Compact Combinatorial Maps: a Volume Mesh Data Structure

Xin Feng[a], Yuanzhen Wang[a], Yanlin Weng[b,*], Yiying Tong[a,*]

[a] *Michigan State University, USA*
[b] *Zhejiang University, China*

Abstract

We propose a compact data structure for volumetric meshes of arbitrary topology and bounded valence that offers cell-face, face-edge, and edge-vertex incidence queries in constant time. Our structure is simple to implement, easy to use, and allows for arbitrary, user-defined 3-cells such as prisms and hexahedra, while remaining very efficient in memory usage compared to previous work. Its time complexity for commonly-used incidence and adjacency queries such as vertex and dart one-rings is analyzed.

Keywords: 3D mesh, Combinatorial maps, Cell complex, Half-face, Compact mesh data structure

1. Introduction

Volume meshes are now ubiquitous in solid modeling, physics-based simulation, computational science, and even rendering of translucent materials. However, the ever-increasing size and complexity of meshes impose undue stress on both memory access times and usage, especially since mesh size typically grows as a cubic function of the resolution. A data structure with small memory footprint that can efficiently handle queries of incidence and adjacency would thus benefit a wide range of applications in graphics and scientific computing in general.

While our data structure is based on the compact, array-based mesh data structure [1], we provide a simple but generic method for defining volume cell

*Corresponding authors.
Email addresses: weng@cad.zju.edu.cn (Yanlin Weng), ytong@msu.edu (Yiying Tong)

types, *complete the data structure with a list of edges, and improve incidence queries within each volume cell.*

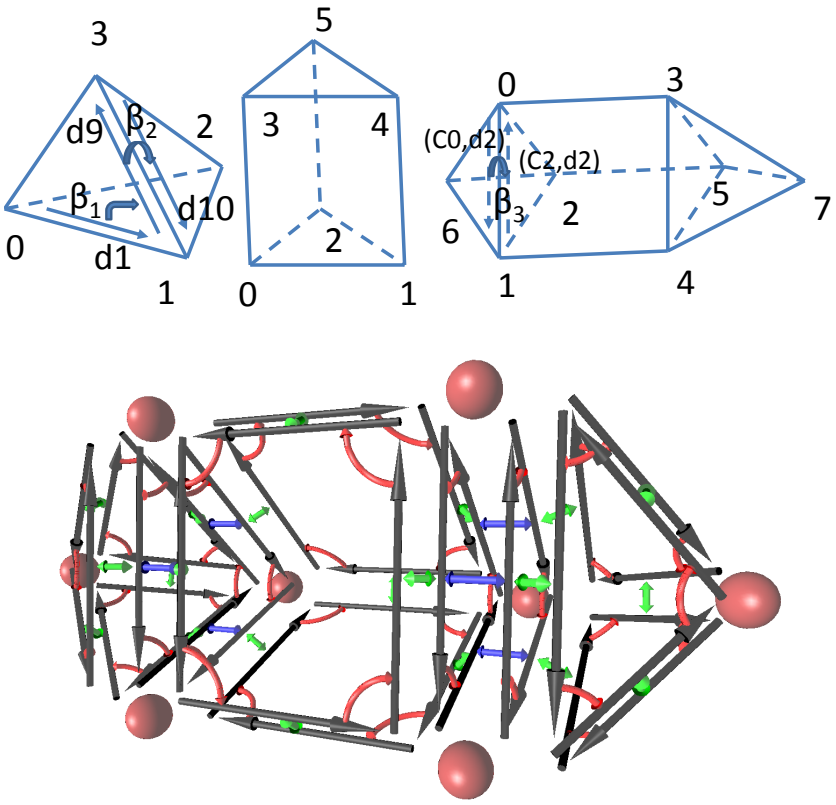

Figure 1: Upper: tetrahedron cell type; prism cell type; a mesh with 3 cells. Bottom: full set of combinatorial maps (β_1 in red, β_2 in green), and β_3 in blue) among darts. One example for each of the maps is given with the labels for the darts involved.

Related work. We limit our discussion of previous work to closely related 3D data structures— for a survey of 2D mesh data structures, see, e.g., [2, 3]. Note, however, that the 2D version of our compact combinatorial map data structure is equivalent to HEDS [1], which is known to be similar in memory usage to a number of compact implementation of half-edge data structures.

In scientific computing, 3D volumes are often assumed to be 3D manifolds, i.e., void of degenerate structures such as "shark-fin" or "hanging rod". Under this assumption, a table of mesh element connectivity that maps volume cells to their corner vertices provides complete information

about incidence among vertices, edges, faces, and volume cells. While this can be sufficient for various geometry processing algorithms[4, 5], many computational applications require constant-time upward or downward incidence queries (to access lower dimension cells from higher dimension cells, or vice versa), which cannot be achieved without auxiliary connectivity information. This requirement was referred to as *comprehensiveness* in [1]. Some applications may only require certain incidence queries, e.g., cell-face relations in ray-tracing [6], cell-vertex relations in Delaunay tetrahedralization. On the other hand, many Finite Element Method-based applications may need all incidence queries, including face-edge relations [7].

