OpenVolumeMesh - A Versatile Index-Based Data Structure for 3D Polytopal Complexes

Michael Kremer[1], David Bommes[2], and Leif Kobbelt[1]

[1] Computer Graphics Group, RWTH Aachen University, Germany. {mkremer,kobbelt}@cs.rwth-aachen.de
[2] INRIA Sophia Antipolis - Méditerranée, France. david.bommes@inria.fr

Summary. We present a data structure which is able to represent heterogeneous 3-dimensional polytopal cell complexes and is general enough to also represent non-manifolds without incurring undue overhead. Extending the idea of half-edge based data structures for two-manifold surface meshes, all faces, i.e. the two-dimensional entities of a mesh, are represented by a pair of oriented *half-faces*. The concept of using directed half-entities enables inducing an orientation to the meshes in an intuitive and easy to use manner.

We pursue the idea of encoding connectivity by storing first-order top-down incidence relations per entity, i.e. for each entity of dimension d, a list of links to the respective incident entities of dimension $d-1$ is stored. For instance, each half-face as well as its orientation is uniquely determined by a tuple of links to its incident half-edges or each 3D cell by the set of incident half-faces. This representation allows for handling non-manifolds as well as mixed-dimensional mesh configurations. No entity is duplicated according to its valence, instead, it is shared by all incident entities in order to reduce memory consumption. Furthermore, an array-based storage layout is used in combination with direct index-based access. This guarantees constant access time to the entities of a mesh.

Although bottom-up incidence relations are implied by the top-down incidences, our data structure provides the option to explicitly generate and cache them in a transparent manner. This allows for accelerated navigation in the local neighborhood of an entity.

We provide an open-source and platform-independent implementation of the proposed data structure written in C++ using dynamic typing paradigms. The library is equipped with a set of STL compliant iterators, a generic property system to dynamically attach properties to all entities at run-time, and a serializer/deserializer supporting a simple file format. Due to its similarity to the OpenMesh data structure, it is easy to use, in particular for those familiar with OpenMesh. Since the presented data structure is compact, intuitive, and efficient, it is suitable for a variety of applications, such as meshing, visualization, and numerical analysis. OpenVolumeMesh is open-source software licensed under the terms of the LGPL.

1 Introduction

Most techniques in computational engineering sciences are based on discretizations of the underlying domain (e.g. 2D or 3D) in terms of meshes consisting of polytopal elements. Depending on the application, the data structures used to handle these meshes have to meet various requirements. Several data structures have been proposed for various applications that significantly differ in the way they encode adjacency and/or incidence relations between the entities of a mesh.

The extent to which navigation on the mesh is possible along with its access time essentially depends on the amount of encoded incidence relations. In many applications it is oftentimes sufficient to solely encode a subset of all possible local incidence/adjacency relations. Mesh generation, for instance, requires the data structure to be flexible in terms of local modifications to the topology while providing means for efficiently navigating on it using local incidence information. In this case, the most suitable data structure is certainly a trade-off between fast local navigation by storing extra incidence information on one hand and keeping the extra amount of storage space needed for these additional information as small as possible on the other hand. When performing finite element analysis on a mesh, most of this additional incidence information is not needed anymore since in most cases it suffices to address the nodes or cells of a mesh only leading to the need for data structures that rather fulfill the requirement to be as compact as possible. Most of the data models proposed in earlier work come with a set of serious limitations in favor of being optimized for the demands of particular application fields. These limitations may comprise the restriction to homogeneous meshes, i.e. meshes only consisting of one type of element (e.g. triangles, quadrilaterals, tetrahedra, hexahedra, etc.), or the restriction to manifold configurations only. As a consequence, many engineering laboratories and researchers develop their own data structures respecting their individual needs but clearly lacking universal applicability.

1.1 Our Contribution

In this work, we propose a data structure for polytopal meshes that is efficient and easy to use regarding local modification operations while being memory efficient.

The basic concept of the presented approach adopts the idea of *half-edge* based surface mesh representations as proposed by Mäntylä [23] (also see [4]) and carries it over to 3-dimensional meshes, leading to the notion of *half-faces* (also described in [20]). For this, each of the faces, the 2-dimensional entities, is represented by a pair of oriented half-faces.

The concepts of this mesh representation are simple and easy to use while still being suitable for various application fields. In the presented data structure, incidence/adjacency relations are not encoded in terms of fixed sets of pointers to neighboring entities per entity as this restricts the class of representable meshes to manifolds only. Instead, an array-based approach is used to store *top-down* and optionally *bottom-up* incidence relations (as in the Incidence Graph data structure described in [12]) that allows for representing non-manifold and even mixed-dimensional meshes. Only storing one half of each paired half-entity in combination with the use of indices instead of pointers to reference entities additionally reduces the amount of consumed memory. For instance, in our test setting, the memory footprint of a hexahedron in a regular hexahedral mesh (cf. Sec. 5) is approximately 254 byte with bottom-up incidence relations and 133 bytes without.

We offer a complete open-source implementation of the presented data structure. Additionally, the framework is equipped with a set of useful tools such as a dynamic property system that can be used to attach generic properties to the entities of a mesh at run-time, a file reader/writer implementation, and means to keep a mesh consistent after the deletion of entities. Since the handling of the data structure is similar to OpenMesh, a half-edge based data structure for two-manifold surface meshes introduced by Botsch et al. [3], we name the proposed data structure *OpenVolumeMesh*.

2 Related Work

Data structures for special purposes, such as meshing or finite element analysis, have been subject to research for many years. Therefore, a variety of concepts aiming at different applications has been proposed in the last decades. In this section we are going to provide a brief overview of recent work that has been done in this field.

A detailed overview and comparison of available data structures for simplicial and cell complexes is given by De Floriani et al. in [14] and [15]. Some of the data structures for simplicial complexes proposed in the work of De Floriani are implemented in the Mangrove Topological Data Structure Library which is publicly available [5]. For a detailed description and comparison of a selection of data structures focusing on mesh generation and finite element analysis, we refer to the work of Garimella [16]. Also a concise overview and comparison of array-based data structure concepts has been given by Alumbaugh et al. in [1].

