SOT: Compact representation for tetrahedral meshes

Topraj Gurung and Jarek Rossignac
School of Interactive Computing, College of Computing,
Georgia Institute of Technology, Atlanta, GA
{topraj,jarek}@cc.gatech.edu

ABSTRACT
The *Corner Table* (CT) promoted by Rossignac et al. provides a simple and efficient representation of triangle meshes, storing 6 integer references per triangle (3 vertex references in the V table and 3 references to opposite corners in the O table that accelerate access to adjacent triangles). The Compact Half Face (CHF) proposed by Lage et al. extends CT to tetrahedral meshes, storing 8 references per tetrahedron (4 in the V table and 4 in the O table). We call it the *Vertex Opposite Table* (*VOT*) and propose a sorted variation, *SVOT*, which does not require any additional storage and yet provides, for each vertex, a reference to an incident corner from which an incident tetrahedron may be recovered and the star of the vertex may be traversed at a constant cost per visited element. We use a set of powerful wedge-based operators for querying and traversing the mesh. Finally, inspired by tetrahedral mesh encoding techniques used by Weiler et al. and by Szymczak and Rossignac, we propose our *Sorted O Table* (*SOT*) variation, which eliminates the V table completely and hence reduces storage requirements by 50% to only 4 references and 9 bits per tetrahedron, while preserving the vertex-to-incident-corner references and supporting our wedge operators with a linear average cost.

Keywords
Modeling, Tetrahedral Meshes, Data Structures, Storage

1. INTRODUCTION

1.1 Problem
Unstructured tetrahedral meshes are used in numerous applications, including finite element analysis [1, 11, 26], interpolation of samples [55], shape reconstruction [7], and medical image analysis [32, 47].

A variety of data structures and operators have been proposed [2, 10, 22, 21, 30, 43] for storing the connectivity of the tetrahedral mesh and for caching additional information that simplifies and accelerates common queries and traversal operators needed to support applications. In some applications, typical meshes contain millions of tetrahedra [29] and this complexity continues to increase. Therefore, it is desired to strive for further **reduction of the storage cost** associated with these data structures.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling (SPM '09), October 4-9, 2009, San Francisco, CA.
Copyright 2009 ACM 978-1-60558-711-0/09/10…$10.00.

Several tetrahedral mesh compression schemes have been proposed [31, 61]. Some support progressive refinements [45] or streaming [8, 33, 66]. Unfortunately, the compressed format they offer is not suitable for traversing, simplifying [14, 22, 21, 67], refining [41], or improving [57, 42, 58] the mesh. Thus, an effective representation scheme is needed that provides efficient support for **random access operators that traverse the mesh** and which may be constructed efficiently from other (possibly compressed) formats or updated to reflect mesh modifications.

1.2 Foundation
The *Corner Table* [49] provides a simple and efficient representation of triangle meshes, storing 6 integer references per triangle (3 references to the vertices of a triangle are stored as consecutive entries in the V table and 3 references to opposite corners are stored in the corresponding entries of the O table). The Corner Table has been extended by Bischoff and Rossignac [8] and by Lage et al. [37] to support tetrahedral meshes. The resulting *Vertex Opposite Table* (*VOT*), which is called the Compact Half Face (CHF) in [37], stores 8 references per tetrahedron (4 references to the vertices of a tetrahedron—one per corner—stored as consecutive entries in the V table and 4 references to opposite corners stored as corresponding entries in the O table). References to opposite corners cached in the O table are used to provide constant cost access to adjacent tetrahedral and their bounding cells. We illustrate it on a mesh of two tetrahedra in Fig. 1 right, where we have numbered the corners. The corner pairs (1,5), (2,7) and (3,6) each share the same vertex. Corners 0 and 4 are opposites of each other. The other corners do not have opposites. For each such border corner c, we set O[c]=c.

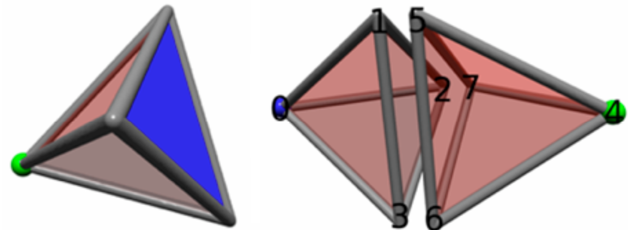

Fig. 1: Left: The blue face f(c) is the opposite face of corner c (green vertex). Right: Corners 0 and 4 (green and blue balls) are opposites: O[0]=4 and O[4]=0. The two tetrahedra have been shrunk for clarity, but are in fact adjacent to each other: f(g)=f(b).

c	0	1	2	3	4	5	6	7
V[c]	0	1	2	3	4	1	3	2
O[c]	4	1	2	3	0	5	6	7

Table 1: VOT for Fig. 1, right.

1.3 Contributions

For each ***corner*** c of each tetrahedron, the *VOT* stores the references V[c] to the corresponding vertex and the reference O[c] to the opposite corners in an adjacent tetrahedron, if one exists. It does not store any references from vertices to corners or to incident tetrahedra. Because such **vertex-to-incident-corner or tetrahedron references** are important in some applications, we introduce a ***Sorted VOT*** (***SVOT***) representation, which associates with each vertex v a reference to one of its incident corners V(c). Remarkably, SVOT caches this **vertex-to-incident-corner reference without any additional storage**. This "trick" is accomplished by rearranging the order in which the n_T tetrahedra and their corners are stored in the VOT: When the mesh has n_V vertices, for any index $v<n_V$, vertex v is the location of the first corner of the vth tetrahedron.

We provide a **linear cost algorithm** for computing the O table of the *VOT* from the V table and an algorithm for converting a *VOT* into a *SVOT* that has expected linear time complexity.

Finally, we propose another extension of the *VOT,* which we call the ***Sorted O Table*** (***SOT***). The *SOT* further reduces the storage requirements to only **4 references** and **9 *service bits* per tetrahedron**, while preserving the direct vertex-to-incident-corner access. We hide the *service bits* in the integer representation of the references and hence, store the SOT using $4n_T$ integers. To accomplish this saving, we eliminate the need for storing the V table entirely. The vertex references V(c) of a corner is inferred from information stored in the O table and the service bits.

To support the constructions of our SVOT and SOT and the traversal of the tetrahedral mesh, we have developed a set of powerful corner and **wedges operators** that extend *half-edge* operators proposed by Lage et al. [37]. A wedge is the association of an edge with an incident tetrahedron and with a bounding vertex. As they operate on wedges, a set of our operators **mimic** the effect of corresponding triangle-mesh corner operators (next, previous, opposite, left, right, and swing) that operate on the triangle-mesh boundary of the star of the starting vertex of a wedge. We include the details of an **efficient implementation** of these operators, which have constant cost for VOT and SVOT, and average constant cost for SOT (where their expected cost is proportional to the valence of the base vertex).

Finally, we provide **examples** that demonstrate the ease of use of these data structures and operators for retrieving—at a constant (or average constant for SOT) cost per element—the **tetrahedra around an edge**, the **star of a vertex**, and the **connected component of the boundary** of the mesh.

2. BACKGROUND AND PRIOR ART

Vertices are stored in the *G Table* and hence implicitly associated with ***integer references*** in $[0,n_V-1]$ where n_V is the **number of vertices**. G[v] is the location of the vertex with reference v. To simplify exposition, we use the term vertex and symbol v to define the reference or the location, depending on the context.

