# COMPACT REPRESENTATIONS OF SIMPLICIAL MESHES IN TWO AND THREE DIMENSIONS*

Daniel K. Blandford dkb1@cs.cmu.edu
Guy E. Blelloch blelloch@cs.cmu.edu
David E. Cardoze cardoze@cs.cmu.edu
Clemens Kadow kadow@cmu.edu

*Carnegie Mellon University, Pittsburgh, PA, U.S.A.*

## ABSTRACT

We describe data structures for representing simplicial meshes compactly while supporting online queries and updates efficiently. Our representation requires about a factor of five less memory than the most efficient standard representations of triangular or tetrahedral meshes, while efficiently supporting traversal among simplices, storing data on simplices, and insertion and deletion of simplices.
Our implementation of the data structures uses about 5 bytes/triangle in two dimensions (2D) and 7.5 bytes/tetrahedron in three dimensions (3D). We use the representations to implement 2D and 3D incremental algorithms for generating a Delaunay mesh. The 3D algorithm can generate 100 Million tetrahedrons with 1 Gbyte of memory, including the space for the coordinates and all data used by the algorithm. The runtime of the algorithm is as fast as Shewchuk's Pyramid code, the most efficient we know of, and uses a factor of 3.5 less memory overall.

**Keywords: mesh representations, computational geometry, compression**

## 1. INTRODUCTION

The space required to represent large unstructured meshes in memory can be the limiting factor in the size of a mesh used in various applications. Standard representations of tetrahedral meshes, for example, can require 300-500 bytes per vertex. One option for using larger meshes is to maintain the mesh in external memory. To avoid thrashing, this requires designing algorithms that carefully orchestrate how they access the mesh. Although several such external memory algorithms have been designed [17, 12, 10, 27, 41, 2, 40, 1], these algorithms can be much more complicated than their main-memory counterparts, and can be significantly slower.

Another option for using larger meshes is to try to compress the representation within main memory. There has in fact been significant interest in compressing meshes [11, 21, 39, 31, 32, 38, 24, 22, 16]. In three dimensions, for example, these methods can compress a tetrahedral mesh to less than a byte per tetrahedron [38]—about 6 bytes/vertex. These techniques, however, are designed for storing meshes on disk or for reducing transmission time, not for representing a mesh in main memory. They therefore do not support dynamic queries or updates to the mesh while in compressed form.

We are interested in compressed representations of meshes that permit dynamic queries and updates to the mesh. The goal is to solve larger problems while using standard main-memory algorithms. In this paper we present data structures for representing two and three dimensional simplicial meshes. The representations support standard operations on meshes including traversing among neighboring simplices, inserting and deleting simplices, and the ability to store data on simplices. For a class of well shaped

* This work was supported in part by the National Science Foundation as part of the Aladdin Center (www.aladdin.cmu.edu) and Sangria Project (www.cs.cmu.edu/~sangria) under grants ACI-0086093, CCR-0085982, and CCR-0122581.

meshes [29] these operations all take constant time. The precise definition of our interface is described in Section 4. Although our representations are not as compact as those designed for disk storage, they still save a factor of between 5 and 10 over standard representations.

Our data structures are described in Section 5. They take advantage of the separator property of well-shaped meshes [29], and use recent results in compressing graphs [4]. In particular our technique uses separators to relabel the vertices so that vertices that share a simplex are likely to have labels that are close in value. Pointers are then difference encoded using variable length codes. We use this technique to radially store the neighboring vertices around each vertex in 2D and around a subset of the edges in 3D. A query need only decode a single vertex in 2D or vertex and edge in 3D. For applications that need to generate new vertices, *e.g.*, Delaunay refinement, we leave extra space in the label space and assign new labels based on the labels of the neighbors.