To address this issue, several data structures were proposed to store sufficient adjacency information to be comprehensive. Guibas and Stolfi [8] initially proposed a face-edge data structure. A more abstract "cell-tuple" data structure was formulated in [9] using the idea of boundary representation for a mesh, based on the fact that it is theoretically possible to "order" all the (k–1)-cells and k-cells around a (k–2)-cell on the boundary of a (k+1)-cell. Orientable quasi-manifolds can be also represented by a notion called Combinatorial Map, originally defined for polygonal meshes [10]. Combinatorial maps can be extended to generalized d-maps to encode non-orientable manifolds [11]. Beall and Shephard [12] developed a topology-based mesh data structure which stores the adjacent information in the boundary representation; however, their direct implementation of adjacent relationships requires a relatively large amount of additional memory storage. Generalized d-maps can also be compressed [13], but the adjacency information will only be restored through decompression.

Recently, a number of compact data structures have been proposed. For instance, [14] introduced a method capable of providing comprehensive connectivity information using only about 7.5 bytes per tetrahedron. However, this approach is only applicable to simplicial meshes. [1] presented a compact array-based data structure for 3D orientable manifold cell complexes. With their concept of anchored half-faces, they were able to compute incident cells in constant time. However, while they can produce incident cell representations such as an edge represented by two vertex indices, *it is not possible to find an identifier for the edge using only the proposed connectivity representation*. [15] independently developed a similar data structure which only explicitly stores the incidence information of nodes and mesh elements, with pre-defined element types. Edges and faces are still implicitly represented, but bit flags are used to ensure uniqueness. They also improved the speed of

adjacency queries with "reverse indices".

There are a number of libraries providing practical implementations of volume mesh data structures. [16] already contains an implementation of Combinatorial Maps; OpenVolumeMesh [17], released recently, is based on OpenMesh [18], which stores incidence information for cells with those with one dimension less; libMesh [19] provides a complete but *not* comprehensive connectivity description; CGoGN [20] provides an implementation of Generalized Maps. However, none of these existing implementations are optimized for storage.

Contributions. Our main contributions include:

- a concise local connectivity description of generic 3-cell (volume cell) types, suitable for both file format and data structure;
- an efficient way to store all the required combinatorial maps for darts in volume meshes;
- a straightforward way of associating attributes to k-cells ($k \in \{0, 1, 2, 3\}$); and
- a constant time complexity access to adjacency information, including *face-edge* incidence.

Note that unique edge identifiers and the face-edge incidence are the main missing components in the compact array-based mesh data structures [1] compared to our implementation. On the other hand, one can replace integer indices with memory pointers and use linked lists to make our data structure able to handle dynamic connectivity, at the cost of slightly increased memory usage, possible fragmentation and worse spatial consistency. Array-based data structure, however, are often more convenient in languages dedicated to scientific computing, such as FORTRAN. It is also easier to parallelize when distributed over several CPUs [1]. In fact, most of the afore-mentioned implementations provide the users with the option of using arrays and integers. Note that while we discuss in this paper the details and implementation of our data structure to encode orientable 3D manifolds, it can be generalized to orientable d-dimensional manifold meshes.

The rest of the paper is organized as follows. In Sec. 2, we briefly introduce the combinatorial maps data structure for volume meshes. In Sec. 3, we describe our compact array-based data structure, and briefly analyze its

space complexity. In Sec. 4, we discuss adjacency queries and show typical operations our data structures can efficiently handle, before concluding in Sec. 5.

2. Combinatorial Maps

In order to introduce the notion of combinatorial maps, we loosely follow the notation used in [16] and call k-dimensional cells *k-cells*. Hence, vertices are 0-cells, edges are 1-cells, faces are 2-cells, and volume cells (such as tetrahedra, prims, etc) are 3-cells. Two cells of different dimensions are said to be incident if one is a subset of the other. Two k-cells of the same dimension are adjacent if they share a common $(k-1)$-cell.

A combinatorial map describes the incidence and adjacency relations among cells of the mesh using a basic element called *dart*, and a group of relations between darts. For an orientable 3D manifold, a 3D dart corresponds to a cell tuple (v, e, f, c), where v is a starting vertex of an edge e that lies in a face f of 3-cell c. For 2D orientable surfaces, a 2D dart would be the same as the usual half-edge.

An abstract way to define a whole 3D combinatorial map M is to use a 4-tuple $M = (D, \beta_1, \beta_2, \beta_3)$, with:

- D is a finite set of darts;
- for $i = 1, 2, 3$, $\beta_i : D \to D$ is a mapping;
- β_1 is a permutation;
- β_2, β_3, and $\beta_1 \circ \beta_3$ are involutions, i.e., $\forall d \in D$, $\beta_2 \circ \beta_2(d) = d$, $\beta_3 \circ \beta_3(d) = d$, and $(\beta_1 \circ \beta_3) \circ (\beta_1 \circ \beta_3)(d) = d$.