Mesh elements (triangles, quads, polygons, tetrahedra) may be represented by ordered sets of vertex references. When all elements have the same vertex-count k, these references may be stored as consecutive integer entries in the *V Table*. Although that information is sufficient for processing individual elements, it is not sufficient for providing efficient access to neighboring elements, which is important in many applications (such as curvature calculation or connected component identification). To accelerate these queries, a variety of data structures have been proposed for caching additional incidence and adjacency information.

Several data structures for polygonal meshes operate on edge-uses, which are each the association of an edge with a bounding vertex and with an incident face. Examples include Baumgart's *Winged-Edge* [4, 3]; Guibas and Stolfi's *Quad-Edge* [30], Mantyla's *Half-Edge* [44], and Lienhardt's *dart* [40].

Extensions of these schemes to non-manifold and non-regularized complexes include Weiler's *Radial-Edge* [68]. Extensions to three-cells arrangements include Dobkin and Laszlo's *Facet-Edge* [22, 21] extension of the *Quad-Edge* and Lopes and Tavares's *Handle-Face* [43] extension of the *Half-Edge*. Extensions to n-complexes include Paoluzzi et al. model [46]. Extensions to n-manifolds include Brisson's *Cell-Tuple* [9] generalizing the *Quad-Edge* and *Facet-Edge*. Extensions to more general n-dimensional complexes include Lienhardt's *n-Generalized Maps* [39], Rossignac and O'Connor's proposed *Selective Geometric Complexes* [52], and De Floriani and Hui's *Non-Manifold Indexed data structure with Adjacencies* [25]. For additional discussion on data structures for simplicial complexes, we refer the interested reader to [24] and [35].

Although these techniques are suitable for representing triangle meshes and support efficient query and traversal operators [8], they require significantly more storage [36, 51] than the representations discussed below, which have been optimized for triangular or tetrahedral meshes. For example, the *quad-edge* stores 3 references per edge-use (1 to a vertex and 2 to other edge-uses), which amounts to $9n_T$ references.

More compact representations customized for triangle meshes include Campagna's et al. *Directed Edges* [12] and Kallmann and Thalmann's *Star-Vertices* representation [36], which, for each vertex, stores only the sorted list of references to its neighbors, and hence requires a total of $3n_T$ references, plus a few indices for locating the lists boundaries.

The *Corner Table* [54, 50, 49], which is the basis of the proposed solution, represents the connectivity of a manifold triangle mesh of n_t triangles by two tables of $3n_t$ integers each. The V-table lists the triangle/vertex incidence, such that the 3 vertices bounding a triangle are consecutive and listed in an order that is compatible with a consistent orientation of the mesh. Hence, each entry to the V[c] table represents a ***corner*** c associating a face f with a bounding vertex. The O-table stores the integer reference of the ***opposite corner***. A set of ***corner operators***, listed below and illustrated in Fig. 2, may be trivially implemented from the information contained in these two tables:

```
int t(int c) {return int(c/3);}                        // triangle of c
int v(int c) {return V[c];}                             // vertex of c
int o(int c) {return O[c];}                             // opposite
int n(int c) {if ((c%3)==2) return c–1; else return c+1;}// next in t(c)
int p(int c) {return n(n(c));}                          // previous corner
int l(int c) {return o(p(c));}                          // tip on left
int r(int c) {return o(n(c));}                          // tip on right
int s(int c) {return n(l(c));}                          // next around v(c)
```

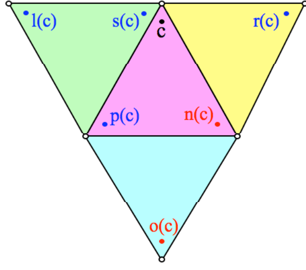

Fig. 2: The corner operators for a triangle mesh.

Several data structures have been customized for tetrahedral meshes [62, 43, 2, 30, 10, 26, 27, 48, 5, 56]. They, or their variations, have been used for mesh generation and processing [23, 28, 34, 63, 38, 17].

The Corner Table has been extended by Bischoff and Rossignac's *VOT* [6] and the equivalent Lage et al. *Compact Half Faces* (*CHF*) [37] independently extend the *Corner Table* to tetrahedral meshes. The VOT requires 8 references per tetrahedron (4 for vertex references and 4 for opposite corners). An index to these tables identifies a particular **corner** of a particular tetrahedron. Therefore, the V and O tables each have $4*n_T$ entries. As was done for the triangle meshes, the corners of each tetrahedron are ***consecutive*** in the *VOT* (the 4 corners for the ith tetrahedron are stored at entries 4*i+j, where j = 0,1,2,3) and are listed in an order that is consistent with the orientation of the tetrahedron (the vertices of corners j=1,2,3 appear counter-clockwise from the vertex of corner j=0). When two tetrahedra share a face, the two corners, b and c, that do not lie on the shared face (f(b)=f(c)) are opposite (Fig. 1) and we cache this relation: O[b]=c and O[c]=b.

Lage et al. [37] propose options to cache additional references, including a reference from each vertex to an incident tetrahedron.

Weiler et al. [69] encode tetrahedral meshes in strips. Using a greedy stripification algorithm [69], they obtain an average of 4.3 tetrahedra per strip. They also discuss a variation that builds longer strips by allowing duplicate vertex entries in a strip. A tetrahedral strip is stored as an ordered list of vertex references such that any 4-tuple of consecutive vertices bound a tetrahedron. Furthermore, each tetrahedron in a strip is face-adjacent to its predecessor and successor in the strip (when they exist). A strip with k tetrahedra has 3+k entries in V (one per vertex) and has 2+2k external faces, for which they store opposites in the O-table. Hence, to produce a regular structure, they use 3 tables, each of size (3+k), one for the vertices, and two for the opposites, resulting in storage cost of 3(3+k) per strip, which results in $3n_T + 9n_S$ total storage, where n_T is the number of tetrahedra and n_S is the number of strips. Since there are 2+2k external faces, but 6+2k locations in the opposites table, 4 locations in the opposites table do not contain any information. A bit per corner is used to identify the beginning and ending of strips. Assuming 4.3 tetrahedra per strip, the total storage cost is $(3n_T+9(n_T/4.3))= 5.1n_T$, i.e. an average of 5.1 references per tetrahedron.

Several techniques were proposed for compressing tetrahedral meshes [61, 31, 13], for streaming them [70, 6, 33], and for transferring refinements or simplifications [60, 45, 67]. But random access operators that traverse the mesh (such as those developed for triangle meshes [71]) are not supported in these approaches.

Several representation schemes were developed to support multi-resolution tetrahedral meshes [64, 15, 14, 35, 19, 16, 18, 20, 59]. In more direct relevance to our work, Szymczak and Rossignac [61] propose the Grow&Fold compression algorithm for tetrahedral meshes. In their Grow&Fold compression algorithm, Szymczak and Rossignac compute a ***Tetrahedron Spanning Tree*** (***TST***) and encode it using 3 bits per tetrahedron. Except at the root, the traversal of the *TST* enters a tetrahedron T by a face f. Each one of the 3 bits associated with T corresponds to a different face of T (excluding f) and indicates whether T has a child in the *TST* incident upon that face. Because the *TST* does not encode the complete connectivity, Szymczak and Rossignac also store two bits per border face of the *TST* to control a folding process that reconstructs the full connectivity. These bits indicate whether the face is a border face of the mesh and, when not, select one of its edges for folding. Since there are roughly $2n_T$ such border faces, their scheme requires about $7n_T$ bits to encode the connectivity of the mesh. It is impractical to require a full traversal of the *TST* to identify the parent of a tetrahedron and to require the executing of the folding algorithm to recover the references of the other two adjacent tetrahedra. Hence, these references must be cached. Furthermore, a tetrahedron may have 0, 1, 2, or 3 children in the *TST*. We must be able to locate these children in constant time, without having to traverse the rest of the *TST*.