In [13] De Floriani et al. introduce a scalable index-based mesh data structure that can be used to represent mixed-dimensional non-manifolds. Their approach differs from the one presented in this work in that it is restricted to simplicial complexes. Furthermore, there is the work of Gross et al. [18], Dobkin et al. [11], a survey on data structures for level-of-detail models by De Floriani et al. [10], and the 3-dimensional triangulations which are part of the CGAL library described by Teillaud in [31] that are also restricted to simplicial complexes.

In [6] Celes et al. present a compact data structure restricted to manifolds and focusing on the requirements of finite element analysis. Also refer to [2] for a more detailed description. Their approach only encodes vertices and top elements causing low memory consumption at one hand but high computational costs when generating edge and face definitions at the other hand. Related concepts are presented in the MOAB-SD data structure [30]. In [21] a memory-optimized data structure for arbitrary cell complexes is presented that is also restricted to manifold mesh configurations.

Another point of view offers the work from Remacle et al. in [26] where the *Algorithm Oriented Mesh Database* is presented. In their approach and its extension for parallel analysis [28], a dynamic mesh representation is used, i.e. a representation which is able to adapt to the needs of an individual algorithm. This data structure is very flexible while lacking the ease-of-use of the presented one. An entirely different approach was introduced with the concept of Nef-polyhedra as described by Granados et al. in [17]. Nef-polyhedra in d-dimensional space are the closure of half-spaces under Boolean set operations. This concept is a powerful tool to handle non-manifold complexes that may even contain elements of infinite volume. Yet, obviously, the approach is hardly usable in meshing and finite element analysis applications because it is not intuitive and due to its generality lacks efficiency. The radial-edge data structure [33] is capable of representing non-manifold 3-dimensional meshes but comes at the price of comparatively high storage costs.

Recently, the *Combinatorial Maps* data structure [7] based on the work of Damiand [8] was released as part of the CGAL library. The underlying concept of this data structure is the generalization of half-edges in arbitrary dimensions, also see the concept of n-dimensional generalized maps as described by Lienhardt [22]. Combinatorial Maps are used to represent the topology of a polytopal complex whereas its extension, the *Linear Cell Complex* data structure [9] which is also part of the CGAL library is used to attach geometric information to the vertices of the complex. Due to its similarity to OpenVolumeMesh, we focus on the comparison to CGAL's Linear Cell Complexes in the evaluation section of this paper.

3 Design

3.1 Terminology

A *k-manifold* with boundary is a subset of the Euclidean space for which the neighborhood of each internal point is topologically homeomorphic to an open k-ball and the neighborhood of each boundary point to an open k-half-ball. A *combinatorial* 3-dimensional polytopal complex is a mesh that consists of a set of conforming d-polytopes, $0 \leq d \leq 3$, with underlying incidence and adjacency relations. The 3-dimensional polytopes of such complex are called *cells*. They are topological polyhedra and thus bounded by a set of

2-dimensional entities called *faces.* Each face is bounded by a set of *edges*, the 1-dimensional entities of the complex. Entities of dimension 0 are called *vertices.* Each edge spans a 1-manifold between a pair of distinct vertices. If an entity of dimension d is entirely part of the boundary of an entity of dimension greater than d they are said to be *incident.* Two distinct vertices are said to be *adjacent* if they are incident to the same edge. Two distinct entities of dimension d, with $1 \leq d \leq 3$, are adjacent if they are incident to the same entity of dimension $d-1$. Those faces of a manifold 3-dimensional complex that are incident to exactly two cells are called *interior* or simply *inner* faces, whereas faces incident to exactly one cell are called *boundary* faces.

A non-manifold complex which contains parts of different dimensionality, i.e. entities not incident to at least one entity of maximum dimension, is said to be *non-regular* or *mixed-dimensional.* Analogously, a k-manifold complex that does not contain any entity that is not incident to a k-dimensional entity is said to be *regular.*

In this paper, $\mathcal{V}$ denotes the set of vertices, $\mathcal{E}$ the set of edges, $\mathcal{F}$ the set of faces, and $\mathcal{C}$ the set of cells of a complex. In a combinatorial polytopal complex, the vertices are considered abstract entities without geometric meaning, whereas in a *geometric* polytopal complex, there exists a function $p : \mathcal{V} \rightarrow \mathbb{R}^n$, usually $n = 3$, that assigns each vertex a unique position in n-dimensional Euclidean space. This function is called *geometric embedding.* Note that in engineering sciences this function is also referred to as *geometric classification.*

3.2 Connectivity Representation

Half-Edge Based Data Structures

In half-edge based representations used for 2-manifold surface meshes, the edges of a mesh are split into pairs of directed *half-edges* as described in detail in [23] and [4]. For each half-edge a set of links (e.g. pointers) to neighboring entities is stored. Usually this set comprises a link to one of the incident vertices, the incident face, the opposite half-edge as well as an adjacent half-edge within the current face (cf. Fig. 1). The notion of half-edges intrinsically induces an orientation to the mesh. A face is well-defined if its incident boundary half-edges form a closed and consistently oriented loop. By convention, the front-side of a face is defined to be the one where all incident half-edges are oriented in counter-clockwise manner. In a 2-manifold surface mesh represented by half-edges, each half-edge can be incident to either zero or one face. Half-edges incident to no face are called *boundary.*

This representation benefits from the fact that many of the commonly used iterators can be generated with little computational cost. Furthermore, this data structure concept allows for efficient local modifications of the surface mesh by changing only a few pointers. Hence, this concept is highly suitable for meshing and geometry processing applications. However, one of the major drawbacks is the restriction to manifold mesh configurations since in some meshing applications temporarily non-manifold configurations may occur.

Top-Down Incidence Relations

In the presented data structure we adopt the general idea of representing the edges by directed half-edges. Furthermore, we carry over this concept to the faces. Each face of a mesh is then represented by a pair of oppositely oriented *half-faces* (cf. [20]). A half-face as well as its orientation is uniquely determined by its incident half-edges.

