborhoods cannot be accessed easily. Some implementations therefore additionally store all incident faces per vertex (e.g., [6, 29]), but even then it is still inefficient to enumerate all incident *vertices* of a center vertex—a query frequently required for many algorithms, such as mesh smoothing, decimation, or remeshing. Furthermore, since for a general polygon mesh the number of vertices per face and the number of incident faces per vertex are not constant, they have to be stored using dynamically allocated arrays or lists, which further complicates the data structure. Edges are typically not represented at all.

In contrast, storing the main connectivity information in terms of edges or halfedges naturally handles arbitrary polygon meshes. The data types for vertices, (half-)edges, and faces all have constant size. The vertices and face-neighbors of a face can be efficiently enumerated, as well as the vertices or faces incident to a center vertex. Attaching additional data to vertices, halfedges, and faces is simple, since all entities are explicitly represented. Finally, a halfedge-based data structure allows for simple and efficient implementation of connectivity modifications as required by modern approaches to interleaving mesh generation and optimization [28, 25] or simulation [30]. We therefore choose a *halfedge data structure* to store the connectivity of a polygon mesh. The basic connectivity relations within a typical halfedge data structure are shown in Figure 1.

3.3 Storage

On an implementation level one has to decide whether to store the mesh entities in either doubly-linked lists or simple arrays.

Lists have the advantage that they allow for easy removal of individual vertices, edges, or faces, which is required, e.g., when collapsing edges or removing vertices in a mesh decimation algorithm. However, this flexibility comes at the price of higher memory consumption and less coherent memory layout compared to array-based storage, both resulting in considerable performance loss. We evaluated this on the halfedge data structure [17] of CGAL [8], which allows to switch between a list-based and an array-based implementation. Our

benchmarks in Section 5 show the list-based implementation to be up to twice as slow as the array-based version.

Array-based storage on the one hand is more compact and faster, but on the other hand the removal of mesh entities is more difficult. Typically mesh entities are first marked as deleted and later removed by some form of garbage collection. However, the advantages in terms of performance and memory consumption clearly outweigh the additional effort needed to support removal. For these reasons we choose an *array-based storage* scheme.

3.4 Entity References

When using array-based storage for mesh entities, references (or handles) to entities can be represented either as pointers or indices.

Pointers have three important drawbacks: First, they become invalid upon a relocation of the array, which happens if the array has to allocate more memory (e.g., for refinement or subdivision algorithms). While the data structure can automatically update all *internally* stored pointers, references that are stored externally by the user will inevitably become invalid. Second, on 64-bit architectures pointers consume twice as much memory as 32-bit indices. For larger meshes, however, one has to use 64-bit addressing, since complex meshes easily exceed the 2GB limit for 32-bit architectures. Finally, pointers cannot be used to access additional properties of mesh entities that are stored in additionally "property arrays" (see the next section). We therefore choose *indices* as entity references.

3.5 Custom Properties

Additional information about the mesh entities can be stored either by extending the mesh entities themselves or by using additional arrays. For instance, vertex normals can be incorporated either by adding a member variable `normal` to the class `Vertex`, or by having an additional array `vertex_normals` where the `i`'th entry is the normal of vertex `i`.

The first approach, as e.g. chosen by CGAL, is more elegant from an object-oriented point of view, but has the following drawbacks: Since the class types of mesh entities are extended at compile-time, all custom properties are allocated over the whole running time of the application, even if the properties are used for a short time only. This does not only waste memory, it also slows down the algorithms due to a less compact memory layout: Just adding vertex and face normals to the CGAL mesh by extending the `Vertex` and `Facet` types slowed down our benchmarks (Section 5) by about 25% on average. This can be a significant drawback for larger mesh processing applications, where many individual algorithms need some custom data at some point in time.

In contrast, additional arrays can be dynamically allocated at run-time, such that custom properties are just allocated when needed and deleted afterwards (as implemented in OpenMesh and Mesquite). Keeping all property

arrays synchronized upon resize and swap operations can easily be implemented. Furthermore, computations on the property arrays are also more cache-friendly, thereby increasing performance compared to extended mesh entities. Finally, if the model is meant to be visualized in an interactive application, property arrays can also be used in conjunction with OpenGL vertex arrays (normals, colors, texture coordinates), which speeds up rendering performance considerably. We therefore store custom properties in additional *synchronized arrays*.