3. WEDGE OPERATORS

The wedge operators, which we use for building SVOT and SOT and in our traversal algorithms are based on the following ***auxiliary bit-manipulation operators*** (as in [37])

```
boolean even(int c) {return ((c&1)==0);}          // c is even
int m4(int c) {return c&0x3;}                     // c modulo 4
int d4(int c) {return c>>2;}                      // c divided by 4
int x4(int t) {return t<<2;}                      //t multiplied by 4
int fc(int c) {return x4(d4(c));}                 // first corner of t(c)
```

and on the following ***corner operators***

```
int T(int c) {return d4(c);}                      // tetrahedron of c
int N(int c) {return fc(c)+m4(m4(c)+1);}          // next corner in T(c)
int P(int c) {return fc(c)+m4(m4(c)+3);}          // previous corner
int V(int c) {return V[c];}                       // vertex of c
int O(int c) {return O[c];}                       // opposite corner
boolean B(int c) {return O(c)==c;}                // f(c) is a border face
```

To traverse the mesh and to access the various elements (vertices, edges, faces, and tetrahedra) and their neighbors in an orderly fashion, we use the concept of a ***wedge***, which is the association of a ***base vertex*** v with an incident edge e and an incident tetrahedron t. It corresponds to a "half-edge-use" [44] and to the *half-edge* [37]. In our figures, a wedge w defined by the triplet (v,e,t) is shown as a colored arrow along the half of e away from v. For simplicity, w.a denotes the **starting corner** of w and w.b its **ending corner**.

We define 10 wedge operators (Fig. 3). Consider an interior vertex v. The boundary of its star is a triangle mesh M homeomorphic to a sphere. To each wedge w=(v,e,t) corresponds a corner c of M. We have named some of our wedge operators so that they preserve this correspondence. For example, as shown in Fig. 3 left, n(w) corresponds to n(c), p(w) corresponds to p(c), and o(w) corresponds to o(c). The triangle corners are not shown to avoid clutter, but are on the face to which the corresponding wedge arrow points. Similarly, as shown in Fig. 3 right, l(w) corresponds to l(c) and r(w) corresponds to r(c). Note that we

have underscored here the names of the triangle-mesh corner operators to better distinguish them from the corresponding tetrahedral mesh wedge operators.

Several wedge operators do not have corresponding corner operators. The ***mirror wedge*** m(w) returns Brisson's *swap₀*. The ***cross wedge*** operator k(w) returns the wedge whose edge is not adjacent to the edge of w, and that appears to go left, when seen from an observer aligned with (the arrow used to show) w.

We **do not store wedges** explicitly, since storing them would require a significant amount of memory. Instead, we represent a current wedge by an ordered pair of two references to corners of the same tetrahedron. Clearly, such an ordered pair of corners defines the triplet (v,e,t) of a staring vertex v, a supporting edge e, and an incident tetrahedron t.

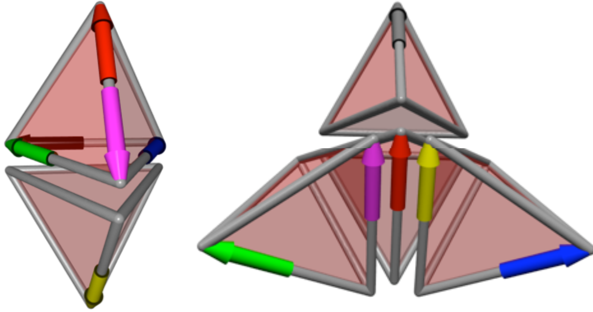

Fig. 3: Wedge w is shown as a red arrow. *Left* (basic wedge operators): *next* wedge n(w) in green, *previous* wedge p(w) in blue, *mirror* (reversed) wedge m(w) in magenta, *cross* wedge k(w) in brown (back), and the *opposite* wedge o(w) in yellow in an adjacent tetrahedron. *Right* (derived wedge operators): *swing right* wedge sl(w) in magenta, *swing left* wedge, sr(w) in yellow, *right* wedge r(w) in green, *left* wedge l(w) in blue, forward wedge f(w) in grey. Note that the left wedge when viewed from this direction appears right relative to w.

The next, mirror and opposite wedge operators are similar to the next, mate, and radial half-edge operators respectively provided by Lage et al. [37].

We use the following function to create a wedge (object).

```
Wedge w(int a, int b) {return new Wedge(a,b);}
```

We need only three ***basic wedge operators*** (Fig 3 left):

```
Wedge m(Wedge w) {return w(w.b,w.a);}                    // mirror
Wedge n(Wedge w) {                                       // next
 int nc=m4(m4(w.b)+(even(w.a)?3:1));
 if(nc==m4(w.a)) {nc=m4(m4(w.b)+2);};
 return w(w.a,fc(w.a)+nc);}
Wedge o(Wedge w) { int na; int oc=O(w.b);                // opposite
 if(oc==c) {return null;};
  if(V(N(oc))==V(w.a)) {na=N(oc);}
  else if(V(P(oc))==V(w.a)) {na=P(oc);}
  else {na=N(N(oc));};
  return w(na,oc);};
```

The **mirror** wedge operator m() simply reverses the starting and ending corner indices. The **next** wedge operator n(), based on whether corner w.a is odd or even, returns the proper next wedge. If we represent a tetrahedron by the corners 0, 1, 2 and 3, then by construction and our adopted tetrahedron orientation, (1,2,3) looks counter-clockwise from 0, (0,2,3) looks clockwise from 1, (0,1,3) looks counter-clockwise from 2 and (0,1,2) looks clockwise from 3. Hence, we can see that even corners c_e view face $f(c_e)$ as counter-clockwise and odd corners c_o view face $f(c_o)$ as clockwise. The n() operator returns the next counter-clockwise wedge, i.e. corner n(w).b is counter-clockwise relative to corner w.b when viewed from corner w.a. The **opposite** wedge operator o() uses the corner operator O(w.b) to identify the corner b opposite to w.b in an adjacent tetrahedron. However, there are **three wedges having b as starting vertex**. We return the one whose end-corner is at a vertex that is not bounding the tetrahedron of w. We could accelerate o() by caching the ***rotation number*** indicating which of the three wedges has b as starting vertex. Although we do not cache the rotation number for the VOT and SVOT, we will cache that rotation number for the SOT, as discussed in Section 5.

Also note that o() returns null when the wedge has no opposite. This software engineering decision facilitates the implementation of derived wedge operators by deferring the testing of border conditions. Our implementation of the m(), n() and o() operators (not shown here) returns null if a null wedge is received as input.

From these three basic wedge operators, we construct seven convenient ***derived wedge operators*** listed below and shown in Fig. 3 right. For practice, we encourage the reader to visually verify their implementation using Fig. 3.

```
Wedge p(Wedge w) {return n(n(w));}            // previous wedge
Wedge l(Wedge w) {return o(n(w));}            // left wedge
Wedge r(Wedge w) {return o(p(w));}            // right wedge
Wedge k(Wedge w) {return n(m(p(w)));}         // cross wedge
Wedge f(Wedge w) {return o(m(w));}            // forward wedge
Wedge sl(Wedge w) {return n(l(w));}           // swing left wedge
Wedge sr(Wedge w) {return p(r(w));}           // swing right wedge
```

3.1 Using wedge operators

To demonstrate the use of the wedge operator, we discuss here the VOT implementation of three common algorithms.