Section 6 describes an implementation of our data structure and Section 7 presents experimental results. The implementation uses about 5 bytes per triangle in 2D and about 7.5 bytes per tetrahedron in 3D when measured over a range of mesh sizes and point distributions. We present experiments based on using our representation as part of incremental Delaunay algorithms in both 2D and 3D. We use a variant of the standard Bowyer-Watson algorithm [8, 42] and the exact arithmetic predicates of Shewchuk [37] for all geometric tests. We also present experiments based on a Delaunay refinement algorithm that removes triangles with small angles by adding new points at their circumcenters. All space is reported in terms of the total space including the space for the vertex coordinates and all other data structures required by the algorithm. The results for 1 Gbyte of memory are summarized as follows.

- We can generate a 2D Delaunay mesh with 110 million triangles (.47 Gbytes for the mesh, .44 Gbytes for the vertex coordinates, and about .1 Gbytes for auxiliary data used by the algorithm). Compared to the Triangle code [36] (the most efficient we know of) our algorithm uses a factor of 3 less memory. It is about 10% slower than Triangle's divide-and-conquer algorithm and much faster than its incremental algorithm.

- We can generate a 3D Delaunay mesh with 100 Million tetrahedrons (.75 Gbytes for the mesh, .17 Gbytes for the vertex coordinates, and .08 Gbytes for auxiliary data). Compared to the Pyramid code [35], our algorithm uses a factor of 3.5 less memory, and is about 30% faster.

- We can generate a refined 2D Delaunay mesh with 80 million triangles with no angle less than 26%. This version dynamically generates new labels, and uses an extra level of indirection in our data-structure.

Our representation can be used in conjunction with external memory algorithms. Also, although we only describe our implementation for 2D and 3D simplicial meshes, the ideas extend to higher dimensions. These topics are discussed, briefly, in Section 8.

## 2. STANDARD MESH REPRESENTATIONS

There have been numerous approaches for representing unstructured meshes in 2 and 3 dimensions. Some are specialized to simplicial meshes and others can be used for more general polytope meshes. For the purpose of comparing space usage, we review the most common of these representations here. A more complete comparison for 2D data structures can be found in a paper by Kettner [25].

In two dimensions most approaches are based on either triangles or edges. The simplest representation is to use one structure per triangle. Each structure has three pointers to the neighboring triangles, and three pointers to its vertices. Assuming no data needs to be stored on triangles or edges, this representation uses 6 pointers per triangle. Storing data requires extra pointers. Shewchuk's Triangle code [36], and the CGAL 2D triangulation data structure [7] both use a triangle-based representation. To distinguish the three neighbors/vertices of a triangle, a handle to a triangle typically needs to include an index from 1 to 3. The data structure used by Triangle, for example, includes such an index in the pointer to each neighbor (in the low 2 bits) so that a neighbor query not only returns the neighbor triangle, but returns in which of three orders it is held.

There are many closely related representations based on edges, including the doubly connected edge list [30], winged-edge [3], half-edge [43], and quad-edge [20] structures. In addition to triangulated meshes, these representations can all be used for polygonal meshes. In these representations each edge maintains pointers to its two neighboring vertices and to neighboring edges cyclically around the neighboring faces and vertices. Each edge might also maintain pointers to the neighboring faces and to edge data. The most space efficient of these representations can maintain for each edge a pointer to the two neighboring vertices and to just two neighboring edges, one around each face and vertex. Assuming no data needs to be stored on a face or edge, this requires 4 pointers per edge, which for a manifold triangulation is equivalent to the 6 pointers per triangle used by the triangle structure ($|E| = 3/2|T|$). The half-edge data structure [43], used by CGAL [25], LEDA [28] and HGAM [18], maintains two structures per edge, one in each direction. These half-edges are cross referenced, requiring an extra two pointers per edge. The winged-edge and quad-edge structures maintain pointers to all four neighboring edges, requiring 6 pointers per edge (9 per triangle).