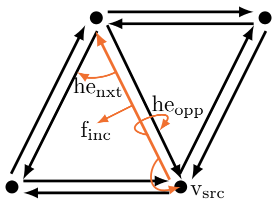


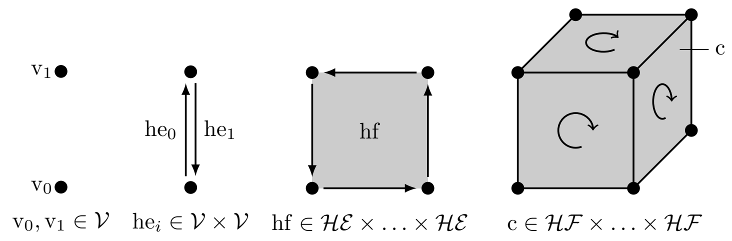
Fig. 1. An illustration of a simple half-edge based mesh. For each half-edge a set of pointers is stored. These pointers are links to the following neighboring entities: Source (or target) vertex ($\mathrm{v_{src}}$), opposite half-edge ($\mathrm{he_{opp}}$), incident face ($\mathrm{f_{inc}}$), next half-edge in face ($\mathrm{he_{nxt}}$). The incident half-edges of each of the faces form a closed and consistently oriented loop. Usually the front-side of a face is defined to be the one where this loop is oriented in counter-clockwise sense.

In contrast to the traditional half-edge based data structures for 2-manifolds, in the proposed data structure connectivity is not encoded as unique links to neighboring entities per entity. Instead, an (ordered) list of links, called *handles*, to the respective incident first-order lower-dimensional entities is stored, similar to the incidence graph data structure [12]. Consequently, each entity of dimension d as well as its orientation is defined by an (ordered) list of handles to its incident entities of dimension $d-1$. For instance, a half-edge is uniquely defined by an ordered 2-tuple of handles to its incident vertices (one source and one target vertex). These relations are called the (intrinsic) *top-down* incidence relations. With the notion of orientation it is easy to check for the consistency of an entity. For example, in analogy to half-edge based representations, a half-face's incident half-edges have to be specified in an order such that their union forms a closed, connected, and consistently oriented 1-manifold. A cell is well-defined if the union of all incident half-faces forms a closed and consistently oriented 2-manifold which is the case if each incident half-edge occurs exactly once and the number of involved edges is half the number of involved half-edges. See Figure 2 for an illustration of this concept.

Handles are simply indices of the entities referring to their position in the respective linear storage container and thus allow for constant access time. Using top-down incidence relations rather than the traditional pointer-based half-edge approach circumvents the restriction to quasi-manifold configurations.

Bottom-Up Incidence Relations

The described top-down incidence relations are complete in a sense that they define all possible incidence and adjacency relations between each pair of



v_1
v_0
he_0
he_1
hf
c
$v_0, v_1 \in \mathcal{V}$
$he_i \in \mathcal{V} \times \mathcal{V}$
$hf \in \mathcal{HE} \times \ldots \times \mathcal{HE}$
$c \in \mathcal{HF} \times \ldots \times \mathcal{HF}$

Fig. 2. Illustration of the top-down incidence relations at the example of a hexahedron. Each entity, except for vertices, is uniquely defined via an (ordered) list of handles to the respective incident first-order lower-dimensional entities. The order in which these handles are given determines the entity's orientation.

distinct entities of a mesh. Consequently, *bottom-up* incidences, i.e. links to incident entities of dimension $d + 1$ for an entity of dimension d, can be entirely extracted from top-down incidences. However, this causes unnecessary computational costs that scale linearly with the complexity of a mesh. Consequently, it is possible to explicitly generate and cache bottom-up incidence relations for all entities except for cells in a straightforward manner.

This is accomplished by storing a list of handles to the respective incident first-order higher-dimensional entities per entity. For instance, for each vertex a list of incident outgoing half-edges is stored. Using these incidence relations allows for constant access time to the higher-dimensional neighborhood of an entity instead of access time in $O(n)$, where n is the number of elements in the mesh. When adding an entity to the mesh, bottom-up incidences are only updated for the local neighborhood of the considered entity and its incident lower-dimensional components. Note that these relations can be computed optionally and come, if needed, at the price of extra memory consumption. The amount of extra memory consumption scales linearly with the mesh complexity. Refer to Section 5.1 for an analysis.

Figure 3 shows an illustration of the bottom-up incidence relations in a simple hexahedral mesh. Note that the order of incident half-faces per half-edge can only be determined in 3-manifold meshes where each edge is incident to at most two boundary faces. In non-manifold mesh configurations the incident half-faces are given in no particular order.

Although bottom-up incidences can be computed optionally, they are required for some of the provided iterators. Furthermore, they are needed in order to determine the boundary of a mesh.

3.3 Storage

Internally, all entity objects are stored in array-based storage containers using handles to access them, i.e. indices referring to their position within the respective array. As mentioned in the previous section, an entity of dimension d, except $d = 0$, is internally represented by a list of handles to incident

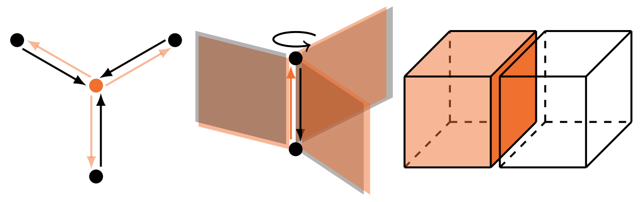
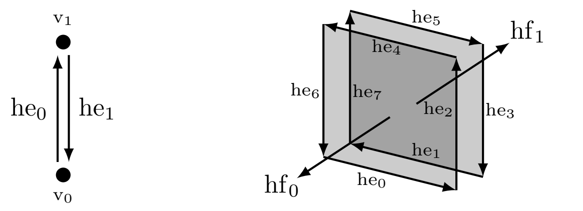


Fig. 3. Illustration of the bottom-up incidence relations at the example of a simple hexahedral mesh. Left: For each vertex a list of incident outgoing half-edges is stored (marked in orange). Middle: For each half-edge an ordered list of incident half-faces is stored. The half-faces are stored in counter-clockwise order with respect to the respective half-edge pointing towards the viewer. Note that this order can only be determined in a 3-manifold mesh. Right: For each half-face a handle to the incident cell is stored. For all boundary half-faces a sentinel handle is stored.

entities of dimension $d-1$. By construction, all opposing half-entities have consecutive indices because they are always created in pairs. We can exploit this fact in a way that we physically store only one of the two half-entities as we know that the opposite of each half-entity is defined by the inverse list of incident lower-dimensional entities. The operation of generating the opposite of an half-entity is accomplished by the opp()-operator as described in Figure 4. When accessing a half-entity that is not explicitly stored, it is generated on-the-fly using this operator.