3.1.1 Swinging around an edge

Here, we explain how to visit all tetrahedra incident on an edge. Given a wedge w, we iteratively use the *swing left* operator until we return to w or reach a null wedge (meaning, we are outside of the mesh). If we reach the w, we are done. If we reach a null wedge, we repeat the process, but swinging right. The wedges we visit identify the tetrahedra incident on the given wedge and provide a starting reference for processing them (not included).

```
Wedge swing(Wedge w) {                      //swing around wedge
   Wedge sw = w(w.a, w.b);                  //start wedge
   while(w!=null){                          //not a boundary wedge
      if(eqW(sw, w)) break;                 //reached start wedge
      w = sl(w);}                           //swing left
  if(w==null) {                             //start wedge
     w = w(sw.a, sw.b);
     while(w!=null){ w = sr(w);  }}}        //swing right
boolean eqW(Wedge u, Wedge w) {             //equal wedges
       return (u.a==w.a && u.b==w.b);}      //u and w equal?
```

3.1.2 Visiting components of the boundary

Here, we explain how to traverse connected components of the boundary of a tetrahedral mesh using the VOT.

We can traverse a *shell* (connected manifold component) of a triangle mesh by starting from a corner c and recursively walking

to neighboring triangles using the l() and r() *Corner Table* operators. We can either mark the visited triangles and use recursive calls, or mark visited vertices and triangles to avoid most recursive calls, as done in the Edgebreaker traversal [53].

Note that to each corner b of a border triangle of a tetrahedron mesh corresponds a unique wedge w=wedge(a,b). Using the analogy between wedge and corner operators, we can execute the above traversal algorithm and visit the faces of the connected component of the boundary of the mesh. All we need is the wedge counterparts lc(), and rc() of the l() and r() operators. We provide their implementation below, along with oc().

```
Wedge rc(Wedge w){return swing(p(m(w)));}      //right boundary
Wedge lc(Wedge w){return swing(n(k(w)));}       //left boundary
Wedge oc(Wedge w){return swing(m(k(w)));}         //opposite
Wedge swing(Wedge w){                  //swing around w until we..
 while(true){ if(B(k(w).a) break; w = sr(w);}       //.. hit boundary
 return k(w);}                                  //return proper wedge
```

3.1.3 *Visiting the TST*

Here we describe how to visit a Tetrahedron Spanning Tree, which is used in several compression solutions [61, 31].

We use the *right*, *left*, *opposite* and *forward* wedge operators. A temporary array of bits or flags is maintained to record the visited status of all tetrahedra. (We use the most significant bit of the Vertex Table to store them.) The recursive version of the code is provided below.

```
void dfs(Wedge w) {                              //depth first traversal
 if(w!=null && !VisitedT(T(w.a))) {                  //if tet not visited
  setVisitedT(T(w.a), true);                         //set visited status
  dfs(r(w)); dfs(l(w));                          //visit right and left tets
  dfs(o(w)); dfs(f(w));}                   //visit opposite and forward tets
```

4. SVOT

4.1 Motivation

All algorithms that we have encountered may be easily expressed by looping through or manipulating tetrahedra, corners (or equivalently their opposite oriented faces) or wedges. For example, one may find all tetrahedra that lie inside a given ball visiting all tetrahedra and accessing their corners and testing the corresponding vertices for inclusion in the ball. Temporary flags could be used to avoid testing the same vertex more than once. Similarly, one may find the tetrahedron with the sharpest edge by looping through all tetrahedra and for each one, by looping through its six wedges.

However, some developers prefer to have **direct access from a vertex to its star**, because some of their algorithms operate directly on vertices (not through corners) or because some of their auxiliary data structures refer directly to vertices. A natural solution [37] is to add a **vertex-to-corner lookup table** C, such that C[v] contains the index of corner c, such that V[c]=v. This approach requires storing additional n_V references. The *SVOT* solution described below provides the same information though a constant cost function call C(v) and avoids storing the C table.

4.2 Proposed SVOT solution

To provide constant cost access to a corner C(v) for each vertex v, and this without additional storage, we reorder the tetrahedra and their corners in the VOT so the corner C(v) incident upon vertex v may be simply computed as 4*v. This mapping works for most meshes. However, for thin meshes, we may need a slightly more complex special mapping, as shown in Fig. 4 which we discuss in Section 4.3.4.

4.3 SVOT construction algorithm

We explain here how to compute SVOT from the V table alone in linear time. The process involves 3 steps:

1) CONSTRUCTION: Compute O
2) MAPPING: Establish a **mapping,** M(t)=v, between each tetrahedron t and a bounding vertex v so that **no two tetrahedra map to the same vertex**.
3) SORTING: Reorder the *VOT*

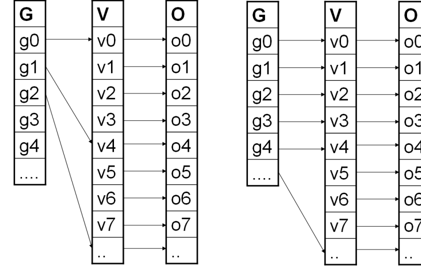

Fig. 4: Left: General SVOT mapping. Right: Special mapping.

4.3.1 Construction

The O-table does not need to be archived since it may be recomputed from the V-table when the V table is loaded. We discuss here four algorithms for rebuilding the O-table from the V-table. Their timings on two meshes of different complexity are listed in Table 2.

The ***all-pairs*** algorithm (trivial, yet impractical for large values of n_T) has asymptotic computational complexity $O(n_T^2)$. It considers each pair of different corners c and b: if f(b)=f(c), then b and c are opposite. We simply set O[b]=c and O[c]=b.

The ***tuple-sort*** algorithm, which is $O(n_T \log n_T)$, first constructs a table of 4-tuples (v_1,v_2,v_3,c), where the v_i are vertices of f(c) sorted so that $v_1<v_2<v_3$, and then sorts the table. Consecutive entries (v_1,v_2,v_3,b) and (v_1,v_2,v_3,c) with matching first 3 references define opposite corners b and c. Entries with no matching triplets define border corners with no opposite.

The ***bin-sort*** algorithm, which has complexity $O(n_V d^2)$ where d is the maximum vertex valence, avoids the above global sort by using v_1 to bin all the 4-tuples incident upon vertex v_1. All quadruplets in a bin are then sorted and used as above. Since typically d is small, we use a trivial $O(d^2)$ sort, which of course may be replaced with an $O(d \log d)$ sort if desired.

The ***hash-sort*** algorithm, which has expected $O(n_T)$ computational complexity, uses hashing to sort the c-values of the above 4-tuples using the 3-tuples (v_1,v_2,v_3) as the key. Given a good hash key, hash table lookups can be performed in constant time. The ***hash-sort*** algorithm is used by Lage et al. [37] where they use an associative container to construct the O table.

Our ***bbin-sort*** algorithm was inspired by [65, 66]. It has $O(n_T)$ time and $O(n_T+n_V)$ space complexity. First, with each vertex u, we associate a bin B_u of wedges emanating from u. To do so, for each corner c, we visit the 3 wedges of T(c) that start at c and add them to $B_{V(c)}$. This process has linear complexity. Then, for each vertex u, we (i) bin-sort the wedges of B_u into edge-bins $B_{u,v}$ and (ii) process the edge-bins.