In three dimensions there are analogous representations based either on tetrahedrons or on faces and edges. Again the simplest representation is to use a structure per tetrahedron. Each tetrahedron has 4 pointers to adjacent tetrahedrons, and 4 to its corner vertices. Assuming no data this requires 8 pointers per tetrahedron. This representation is used by Pyramid [35] and CGAL [7]. The face and edge representations are often called boundary representations (b-reps). Such boundary representations are more general than the tetrahedron representations, allowing the representation of polytope meshes, but tend to take significantly more space. Dobkin and Laszlo [13] suggest a data structure based on edge-face pairs, which in general requires 6 pointers per edge-face. For tetrahedral meshes this representation can be optimized to 9 pointers per face (6 to the adjacent faces rotating around its 3 edges, and 3 to the corner vertices). This corresponds to 18 pointers per tetrahedron. Weiler's radial-edge representation [44], Brisson's cell-tuple representation [9], and Linehard's G-map representation [26] all take more space.

In summary, the most efficient standard representations of simplicial meshes use 6 pointers per triangle in 2D and 8 pointers per tetrahedron in 3D. At least one extra pointer is required to store data on triangles in 2D or tetrahedrons in 3D.

## 3. PRELIMINARIES

In this section we review some basic notions of combinatorial topology used in this paper. For a more detailed discussion the reader can refer to [14] and [33] among others.

An *(abstract) simplicial complex* $K$ is a non-empty collection of finite sets which is closed under taking non-empty subsets. The elements of $K$ are called *simplices.* The underlying set $\cup K$ is called the *vertex set* and its elements are called *vertices.* The *dimension* of a simplex with $d$ vertices is $d-1$. The dimension of $K$ is the maximum dimension among its simplices. A simplex $\tau$ is a *face* of a simplex $\gamma$ iff $\tau \subseteq \gamma$, and iff $\tau \neq \gamma$ we say that $\tau$ is a *proper* face of $\gamma$. We say that $K$ is *pure* if every simplex is a face of a simplex of highest dimension. Let $S$ be a subset of $K$. We call the collection of all simplices in $S$ together with all their faces, $Cl(S)$, the *closure* of $S$. The *star* of a simplex is the union of its superfaces, $St(\sigma) = \{\gamma : \sigma \subseteq \gamma\}$. The *link* of a simplex $\sigma$ is the set of simplices in the closure of its star that do not intersect it, $Lk(\sigma) = Cl(St(\sigma)) - St(\sigma)$.

Let $E$ be a mapping from the vertices of $K$ to $R^m$. We let $|\sigma|$ denote the convex hull of the images of their vertices of $\sigma$ under $E$, and let $|K| = \cup_{\sigma \in K}|\sigma|$. We say that $|K|$ is an embedding of $K$ iff for all simplices $\sigma$ and $\tau$ it holds that $|\sigma| \cap |\tau| = |\gamma|$ where $\gamma$ is their maximum common face (which may be empty). We say that $K$ is a $d$-manifold (with boundary) iff $|K|$ is a $d$-manifold (with boundary). If $K$ is a manifold of dimension $d$ then the link of every $(d-2)$-simplex is a cycle of edges and vertices (*i.e.*, a 1-manifold). If $K$ is a manifold with boundary, then the link of every $(d-2)$-simplex is either a cycle or a path, *i.e.*, a 1-manifold with or without boundary (see Figure 1 (a)). We will make use of this fact in our representation described in section 5.

An *ordering*, $\vec{s^d}$, of a $d$-simplex, $s^d$, is a total ordering of its vertices. An *orientation*, $\overline{s^d}$, of a simplex, $s^d$, is a maximal set of orderings which are even permutations of each other[1]. Every ordering $\vec{s^d}$ on a simplex implies an orientation $\overline{s^d}$ on the simplex, and every $d$-simplex, $d > 0$, has two possible orientations. The orientation $\overline{s^d}$ of a simplex *induces* an orientation $\overline{s^{d-1}}$ on every $d-1$ subsimplex—*i.e.*, for all $\vec{s^{d-1}} \in \overline{s^{d-1}}$ there exists $\vec{s^d} \in \overline{s^d}$ such that $\vec{s^{d-1}}$ is a prefix of $\vec{s^d}$.