Intuitively speaking, β_i maps a dart to another dart with a different i-cell and a different vertex. If we identify the darts with (v, e, f, c) in the regular cell complex description, $\beta_1((v, e, f, c)) = (v', e', f, c)$, $\beta_2((v, e, f, c)) = (v', e, f', c)$, and $\beta_3((v, e, f, c)) = (v', e, f, c')$. Note that β_1 and β_2 are the 3D analogues of a half-edge's next() and opposite() operations, respectively.

In this abstract sense, we can define k-cells by orbits $\langle S \rangle(d)$, i.e., the set of darts that can be reached by arbitrary combination of maps $m \in S$:

- the 3-cell containing d is $\langle \{\beta_1, \beta_2\} \rangle(d)$;
- the 2-cell containing d is $\langle \{\beta_1, \beta_3\} \rangle(d)$;

- the 1-cell containing d is $\langle\{\beta_2, \beta_3\}\rangle(d)$,
- the 0-cell containing d is $\langle\{\beta_1 \circ \beta_2, \beta_1 \circ \beta_3\}\rangle(d)$.

3. Compact Data Structure

3.1. Overview

File format. For a 2D polygonal mesh, the complete connectivity information can be encoded by a face list, with each entry corresponding to the list of vertices in the polygon face. However, for a polyhedral mesh, the same list of vertices can correspond to different polyhedra. For instance, an octahedron and a prism both have six vertices. As there are only a handful of k-cell types in most k-dimensional meshes used in practice, we opt to describe all the k-cell types in the header part of the file, and to describe each polyhedron by an ordered vertex list and its k-cell type.

Comprehensive data structure. All low dimensional ($\leq k-1$) relations ($\beta_1, \ldots, \beta_{k-1}$) map darts within the same k-cell. Given the type of a k-cell, we may assign each dart in that cell a local id, and the maps among the darts can be precomputed when the k-cell type. One can easily assemble a global ID for each dart by (C, d), where C is the global ID of the k-cell, and d is the local dart ID. Additional auxiliary local incidence mapping to increase efficiency can also be created for each k-cell type at a constant memory cost (independent of the mesh size).

β_k maps a dart in one k-cell C_1 to another dart in an adjacent k-cell C_2. Noticing the relation among β's, we only store β_k for one dart in the common $k-1$-cell in C_1. Thus, the size of β_k can be reduced to one dart per pair of k-cell and $k-1$-cell.

The relation between k-cells and darts is implicitly given in the way we express a global ID for each dart (C, d). The mapping from darts to vertices (0-cells) is stored in the vertex lists for k-cells, also called the element connectivity in array-based methods such as [1], denoted $Cv2V$ below. The map from each vertex to one of its darts is stored in a table, denoted by $V2D$ below.

The above information enables constant time incidence/adjacency inquiries among vertices, k-cells, and "half"-$k-1$-cells, akin to [1] except some subtle differences. However, no unique IDs are actually given to 1-cells, 2-cells, through $k-1$-cells, hence no constant time incidence inquiries involving these cells can be achieved, without additional memory cost.

We propose to build a minimal set of additional connectivity tables to provide these incidence relations crucial to real world applications. We describe them as optional, since often one may only need some of the tables in this set, although at least one of them is, in many cases, indispensable. Here we restrict our discussion to 3D. To create a unique edge identifier we use a table called $E2D$, which maps a global edge ID to one of its darts. The map from darts back to edges can be implemented through a table $V2E$ mapping a vertex to the edge starting from it with the smallest ID, as elaborated below. Similarly, but less frequently required, we assign unique face IDs through the table $F2D$, and the backward mapping by $V2F$.

3.2. Details for 3D

To illustrate the detailed actual data structure, we use as a running example the description of the simple meshes shown in Figure 1, as found in a mesh file—skipping the list of vertex coordinates since our focus is on connectivity information. As in the compact array-based half-face data structure (HFDS) [1], we leverage the fact that there are only a few types of cells typically used in engineering or graphics applications. However, unlike in HFDS, we will not limit ourselves to 3-cells used in the CFD General Notation System (tetrahedron, pyramid, prism, and hexahedron): any 3-cell type for which faces are locally defined can be specified in the header of a mesh file.

Local information within each 3-cell. Each 3-cell is treated locally as a 2-manifold cell complex, which can be represented by a local half-edge structure, i.e., a 2D combinatorial map. For a given type of 3-cell with n_v vertices, n_e edges, n_f faces:

- locally denote each vertex by v_i, with $i \in \{0, \ldots, n_v - 1\}$;
- locally label each face as $f_m = (v_i, v_j, v_k, \ldots)$, with $m \in \{0, \ldots, n_f - 1\}$.
- (optionally) locally label each of the $2n_e$ darts as $e_k = (v_i, v_j)$, with $k \in \{1, \ldots, 2n_e\}$;

Darts are indexed starting from 1, as 0 is reserved for boundaries.

