OVM-An Open Source Data Structure for Tetrahedral and Hexahedral Meshes

Chuhua Xian

School of Computer Science and Engineering, South China University of Technology, Guangzhou, 510006, China

chuhuaxian@gmail.com

Abstract: This work introduces a scalable and efficient topological structure for tetrahedral and hexahedral meshes. The design of the data structures aims at maximum flexibility and high performance. It provides a high scalability by using hierarchical representations of topological elements. The proposed data structure is array-based, and it is a compact representation of the *half-edge* data structure for volume elements and *half-face* data structure for volumetric meshes. This guarantees constant access time to the neighbors of the topological elements. In addition, an open-source implementation named ***OpenVolumetricMesh(OVM)*** of the proposed data structure is written in C++ using generic programming concepts.

Key words: tetrahedral mesh; hexahedral mesh; geometry modeling; data structure; generic programming

1 Introduction

With the development of the meshing techniques, tetrahedral and hexahedral meshes are widely used in many applications, including scientific visualizations, physical based simulation, and product quality evaluation. For these and many other applications, the efficiency and scalability of the data structure of volumetric meshes is sometimes critical.

Nowadays, there have been several data structures and algorithms to visualize and manipulate [1-3]. However, these data structures do not consider the memory consuming and scalability when handling large models. In [4], Alumbaugh et al. proposed an array based data structure for volumetric mesh. By extending the *half-edge* data structure to 3-dimension, Lage et al. proposed the *compact half-face* (CHF) data structure to represented the tetrahedral mesh. The CHF data structure is able to access the adjacent elements in constant time by ordering the vertices, half-edges and half-faces for tetrahedral meshes. However, it can only process the tetrahedral meshes.

In this paper, we describe the design of a scalable data structure for tetrahedral and hexahedral meshes. We also offer a complete open-source implantation of the presented data structure. The features of the data structure are following:

- Efficient in both storage and time. It requires few memory since it need not to store the topological relationships of the same element, and the access of the neighbors of the same element can be achieved in constant time through a constant pre-defined array for the special structure of tetrahedral and hexahedral meshes.
- It offers dynamic properties allowing the user to attach and detach to the volumetric mesh during running time.
- It is object-oriented using class inheritance and virtual instantiation and provides a unique interface for both tetrahedral meshes and hexahedral mesh.
- It provides local topological operations, such as face flip, split, and edge collapse operations for tetrahedral mesh.
- It is easy to understand and use for non-experts and it can be very easy to be integrated into other application projects. It offers iterators to access the elements of the mesh like C++ STL and hides the complex underlying structure with simple APIs.

Note that another open-source data structure named OpenVolumeMesh [5] is also available, our software package is developed independently and focus on processing tetrahedral and hexahedral meshes.

The remainder of this paper is organized as follows. Section 2 will introduce the design of OVM data structure. In section 3, the implementation will be

described. And finally, section 4 concludes this paper and will discuss our future work.

2 OVM Data Structure

The objective of this section is to introduce the basic design of the current OVM implementation. The source code and more detailed description of the implementation are available in [6].

2.1 Hierarchical Representation of OVM

The data structure of the OVM is designed in a Top-down way. As shown in Figure 1, the hierarchical representation of our implementations contains four levels, including tetrahedron (or hexahedron), half-face, half-edge and vertex. The tetrahedrons (or hexahedrons) describe the volumetric elements of the mesh. We will give more details of the half-face and half-edge levels in the following subsections.