(i) To bin-sort the wedges, we proceed as follows. Assume that we have w wedges in B_u. Using the Euler formula, we have t=w/3 tetrahedra and assume we have e edges incident upon u. We construct a hash injective function, h_e, that maps integers in {0..n_V-1} into integers in {0..e-1} and use it to bin-sort [37] each wedge of B_u into the edge-bin $B_{u,e}$ associated with edge(u,h_e(V(w.b))). We can construct the mapping h_e in linear time O(e) and linear space O(e+n_V) by maintaining appropriate lookup data structures. Also, note that $B_{u,e}$ is a temporary data structure constructed for each vertex u. The cost of storing all $B_{u,e}$ is O(t).

(ii) We process an edge-bin, $B_{u,v}$ as follows. Assume that it has s entries. This implies that there are s tetrahedra and assume it has f faces incident upon edge(u,v). We use a hash injective function h_f, that maps integers in {0..n_V-1} into integers in {0..f-1}. We can construct the mapping h_f in linear time O(f) and linear space O(f+n_V) by maintaining appropriate lookup data structures. For each wedge w of $B_{u,v}$, for each corner c of T(w) that is not in {w.a,w.b}, we bin-sort c into a face-bin $B_{u,v,x}$, where x is the fourth vertex of T(w). $B_{u,v,x}$ is a temporary data structure constructed for each (u,v) vertex pair. The cost of storing all $B_{u,v,x}$ is O(s). Now each face-bin $B_{u,v,x}$ contains the opposite corners of face(u,v,x). When it has a single entry, c, then we set O[c]=c. When it has two entries, c, and d, we set O[c]=d and O[d]=c.

Note that each wedge and each face is hashed only once and the number of wedges and faces are linear functions of n_T. Hence, the total number of hashing steps is O(n_T). The final pass that sets the O table is also linear in the number of faces. Consequently, the expected running cost of ***bbin-sort*** is O(n_T). Additionally, the B_u, $B_{u,e}$, $B_{u,v,x}$ data structures each require O(n_T), O(t) and O(s) space, which is bounded by O(n_T). Constructing the hash injective function requires additional O(e+n_V) and O(f+n_V) space. Therefore, the expected space cost of ***bbin-sort*** is O(n_T+n_V).

Tetrahedra	BBin-sort	Hash-sort	Tuple-sort	Bin-sort
482,867	1,192ms	2,087ms	2,490ms	6,253ms
1,239,990	3,230ms	5,897ms	7,295ms	16,368ms

Table 2: Performance tests for various O-table rebuilding algorithms. Tests were done on a Macbook with 2GB memory and 2.1Ghz dual-core processor. The naive all-pairs algorithm is impractical.

4.3.2 Mapping

The *mapping phase* involves three steps: ***Initialization***, ***Traversal***, and ***Termination***.

To represent the mapping M() computed in the *mapping phase*, we use a temporary table M which stores the vertex M[t] associated with tetrahedron t. We traverse the TST (as discussed in 3.1.3) and, as we enter a new tetrahedron t through a face f, associate T with the reference to its ***tip vertex*** (the vertex not bounding f), unless that vertex has already been associated with another tetrahedron. This idea is similar to the association of the tip vertex of each *type-C* triangle in the Edgebreaker compression scheme [54] for triangle meshes. Unfortunately, unless we take special precautions, this simple idea may not always work, because the traversal may associate each tetrahedron incident upon a vertex v with a vertex other than v, leaving an orphan vertex. To eliminate the possibility of orphan vertices, we propose a special **initialization step**, which **guarantees** that this approach produces a correct mapping (where each vertex is associated with a different tetrahedron).

Initialization: During the initialization step, we pick a ***seed*** tetrahedron S so that **none of its vertices bounds a border face**.

Let c be the first corner of the seed tetrahedron S. We set M[S]=V(c) and **mark (as visited) all vertices of S**.

Note that this approach assumes that a suitable seed exists. Finding S, when it exists is trivial. However, it must be noted that the proposed approach may not work on thin meshes or crusts [28] in which each tetrahedron has at least one external face. We address such issues at the end of the section. Furthermore, our approach requires performing the proposed reordering process for each connected component of the mesh. For multiple components, we need multiple seed tetrahedra. After selecting a seed tetrahedron S, we mark S and start a depth first traversal of the TST with S as root and T(O(c)) as the first child, where c is the first corner of S.

We use three arrays of auxiliary tables: visitedV[v] keeps track of visited vertices, visitedT[t] keeps track of visited tetrahedra, whichCorner[t] stores the corner c in tetrahedron t such that M[T(c)]=V(c).

Traversal: During the traversal step, we reach a new unvisited tetrahedron t by arriving from a parent tetrahedron through the opposite face f(b) of a corner b. We mark t as visited. If the vertex V(b) has not yet been visited, we mark it as visited, set M[t]=V(b), and store in whichCorner[T(b)] corner b.

***Termination*:** We match the three tetrahedra incident upon seed S (T(O(c)) where c is not the first corner of S) with 3 vertices of S, as shown in Fig. 5. Given the precaution we took with the selection of the seed (S has four adjacent tetrahedra), the other 3 tetrahedra adjacent to S have been reached while coming from tetrahedra other than S and hence will not be associated with a vertex (since their tip vertex was marked with S during initialization and is no longer available to be associated with them).

Since we performed a depth first traversal starting from S, we visit all face connected tetrahedra which implies we visit all the vertices.

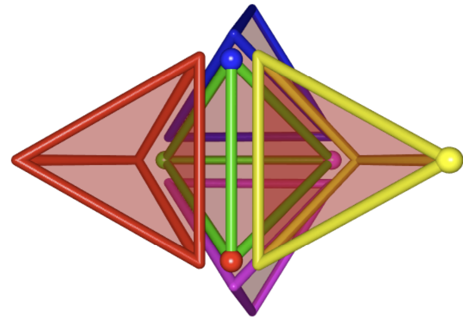

Fig. 5: The seed tetrahedron S (shown in green) with the corner c (shown as a green ball), its first neighbor T(O(c)) (shown in yellow), and the other three adjacent neighbors of S (shown in red, blue and magenta). Each tetrahedron shown is matched with a vertex of the same color. The other tetrahedra are not shown.

4.3.3 Sorting

We sort the tetrahedra in the order of M[T] and write them into a new copy of V, performing a permutation of the corners of each tetrahedron so that the remembered corner b (stored in whichCorner[T]) is listed first as corner of index zero in T. Finally, we recompute the O table as explained in 4.3.1.

The sorting discussed above requires $O(n_T)$ time and $O(n_T+n_V)$ temporary space for marking tetrahedra and vertices. The sorting needs to be performed only once, since its result (i.e. the sorted V-table) may be archived for future uses. As this sorting is a permutation where each element knows the location of the bin it wants to be in the sorted order, it requires $O(n_T)$ time instead of the traditional $O(n_T \log n_T)$ that is associated with sorting.

4.3.4 Special cases of thin meshes

It was pointed out in the mapping phase that an internal seed tetrahedron S might not exist. Should an internal tetrahedron not exist, we use an alternate method. Initially choose any tetrahedron as the seed tetrahedron S. We place S as the first tetrahedron in the V table. Let the four vertex references for S be v_0, v_1, v_2 and v_3. We swap the vertices in the geometry table. We swap v_0 with the 0th entry in the geometry table, v_1 with the 1st entry in the geometry table, likewise for v_2 and v_3. We then reorder the rest of the tetrahedra as described in the construction section for the general case. Now, the vertex-to-corner mapping is as follows: for i<4, the ith vertex maps to the ith corner. For i>=4, the ith vertex maps to the $((i-3)*4)^{th}$ corner. This mapping has been shown in Figure 4, right.