The mesh file for Figure 1 would thus contain the following information:
Cell type 0 (tetrahedron):

faces	0:(0,2,1)	1:(0,1,3)	2:(1,2,3)	3:(2,0,3)		
darts	1:(0,1)	2:(1,0)	3:(0,2)	4:(2,0)	5:(0,3)	6:(3,0)
	7:(1,2)	8:(2,1)	9:(1,3)	10:(3,1)	11:(2,3)	12:(3,2)

Cell type 1 (prism):

faces	0:(0,2,1)	1:(0,1,4,3)	2:(1,2,5,4)	3:(2,0,3,5)	4:(3,4,5)	
darts	1:(0,1)	2:(1,0)	3:(0,2)	4:(2,0)	5:(0,3)	6:(3,0)
	7:(1,2)	8:(2,1)	9:(1,4)	10:(4,1)	11:(2,5)	12:(5,2)
	13:(3,4)	14:(4,3)	15:(3,5)	16:(5,3)	17:(4,5)	18:(5,4)

Cells:

type 0	C0:(1,0,2,6)	C1:(3,4,5,7)
type 1	C2:(0,1,2,3,4,5)	

In all the tables we list, the information before ":" is for illustration purposes only, and is thus not stored in memory or files. For each 3-cell type, defining only the faces would be necessary and sufficient, since we can build the darts based on faces and give them labels. We then build a lookup table for β_1 and β_2 of all darts, with $2n_e$ entries and $2n_e$ possible values in the range for each entry. In our running example, the β_1 and β_2 tables for 3-cell type 0 are

d	1	2	3	4	5	6	7	8	9	10	11	12
$\beta_1(d)$	9	3	8	5	12	1	11	2	6	7	10	4
$\beta_2(d)$	2	1	4	3	6	5	8	7	10	9	12	11

Here the rows labeled β_1 and β_2 contain the images of the darts of the same column in the rows labeled with d, e.g. $\beta_1(1) = 9$ and $\beta_2(1) = 2$.

Assuming a small number of 3-cell types compared to mesh size, these type specifications only use a negligible amount of memory. In fact, storing all the *local* incidence and adjacency information directly for improved speed only requires an additional constant memory cost. We denote local incidence mappings as follows:

- $d2f(d)$ maps a dart d to its local face ID;
- $f2d(f, i)$ is the i-th dart of the local face f;
- $d2v(d)$ maps a dart d to its starting vertex.

We use lower (resp., upper) case in the name of a map to denote whether the index is local (resp., global).

Global information. We load the connectivity table that contains, for each 3-cell, the global indices of its vertices. We denote this table by $Cv2V(C, v)$ since it maps the v-th vertex of 3-cell C to its global index V. Note that this corresponds to the usual way of storing the bare minimum connectivity information of 2D polygonal meshes in files. Similarly, one can organize the file by listing vertex lists of every 3-cell in their order of enumeration; that is, the file first lists the descriptions of 3-cell types, followed by vertex lists

of all 3-cells of the first type, the second type, etc. Once we have the 3-cell connectivity, a dart can be globally indexed by an ordered pair $D = (C, d)$, where C is the global 3-cell ID, and d is the local dart index. Note that instead of using a local face index with a starting vertex (called anchored half face) as in HFDS, we use local indices of darts; for the common case of tetrahedron meshes, this means we can cope with meshes twice as large for the same amount of memory.

To complete incidence and adjacency information in the combinatorial map, we need to construct β_3. We save space by noticing that $\beta_3 = \beta_1 \circ \beta_3 \circ \beta_1$, which means that $\beta_3(D)$ can be inferred if $\beta_3(\beta_1(D))$ is known. Thus, we only store β_3 for the *first* dart in each half face $H = (C, f)$, and denote this additional table by $H2D(C, f)$. If the application requires the use of boundary darts, their β_3 can be stored in a separate list $B2D(B)$, mapping the first dart of each boundary face B to its corresponding dart in the 3-cell adjacent to it. We also need to map from a vertex to one of its darts $V2D(v)$; but the map from a dart to its starting vertex is trivially found by $D2V(C, d) = Cv2V(C, d2v(d))$.

The tables for the 3-cell example are

β_3	C0	f0d3:(2,8)	f1d1:(0,0)	f2d7:(1,0)	f3d4:(2,0)
	C1	f0d3:(2,16)	f1d1:(3,0)	f2d7:(4,0)	f3d4:(5,0)
	C2	f0d3:(0,8)	f1d1:(6,0)	f2d7:(7,0)	f3d4:(8,0)
		f4d13:(1,2)			
B2D	BF0:(0,1)	BF1:(0,7)	BF2:(0,4)	BF3:(1,1)	BF4:(1,7)
	BF5:(1,4)	BF6:(2,1)	BF7:(2,7)	BF8:(2,4)	
V2D	V0:(0,7)	V1:(0,1)	V2:(0,4)	V3:(1,1)	V4:(1,7)
	V5:(1,4)	V6:(0,10)	V7:(1,6)		

Boundary. The map β_3 usually returns an internal dart (C, d) with $d > 0$. However, if the opposite is a boundary dart, it will return $(B, 0)$, i.e., the boundary half-face ID. We carefully choose $V2D$ so that whether a vertex V is on boundary can be determined by examining $\beta_3(V2D(V))$. Darts belonging to boundary half-face do not need to explicitly maintained in most cases .