For our purposes, a *$d$-pseudomanifold* is a pure $d$-complex where every $(d-1)$-simplex is contained in at most two $d$-simplices and where the dual graph is connected. The vertices of the dual graph are the $d$-simplices and the edges are the $(d-1)$-simplices. A $d$-pseudomanifold is *orientable* if its $d$-simplices can be given orientations in such a way that when they meet at a $(d-1)$-simplex $s$, they induce opposite orientations on $s$. Every orientable $d$-pseudomanifold has two possible orientations, which can be specified by the orientation of one of its $d$-simplices. If $K$ is a $d$-pseudomanifold then the link of every $(d-2)$-simplex is a collection of disjoint cycles and/or paths (see Figure 1 (b)).

In this paper we will use the term *simplicial mesh* to refer to a pseudomanifold abstract simplicial complex with a given orientation.

## 4. INTERFACE

In this section we present the interface for simplicial meshes that our representation implements. It is a simplified version of an interface described in [5]. The interface supports standard operations on meshes

[1] An even permutation is a permutation reached by an even number of swaps.

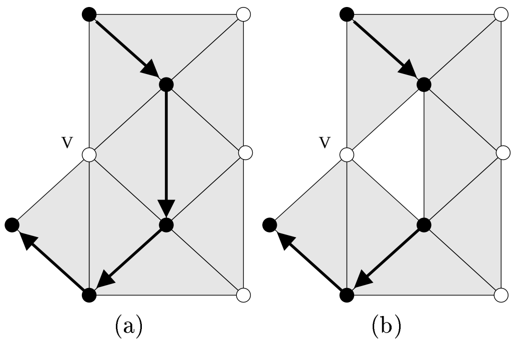


**Figure 1**: Example of a 2D manifold complex with boundary (a) and a pseudomanifold complex (b) along with the link of a vertex $v$. The link is a single path in (a) and two paths in (b).

including, a mechanism to systematically traverse a mesh (*e.g.*, reflect across a face, or rotate around a vertex), and for updating the mesh, including inserting and deleting simplices and associating data with simplices. The interface consists of three operations on ordered simplices (**empty**, **up**, and **down**), and three operations on simplicial meshes (**add**, **delete**, and **findUp**).

Let $s^k = (v_1, \ldots, v_{k+1})$ be an ordered simplex. The **empty** operation creates an empty simplex **empty**() $\to$ (). The **up** operation adds a vertex $v$ to $s$: **up**$(s^k, v) \to (v_1, \ldots, v_{k+1}, v)$. The **down** operation extracts the last vertex from $s^k$: **down**$(s^k) \to (v_1, \ldots, v_k) : v_{k+1}$.

Let $M$ be a $d$-dimensional simplicial mesh. The **add** operation takes $M$ and a highest dimension ordered simplex $\vec{s^d}$ and returns a new mesh $M'$ that results from adding $\vec{s^d}$ to $M$. We require that $s^d$ has consistent orientation with $M$. Note that when we add $\vec{s^d}$, we don't have to store all even permutations of $\vec{s^d}$—just storing $\vec{s^d}$ is enough to determine the orientation of $\vec{s^d}$. The **delete** operation takes $M$ and a highest dimension ordered simplex $\vec{s^d}$ and returns the mesh $M'$ that results from removing $\vec{s^d}$ from $M$. The **findUp** takes an ordered simplex $\vec{s^k}$, $0 \le k \le d$, and $M$. If $s^k$ is not in $M$ it returns **null**. Otherwise it returns an ordered simplex $\vec{s^d}$ such that $\vec{s^d} \in M$ and $\vec{s^k}$ is a prefix of $\vec{s^d}$. In the special case where $k = d - 1$, then there is at most one $s^d$ that can be returned.

In addition to the core interface, we also provide two operations to associate and retrieve data from simplices in a complex. The **addData** operation takes $M$, an ordered simplex $\vec{s^k}$, $0 \le k \le d$ and some user supplied data $u$. It associates $u$ with $s^k$ in $M$. The operation **findData** takes $M$ and an ordered simplex $\vec{s^k}$ and returns the user data $ud$ associated with $s^k$ in $M$. If there is no associated data then **null** is returned.