4.4 Reaching for the stars

Given an integer reference v to a vertex, the *SVOT* gives us direct access to the corresponding corner C(v), using:

```
int C(int v) {return x4(v);}                    //x4(v) = v*4
```

The auxiliary operator `x4` was introduced in Section 3. The corner and wedge operators for the *VOT* work without modification on the *SVOT*.

Since the ith vertex is mapped to the $(4*i)^{th}$ corner in the SVOT, to visit the star of the ith vertex, we simply call star(4*i). Using this, we can traverse all tetrahedra incident on a vertex by performing a depth first traversal utilizing the right, left and opposite wedge operators. It is similar to depth first traversal but we do not use the forward wedge operator. If we are given the corner c then we recursively visit all the right, left and opposite wedges of w where w.a=c and w.b = N(c).

```
void star(int c) { Wedge w = w(c, N(c)); star(w);} //star of corner c
void star(Wedge w) {                               //star traversal
 if(w!=null && !VisitedT(T(w.a))) {               //if tet not visited
  setVisitedT(T(w.a), true);                       //set visited status
  process(T(w.a));                                 //process the tetrahedron
  star(r(w)); star(l(w)); star(o(w));}}            //visit neighbors
```

5. SOT

The *VOT* and our *SVOT* variation each store 8 references per tetrahedron (4 to vertices and 4 to opposite corners). We discuss here an approach to reduce this storage to 4 references and 9 ***service bits*** per tetrahedron (2 bits per opposite corner = 8 bits per tetrahedron and 1 bit for visited state).

5.1 High level description

The *Sorted Opposite Table* (*SOT*) uses the same O-table as the one produced in *SVOT*, **but does not store the V table at all.** Therefore, the resulting *SOT* has no references to vertices. How then is it possible to find vertex references V(c) of a corner c? The solution comes from a combination of three ideas.

1) Because the O table is sorted in *SVOT*, to each vertex v corresponds a ***matching*** tetrahedron t_v=v of which the first corner is incident on v. Note that such tetrahedra are easily **recognized** because their index t is less than the number of vertices n_V. A slightly modified test is used for thin meshes.

2) By construction of the SVOT, in the **star** of every vertex v, there is a ***matching*** tetrahedron t_v = v.

3) Starting from any corner c, we can visit the *star* of its vertex V(c), even though we do not yet know the index of v. On average, we need to visit 13.3 tetrahedra of the star of v, before finding a ***matching*** tetrahedron, but the associated performance cost may be justified by the reduction of storage, and hence of page faults.

To visit the *star*, our operators use the O-table and the service bits. Two of these bits per corner are used to indicate the rotation number (as described in section 3 for the o() wedge operator).

To traverse the *star*, we use a tetrahedral version of the corner table traversal of the triangle mesh that bounds the star (as explained in section 4.4). We use one *service bit* per tetrahedron to remember which ones were visited. We perform a second pass to erase them.

5.2 Construction of SOT:

We first construct the *SVOT*. We then construct the *SOT* by eliminating the V table in the *SVOT*. In the O table, we store the two *service bits* per corner to encode the *rotation number*, which provides information about the relative orientation of two face adjacent tetrahedra.

Specifically, consider two tetrahedra t_1 with corner c and t_2 with corner O(c) that share a common face f(c). Imagine that we do not know the vertex references for the individual corners in f(c). There are three ways in which we can "glue" t_1 and t_2 at f. We can rotate t_1 clockwise, counter-clockwise, or not at all. Correspondingly, the *rotation number* rn(c) for corner c is either be 0, 1 or 2. The rotation number is computed by using the V table in SVOT. We cache the rotation number, rn(c) for each corner c as the most significant bits of each entry in the opposites table. The code below computes the rotation number for a given corner.

```
int RotationNumber(int c) {          //rotation number for corner c
 Wedge w = w(O(c), N(O(c)));                          //create a wedge
 if(V(N(c))==V(w.b)) return 0;                  //no rotation required
 if(V(N(c))==V((n(w)).b)) return 1;              //1 rotation required
 return 2;}                                     // 2 rotations required
```

Note that the *SOT* maintains the property of the *SVOT* where the ith vertex v_i maps to the ith tetrahedron T_i and the corner c_i=4*i is incident on the vertex v_i. The special case of thin meshes also generalizes easily to the *SOT*.

5.3 Determining vertex references:

We explain here in details how the vertex for an arbitrary corner b can be inferred from the O table of the SOT and from the *service bits*.

The idea is to traverse the ***star*** (see Section 4.4) of V(b) to determine the corners $\mathbf{b_i}$ incident on V(b). We must do that of course without knowing V(b), since V(b) is the desired result. The traversal stops when we find a ***matching*** tetrahedron $t<n_V$.

Finding the vertex reference can require that we visit at most d tetrahedra, where d is the valence of the vertex v. Our experiments indicate that we need to visit, on average, about 13.3 tetrahedra.

5.4 Wedge operators on SOT

In the SOT, we need to redefine two operators, the V() corner operator and the o() wedge operator. All other corner and wedge operators from the *VOT* remain the same. In the original *VOT* implementation, the *o*() wedge operator utilizes the V Table, but since the *SOT* doesn't store the V table, we rely on the *rotation number* to determine the proper o() wedge operator. The o() wedge operator takes constant steps for each call. The V() operator traverses the star to determine the vertex reference. When computing the star of the vertex, we need to maintain a ***visited state*** for each tetrahedron. We store it as the most significant bit of O[c], where c is the first corner in its tetrahedron. To unmark the tetrahedra that were visited, we use the star traversal again. The code for these two operators is provided below.

```
int V(int c) {                              //returns SOT vertex id
  Wedge w = w(c, N(c));                     //create wedge w
  return StarV(w);}                         //utilize star to get vertex id

int StarV(Wedge w) {                        //star to determine vertex id
  int rv = -1;                              //default value
  if(w!=null) {                             //if wedge exists
   int t = T(w.a);                          //tetrahedron id
   if(!VisitedT(t)) {                       //if tetrahedron not visited
    setVisitedT(t, true);                   //mark as visited
    if(t<v && m4(w.a)==0) {rv = t;}         //if first corner and t<|G|
    if(rv==-1) v = StarV(r(w));             //visit right wedge
    if(rv==-1) v = StarV(l(w));             //visit left wedge
    if(rv==-1) v = StarV(o(w));}}           //visit opposite wedge
  return rv;}                               //return vertex id


Wedge o(Wedge w) {                          //SOT opposite operator
  if(w==null) return null;                  //wedge does not exist
  if(O(c)==c) return null;                  //wedge does not exist

int c = w.b;                                //corner c
  Wedge ow = w(O(c), N(oc));                //no rotation
  if(rn(c)==1) ow = n(ow, tm);              //1 rotation
  if(rn(c)==2) ow = p(ow, tm);              //2 rotation

Wedge cw = w(c, N(c));                      //no rotation
  if(cw.b==w.a) {return w(ow.b, ow.a);}     //no aligning
  if(n(cw, tm).b==w.a) { return w(p(ow, tm).b, ow.a);}//align next
return w(n(ow, tm).b, ow.a);}               //align with prev
```

6. CONCLUSION

The *VOT* representation of tetrahedral meshes requires 8 references per tetrahedron. We propose two variations: (1) The *SVOT* affords references from a vertex to one of its incident corners without increasing storage. It permits to access all incident and adjacent cells to a corner, a vertex, or a tetrahedron with a constant cost per cell. (2) The *SOT* further reduces storage cost to 4 references and 9 service bits per tetrahedron, but makes the computational cost of accessing neighboring cells proportional to the valence of a common vertex.