3.6 Ease of Use

Up to this point, our previous mesh data structure OpenMesh [5], at least in its current version [21], follows most of the design decisions made so far. From our experience in research and teaching, however, the level of genericity offered by OpenMesh is not needed in practice. For instance, custom properties can be allocated both by extending mesh entities as well as using additional arrays, where due to the former the mesh entities (and hence the whole mesh) become template classes. Furthermore, the large (template-parametrized) inheritance hierarchy makes the code unnecessarily hard to document and understand. In terms of `C++` sophistication, the polyhedral data structure of CGAL requires an even higher level of template expertise, which makes it hard to use this data structure with students or inexperienced programmers, too.

To reduce the negative effect that heavy use of templates and complicated inheritance hierarchies have on the ease of use of the data structure, we made our design *as simple as possible* while maintaining maximum applicability.

4 Implementation

In the following we highlight the most important aspects of our implementation. We first describe the fundamental organization of our new data structure and successively proceed to higher-level functionality.

Since OpenMesh already satisfies all design choices except simplicity, we started our implementation from a massively stripped-down and simplified version of OpenMesh. In contrast to other implementations, ours is concentrated within a single class, namely `Surface_mesh`. While the core of `Surface_mesh` (without file I/O) is implemented in three files using about 2250 lines of code, the part of OpenMesh that implements the same functionality requires 41 files or 8400 lines of code. In contrast to CGAL and OpenMesh, `Surface_mesh` is not a class template, i.e., it does not require so-called traits classes as template parameters. However, the fundamental types `Scalar` and `Point` can still be defined by simple `typedef`s.

`Surface_mesh` implements an array-based halfedge data structure. The basic entities of the mesh, i.e., vertices, (half-)edges, and faces are represented by the types `Vertex`, `Halfedge`, `Edge`, and `Face`, respectively, all of which are

```
Surface_mesh mesh;

// allocate property storing a point per edge
Surface_mesh::Edge_property<Point> edge_points
  = mesh.add_edge_property<Point>("property-name");

// access the edge property like an array
Surface_mesh::Edge e;
edge_points[e] = Point(x,y,z);

// remove property and free memory
mesh.remove_edge_property(edge_points);
```

Listing 1. Working with a custom edge property.

basically 32-bit indices. Edges are represented implicitly, since two opposite halfedges (laid out consecutively in memory) build an edge.

The connectivity information is stored in form of custom properties (i.e., synchronized arrays) of vertices, faces, and halfedges: Each vertex stores an outgoing halfedge, each face an incident halfedge. Each halfedge stores its incident face, its target vertex, and its previous and next halfedges within the face. Since opposite halfedges are laid out consecutively in memory, the opposite halfedge can be accessed by simple modulo operations on the `Halfedge` indices and therefore does not have to be stored explicitly.

Managing internal mesh data as well as dynamically allocated user-defined properties within the same framework for synchronized arrays on the one hand simplifies implementing and maintaining the data structure. On the other hand the performance of the data structure then crucially depends on efficient access to these properties. Our property mechanism deviates from Mesquite and OpenMesh in that it (i) avoids inefficient virtual function calls, (ii) does not require error-prone casting of `void`-pointers, (iii) avoids unnecessary indirections, and (iv) offers a cleaner interface. From a user's point of view, working with a custom property is as simple as shown in Listing 1.

In addition to access to all incidence relations and custom properties, `Surface_mesh` also offers higher-level topological operations, such as adding vertices and faces, performing edge flips, edge splits, face splits, or halfedge collapses. Based on these methods typical geometry processing algorithms (smoothing, decimation, subdivision, remeshing) can be implemented conveniently. Since `Surface_mesh` uses an array-based storage special care has to be taken when removing items from the mesh. Such operations do not delete mesh entities immediately, but instead mark them as being to be deleted. The function `garbage_collection()` eventually deletes those items from the arrays, while preserving the integrity of the data structure.