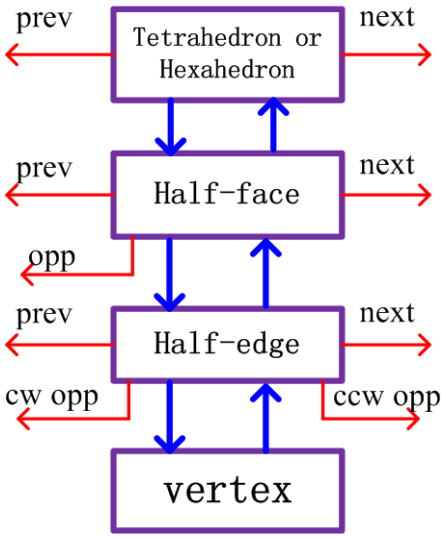

Figure 1 Hierarchical representation of OVM

2.2 Half-Edge Data Structure

In OVM, we use half-edge based data structure to represent a volumetric element (tetrahedron or hexahedron). As described in [7, 8], the *half-edge* data structure is used to represent 2-dimension manifold models. Using this representation, each physical edge is split into a pair of directed *half-edges*, and for each half edge, some of its adjacent entities are stored, such as the previous, next and opposite half edges, as shown in Figure 2(a). The corresponding face where the half edge lies is called the incident face of the half edge. Because a single tetrahedron or hexahedron is a 2d manifold, we use the half-edge data structure to represent it in our data structure. Figure 2(b) and (c) show the half-edge structure of the tetrahedron and hexahedron respectively. However, it needs not to store the links of a half edge on the same tetrahedron (or hexahedron) in our data structure using our numbering strategy to the half edges. This will greatly saves the amount of storage while it can also guarantee the constant access time to the neighbors. We will describe the number strategy in section 2.5.

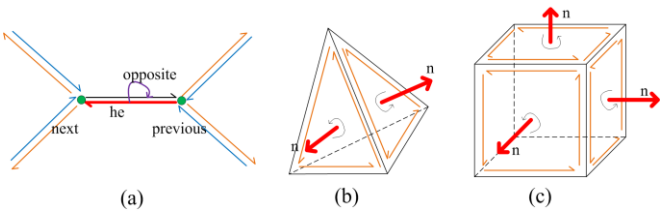

Figure 2 Half-edge data structure

2.3 Half-Face Data Structure

Similar to the *half-edge* data structure, the *half-face* data structure represents a physical face by a pair oriented half faces, as shown in Figure 3. In OVM, each half-face stores the handle (or index) of its opposite half-face. If one half-face has no opposite half-face, it is a boundary half-face and its corresponding tetrahedron (or hexahedron) is a boundary element.

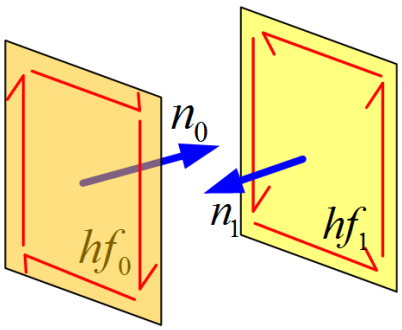

Figure 3 Illustration of a pair of *half face*

2.4 Connectivity Representation

In order to access the neighboring volumetric element, it needs to store the adjacent relationship of two neighboring elements. Actually, a half edge has two opposite half edges: one is on the same volumetric element, and the other is on the opposite half face of its incident half face, as shown in Figure 4(a). As illustrated in Figure 4(d), when looking from the reverse direction of the half edge *he*, the opposite half edge on the same volumetric element lies on the

clockwise position, so we call it clockwise opposite (abbr. *cw opp*, as shown in Figure 4(b)) half edge. And the one on the half face of its incident half face lies on the counter clockwise position, so we call it counter clockwise (abbr. *ccw opp*, as shown in Figure 4(c)) half edge.