Edge and face incidence information. If we need to use a unique edge identifier, a table for $E2D(E)$ is maintained to map an edge to one of its darts. We sort the edges in the $E2D$ table by lexicographic order of their vertices (V_{start}, V_{end}) assuming that it always points from the vertex with a smaller index to the one with a larger index. A backward mapping $D2E$ can be implemented by a table $V2E(V)$, mapping vertex V to the first edge starting

from it. We can avoid sorting the edges by using a linked list at the cost of storing another n_1 integers. The map $V2E$ would then be made to map a vertex to a linked list of edges starting from it.

If only half faces need identifiers, (C, f) can be used instead. Otherwise, a table $F2D(F)$ is required. Similar to the edge case, we can sort the faces by their first three vertices, assuming vertices are in ascending order within each face F. Then the backward mapping $D2F$ can be implemented by $V2F(V)$, mapping vertex V to the first face that has V as its smallest-indexed vertex.

For our running example, the (optional) edge tables are

E2D	E0(V0,V1):(0,2)	E1(V0,V2):(2,3)	E2(V0,V3):(2,5)
	E3(V0,V6):(0,9)	E4(V1,V2):(0,3)	E5(V1,V4):(2,9)
	E6(V1,V6):(0,5)	E7(V2,V5):(2,11)	E8(V2,V6):(0,11)
	E9(V3,V4):(2,13)	E10(V3,V5):(1,3)	E11(V3,V7):(1,5)
	E12(V4,V5):(2,17)	E13(V4,V7):(1,9)	E14(V5,V7):(1,11)
V2E	V0:0 V1:4 V2:7	V3:9 V4:12 V5:14	V6: V7:

Example table construction. The construction of most tables is straightforward since the mesh connectivity information is complete. We only give an example of how to build $E2D$ in Algorithm 1. Note that the procedure ensures that a quick counter-clockwise traversal of the edge's one-ring is possible even when it is on the boundary, and an easy boundary test through $\beta_3 \circ \beta_2(E)$.

Spatial complexity. Tetrahedron meshes are the easiest to establish comparisons between various data structures: for such meshes, we can approximate all k-cell counts n_k as a function of the number of tetrahedra n_3 and boundary faces n_b—other mesh types must be analyzed using the count of darts, and its estimated relation with k-cell numbers. Following [12], we assume the average valence of a vertex is around 4π divided by the solid angle for a vertex of an equilateral tet 0.5513, i.e., 22.8. Additionally, we assume that the average solid angles at boundary nodes are about half of the average angle. Based on these assumptions, the fact that each tetrahedron has 4 vertices and 4 faces, and Euler's formula, we have

$$22.8n_0 \approx 4n_3,\ 4n_3 + n_b = 2n_2,\ n_0 - n_1 + n_2 - n_3 \approx 0.$$

The k-cell counts are therefore

$$n_0 \approx 0.175n_3,\ n_1 \approx 1.175n_3 + 0.5n_b,\ n_2 = 2n_3 + 0.5n_b.$$

For the models shown in Table 1, these estimates are very close to the actual k-cell counts. Some of the models are shown in Figure 2, with cross-sections

Algorithm 1 Build $E2D$ table

1: init flag table $visited()$, $E \leftarrow 0$
2: **for all** non-boundary dart D **do**
3: **if** $visited(D)$ **then**
4: continue
5: **end if**
6: $D_0 \leftarrow D$
7: **while** true **do**
8: $D' \leftarrow \beta_3 \circ \beta_2(D)$ {rotate clockwise}
9: **if** $Boundary(D')$ or $D' = D_0$ **then**
10: break
11: **end if**
12: $D \leftarrow D'$
13: **end while**
14: $E2D(E) \leftarrow D$, $E \leftarrow E + 1$, $D_0 \leftarrow D$
15: **repeat**
16: $visited(D) \leftarrow true$, $vistied(\beta_2(D)) \leftarrow true$
17: $D \leftarrow \beta_2 \circ \beta_3(D)$ {rotate counter-clockwise}
18: **until** $Boundary(D)$ or $D = D_0$
19: **end for**

revealing the internal tetetrahedral structure. The memory usage for these models in OpenVolumeMesh and CGAL combinatorial maps data structures is listed in Table 2.