## 5. REPRESENTATION

Here we describe our 2D and 3D representations for simplicial meshes (simplicial orientable pseudo manifolds). We first describe uncompressed versions of the representations and then describe how to compress them. Our representations are based on storing the link for a set of $(d-2)$-simplices. In 2D this is similar to the half-edge structure [43], and in 3D it is similar to the Dobkin and Laszlo [13] structure. We note, however, that all references are to vertex labels instead of pointers to other higher-dimensional simplex structures, allowing us to compress based on vertex labels. Our representations have the property that if the degree of all vertices is bounded all queries take constant time. We first describe a version for manifold complexes.

Our 2D representation maps each vertex to its link, represented as a cycle of the labels of its neighboring vertices. The cycle is ordered radially around the vertex in the orientation of the complex, *e.g.*, clockwise. A **findUp** query on the ordered edge $(v_1, v_2)$ can be answered by looking up the link for $v_1$, finding $v_2$ in the link, and returning the next vertex in the link. A **findUp** on a vertex can be answered by selecting the first two vertices off of its link.

The link can be stored as a list of labels starting at an arbitrary point on the cycle. If the vertex has bounded degree, the lookup takes constant time. To analyze the space note that each edge appears in two cycles, and each appearance requires two pointers, one to the vertex label and one to the next element in the list. The total space is therefore 4 pointers/edge + 1 pointer/vertex. This is identical in space usage to the triangle-based structure, assuming that it also maintains a pointer from each vertex to one of its incident triangles. Our representation is similar to the half-edge structure since there are effectively two structures per edge, one pointing in each direction. It differs, however, in that there are no direct cross pointers between the matching half edges.

In 3D the representation maps a subset of all ordered edges to their link, represented as a cycle of vertex labels. The cycle is maintained in a consistent orientation, *e.g.*, obeying a right-hand rule with respect to the order (direction) of the edge. The *representative* subset $E'$ is selected to include only the edges $\{v_1, v_2\}$ for which either the labels of $v_1$ and $v_2$ are both odd, or they are both even. Furthermore an edge is only stored in one of its two orders, chosen using a fixed rule, *e.g.*, lower labeled vertex first. Since for any triangle (2-simplex) at least two labels have to be either odd or even, this sampling of the edges guarantees that every triangle has at least one representative ordered edge in $E'$. The representation also needs to supply a way to access the link given the vertex labels of any edge in $E'$. This can be implemented using an adjacency list for each $v \in V$ of all outgoing representative edges $(v, v') \in E'$. Each element of the list stores $v'$ and a pointer to the link of $(v, v')$.

A **findUp** on an ordered triangle $(v_1, v_2, v_3)$ works as follows. It first finds a representative ordered edge $(v_a, v_b)$ from the triangle. Let's call the third vertex on the triangle $v_c$. It looks up the link of $(v_a, v_b)$ in the adjacency list for $v_a$, and searches for $v_c$ in the link. If $(v_1, v_2, v_3)$ and $(v_a, v_b, v_c)$ have the same orientation (are an even permutation of each other) **findUp** returns the next vertex in the link, otherwise it returns the previous vertex in the link. A **findUp** on a vertex can be implemented by selecting any of its outgoing edges, and selecting the first two vertices of the edge's link. A vertex, however, might have no outgoing edges in $E'$. For such a vertex $v$ the representation can store $(v_1, v_2)$ for any triangle $(v, v_1, v_2)$. The triangle can be used to find the tetrahedron. To support **findUp** on edges requires storing all edges (in one direction), but not necessarily their links. For edges not in $E'$ (*i.e.*, odd-even edges), the representation needs only store a single vertex in their link.