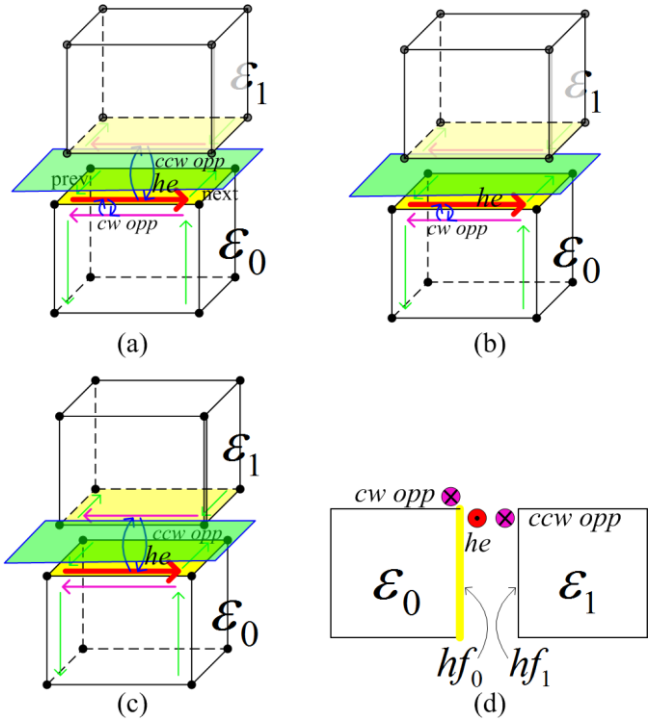

Figure 4 Two types of opposite half-edge. (a) Two types of opposite half edge (b) The opposite half edge on the same volumetric element (c) The opposite half edge on the opposite half face (d) Illustration of the two opposite half edges.

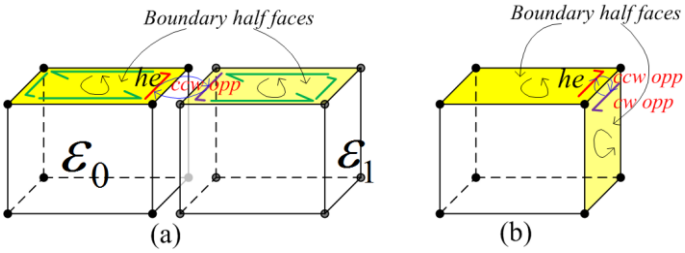

Figure 5 The *ccw opp* half edges on boundary *half faces*

If the half edge lies on a boundary half face, its *ccw opp* half edge points to the corresponding half edge of the neighboring boundary half face (as shown in Figure 5(a)). Moreover, if the neighboring boundary half face lies on the same volumetric element, the *ccw opp* half edge and *cw opp* half edge are the same (see Figure 5(b) for illustration). Using these two types of opposite half edges, it can traverse all half faces and all volumetric elements around an edge, as shown in Figure 6. We can see that the surface mesh of the model is also represented by *half-edge* data structure with these connective relationships and we unify the representations of the surface mesh and volumetric mesh in an elegant way.

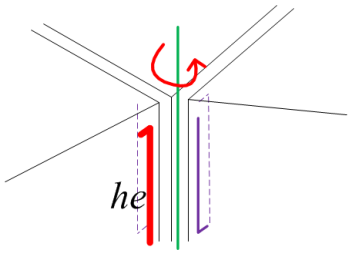

Figure 6 Traverse the half faces around an edge from a half edge

2.5 Numbering Half-Face and Half-Edge

Generally speaking, the links of a half edge should be stored for general type of volumetric element in order to achieve constant access time. But for tetrahedron or hexahedron, it does not need to store such links since the numbers of their half edges and half faces are constant. To achieve this goal, we need to number the half face and the half edges of a tetrahedron or hexahedron.

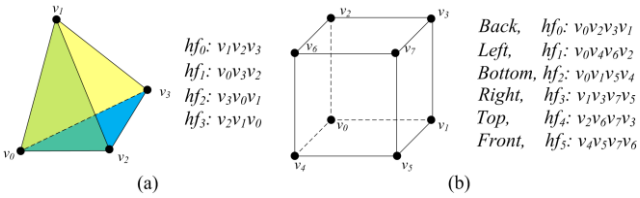

Figure 7 Numbering the half faces

There may be many numbering rules of the half faces and half edges on a tetrahedron (or hexahedron). In the following, we will take tetrahedron for example to describe our number strategy. In OVM, we number the vertices and *half faces* of a tetrahedron like Figure 7(a). Here, we use v_i and hf_i to represent the *i-th* vertex and half face of a tetrahedron respectively. Then, we number the half edges as shown in Figure 8.