In the following analysis, we assume that the lowest four or more bits are sufficient to encode the local dart index or the local half face index; thus we need only one integer for (C, d) or (C, f). Alternatively, for tetrahedron meshes with fixed connectivity, we can use an integer D such that it represents $C = D/12$ and $d = D\%12$. When 3-cells are sorted by type, this method can be easily extended to cope with hybrid meshes and to include boundary darts. The memory size required for the various connectivity tables are listed below:

Table	V2XYZ	Cv2V	H2D	V2D	B2D
Space	$3n_0$	$4n_3$	$4n_3$	n_0	n_b
Table(optional)		E2D	V2E	F2D	V2F
Space		n_1	n_0	n_2	n_0

By tallying up these numbers, we find that $8n_3 + n_0 + n_b \approx 8.175\, n_3 + n_b$ integers are required for the basic tables, in par with the basic eight pointers per tetrahedron (pointing to adjacent tetrahedra and corner vertices) plus one pointer per node (to one incident tetrahedron) used to encode connectivity in Pyramid and CGAL, and close to [14]'s tetrahedron mesh structure prior to difference code compression. Data structures capable of handling generic polytope meshes require more memory space when used for simplicial meshes, e.g., Dobkin and Laszlo's structure [21] would require around $18n_3$ pointers, while radial-edge, cell-tuple, and G-map representations, as well as CGAL's combinatorial map, would use even more memory. If unique edge identifiers are needed, we require $n_0 + n_1 \approx 1.35\, n_3$ additional integers, which is more compact than the pure tet mesh encoding of [14] before difference coding.

HFDS [1] uses the same amount of basic space $(8.175\, n_3 + n_b)$. However, their encoding of a local dart (anchored face) identifier (C, f, v) uses a separate local index f for a face within the tetrahedron and a local index v of a vertex within the face. Thus, it would be less memory efficient when dealing with generic 3-cells, for example, 3-cells that have 5-edge faces or more. In addition, even in the common case of tetrahedron meshes, HFDS requires 5 bits for local indices ($f = 0$ is reserved for boundary), while we only need 4 bits, enabling us to handle meshes with 256M 3-cells with a 32-bit integer representation, instead of their 128M limit.

Furthermore, and *key to runtime efficiency*, we provide a simple way to

give edges and faces unique identifiers. As we elaborate upon next, this enables constant time incidence queries, and allows appending attributes to edges and faces, which are important in simulation and other computational tasks. The HFDS data structure does not actually provide any means to get unique adjacent edge IDs in constant time.

4. Incidence/Adjacency Queries

As our data structure can be seen as an internal representation of a combinatorial map, it can directly leverage any implementation of combinatorial maps to get incidence and adjacency information in constant time. In addition, with integer IDs, additional attributes associated to vertices, darts, half faces, cells, edges, and faces, can be directly allocated as an array with the appropriate size, making it highly efficient and flexible for static meshes. We will first give a few examples of commonly-used neighborhood constructions such as one-rings in Algorithms 2 and 3 ('.' symbol denotes member access). Assuming constant maximum valence, both algorithms run in constant time. To map a dart to a unique edge ID, we find the end vertices (V_{start}, V_{end}) with $V_{start} < V_{end}$. We then perform a linear search in $E2D$ starting from $V2E(V_{start})$, this again would terminate in constant time.

In most cases, faces do not need a unique ID, as attributes are often associated to half faces; but if needed, our $F2D$ and $V2F$ tables can be used to provide a unique face ID.

All other incidence information can be similarly assembled from the mappings between cells and darts and the mappings among darts.

5. Conclusion

We presented an efficient internal representation of combinatorial maps. All necessary components in combinatorial maps can be implemented in

model name	n_0	n_1	n_b	n_3	+V2XYZ	est.	+Edges
1mag	95,156	648,969	48,308	529,652	19,858k	18,625k	23,006k
Armadillo	189,919	1,314,767	77,704	1,085,997	39,502k	38,103k	44,634k
david	140,592	965,377	65,402	792,038	29,486k	27,824k	33,334k
dc-wt	550,770	3,819,288	224,024	3,156,497	111,286k	110,742k	125,702k
emd1590	23,419	150,930	19,540	117,736	5,346k	4,175k	6,110k
fertility	341,924	2,385,564	125,450	1,980,912	70,098k	69,438k	79,490k
neptune	358,647	2,498,975	133,476	2,073,588	73,622k	72,695k	83,442k

Table 1: Actual memory usage for a variety of meshes.

model name	OpenVolumeMesh	CGAL CM
1mag	246,284k	334,028k
Armadillo	502,028K	673,587k
david	366,988k	483,532k
dc-wt	1,402,900K	1,929,379k
emd1590	54,696K	67,789k
fertility	885,876K	1,205,862k
neptune	921,616K	1,258,291k

Table 2: Actual memory usage for the same meshes as in Table 1 using OpenVolumeMesh library and CGAL's combinatorial maps, respectively.

Algorithm 2 One-ring darts and cells around vertex V_0

Ensure: {darts} and {cells} contain darts and cells in the one-ring.