$$\mathrm{he}_0 = (\mathrm{v}_0, \mathrm{v}_1)$$
$$\mathrm{he}_1 = (\mathrm{v}_1, \mathrm{v}_0)$$
$$\mathrm{he}_1 = \mathrm{opp}(\mathrm{he}_0) = (\mathrm{v}_1, \mathrm{v}_0)$$
$$\mathrm{he}_0 = \mathrm{opp}(\mathrm{he}_1) = (\mathrm{v}_0, \mathrm{v}_1)$$

$$\mathrm{hf}_0 = (\mathrm{he}_0, \mathrm{he}_2, \mathrm{he}_4, \mathrm{he}_6)$$
$$\mathrm{hf}_1 = (\mathrm{he}_7, \mathrm{he}_5, \mathrm{he}_3, \mathrm{he}_1)$$
$$\mathrm{hf}_1 = \mathrm{opp}(\mathrm{hf}_0) =$$
$$(\mathrm{opp}(\mathrm{he}_6), \mathrm{opp}(\mathrm{he}_4), \mathrm{opp}(\mathrm{he}_2), \mathrm{opp}(\mathrm{he}_0))$$
$$\mathrm{hf}_0 = \mathrm{opp}(\mathrm{hf}_1) =$$
$$(\mathrm{opp}(\mathrm{he}_1), \mathrm{opp}(\mathrm{he}_3), \mathrm{opp}(\mathrm{he}_5), \mathrm{opp}(\mathrm{he}_7))$$

Fig. 4. Description of the opp()-operator. Left: The opposite of a given half-edge is generated by simply inverting the ordered list of handles to the respective incident vertices. Right: The definition of the opposite of a given half-face is obtained by inverting the ordered list of handles to the incident half-edges and replacing each element in this list by its opposite, respectively.

Since these handles are only indices, the inversion can be performed at very small computational cost. The benefit of this technique is the reduction of memory consumption for the half-edges and half-faces by a factor of two.

Access to the half-entities is accomplished via the following mapping (at the example of half-edges):

$$\text{half\_edge}(h) = \begin{cases} \text{he\_array}[\frac{h}{2}] & \text{if } h \text{ is even,} \\ \text{opp}(\text{he\_array}[\frac{h-1}{2}]) & \text{otherwise,} \end{cases}$$
$$\text{with } h \in [0, 2 \cdot |\mathcal{E}| - 1],$$

where he_array is the array in which the half-edge objects are stored, h is a half-edge handle, and he_array$[i]$, with $0 \leq i < |\mathcal{E}|$, evaluates to the half-edge object stored at position i in the array. This works analogously for the half-faces.

4 Implementation Details

The presented library is entirely written in C++ using the standard template library and template programming paradigms. At the implementation level we make a clear distinction between the topology and geometry of a complex. The topology as well as the geometry is implemented in so called *kernel* classes. Class `TopologyKernel` provides all basic topological functions and relations of a complex including top-down and bottom-up incidences, as well as means to add and/or delete entities. It is designed to handle heterogeneous polytopal meshes. If needed, it is possible to derive specializations from this class in order to handle e.g. homogeneous polytopal meshes of certain kinds of polytopes, such as tetrahedra, hexahedra, etc., where special functionality may be needed. The library already includes a specialized topology class for hexahedra where an additional consistency check assures that the number of incident half-faces of a hexahedron is exactly six and that they induce a “virtual” coordinate system as shown in Figure 5. In this figure “XF” denotes the front-face and “XB” denotes the back-face on the virtual x-axis oriented front-to-back. This works analogous for the two other axes.

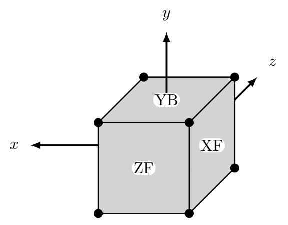


Fig. 5. The half-faces of a hexahedron induce a “virtual” coordinate system if given in a fixed order, e.g. (XF, XB, YF, YB, ZF, ZB), where “XF” denotes the front-face and “XB” the back-face on the virtual x-axis, etc.

If, in addition to the topology, a geometric embedding is needed, one may use class `GeometryKernel` which inherits a specified topology kernel and which provides a geometric embedding of the vertices into some vector space as well as some common functions such as the computation of barycenters, edge lengths, etc. The embedded space is specified as template parameter to the

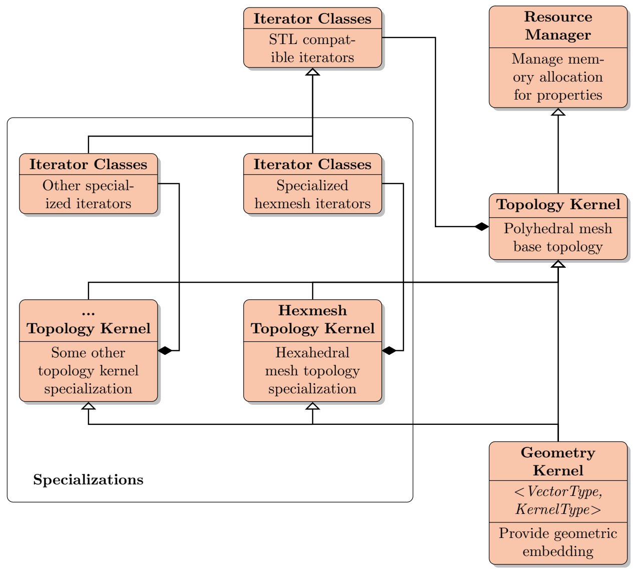

Iterator Classes
STL compatible iterators
Resource Manager
Manage memory allocation for properties
Iterator Classes
Other specialized iterators
Iterator Classes
Specialized hexmesh iterators
Topology Kernel
Polyhedral mesh base topology
...
Topology Kernel
Some other topology kernel specialization
Hexmesh Topology Kernel
Hexahedral mesh topology specialization
Specializations
Geometry Kernel
<VectorType, KernelType>
Provide geometric embedding