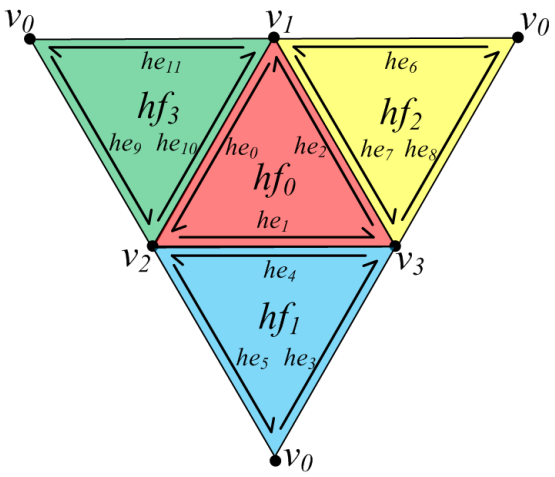

Figure 8 Numbering the half edges of a tetrahedron

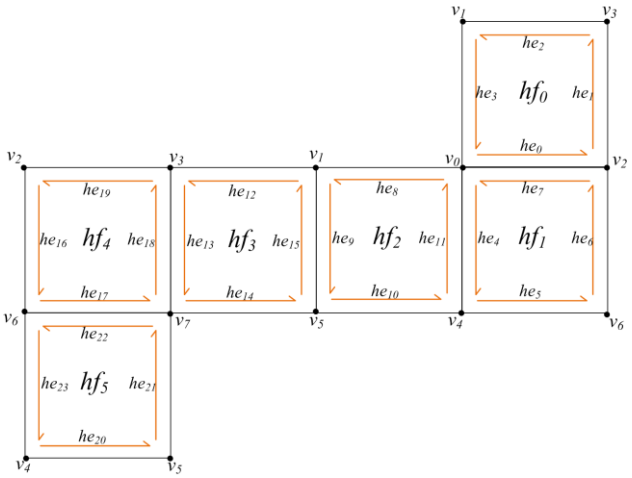

Figure 9 Numbering the half edges of a hexahedron.

For a tetrahedron, we can see that each half edge has its fixed *cw opp* half edge, e.g. the *cw opp* half edge of he_1 is he_4. So we define an array with constant length to achieve our goal. In OVM, this array is defined as,

$$TCW\ [\ ] = \{\ 10, 4, 7, 8, 1, 9, 11, 2, 3, 5, 0,\ 6\ \} \quad (1)$$

In OVM, we store the half in an array. That means the number of the half edges is *12n* if a tetrahedral mesh has *n* tetrahedrons. Then, suppose the index of a half edge is *h*, then that of its *cw opp* half edge $h_{cw}$is,

$$h_{cw} = \lfloor h/12 \rfloor \times 12 + TCW\ [h\ \%\ 12] \quad (2)$$

Using this index, we can get the information of the half edge from the half array. Since *h* can be the index of any half edge, we get constant access time to *ccw opp* half edge from (2).

Similar strategy applies to hexahedron as shown in Figure 7(b) and Figure 9. Then, the *CW* array is,

$$\begin{aligned} HCW\ [\ ] = \{\ & 7,\ 19, 12,\ \ 8, 11, 23, \\ & 16,\ \ 0,\ \ 3, 15, 20,\ \ 4, \\ & 2, 18, 21,\ \ 9,\ \ 6, 22, \\ & 13,\ \ 1, 10, 14, 17,\ \ 5\ \} \end{aligned} \quad (3)$$

and the corresponding index of the *cw opp* half edge h_{cw} is,

$$h_{cw} = \lfloor h/24 \rfloor \times 24 + HCW\ [h\ \%\ 24] \quad (4)$$

2.6 Storage

All topological elements, including vertices, half edges, half faces and tetrahedrons (or hexahedrons) are stored in array-based storage containers, and we access those using handles, e.g. indices referring to their positions within the array. As shown in Figure 1, some handles of different levels of topological elements should be stored in order to make the access can cross these levels. In OVM, the number of half face is *6n* and the number of half edges is *12n* if a tetrahedral mesh contains n tetrahedrons. As analyzed in section 2.5, we can see that there are special relationships among the half edges, half faces, and their corresponding tetrahedron in the arrays. So it does not need to store other information between these levels. Similar relationships exist in hexahedral mesh. But in order to make the access between vertex and half edge levels, , it needs to store its start vertex for a half edge, and store one of attached half edges for a vertex.

3 Implementation

The proposed data structure is implemented in C++, using generic programming ideas and the standard template library. The *BaseKernel* defines the topological element containers and property operations. The *TopologyKernel* inherits *BaseKernel* and it provides all basic topological operations and the relations of these topological elements. And we use *THKernel* inheriting *TopologyKernel* to specify the tetrahedral and hexahedral meshes. The inheritance is shown in Figure 10. In OVM, the coordinates of the vertices are stored as property instead of using explicit containers. Moreover, OVM also provides

APIs of IO to read and write MEdit files [9] and NETGEN [10] mesh files. Details of the implementation can be found in [6]. In Figure 12, we show an application of the proposed data structure.