$C \leftarrow (V2D(V_0).C)$
Queue $Q.push(C)$, save C in {cells}
while Q not empty **do**
  $C \leftarrow Q.pop()$
  $\{D_i\} \leftarrow$ all darts starting at V_0 in C
  save $\{D_i\}$ in {darts}
  for all D in $\{D_i\}$ **do**
    $C \leftarrow (\beta_3(D).C)$
    if C not in {cells} **then**
      $Q.push(C)$, save C in {cells}
    end if
  end for
end while

compact form. Compared to previous work, our data structure can handle arbitrary 3-cell types, and it provides adjacency and boundary inquiries in constant time. Appending attributes to *cells of any dimension* is also straightforward.

One limitation of the compact combinatorial map data structure we described is its apparent inability to deal gracefully with dynamically changing connectivity, in particular with possible changes of 3-cell types. (On the other hand, if 3D cells are kept intact as in the case of cutting or merging meshes along faces, the mesh can be easily modified accordingly.) However, we believe that our data structure can be readily altered to efficiently handle connectivity changes as well: one could use pointers instead of integers for the IDs of 3-cells and vertices—and the last few bits of the pointer can actually be

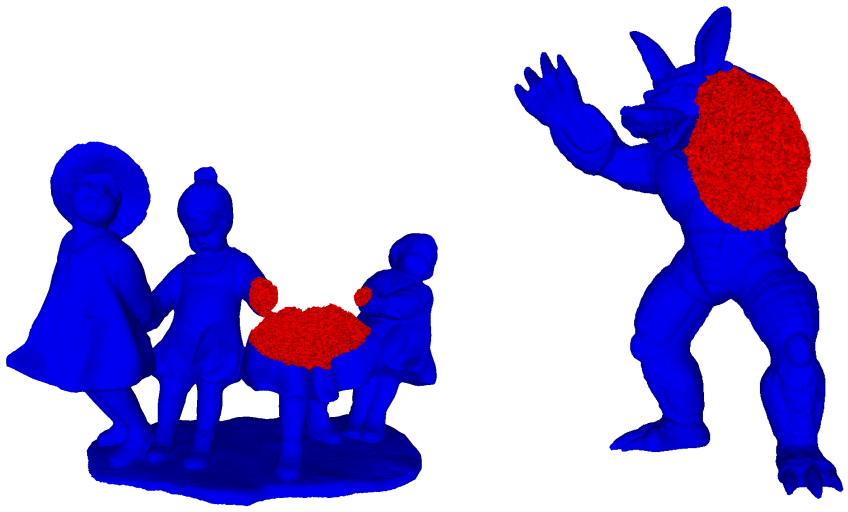

Figure 2: Some of the meshes in the statistics. The cross-sections reveal the internal tetrahedra. The surface triangles are rendered blue, and the internal triangles red.

used to encode local dart index as in the integer case. The linked list version of $V2E$ will be necessary, increasing the memory space by $n_1 = 1.175\, n_3$.

Thus, a possible research direction worth exploring is the design of admissible local connectivity changes (such as edge removal or 2-3 flip) that maintain the validity of our compact data structure. Compression of neighboring information (β_3) using difference coding after sorting the cells along space-filling curves could also lead to further reduction of memory usage. Additionally, the extension to dimension $n > 3$ could be done by encoding the local connectivity $(\beta_1, \ldots, \beta_{n-1})$ of n-cell types, and store only β_n.

Another future work would be to explore the application of the data

Algorithm 3 One-ring (internal) HF around Edge E_0

Ensure: Array {HF} is the CCW ordered one-ring.

1: $D_0 \leftarrow E2D(E_0)$, $D \leftarrow D_0$
2: **repeat**
3: $\quad C \leftarrow (D.C)$, $d \leftarrow (D.d)$
4: $\quad$ save $(C, d2f(d))$ and $(C, d2f(\beta_2(d)))$ in {HF}
5: $\quad D \leftarrow \beta_2 \circ \beta_3(D)$ {rotate counter-clockwise}
6: **until** $Boundary(D)$ or $D = D_0$

structure in tasks involving volume data, such as 3D field design, solid texturing [22, 23], etc.

Acknowledgments

This work was supported in part by NSF grants CCF-0936830, IIS-0953096, CMMI-0757123 and CCF-0811313. Yanlin Weng was supported by the NSF of China (No. 61003145) and the National High-Tech R&D program of China (No. 2012AA011503). An earlier version of this work appeared in the Proceedings of Computational Visual Media Conference 2012 [24].

References

[1] T. J. Alumbaugh, X. Jiao, Compact Array-Based Mesh Data Structures, in: B. W. Hanks (Ed.), Engineering, IMR '05, Springer Berlin Heidelberg, 2005, pp. 485–503. doi:10.1007/3-540-29090-7_29. 1, 2, 3, 4, 6, 7, 12

[2] D. Sieger, M. Botsch, Design , Implementation , and Evaluation of the Surface mesh Data Structure, in: International Meshing Roundtable, 2011, pp. 533–550. 2

[3] S. P. Serna, A. Stork, D. W. Fellner, Considerations toward a Dynamic Mesh Data Structure, in: SIGRAD Conference, 2011, pp. 83–90. 2