In order to sequentially access mesh entities we provide iterators for each entity type, namely `Vertex_iterator`, `Halfedge_iterator`, `Edge_iterator` and `Face_iterator`. Each iterator stores a reference to the current entity and to the mesh. The latter is used to automatically detect and skip deleted

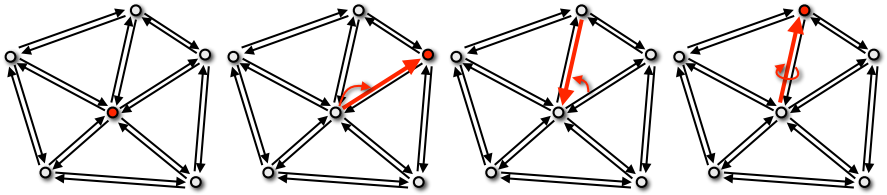


Fig. 2. Traversal of one-ring neighbors of a center vertex. From left to right: 1. Start from center vertex. 2. Select outgoing halfedge (and access its target vertex). 3. Move to previous halfedge. 4. Move to opposite halfedge (access its target vertex). Steps 1 and 2 correspond to the initialization of a `Vertex_around_vertex_circulator` using `mesh.vertices(Vertex)`, steps 3 and 4 to the `++`-operator of the circulator.

entities, for instance when the user collapsed some edges but did not yet clean-up using `garbage_collection()`. We decided for these "safe iterators" despite their small performance penalty, since "unsafe" iterators turned out to be a frequent source of errors for novice OpenMesh users.

Similar to iterators, we also provide circulators for the ordered enumeration of all incident vertices, halfedges, or faces around a given face or vertex. Since there is no clear begin- and end-circulator, we follow the CGAL convention and use `do-while` loops for circulators. The traversal of the one-ring neighborhood of a vertex—which corresponds to a `Vertex_around_vertex_circulator`—is shown in Figure 2. An example usage of iterators and circulators is demonstrated in the smoothing example in Listing 2.

5 Evaluation and Comparison

In this section we evaluate our mesh data structure and compare it to three other widely used data structures: OpenMesh, CGAL, and Mesquite. Our evaluation criteria are ease of use, run-time performance, and memory usage.

All tests were performed on a Dell T7500 workstation with an Intel Xeon E5645 2.4 GHz CPU and 6GB RAM running Ubuntu Linux 10.04 x86_64. All libraries and tests were compiled with `gcc` version 4.4.3, optimization turned on (using `-O3`) and debugging checks disabled (`-DNDEBUG`).

For each of the mesh libraries in our comparison we used the latest version available, i.e., OpenMesh 2.0.1, CGAL 3.8, and Mesquite 2.1.4. To achieve comparable results, we chose double-precision floating point values for scalars, vertex coordinates, and normal vectors for all benchmarks and data structures. Since one benchmark requires vertex and face normals, all data structures allocate these properties, either by extending vertex and face types (CGAL) or using property arrays (Mesquite, OpenMesh, `Surface_mesh`).

Note that regarding CGAL we compare to both the list-based and the vector-based version of the `Polyhedron_3` mesh data structure, denoted as CGAL_list and CGAL_vector, respectively. Furthermore, following [24], we

```
#include <Surface_mesh.h>

int main(int argc, char** argv)
{
    Surface_mesh mesh;

    // read mesh from file
    mesh.read(argv[1]);

    // get (pre-defined) property storing vertex positions
    Surface_mesh::Vertex_property<Point> points
      = mesh.get_vertex_property<Point>("v:point");

    // iterators and circulators
    Surface_mesh::Vertex_iterator vit, vend = mesh.vertices_end();
    Surface_mesh::Vertex_around_vertex_circulator vc, vc_end;

    // loop over all vertices
    for (vit = mesh.vertices_begin(); vit != vend; ++vit)
    {
        if (!mesh.is_boundary(*vit))
        {
            // move vertex to barycenter of its neighbors
            Point  p(0,0,0);
            Scalar c(0);
            vc = vc_end = mesh.vertices(*vit);
            do
            {
                p += points[*vc];
                ++c;
            }
            while (++vc != vc_end);
            points[*vit] = p / c;
        }
    }

    // write mesh to file
    mesh.write(argv[2]);
}
```

Listing 2. A simple smoothing program implemented using `Surface_mesh`.

removed the storage for the plane equation from face entities in order to increase performance.

In contrast to CGAL, OpenMesh, and `Surface_mesh`, which are all halfedge data structures, Mesquite employs a face-based data structure that stores both downward adjacency (vertices of a face) and upward adjacency (all incident faces of a vertex).

5.1 Ease of Use

Being our primary design goal, we begin our evaluation by comparing the ease of use of `Surface_mesh` to the other libraries.