To facilitate the use of these new representations, we introduce a small set of powerful wedge operators for querying and traversing the tetrahedral mesh and provide efficient implementations that work directly off the *VOT or SOT*. These wedge operators are a natural and intuitive extension to tetrahedral meshes of the familiar *Corner Table* operators originally developed for triangle meshes. We illustrate their power by providing reasonably simple source code for several common algorithms that process tetrahedral meshes.

7. REFERENCES

[1] H. Allik and T. J. R. Hughes, *Finite element method for piezoelectric vibration*, International Journal for Numerical Methods in Engineering, 2 (1970), pp. 151-157.

[2] F. Aurenhammer, *Voronoi diagrams- a survey of a fundamental geometric data structure*, ACM Comput. Surv., 23 (1991), pp. 345-405.

[3] B. G. Baumgart, *A polyhedron representation for computer vision*, *Proceedings of the May 19-22, 1975, national computer conference and exposition*, ACM, Anaheim, California, 1975.

[4] B. G. Baumgart, *Winged edge polyhedron representation*, Stanford University, 1972.

[5] M. W. Beall and M. S. Shephard, *A General Topology-based Mesh Data Structure*, International Journal for Numerical Methods in Engineering, 40 (1997), pp. 1573-1596.

[6] U. Bischoff and J. Rossignac, *TetStreamer: Compressed Back-to-Front Transmission of Delaunay Tetrahedra Meshes*, *Proceedings of the Data Compression Conference*, IEEE Computer Society, 2005.

[7] J.-D. Boissonnat, *Shape reconstruction from planar cross sections*, Comput. Vision Graph. Image Process., 44 (1988), pp. 1-29.

[8] M. Botsch, M. Pauly, C. Rössl, S. Bischoll and L. Kobbelt, *Geometric modeling based on triangle meshes*, *Course Notes, ACM SIGGRAPH 2006*, ACM Press, 2006.

[9] E. Brisson, *Representing geometric structures in d dimensions: topology and order*, *Proceedings of the fifth annual symposium on Computational geometry*, ACM, Saarbruchen, West Germany, 1989.

[10] E. Bruzzone and L. D. Floriani, *Two data structures for building tetrahedralizations*, Vis. Comput., 6 (1990), pp. 266-283.

[11] J. C. Caendish, D. A. Field and W. H. Frey, *An apporach to automatic three-dimensional finite element mesh generation*, International Journal for Numerical Methods in Engineering, 21 (1985), pp. 329-347.

[12] S. Campagna, L. Kobbelt and H.-P. Seidel, *Directed edges-A scalable representation for triangle meshes*, Journal of Graphics Tools, 3 (1998), pp. 1-11.

[13] D. Chen, Y. J. Chiang, N. Memon and X. Wu, *Geometry compression of tetrahedral meshes using optimized prediction*, European Conference on Signal Processing (2005).

[14] P. Chopra and J. Meyer, *TetFusion: an algorithm for rapid tetrahedral mesh simplification*, *Proceedings of*

the conference on Visualization '02, IEEE Computer Society, Boston, Massachusetts, 2002.
[15] P. Cignoni, D. Constanza, C. Montani, C. Rocchini and R. Scopigno, *Simplification of Tetrahedral meshes with accurate error evaluation*, *Proceedings of the conference on Visualization '00*, IEEE Computer Society Press, Salt Lake City, Utah, United States, 2000.
[16] P. Cignoni, L. D. Floriani, P. Magillo, E. Puppo and R. Scopigno, *Selective Refinement Queries for Volume Visualization of Unstructured Tetrahedral Meshes*, IEEE Transactions on Visualization and Computer Graphics, 10 (2004), pp. 29-45.
[17] CUBIT, *CUBIT Mesh Generation Toolkit*, Tech. report Sandia National Laboratories (2001).
[18] B. Cutler, J. Dorsey and L. McMillan, *Simplification and improvement of tetrahedral models for simulation*, *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, ACM, Nice, France, 2004.
[19] E. Danovaro, L. d. Floriani, M. Lee and H. Samet, *Multiresolution Tetrahedral Meshes: An Analysis and a Comparison*, *Proceedings of the Shape Modeling International 2002 (SMI'02)*, IEEE Computer Society, 2002.
[20] E. Danovaro, L. D. Floriani, P. Magillo, E. Puppo, D. Sobrero and N. Sokolovsky, *The Half-Edge Tree: A Compact Data Structure for Level-of-Detail Tetrahedral Meshes*, *Proceedings of the International Conference on Shape Modeling and Applications 2005*, IEEE Computer Society, 2005.
[21] D. P. Dobkin and M. J. Laszlo, *Primitives for the manipulation of three-dimensional subdivisions*, *Algorithmica*, 1989, pp. 3-32.
[22] D. P. Dobkin and M. J. Laszlo, *Primitives for the manipulation of three-dimensional subdivisions*, *Proceedings of the third annual symposium on Computational geometry*, ACM, Waterloo, Ontario, Canada, 1987.
[23] H. Edelsbrunner, *Geometry and Topology for Mesh Generation*, Cambridge University Press, 2001.
[24] L. D. Floriani and A. Hui, *Data structures for simplicial complexes: an analysis and a comparison*, *Proceedings of the third Eurographics symposium on Geometry processing*, Eurographics Association, Vienna, Austria, 2005.
[25] L. D. Floriani and A. Hui, *A scalable data structure for three-dimensional non-manifold objects*, *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, Eurographics Association, Aachen, Germany, 2003.
[26] R. V. Garimella, *Mesh data structure selection for mesh generation and FEA applications*, *International Journal for Numerical Methods in Engineering*, 2002, pp. 451-478.
[27] R. V. Garimella, *MSTK - A Flexible Infrastructure Library for Developing Mesh Based Applications*, *Proceedings of 13th International Meshing Roundtable*, 2004, pp. 203-212.
[28] R. V. Garimella and M. S. Shephard, *Tetrahedral Mesh Generation With Multiple Elements Through the Thickness*, *International Meshing Roundtable*, 1995, pp. 321-333.
[29] B. Gregorski, M. Duchaineau, P. Lindstrom, V. Pascucci and K. I. Joy, *Interactive view-dependent rendering of large isosurfaces*, *Proceedings of the conference on Visualization '02*, IEEE Computer Society, Boston, Massachusetts, 2002.
[30] L. Guibas and J. Stolfi, *Primitives for the manipulation of general subdivisions and the computation of Voronoi*, ACM Trans. Graph., 4 (1985), pp. 74-123.
[31] S. Gumhold, S. Guthe and W. Straser, *Tetrahedral mesh compression with the cut-border machine*, *Proceedings of the conference on Visualization '99: celebrating ten years*, IEEE Computer Society Press, San Francisco, California, United States, 1999.
[32] U. Hartmann and F. Kruggel, *A Fast Algorithm for Generating Large Tetrahedral 3D Finite Element Meshes from Magnetic Resonance Tomograms*, *Proceedings of the IEEE Workshop on Biomedical Image Analysis*, IEEE Computer Society, 1998.
[33] M. Isenburg, P. Lindstrom, S. Gumhold and J. Shewchuk, *Streaming compression of tetrahedral volume meshes*, *Proceedings of Graphics Interface 2006*, Canadian Information Processing Society, Quebec, Canada, 2006.
[34] B. Joe, *GEOMPACK - A Software Package for the Generation of Meshes Using Geometric Algorithms*, *Advances in Engineering Software*, 1991, pp. 325-331.
[35] K. I. Joy, J. Legakis and R. MacCracken, *Data Structures for Multiresolution Representation of Unstructured Meshes*, *Hierarchical Approximation and Geometric Methods for Scientific Visualization*, 2002.
[36] M. Kallmann and D. Thalmann, *Star-vertices: a compact representation for planar meshes with adjacency information*, Journal of Graphics Tools, 6 (2001), pp. 7-18.
[37] M. Lage, T. Lewiner, H. Lopes and L. Velho, *CHF: A Scalable Topological Data Structure for Tetrahedral Meshes*, *Proceedings of the XVIII Brazilian Symposium on Computer Graphics and Image Processing*, IEEE Computer Society, 2005.
[38] LaGrit, *LaGriT - Los Alamos Grid Toolbox*, Tech. report Los Alamos National Laboratory (1995).
[39] P. Lienhardt, *N-dimensional Generalized Combinatorial Maps and Cellular Quasi-Manifolds*, *International Journal of Computational Geometry & Applications*, 1994, pp. 275-324.
[40] P. Lienhardt, *Subdivisions of n-dimensional spaces and n-dimensional generalized maps*, *Proceedings of the fifth annual symposium on Computational geometry*, ACM, Saarbruchen, West Germany, 1989.
[41] A. Liu and B. Joe, *Quality local refinement of tetrahedral meshes based on 8-subtetrahedron subdivision*, Math. Comput., 65 (1996), pp. 1183-1200.
[42] Y. Liu and J. Snoeyink, *A comparison of five implementations of 3d Delaunay tessellation*, *Combinatorial and Computational Geometry, MSRI series*, 2005, pp. 439-458.
[43] H. Lopes and G. Tavares, *Structural operators for modeling 3-manifolds*, *Proceedings of the fourth ACM symposium on Solid modeling and applications*, ACM, Atlanta, Georgia, United States, 1997.