[4] T. J. Tautges, T. Blacker, S. A. Mitchell, The Whisker Weaving Algorithm: a Connectivity-Based Method for Constructing All-Hexahedral Finite Element Meshes, International Journal for Numerical Methods in Engineering 39 (19) (1996) 3327–3349. doi:10.1002/(SICI)1097-0207(19961015)39:19<3327::AID-NME2>3.0.CO;2-H. 3

[5] P. Murdoch, The spatial twist continuum: A connectivity based method for representing all-hexahedral finite element meshes, Finite Elements in Analysis and Design 28 (2) (1997) 137–149. doi:10.1016/S0168-874X(97)81956-7. 3

[6] P. Muigg, M. Hadwiger, H. Doleisch, E. Gröller, Interactive volume visualization of general polyhedral grids., IEEE transactions on visualization and computer graphics 17 (12) (2011) 2115–24. doi:10.1109/TVCG.2011.216. 3

[7] M. Desbrun, E. Kanso, Y. Tong, Discrete Differential Forms for Computational Modeling, in: G. Bobenko, A. Schröder, P. Sullivan, J. Ziegler (Ed.), Discrete Differential Geometry, Springer, 2008, pp. 287–324. doi:10.1007/978-3-7643-8621-4_16. 3

[8] L. Guibas, J. Stolfi, Primitives for the manipulation of general subdivisions and the computation of voronoi, ACM Trans. Graph. 4 (2) (1985) 74–123. doi:10.1145/282918.282923. 3

[9] E. Brisson, Representing geometric structures in d dimensions: topology and order, in: SCG 89 Proceedings of the fifth annual symposium on Computational geometry, Vol. 9, ACM Press, 1989, pp. 218–227. doi:10.1145/73833.73858. 3

[10] J. R. Edmonds, A combinatorial representation for polyhedral surfaces, Notices Amer Math Soc 7 (1960) 646. 3

[11] P. Lienhardt, Topological models for boundary representation : a comparison with n-dimensional generalized maps, Computer-Aided Design 23 (1) (1991) 59–82. doi:10.1016/0010-4485(91)90082-8. 3

[12] M. W. Beall, M. S. Shephard, A general topology-based mesh data structure, International Journal for Numerical Methods in Engineering 40 (9) (1997) 1573–1596. doi:10.1002/(SICI)1097-0207(19970515)40:9<1573::AID-NME128>3.0.CO;2-9. 3, 10

[13] S. Prat, P. Gioia, Y. Bertrand, Connectivity compression in an arbitrary dimension, The Visual Computer 21 (8-10) (2005) 876–885. doi:10.1007/s00371-005-0325-z. 3

[14] D. K. Blandford, G. E. Blelloch, D. E. Cardoze, C. Kadow, Compact Representations of Simplicial Meshes in Two and Three Dimensions, International Journal of Computational Geometry and Applications 15 (1) (2005) 3–24. 3, 12

[15] W. Celes, G. H. Paulino, R. Espinha, A compact adjacency-based topological data structure for finite element mesh representation, International Journal for Numerical Methods in Engineering 64 (11) (2005) 1529–1556. doi:10.1002/nme.1440. 3

[16] G. Damiand, Combinatorial maps, in: CGAL User and Reference Manual, 4.0 Edition, CGAL Editorial Board, 2012. 4, 5

[17] OpenVolumeMesh - A Generic and Versatile Index-Based Data Structure for Polytopal Meshes, http://www.openvolumemesh.org/ (2012). 4

[18] M. Botsch, S. S. S. B. L. Kobbelt, OpenMesh - a generic and efficient polygon mesh data structure, Structure 2002. 4

[19] B. S. Kirk, J. W. Peterson, R. H. Stogner, G. F. Carey, libmesh: a c++ library for parallel adaptive mesh refinement/coarsening simulations, Eng. with Comput. 22 (3) (2006) 237–254. doi:10.1007/s00366-006-0049-3. 4

[20] Combinatorial and Geometric modeling with Generic N-dimensional Maps, http://cgogn.u-strasbg.fr/Wiki/index.php/CGoGN (2012). 4

[21] D. P. Dobkin, M. J. Laszlo, Primitives for the manipulation of three-dimensional subdivisions, in: Proceedings of the third annual Symposium on Computational Geometry, SCG '87, ACM, New York, NY, USA, 1987, pp. 86–99. doi:10.1145/41958.41967. 12

[22] X. Zhao, B. Li, L. Wang, A. Kaufman, Texture-guided volumetric deformation and visualization using 3d moving least squares, Vis. Comput. 28 (2) (2012) 193–204. doi:10.1007/s00371-011-0635-2. 16

[23] G.-X. Zhang, S.-P. Du, Y.-K. Lai, T. Ni, S.-M. Hu, Sketch guided solid texturing, Graphical Models 73 (3) (2011) 59–73. doi:10.1016/j.gmod.2010.10.006. 16

[24] X. Feng, Y. Wang, Y. Weng, Y. Tong, Compact combinatorial maps in 3d, in: Proceedings of Computational Visual Media Conference, Springer, 2012. 16