Simplicity

As already outlined in Section 3.6, simplicity is a key criterion for the ease of use of a software library. By design, `Surface_mesh` is as simple as possible

```
typedef CGAL::Simple_cartesian<double>  Kernel;
typedef Kernel::Point_3                 Point_3;

template <class Refs>
struct My_halfedge : public CGAL::HalfedgeDS_halfedge_base<Refs>
{
    Point_3 halfedge_point;
};

class Items : public CGAL::Polyhedron_items_3
{
public:
    template <class Refs, class Traits>
    struct Halfedge_wrapper
    {
        typedef My_halfedge<Refs> Halfedge;
    };
};

typedef CGAL::Polyhedron_3<Kernel, Items>  Mesh;
```

Listing 3. Declaring a custom halfedge property in CGAL.

while maintaining high applicability. In contrast, both OpenMesh and CGAL offer a higher level of genericity. While this enables the customization of the mesh data structure for specialized applications, it also makes the library less accessible for students and inexperienced programmers.

The differences in complexity are demonstrated best by example. Listing 3 shows how to declare a custom halfedge property in CGAL, which is roughly equivalent to Listing 1 showing the usage of properties in `Surface_mesh`.

Compared to OpenMesh, our increased simplicity (and decreased genericity) is due to the definition of basic types (e.g., use `float` or `double` as scalar type, 2D or 3D vertex coordinates) through `typedef`s instead of through template parameters. While this allows `Surface_mesh` not to be a class template, it restricts each application to use a single `Surface_mesh` definition. In contrast, OpenMesh and CGAL allow for several custom-tailored template instances in a single application.

Properties

Comparing Listings 1 and 3 not only serves as an example for evaluating simplicity, but also demonstrates the differences between CGAL's extended entities and and `Surface_mesh`'s synchronized arrays for property handling. While the declaration of the former is rather involved and bound to compile-time properties, the latter is easy to use and dynamically allocated at run-time. Both OpenMesh and Mesquite also support dynamic property arrays. In case of OpenMesh however, the interface is slightly more complicated. Mesquite's implementation of properties relies on casting `void`-pointers, a practice generally discouraged and also relevant to our next evaluation criterion.

Mesquite	1.57
CGAL_list	2.83
CGAL_vector	2.75
OpenMesh	3.37
Surface_mesh	1.13

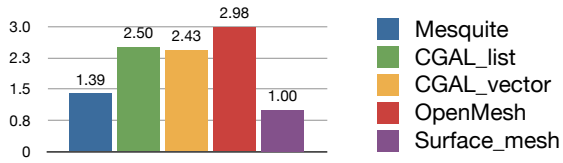


Table 1. Compilation times (in seconds) of our benchmark program for the different mesh data structures. The chart shows timings relative to `Surface_mesh`.

Safety

Especially for inexperienced programmers protection against common sources of errors is a crucial aspect of usability. The use of `void` pointers in Mesquite mentioned above can be considered harmful in this context, since this practice essentially circumvents the static type-safety of the programming language. The use of pointers as entity references for CGAL's array-based mesh data structure is prone to errors, since the pointers (and iterators) become invalid upon resizing. While OpenMesh uses safe, index-based entity references, its iterators by default do not skip deleted items, which turned out to be a common source of errors. In contrast, `Surface_mesh`'s implementation of safe iterators protects the user from iterating over deleted entities.

Compilation Time

Finally, compilation time is a usability factor frequently overlooked. While the times to compile the individual programs in our test suite are relatively short, compilation time becomes a significant factor for the speed and efficiency of the development process in more complex projects. As can be seen from Table 5.1, `Surface_mesh` offers the fastest compilation times, mostly due to minimizing the use of templates.

User Study

We evaluated the usability of `Surface_mesh` in a user study among the participants of a two-day course of mesh processing (involving lectures and programming exercises) held at the Symposium on Geometry Processing 2011. The attendees had a varying degree of programming experience and exposure to other mesh libraries. After the two-days the participants were asked anonymously if `Surface_mesh` was easy to use and understand for them. Out of 18 participants seven strongly agreed to this statement (5/5 points), another seven agreed (4/5 points). On average, `Surface_mesh` received 4.1/5 points. While this is not a representative survey, the results are still encouraging.