Fig. 6. The class diagram of OpenVolumeMesh. All kernel classes inherit from the resource manager which manages memory allocation for the dynamic properties. Class `TopologyKernel` as well as its derived specializations represent the topology of a polytopal complex. If needed, a geometric embedding is provided by class `GeometryKernel` which inherits a topology kernel and expects the vector type of the respective embedding as template parameter. All iterator classes expect a mesh reference as construction parameter.

geometry kernel. The library provides some commonly used predefined vector types that can be used for this purpose. The library also provides means to dynamically attach properties of arbitrary type to the entities and the mesh itself at run-time. The memory management for these properties is implemented in the `ResourceManager` class which is at the top of the inheritance hierarchy. The (de)allocation of memory for a property works analogously to the concept of smart pointers [29]. Furthermore, OpenVolumeMesh is equipped with many common STL-compliant iterators allowing for navigation on the mesh. Each specialized topology kernel may additionally provide its own set of iterators adapted to the respective topology. Figure 6 shows an overview of the main classes in OpenVolumeMesh as well as their inheritance relations.

In addition to the discussed core components of the data structure itself, OpenVolumeMesh is equipped with various useful tools. The set of available tools comprises a (de)serializer implementation that can be used to load meshes from files as well as to save meshes to files. It uses a simple ASCII file format that also supports serialization of properties. Also a file converter tool is included that can be used to convert finite element mesh data generated with the help of external tools, e.g. NETGEN [27], to OpenVolumeMesh's file format. An additional attribute class can be used to mark the entities of a mesh with several flags and which also provides a garbage collection implementation that is used to keep the mesh consistent and optionally manifold after the deletion of a subset of entities. Many of the library's core functions are verified using a unit testing framework.

5 Evaluation and Comparison

In this section we evaluate the performance of the presented data structure. Our main focus is the memory consumption as well as the CPU load caused by a set of various algorithms computed on some meshes shown in Figure 7. Due to its similarity, we directly compare the presented data structure to CGAL's Linear Cell Complex data structure [9]. The test code has been compiled using GCC 4.6.3 with level 2 optimization. It is run on an Intel Core i7 CPU at 2.6 GHz and 6 GB of RAM.

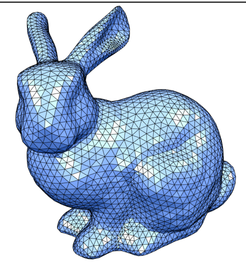
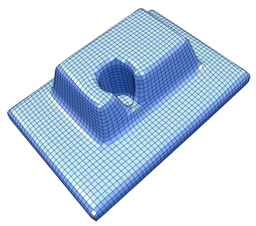
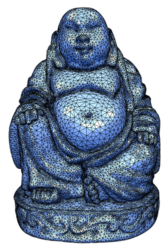
Bunny		**Drill Hole**		**Buddha**	
					
Type	Tetrahedra	Type	Hexahedra	Type	Tetrahedra
Vertices	13,247	Vertices	12,782	Vertices	1,017,902
Edges	82,841	Edges	35,734	Edges	4,155,275
Faces	135,361	Faces	33,213	Faces	6,250,094
Cells	65,766	Cells	10,260	Cells	3,112,720

Fig. 7. The meshes used for the evaluation of the proposed data structure. All meshes are 3-manifold and homogeneous with respect to their cell type.

5.1 Storage Costs

In the Linear Cell Complex data structure, the notion of *darts* is the extension of the classical half-edge to arbitrary dimensions. Links to neighboring entities are represented by a set of so-called β-pointers. The entire topology of a manifold cell complex can be represented by darts and β-pointers. Also only darts are used to represent the entities of any dimension of a complex. In a 3-dimensional mesh, each dart contains four β-pointers, one link to an incident entity of each dimension. For a more detailed description of this concept, refer to the work of Damiand [8, 7] and Lienhardt [22].

The number of required darts in a manifold 3-dimensional mesh can be approximated as follows:

$$n_D \approx |\mathcal{C}| \cdot v_{\mathcal{C}} \cdot v_{\mathcal{F}},$$

where $|\mathcal{C}|$ is the number of cells, $v_{\mathcal{C}}$ the average cell valence, and $v_{\mathcal{F}}$ the average face valence. Furthermore, let the amount of memory (in byte) used to encode the geometric embedding of a single vertex be denoted as $s_{\mathcal{V}}$ and let s_β be the physical size of a β-pointer. So the theoretical overall memory consumption of a 3-manifold cell complex represented in the Linear Cell Complex data structure is computed as follows:

$$s_{\mathrm{LCC}} = n_D \cdot 4 \cdot s_\beta + |\mathcal{V}| \cdot s_{\mathcal{V}} \text{ byte}, \tag{1}$$

where $|\mathcal{V}|$ is the number of vertices in the mesh.

In OpenVolumeMesh all entities except vertices are represented by lists of handles. The number of required handles to encode the top-down incidences (abbr. TD) of a mesh is approximated as follows:

$$n_{\mathrm{TD}} \approx |\mathcal{E}| \cdot 2 + |\mathcal{F}| \cdot v_{\mathcal{F}} + |\mathcal{C}| \cdot v_{\mathcal{C}},$$

where $|\mathcal{E}|$ is the number of edges and $|\mathcal{F}|$ the number of faces in the mesh. For the representation of the bottom-up incidences (abbr. BU) the number of stored handles is:

$$n_{\mathrm{BU}} \approx |\mathcal{V}| \cdot v_{\mathcal{V}} + |\mathcal{E}| \cdot v_{\mathcal{E}} + |\mathcal{F}| \cdot 2,$$

where $v_{\mathcal{V}}$ is the average vertex valence, i.e. the number of incident edges, and $v_{\mathcal{E}}$ is the average edge valence, i.e. the number of incident faces.

In analogy to Linear Cell Complexes, each vertex uses another $s_{\mathcal{V}}$ bytes of memory for its embedding. Let the size of a handle be denoted as s_h. So the overall memory consumption of a mesh represented with the OpenVolumeMesh data structure is computed as follows:

$$s_{OVM} = (n_{\mathrm{TD}} + n_{\mathrm{BU}}) \cdot s_h + |\mathcal{V}| \cdot s_{\mathcal{V}} \text{ byte}. \tag{2}$$

Note that n_{BU} is 0 when no bottom-up incidences are needed.