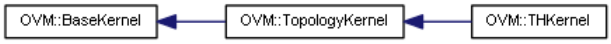

Figure 10 Kernel inheritance

Topological Operations the software package also provides some topological operations for tetrahedral mesh, such as collapsing an edge (as shown in Figure 11), flipping face, splitting an edge and splitting a tetrahedron. These operations are implemented using the basic topological operations in *TopologyKernel*, and provides a convenient way for user to modify the topology of a mesh.

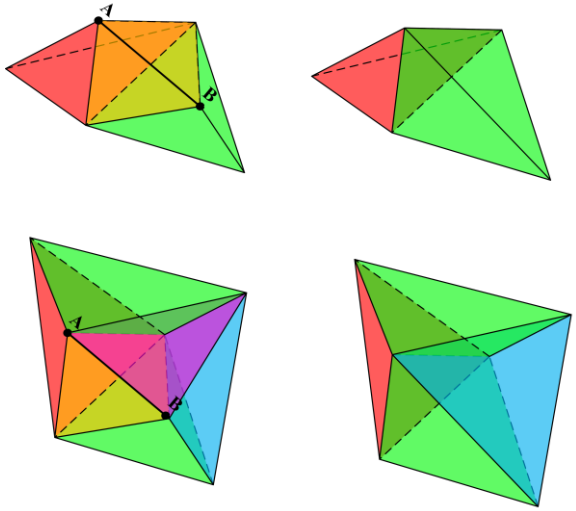

Figure 11 Collapse edge *AB*

4 Conclusions and Future Work

The OVM software package is an efficient and scalable data structure for tetrahedral and hexahedral meshes, both in time and storage. It also provides some topological operations for tetrahedral mesh. By involving C++ template concepts, it is easy to understand and use.

The limitation of OVM is it currently only supports tetrahedral and hexahedral mesh. One of our future works is to extend it to arbitrary volumetric mesh. And we are going to provide common algorithms on that structure.

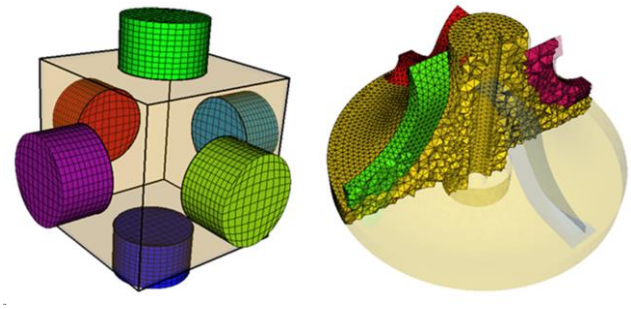

Figure 12 Visualization using OVM

References:

[1]. Lopes, H.E.L. and G. Tavares, Structural operators for modeling 3-manifolds, in Proceedings of the fourth ACM symposium on Solid modeling and applications

. 1997, ACM: New York, NY, USA. pp. 10--18.

[2]. De Floriani, L. and A. Hui, A scalable data structure for three-dimensional non-manifold objects, in SGP '03Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing

. 2003, Eurographics Association: Aire-la-Ville, Switzerland, Switzerland. pp. 72--82.

[3]. Nonato, G., et al., Topological tetrahedron characterization with application in volume reconstruction. Journal of Shape Modeling, 2005. 11(2).

[4]. Alumbaugh, T.J. and X. Jiao, Compact array-based mesh data structures, in Proceedings of the 14th International Meshing Roundtable. 2005. pp. 485--503.

[5]. http://openvolumemesh.org/.

[6]. http://openvm.org.

[7]. Campagna, S., L. Kobbelt and H.P. Seidel, Directed edges — A scalable representation for triangle meshes. Journal of Graphics tools, 1998. 3(4): pp. 1--11.

[8]. Mantyla, M., An Introduction to Solid Modeling. 1987, New York: Computer Science Press, Inc.

[9]. http://tetgen.berlios.de/.

[10]. http://sourceforge.net/projects/netgen-mesher/.