5.2 Performance

In order to compare the efficiency of our implementation with other mesh data structures we designed several benchmarks, which either evaluate a fundamental functionality of a data structure (e.g., iterators or adjacency queries) or test the performance in common application domains (e.g., mesh smoothing or subdivision). The benchmark tests are described below and their pseudo-code is shown in Algorithms 1–6:

1. Circulator Test: For each vertex enumerate its incident faces. For each face enumerate its vertices. This test measures the efficiency of iterators and circulators.
2. Barycenter Test: Center the mesh at the origin by first computing the barycenter of all vertex positions and then subtracting it from each vertex. This test evaluates the performance of iterators and of the access to and basic computations on the vertex coordinates.
3. Normal Test: First compute (and store) face normals, then compute vertex normals as the average of the incident faces' normals. This test measures the performance of iterators, circulators, vertex computations, and custom properties (storing face and vertex normals).
4. Smoothing Test: Perform Laplacian smoothing by moving each (non-boundary) vertex to the barycenter of its neighboring vertices. This test requires (and evaluates) the enumeration of incident vertices of a vertex.
5. Subdivision Test: Perform one step of $\sqrt{3}$-subdivision [19] by first splitting all faces at their centers, smoothing the old vertices, and then flipping all the old edges. This test mainly evaluates the performance of the face split and edge flip operators.
6. Edge Collapse Test: First split all faces at their center and then collapse each newly introduced vertex into one of its (old) neighbors, thereby restoring the original connectivity. This test evaluates the operators face split and halfedge collapse.

These benchmarks were performed on the Imp model, consisting of 300k vertices and 600k triangles, and the Dual Dragon model, a dualized triangle mesh consisting of 100k vertices and 50k polygonal faces. The models are shown in Figure 3. All tests were iterated sufficiently many times in order to get more reliable accumulated timings. The results are listed in Tables 2 and 3. Note that we also performed the tests with other models and setups (CPU, compiler version, and operating system). While the results quantitatively vary to a certain extent, they were qualitatively equivalent to the ones shown here.

It can be observed that for some tests the performance varies significantly between different libraries. While it is hard to track down the reasons in detail, we point out the most important issues we identified.

For Mesquite, a significant performance penalty comes from the large number of virtual functions (e.g., to access incidences or vertex coordinates), as well as from memory fragmentation due to dynamically allocated arrays for

Fig. 3. The three models used in the evaluation.

Algorithm 1: Circulator Test

```
Initialize counter = 0;
for each vertex v do
    for each face f incident to v do
        counter = counter + 1;
    end
end
for each face f do
    for each vertex v incident to f do
        counter = counter − 1;
    end
end
```

Algorithm 2: Barycenter Test

```
Initialize p = (0, 0, 0);
for each vertex v do
    p = p + point(v);
end
p = p/number_of_vertices();
for each vertex v do
    point(v) = point(v) − p;
end
```

Algorithm 3: Normal Test

```
for each face f do
    Compute the face normal of f;
end
for each vertex v do
    n = (0, 0, 0);
    for each face f incident to v do
        n = n + face_normal(f);
    end
    vertex_normal(v) = normalize(n);
end
```

Algorithm 4: Smoothing Test

```
for each vertex v do
    if v is not a boundary vertex then
        p = (0, 0, 0);
        c = 0;
        for each vertex w incident to
        v do
            p = p + point(w);
            c = c + 1;
        end
        point(v) = p/c;
    end
end
```

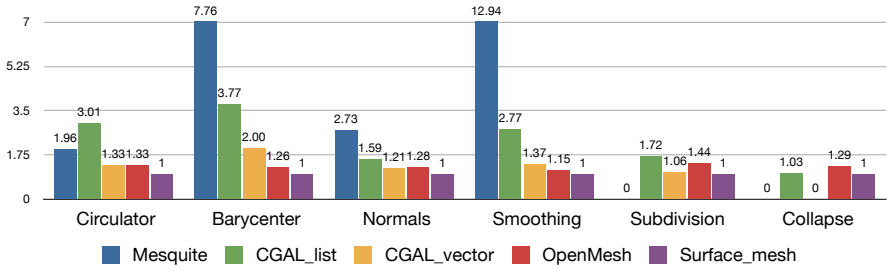
Algorithm 5: Subdivision Test

```
for each face f do
    Compute centroid c;
    Split f at centroid c;
end
for each old vertex v do
    Smooth vertex position;
end
for each old edge e do
    Flip e;
end
```