Assuming a β-pointer's size to be 8 byte on a 64-bit computing architecture, the theoretical size of a dart in a 3-dimensional mesh is 32 byte due to the mentioned four β-pointers. In practice, a dart's size turns out to be 48 byte because it does not only encapsulate the β-pointers but also includes a bitset used to mark darts as well as some attribute related data. In contrast, the size of a handle in the proposed data structure is 4 byte (the size of a 32-bit integer). In both cases, we use an embedding into 3-dimensional Euclidean space with double precision, thus $s_{\mathcal{V}} = 24$ byte.

Table 1 shows the amount of memory consumed at the example of the test meshes. In all cases the total number of used darts in a Linear Cell Complex is slightly smaller than the number of stored handles in an OpenVolumeMesh. However, the amount of stored links (in terms of β-pointers of handles) in Linear Cell Complexes is approximately 2 times (4 times without bottom-up incidences) greater than in OpenVolumeMesh. In addition, the size of a dart (48 byte) is comparatively large, so the used memory is significantly smaller for a mesh represented with OpenVolumeMesh. Of course, these results may vary on different computer architectures.

Linear Cell Complex	**Bunny**	**Drill Hole**	**Buddha**
Number of darts	789,192	246,240	37,352,640
Total # of links (β-pointers)	3,156,768	984,960	149,410,560
KByte (cf. Eq. 1)	37,303	11,842	1,774,762
OpenVolumeMesh	**Bunny**	**Drill Hole**	**Buddha**
Number of handles (TD)	834,829	265,880	39,511,712
Number of handles (BU)	857,138	286,054	42,437,681
Total # of links (handles)	1,691,967	551,934	81,949,393
KByte (cf. Eq. 2, TD only)	3,571	1,338	178,199
KByte (cf. Eq. 2, TD + BU)	6,919	2,455	343,971

Table 1. A comparison of the memory consumption of three different meshes (cf. Fig. 7) represented in the Linear Cell Complex and in the OpenVolumeMesh data structure. In this example, all average valences have been rounded up to the next higher integer. One can see that even with bottom-up incidences the amount of consumed memory is significantly smaller for meshes in OpenVolumeMesh compared to Linear Cell Complexes.

5.2 Computational Costs

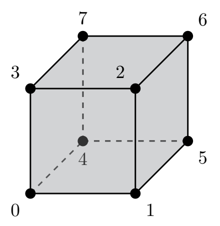
For the evaluation of the computational performance we compare timing results of six different algorithms performed on the bunny and the drill hole mesh (cf. Fig. 7). We test two different classes of algorithms: Three algorithms that leave the topology of the meshes static but only affect the geometry as well as three algorithms that change the topology of the meshes.

In the first example, 100 iterations of Laplacian smoothing are performed on the interior vertices of the meshes by simply placing each vertex into its barycenter in each iteration step. A vertex attribute/property is used to temporarily store the new position of each vertex during one iteration. In this algorithm only the vertex positions of the mesh are modified, the topology is left unchanged.

The second static algorithm simply computes the barycenter of each cell by summing up the positions of all incident vertices interpreted as vectors and dividing them by the total number of incident vertices per cell.

In the third algorithm, the *Scaled-Jacobian* metric is evaluated per vertex per cell of a hexahedral mesh. The Scaled-Jacobian is computed as depicted in Figure 8.

For the class of non-static algorithms we test two subdivision schemes (refinement) as well as a series of edge collapses (decimation) on the meshes. For the refinement operations, a 1-4-split is performed on each element of the bunny mesh by inserting a vertex in the cell's barycenter and creating 4 tetrahedra by connecting all vertices of the tetrahedron with the new center vertex. On the hexahedral mesh, a 1-7-split is performed, that is, for each hexahedron another smaller hexahedron is placed in the center of it such that both hexahedra, the outer and the inner one, are equally aligned. Then another six hexahedra are created by connecting all edges of the inner hexahedron with the corresponding edges of the outer hexahedron forming a face each. This is also known as the *pillow operation* performed on each hexahedron separately as described in [19]. These subdivision schemes serve as an artificial example of refinement applications. In the third non-static algorithm a series of edge collapses is performed on the bunny mesh. These operations are used to simplify tetrahedral meshes. See [32] for a detailed description of the operation. We use bottom-up incidences for all algorithms, whereas for the subdivision algorithms we test different variants: With bottom-up incidences enabled and with only a subset of the bottom-up incidences enabled (those necessary for the computations). Those bottom-up incidences turned off in the second variant are generated afterwards in a third variant.



$c_0 = (0, 1, 4, 3)$
$c_1 = (1, 5, 0, 2)$
$c_2 = (2, 3, 6, 1)$
$c_3 = (3, 7, 2, 0)$
$c_4 = (4, 0, 5, 7)$
$c_5 = (5, 4, 1, 6)$
$c_6 = (6, 2, 7, 5)$
$c_7 = (7, 6, 3, 4)$

$$\delta(i,j,k,l) = \det[\frac{\mathbf{v}_j - \mathbf{v}_i}{\|\mathbf{v}_j - \mathbf{v}_i\|}, \frac{\mathbf{v}_k - \mathbf{v}_i}{\|\mathbf{v}_k - \mathbf{v}_i\|}, \frac{\mathbf{v}_l - \mathbf{v}_i}{\|\mathbf{v}_l - \mathbf{v}_i\|}]$$

$$\text{Scaled-Jacobian} = \min_{0 \le j \le 7} \delta(c_j)$$

Fig. 8. The Scaled-Jacobian metric for all-hexahedral meshes. For each vertex of a cell its three incident edges are interpreted as normalized vectors having the vertex as origin. For each of these triplets of vectors an orthonormal matrix is formed such that the vectors are the column entries ordered so that they form a right-handed coordinate system. The minimum determinant of these matrices is then the desired value per cell. The metric ranges between -1 and 1. The higher the metric the least distorted the hexahedron.

The timing results for the algorithms run on the tetrahedral bunny and the hexahedral drill hole mesh can be seen in Figure 9.