To analyze the space for this representation we assume that the links of representative edges are stored as lists of vertices. Each list element has two pointers: one to the vertex and one to the next element in the list. For an edge $e \in E'$ there is a one-to-one correspondence between the triangles for which $e$ is a face and list elements in the link of $e$. Since a triangle has 3 edges, and on average half the edges will appear in $E'$[2], every triangle will contribute and average of $3 * .5 = 1.5$ list elements to the overall data structure. Since there are twice as many triangles as tetrahedrons, each tetrahedron will contribute an average of 3 list elements, which corresponds to 6 pointers. We also need to store the vertex adjacency lists for out-edges in $E'$. Each edge $(v_1, v_2) \in E'$ will appear as an element in one list $(v_1)$, and will require three pointers: one to $v_2$, one to the link of $(v_1, v_2)$, and one to the next element in the list. Additionally a pointer from each vertex to its list is required. The total space to support **findUp** on triangles is therefore $6|T| + 3/2|E| + |V|$. For a typical reasonably shaped mesh $|E| \approx 7/6|T|$ and $|V| \approx 1/6|T|$, giving approximately 8 pointers/tetrahedron. This is the same as the representation based on tetrahedrons.

The additional space to support **findUp** on vertices is trivial since most vertices already have an outgoing edge. To support **findUp** on edges, we need to separately store the excluded edges $(v_1, v_2)$ that are not in $E'$ in either direction. These can be stored off of $v_1$ using a linked list with 3 pointers per edge—one for $v_2$, one for some $v$ in the link of $(v_1, v_2)$, and one for the next pointer. This comes to about $3 * 1/2 * |E| = 7/6 * 3/2|T| = 7/4|T|$. Many applications will not need **findUp** on edges, so in these cases this extra data need not be stored.

For manifolds with boundaries, the link might be a path of vertices instead of a cycle. We can simply keep the path starting at the first element. For pseudomanifolds the link of singular vertices (2D) or edges (3D) can consist of a set of cycles and/or paths. We call this set the *link set* and it can be represented as multiple lists.

A $d$-simplex $s$ can be deleted by finding the representative $(d-2)$-simplices that faces of $s$, and splitting a cycle or path of each of their links. For example, in Figure 1 when the triangle is deleted from (a) going to (b), the path for the link of vertex $v$ is split into two paths. Similarly the cycles for the other two vertices on the triangle are each slit into a path. If splitting a link leaves the link set empty, then the $(d-2)$-simplex is deleted. A $d$-simplex $s$ can be added by finding the representative $(d-2)$-simplices of $s$, and extending each of their link sets. This extension might add a new path to the set (if neither of the two new vertices are in the set), it might extend an existing path (if one vertex is in the set), it might join two existing paths (if the two vertices are in separate paths), or it might joint a path into a cycle (if the two vertices are the ends of the same path). If the representation is restricted to manifolds with boundary, then the single path must either be extended by one (on either side), or jointed into a cycle.

Data can be added to the $d$ simplices (or $d-1$ simplices) by adding a data field to each element of the link. Since a $d$ simplex will appear in multiple links, the data only needs to be stored on one of them (chosen in a fixed manner to make lookup easy). We make use of this in the compressed representation.

**Compressed Representation:** We first discuss how to compress the representation in 2D. Compression in 3D is similar. We make use of *difference coding*, in which each element in a vertex's link is represented by its difference from the original vertex. If these differences are small, then a variable-length prefix code (such as the Gamma code of Elias [15]) can represent them efficiently. An additional sign bit can be added to allow for negative differences. To ensure that the differences are small, our algorithm relabels the vertices in a preprocessing phase which we will discuss later.

[2]This is only strictly true for randomly selected labels. However, for non-random labels one can use a hash on the labels to decide on which edges to include.

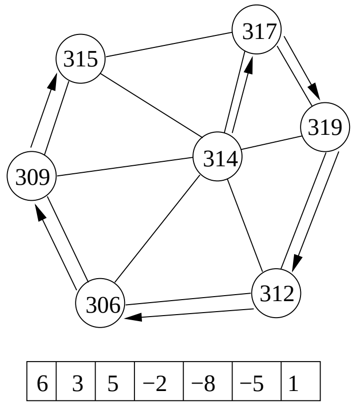


**Figure 2**: The neighborhood and corresponding difference code data for vertex 314. The first entry, 6, is the degree of the vertex. Other entries are the offsets of the neighbors.