Algorithm 6: Collapse Test

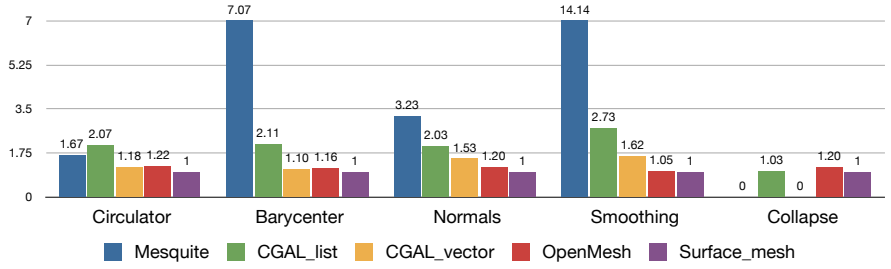
```
for each face f do
    Split f;
end
for each new vertex v do
    Collapse v into one of its neighbors;
end
```

Fig. 4. The six benchmark tests used to evaluate and compare the run-time performance of `Surface_mesh` to Mesquite, CGAL, and OpenMesh.



	Circulator	Barycenter	Normals	Smoothing	Subdivision	Collapse
Mesquite	3479.57	15039.9	11406.4	23228.9	—	—
CGAL_list	5329.89	7298.91	6642.29	4976.79	506.158	1582.94
CGAL_vector	2358.51	3879.86	5064.38	2467.66	312.607	—
OpenMesh	2359.36	2443.59	5356.68	2071.79	423.925	1987.44
Surface_mesh	1673.34	1412.28	4181.92	1757.07	294.24	1547.53

Table 2. Timings for performing Algorithms 1–6 on the Imp model of 300k vertices and 600k triangles. The table lists timings in milliseconds, the chart visualizes the performance relative to `Surface_mesh`.



	Circulator	Barycenter	Normals	Smoothing	Collapse
Mesquite	650.47	4632.11	2234.09	5554.18	—
CGAL_list	804.118	1381.48	1403.99	1070.94	74.376
CGAL_vector	460.168	718.312	1057.46	636.868	—
OpenMesh	475.175	760.832	830.638	410.626	86.358
Surface_mesh	388.47	655.341	690.949	392.901	71.888

Table 3. Timings for performing Algorithms 1–4 and 6 on the Dual Dragon model consisting of 100k vertices and 50k arbitrary polygonal faces. The table lists timings in milliseconds, the chart visualizes the performance relative to `Surface_mesh`.

	Imp	Dragon	Lucy
Mesquite	88M	16M	2.8G
CGAL_list	172M	30M	5.5G
CGAL_vector	105M	19M	3.4G
OpenMesh	67M	14M	2.2G
Surface_mesh	60M	12M	1.9G

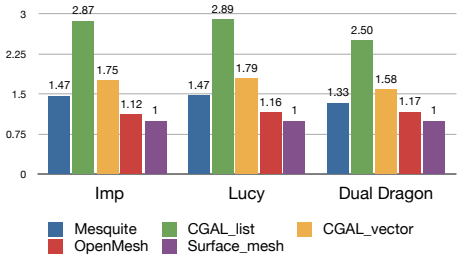


Table 4. Memory usage for the Imp, Lucy, and Dual Dragon models. The table lists resident size memory usage after reading the meshes, without performing any further tests or processing. The chart visualizes the relative difference to `Surface_mesh`.

storing per-vertex and per-face incidences. Moreover, enumerating incident *vertices* of a center vertex is not directly supported by this face-based data structure and therefore has to be implemented less efficiently by looping over the vertices of the incident faces. Since Mesquite does not support connectivity modifications, the subdivision and collapse test were not implemented.

The performance difference between CGAL_list and CGAL_vector is due to the higher memory consumption and memory fragmentation of the list-based version. Both CGAL mesh data structures store 64-bit references, vertex positions, and normal vectors in extended mesh entities, leading to a less compact memory layout, which in turn results in performance penalties. Note that the array-based version does not support removal of entities, so that the collapse test could be implemented with the slower list-based version only.