5.3 Discussion

One can see that, in the tested static and some refinement algorithms, OpenVolumeMesh performs better than Linear Cell Complex. Iterating over all entities of a type is very efficient in OpenVolumeMesh since all elements are stored in STL vectors that are coherently aligned in memory and thus benefit

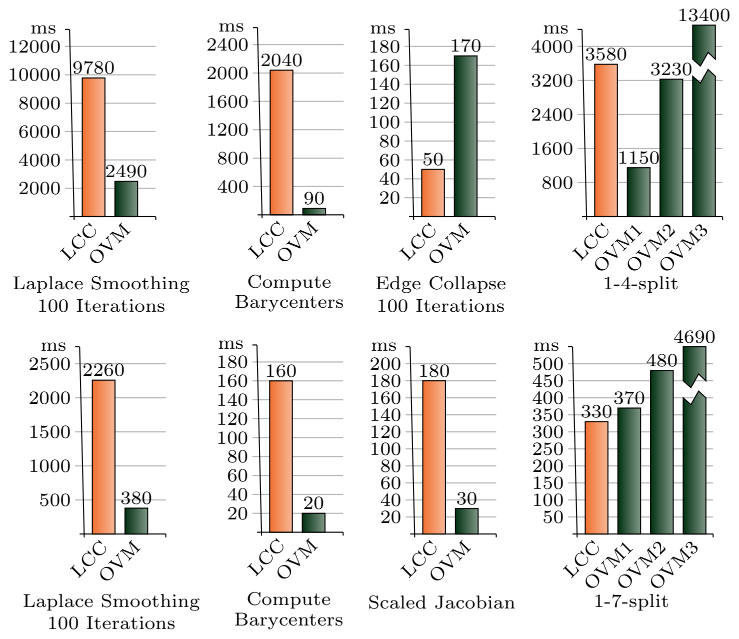


Fig. 9. Top row: The timing results of a selection of algorithms run on the all-tetrahedral bunny mesh (cf. Fig. 7, left). Bottom row: The timing results of a selection of algorithms run on the all-hexahedral drill hole mesh (cf. Fig. 7, center). In OVM1 only those bottom-up incidences needed for the computations are enabled. In OVM2 the disabled bottom-up incidences from variant OVM1 are additionally computed in one chunk afterwards. In OVM3 all bottom-up incidences are enabled.

greatly from modern processors' caching strategies. In contrast, iterating over the darts of a certain entity type in the Linear Cell Complex data structure causes a lot of pointer dereferences and random memory access (unless they are precached) which in general is more expensive than addressing linearly aligned memory. Especially the static algorithms benefit from the bottom-up incidences because they provide precached arrays of handles to an entity's local neighborhood. This may also be a major advantage when creating finite-element matrices as this is a typical example of such a static application. In the refinement applications, Linear Cell Complex supposedly benefits from the fact that the insertion and deletion of entities is accomplished via only a few pointer modifications. In particular when it comes to more complex meshing algorithms as in the 1-7-split, results showed that Linear Cell Complexes perform slightly better than OpenVolumeMesh. If bottom-up incidences are used in meshing algorithms, a lot of undue computational overhead is caused since at every insertion and deletion of an entity all locally affected bottom-up incidences are re-computed by extracting them from the top-down incidences of the local neighborhood of the entity. Therefore, in these cases it is better to disable unnecessary bottom-up incidences beforehand and, if needed, completely generate them in one chunk afterwards. However, at the example of decimation algorithms such as the edge collapses, one can see that array-based representations as implemented in OpenVolumeMesh suffer from the disadvantage that entity deletions are computationally expensive (compared to deletions from a linked list representation as in Linear Cell Complex).

Furthermore, OpenVolumeMesh also has some important limitations by design that we do not want to hold back. Since Linear Cell Complex is the general extension of the half-edge concept, it is suitable to handle meshes in arbitrary dimensions. In contrast, OpenVolumeMesh can only handle meshes of dimension up to three. Indeed, OpenVolumeMesh is suitable for many of the common engineering applications, but there exist as well applications where higher dimensional manifolds are needed. Furthermore, due to the fact that in OpenVolumeMesh each half-entity is shared by its incident entities and thus only exists once, no matter which valence, no unique intrinsic ordering of the bottom-up incidences is provided. For example, unless explicitly computed, there exists no specific order in which a half-edge's incident half-faces are given. In Linear Cell Complex these orderings are intrinsically predetermined. Note that, in general, this order can be determined in manifold meshes only.

6 Conclusion

The results presented in the previous section are promising and legitimate OpenVolumeMesh's position as general purpose data structure for arbitrary 3-dimensional polytopal meshes among other publicly available data structures. We showed that it is suitable for a variety of applications. Furthermore, it is possible to represent non-manifold mesh configurations including mixed-dimensional entities. Additionally, OpenVolumeMesh is fully integrated into the OpenFlipper geometry processing framework [24]. OpenVolumeMesh is open-source software licensed under the terms of the GNU LGPL and available at `www.openvolumemesh.org`.

7 Acknowledgment

The authors thank Hans-Christian Ebke and Marcel Campen for valuable advice and support. We are also very grateful to Guillaume Damiand from LIRIS, Lyon, for kind help with CGAL's Linear Cell Complex data structure. The bunny mesh is courtesy of the Stanford Computer Graphics Laboratory.

This work was funded by the AICES Graduate School and the European Research Council (ERC Starting Grant "Robust Geometry Processing", Grant agreement 257474).

References

[1] T. J. Alumbaugh and X. Jiao. "Compact Array-Based Mesh Data Structures". In: *IMR*. Ed. by Byron W. Hanks. Springer, 2005, pp. 485–503. ISBN: 978-3-540-25137-8.

[2] M. W. Beall and M. S. Shephard. "A General Topology-Based Mesh Data Structure". In: *International Journal for Numerical Methods in Engineering* 40.9 (1997), pp. 1573–1596. ISSN: 1097-0207.

[3] M. Botsch et al. "OpenMesh – a generic and efficient polygon mesh data structure". In: *In OpenSG Symposium.* 2002.

[4] S. Campagna, L. Kobbelt, and H.-P. Seidel. "Directed Edges - A Scalable Representation for Triangle Meshes". In: *Journal of Graphics Tools* 3.4 (1998).

[5] D. Canino. *Mangrove Topological Data Structure.* URL: `http://mangrovetds.sourceforge.net`.