[44] M. Mantyla, *Introduction to Solid Modeling*, W. H. Freeman & Co., 1988.

[45] R. Pajarola, J. Rossignac and A. Szymczak, *Implant sprays: compression of progressive tetrahedral mesh connectivity*, *Proceedings of the conference on Visualization '99: celebrating ten years*, IEEE Computer Society Press, San Francisco, California, United States, 1999.

[46] A. Paoluzzi, F. Bernardini, C. Cattani and V. Ferrucci, *Dimension-independent modeling with simplicial complexes*, ACM Trans. Graph., 12 (1993), pp. 56-102.

[47] J. Pescatore, L. Garnero and I. Bloch, *Tetrahedral finite element meshes of head tissues from MRI for the MEG/EEG forward problem*, *12th Scandinavian Conference on Image Analysis*, 2001, pp. 71-80.

[48] J. F. Remacle, B. K. Karamete and M. S. Shephard, *Algorithm Oriented Mesh Database*, Proceedings of the 9th International Meshing Roundtable (2000), pp. 349-359.

[49] J. Rossignac, *3D Mesh Compression*, in C. Hansen and C. Johnson, eds., *The Visualization Handbook*, Academic Press, 2006.

[50] J. Rossignac, *Surface simplification and 3D geometry compression*, in Goodman and O'Rourke, eds., *The Handbook of Discrete and Computational Geometry (2nd edition)*, CRC Press, 2004.

[51] J. Rossignac, *Through the cracks of the solid modeling milestone*, in S. Coquillart, W. Strasser and P. Stucki, eds., *From object modelling to advanced visualization*, Springer Verlag, 1994, pp. 1-75.

[52] J. Rossignac and M. O'Connor, *SGC: A Dimension-independent Model for Pointsets with Internal Structures and Incomplete Boundaries*, *Geometric Modeling for Product Engineering*, 1989, pp. 145-180.

[53] J. Rossignac, A. Safonova and A. Szymczak, *3D Compression Made Simple: Edgebreaker on a Corner-Table*, *3D Compression Made Simple: Edgebreaker with Zip&Wrap on a Corner-Table*, IEEE Computer Society, 2001.

[54] J. Rossignac, A. Safonova and A. Szymczak, *Edgebreaker on a Corner Table: A simple technique for representing and compressing triangulated surfaces*, *Hierarchical and Geometrical Methods in Scientific Visualization*, 2003, pp. 41-50.

[55] M. Sambridge, J. Braun and H. McQueen, *Geophysical parametrization and interpolation of irregular data using natural neighbours*, Geophysical Journal International, 122 (1995), pp. 837-857.

[56] M. S. Shephard and P. M. Finnigan, *Integration of geometric modeling and advanced finite element preprocessing*, *Finite Elements in Analysis and Design*, 1988, pp. 147-162.

[57] J. R. Shewchuk, *Tetrahedral mesh generation by Delaunay refinement*, *Proceedings of the fourteenth annual symposium on Computational geometry*, ACM, Minneapolis, Minnesota, United States, 1998.

[58] H. Si and K. Gaertner, *Meshing piecewise linear complexes by constrained Delaunay tetrahedralizations*, *Proceedings of the 14th International Meshing Roundtable*, 2005, pp. 147-163.

[59] R. Sondershaus and W. Straser, *View-dependent tetrahedral meshing and rendering*, *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, ACM, Dunedin, New Zealand, 2005.

[60] O. G. Staadt and M. H. Gross, *Progressive tetrahedralizations*, *Proceedings of the conference on Visualization '98*, IEEE Computer Society Press, Research Triangle Park, North Carolina, United States, 1998.

[61] A. Szymczak and J. Rossignac, *Grow & fold: compression of tetrahedral meshes*, *Proceedings of the fifth ACM symposium on Solid modeling and applications*, ACM, Ann Arbor, Michigan, United States, 1999.

[62] T. J. Tautges, K. Merkley, C. J. Stimpson and R. J. Meyers, *The Sandia Mesh Database Component (MDB)*, *Proceedings of the Seventh Us National Congress on Computational Mechanics*, 2003.

[63] TetMesh, *TetMesh - GHS3D, Ver. 3.1*, Tech. report INRIA/SIMULOG (2001).

[64] I. J. Trotts, B. Hamann and K. I. Joy, *Simplification of Tetrahedral Meshes with Error Bounds*, IEEE Transactions on Visualization and Computer Graphics, 5 (1999), pp. 224-237.

[65] S.-K. Ueng and K. Sikorski, *A Note on a Linear Time Algorithm for Constructing Adjacency Graphs of 3D FEA Data*, *The Visual Computer*, 1996, pp. 445-450.

[66] S.-K. Ueng and K. Sikorski, *An out-of-core method for computing connectivities of large unstructured meshes*, *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, Eurographics Association, Blaubeuren, Germany, 2002.

[67] H. T. Vo, S. P. Callahan, P. Lindstrom, V. Pascucci and C. T. Silva, *Streaming Simplification of Tetrahedral Meshes*, IEEE Transactions on Visualization and Computer Graphics, 13 (2007), pp. 145-155.

[68] K. Weiler, *The radial-edge data structure: a topological representation for non-manifold geometric boundary modeling*, Geometric Modeling for CAD Appl., 1988, pp. 3-36.

[69] M. Weiler, P. N. Mallon, M. Kraus and T. Ertl, *Texture-Encoded Tetrahedral Strips*, *Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics*, IEEE Computer Society, 2004.

[70] C.-K. Yang, T. Mitra and T.-C. Chiueh, *On-the-Fly rendering of losslessly compressed irregular volume data*, *Proceedings of the conference on Visualization '00*, IEEE Computer Society Press, Salt Lake City, Utah, United States, 2000.

[71] S.-e. Yoon and P. Lindstrom, *Random-Accessible Compressed Triangle Meshes*, IEEE Transactions on Visualization and Computer Graphics, 13 (2007), pp. 1536-1543.