Since OpenMesh is closest to `Surface_mesh` in terms of design and implementation, it also is close in terms of performance. The differences of about 20%–30% are due to our more efficient mechanism for accessing custom properties, which requires fewer indirections. Furthermore, our `do`–`while` circulators are slightly more efficient than the `for` circulators of OpenMesh, which use a rather complex test for detecting the end of the loop.

The results clearly demonstrate the performance of `Surface_mesh` to be (in most cases) superior to or at least on par with the other data structures.

5.3 Memory Efficiency

Besides run-time performance, memory consumption is a key criterion to measure the efficiency of a library, especially when it comes to applications dealing with highly complex data sets. We compare the memory consumption of the data structures on three different models: the Imp model (300k vertices, 600k triangles) and the Dual Dragon (100k vertices, 50k polygons) already used in the performance comparison and the complex Lucy model (10M vertices, 20M triangles). The results are shown in Table 4.

Although a face-based data structure in general consumes less memory than a halfedge data structure, Mesquite requires more memory than

`Surface_mesh` because (i) of the overhead of the dynamic arrays used to store incidences, (ii) the use of 64-bit references, and (iii) the storage of several helper data per face and vertex.

In addition to the memory overhead due to the doubly-linked of the CGAL list-kernel, both CGAL data structures use 64-bit pointers as references, which consume twice as much memory than the 32-bit indices employed by OpenMesh and `Surface_mesh`.

Our slight performance advantage with respect to OpenMesh comes from the different storage of the information whether a vertex, edge, or face is deleted. We store this information in custom `bool` property arrays, which in a `std::vector<bool>` require approximately 1 bit per entity. In contrast, OpenMesh uses one status byte per entity, similar to Mesquite.

These results show that `Surface_mesh` is superior to the other data structures in terms of memory consumption.

6 Conclusion and Future Work

Our results show that the design decisions made during the development of a mesh data structure have a crucial impact on both the usability and the efficiency of the library. By systematically analyzing the design questions we derived design decisions that—if carefully implemented—result in a mesh data structure that is more usable, offers higher performance and consumes less memory than several other mesh data structures publicly available.

Considering the sometimes drastic differences in performance and memory consumption between the individual libraries, it is important to keep in mind that some of them have originally been designed and implemented with a strong focus on a given application domain, such as computational geometry in case of CGAL and mesh optimization in case of Mesquite. As a consequence, both libraries provide significantly more functionality that goes beyond a pure surface mesh data structure. For example, Mesquite supports the optimization of surface and volume meshes within a single framework.

While we are confident with the tests and results achieved thus far, we feel that our benchmark tests should be expanded to a wider variety of different setups (i.e. different hardware, operating systems, compilers and mesh models). Furthermore, additional algorithms and additional mesh data structures, for instance VCGLib, could be included in future evaluations.

Our performance and memory benchmarks can be a first step towards a general benchmark for mesh data structures. We will therefore make the source code and the results of the benchmarks publicly available. Furthermore, in order to facilitate wide adoption of our new data structure, we will also make `Surface_mesh` freely available under an Open Source license allowing for both academic and commercial usage.

While our current work is focused on surface meshes only, we are aware that applications such as physical simulations often require volumetric meshes. We feel that a systematic approach as presented in this paper might also be beneficial for the design and implementation of a volumetric mesh data structure. In particular, design decisions such as array-based storage, indices as entity references and custom properties as synchronized arrays should carry over to such a data structure seamlessly.

Acknowledgments

The authors are grateful to Pierre Alliez and Christian Rössl for helpful discussions and to the participants of the SGP 2011 graduate school for their valuable feedback. Daniel Sieger and Mario Botsch are supported by the Deutsche Forschungsgemeinschaft (Center of Excellence in "Cognitive Interaction Technology", CITEC). The Lucy and Dragon models are courtesy of the Stanford University Computer Graphics Laboratory.