[6] W. Celes, G. H. Paulino, and R. Espinha. "A compact adjacency-based topological data structure for finite element mesh representation". In: *International Journal for Numerical Methods in Engineering* 64.11 (2005), pp. 1529–1556. ISSN: 1097-0207.

[7] G. Damiand. "Combinatorial Maps". In: *CGAL User and Reference Manual.* 4.0. CGAL Editorial Board, 2012.

[8] G. Damiand. "Contributions aux Cartes Combinatoires et Cartes Généralisées: Simplification, Modèles, Invariants Topologiques et Applications". Habilitation à Diriger des recherches. Université Lyon 1, 2010.

[9] G. Damiand. "Linear Cell Complex". In: *CGAL User and Reference Manual.* 4.0. CGAL Editorial Board, 2012.

[10] L. De Floriani, L. Kobbelt, and E. Puppo. "A Survey on Data Structures for Level-of-Detail Models". In: *Advances in Multiresolution for Geometric Modelling Series in Mathematics and Visualization* 243 (2007). Ed. by Neil A. Dodgson, Michael S. Floater, and Malcolm A. Sabin, p. 523.

[11] D. P. Dobkin and M. J. Laszlo. "Primitives for the manipulation of three-dimensional subdivisions". In: *Proceedings of the third annual symposium on Computational geometry.* SCG '87. Waterloo, Ontario, Canada: ACM, 1987, pp. 86–99. ISBN: 0-89791-231-4.

[12] H. Edelsbrunner. *Algorithms in combinatorial geometry.* New York, NY, USA: Springer-Verlag New York, Inc., 1987. ISBN: 0-387-13722-X.

[13] L. De Floriani and A. Hui. "A scalable data structure for three-dimensional non-manifold objects". In: *Symposium on Geometry Processing.* 2003.

[14] L. De Floriani and A. Hui. "Data Structures for Simplicial Complexes: An Analysis And A Comparison". In: *Symposium on Geometry Processing.* 2005.

[15] L. De Floriani and A. Hui. "Shape Representations Based on Simplicial and Cell Complexes". In: *Eurographics 2007 - State of the Art Reports.* Ed. by Dieter Schmalstieg and Jiri Bittner. Prague: Eurographics Association, 2007.

[16] R. V. Garimella. "Mesh Data Structure Selection for Mesh Generation and FEA Applications". In: *International Journal of Numerical Methods in Engineering* 55.4 (Oct. 2002), pp. 451–478.

[17] M. Granados et al. "Boolean Operations on 3D Selective Nef Complexes: Data Structure, Algorithms, and Implementation". In: *Algorithms - ESA 2003.* Ed. by Giuseppe Di Battista and Uri Zwick. Vol. 2832. Lecture Notes in Computer

Science. Springer Berlin / Heidelberg, 2003, pp. 654–666. ISBN: 978-3-540-20064-2.

[18] P. W. Gross and P. R. Kotiuga. "Data Structures for Geometric and Topological Aspects of Finite Element Algorithms". In: *Progress in Electromagnetics Research* 32 (2001), pp. 151–169. ISSN: 1070-4698.

[19] P. Knupp, L. Subcase, and S. A. Mitchell. *Integration of Mesh Optimization with 3D All-Hex Mesh Generation.* 1999.

[20] M. Lage et al. "CHF: a scalable topological data structure for tetrahedral meshes". In: *Sibgrapi 2005 (XVIII Brazilian Symposium on Computer Graphics and Image Processing).* Natal, RN: IEEE, 2005, pp. 349–356.

[21] F. Ledoux, J.-C. Weill, and Y. Bertrand. "GMDS: A Generic Mesh Data Structure". In: *17th International Meshing Roundtable.* United States, 2008.

[22] P. Lienhardt. "Topological models for boundary representation: a comparison with n-dimensional generalized maps". In: *Computer-Aided Design* 23.1 (1991), pp. 59 –82. ISSN: 0010-4485.

[23] M. Mäntylä. *An Introduction to Solid Modeling.* New York, NY, USA: Computer Science Press, Inc., 1987. ISBN: 0-88175-108-1.

[24] J. Möbius and L. Kobbelt. "OpenFlipper: An Open Source Geometry Processing and Rendering Framework". In: *Curves and Surfaces.* Ed. by Jean-Daniel Boissonnat et al. Vol. 6920. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pp. 488–500. ISBN: 978-3-642-27412-1.

[25] P. Murdoch et al. "The spatial twist continuum: a connectivity based method for representing all-hexahedral finite element meshes". In: *Finite Elem. Anal. Des.* 28.2 (Dec. 1997), pp. 137–149. ISSN: 0168-874X.

[26] J.-F. Remacle and M. S. Shephard. "An algorithm oriented mesh database". In: *International Journal for Numerical Methods in Engineering* 58.2 (2003), pp. 349–374. ISSN: 1097-0207.

[27] J. Schöberl. "NETGEN An advancing front 2D/3D-mesh generator based on abstract rules". In: *Computing and Visualization in Science* 1 (1 1997), pp. 41–52. ISSN: 1432-9360.

[28] E. Seegyoung Seol and M. S. Shephard. "Efficient distributed mesh data structure for parallel automated adaptive analysis". In: *Eng. with Comput.* 22.3 (Dec. 2006), pp. 197–213. ISSN: 0177-0667.

[29] B. Stroustrup. *The C++ programming language (3. ed.)* Addison-Wesley-Longman, 1997, pp. I–X, 1–910. ISBN: 978-0-201-88954-3.

[30] T. J. Tautges. "MOAB-SD: integrated structured and unstructured mesh representation." In: *Eng. Comput. (Lond.)* 20.3 (2004), pp. 286–293.

[31] M. Teillaud. *Three Dimensional Triangulations in CGAL.* 1999.

[32] I. J. Trotts et al. "Simplification of tetrahedral meshes". In: *Proceedings of the conference on Visualization '98.* VIS '98. IEEE Computer Society Press, 1998, pp. 287–295. ISBN: 1-58113-106-2.

[33] K. J. Weiler. "Radial Edge Structure: A Topological Representation for Non-manifold Geometric Boundary Modeling". In: *Geometric Modeling for CAD Applications* (1988). Ed. by M.J. Wozney, H.W. McLaughlin, and J.L. Encarnacao, pp. 3–36.