References

1. T. Alumbaugh and X. Jiao. Compact array-based mesh data structures. In *Proceedings of the 14th International Meshing Roundtable*, pages 485–504, 2005.
2. B. G. Baumgart. Winged-edge polyhedron representation. Technical Report STAN-CS320, Computer Science Department, Stanford University, 1972.
3. D. Blandford, G. Blelloch, D. Cardoze, and C. Kadow. Compact representations of simplicial meshes in two and three dimensions. In *Proceedings of the 12th International Meshing Roundtable*, pages 135–146, 2003.
4. M. Botsch, L. Kobbelt, M. Pauly, P. Alliez, and B. Levy. *Polygon Mesh Processing*. AK Peters, 2010.
5. M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt. Openmesh: A generic and efficient polygon mesh data structure. In *Proc. of OpenSG Symposium*, 2002.
6. M. Brewer, L. Freitag Diachin, P. Knupp, T. Leurent, and D. Melander. The Mesquite mesh quality improvement toolkit. In *Proceedings of the 12th International Meshing Roundtable*, pages 239–250, 2003.
7. S. Campagna, L. Kobbelt, and H.-P. Seidel. Directed edges: A scalable representation for triangle meshes. *Journal of Graphics, GPU, and Game Tools*, 3(4):1–12, 1998.
8. CGAL. Computational Geometry Algorithms Library. `http://www.cgal.org`, 2011.
9. H. C. Edwards, A. B. Williams, G. D. Sjaardema, D. G. Baur, and W. K. Cochran. SIERRA toolkit computational mesh conceptual model. Technical Report SAND2010-1192, Sandia National Laboratories, 2010.
10. L. De Floriani and A. Hui. Data structures for simplicial complexes: An analysis and a comparison. In *Proc. of Eurographics Symposium on Geometry Processing*, pages 119–28, Berlin, 2005.
11. L. De Floriani, A. Hui, D. Panozzo, and D. Canino. A dimension-independent data structure for simplicial complexes. In *Proceedings of the 19th International Meshing Roundtable*, pages 403–420, 2010.

12. R. Garimella. MSTK - a flexible infrastructure library for developing mesh based applications. In *Proceedings of the 13th International Meshing Roundtable*, pages 203–212, 2004.
13. L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and computation of Voronoi diagrams. *ACM Transaction on Graphics*, 4(2):74–123, 1985.
14. T. Gurung, D. Laney, P. Lindstrom, and J. Rossignac. SQuad: Compact representation for triangle meshes. *Computer Graphics Forum*, 30(355–364), 2011.
15. T. Gurung, M. Luffel, P. Lindstrom, and J. Rossignac. LR: Compact connectivity representation for triangle meshes. *ACM Trans. Graph.*, 30(3), 2011.
16. A. N. Hirani. *Discrete Exterior Calculus*. PhD thesis, California Institute of Technology, 2003.
17. L. Kettner. Designing a data structure for polyhedral surfaces. In *Proceedings of 14th Symposium on Computational Geometry*, pages 146–154, 1998.
18. L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry – Theory and Applications*, 13(1):65–90, 1999.
19. L. Kobbelt. $\sqrt{3}$ subdivision. In *Proceedings of ACM SIGGRAPH 2000*, pages 103–112, 2000.
20. M. Mantyla. *An Introduction to Solid Modeling*. Computer Science Press, New York, 1988.
21. OpenMesh. `http://www.openmesh.org`, 2011.
22. E. Seegyoung Seol and M. S. Shephard. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Engineering with Computers*, 22(3):197–213, 2006.
23. J. R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148, pages 203–222. 1996.
24. Le-Jeng Shiue, Pierre Alliez, Radu Ursu, and Lutz Kettner. A tutorial on CGAL Polyhedron for subdivision algorithms. In *Symp. on Geometry Processing Course Notes*, 2004.
25. D. Sieger, P. Alliez, and M. Botsch. Optimizing Voronoi diagrams for polygonal finite element computations. In *Proceedings of the 19th International Meshing Roundtable*, pages 335–350, 2010.
26. N. Sukumar and E. A. Malsch. Recent advances in the construction of polygonal finite element interpolants. *Archives of Computational Methods in Engineering*, 13(1):129–163, 2006.
27. T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst. MOAB: A mesh-oriented database. Technical Report SAND2004-1592, Sandia National Laboratories, 2004.
28. J. Tournois, P. Alliez, and O. Devillers. Interleaving Delaunay refinement and optimization for 2D triangle mesh generation. In *Proceedings of the 16th International Meshing Roundtable*, pages 83–101, 2007.
29. VCGLib. `http://vcg.sourceforge.net/`, 2011.
30. M. Wicke, D. Ritchie, B. M. Klingner, S. Burke, J. R. Shewchuk, and J. F. O'Brien. Dynamic local remeshing for elastoplastic simulation. *ACM Transaction on Graphics*, 29:49:1–49:11, 2010.