Once the vertices are relabeled, the link of a vertex can be represented by concatenating the code for its degree to the codes for the differences of its neighbors. (See Figure 2 for an example.) If a vertex has a link consisting of multiple cycles or paths (as can occur in a pseudomanifold), this link set can be represented by putting the cycles/paths one after the other with a count before each. If data is associated with some of the simplices, this can be interleaved with the codes for the neighbors. The resulting vertex encodings are stored in fixed-length blocks; if an encoding is larger than will fit in one block, multiple blocks may be formed into a linked list to hold the encoding. Our representation makes use of a hashing technique to minimize the size of the pointers used in these linked lists.

When the representation is queried, the code for the corresponding vertex is decompressed. When an update is made, the code for the corresponding vertices is decompressed, modified, and then compressed again.

Compression of a 3D representation is similar except that the representation stores the link around representative edges rather than around vertices. For each vertex the representation stores a list of that vertex's representative out-edges, with pointers to the links of those out-edges. These pointers are compressed using the same hashing technique as above.

**Generating Labels.** If all the vertices are known before the algorithm begins, our algorithm can relabel them using a technique based on $x$-$y$ cuts. Given a set of points, the technique first finds which of the $x$ and $y$ axes has the greatest diameter. It finds the approximate median in that coordinate and partitions the points on either side of that median. The points on one side are labeled first, then the points on the other side. This is done recursively to produce a labeling in which points that are near each other have similar labels. This is similar to a separator-based technique for graph compression through relabeling [4] except that it occurs before any edges have been added to the mesh.

If not all vertices are known before the algorithm begins, our algorithm can assign a sparse labeling to the initial vertices. When a new vertex is added, it is assigned a label that is close to the labels of its neighbors. It would be inefficient to allocate storage for every possible label; instead, our algorithm uses an extra level of indirection to map vertex labels to memory blocks.

## 6. IMPLEMENTATION

**2D Triangulation.** Our 2-dimensional compressed data structure is implemented as follows.

For difference encoding our structure uses the *nibble code*, a code of our own devising that stores integers using 4-bit "nibbles". Each nibble contains three bits of data and one "continue" bit. The continue bit is set to **0** if the nibble is the last one in the representation of a integer, and **1** otherwise. We find that this code is much faster than the gamma code while being almost as space-efficient.

It is sometimes necessary to store an extra bit $b$ with a value $v$. This is accomplished with a shift operation: $v' \leftarrow 2v + b$. In particular, if any value might be negative, our difference coder stores its absolute value plus a sign bit: $v' \leftarrow 2|v| + \texttt{sign}(v)$.

A vertex is represented with a nibble code for the degree of the vertex, followed by nibble codes for the differences to each of the vertex's neighbors. Our representation stores two additional "special-case" bits with each neighbor to provide information about the triangle that precedes it in the link. One bit is set to indicate a gap in the link set: it indicates that there is no triangle preceding that neighbor in the mesh. The other bit is set when data is associated with the triangle preceding that neighbor. In this case, the code for that neighbor is followed with a nibble code representation of the data.

As an optimization, note that for many vertices none of the special-case bits will be set. Our representation stores a bit with the degree of each vertex to indicate if none of its special-case bits are set; if this is so, those bits are omitted in the encoding of that vertex.

Our representation stores the nibble codes for each vertex in an array containing one seven-byte block per vertex. If a block overflows (that is, if the storage needed is greater than seven bytes), additional space

| Block Size | Blocks Needed | Total Space |
|---|---|---|
| 5 | 745151 | 10086381 |
| 6 | 475263 | 9998531 |
| 7 | 283559 | 9920446 |
| 8 | 164660 | 10101104 |
| 9 | 94105 | 10537195 |
| 10 | 53399 | 11179987 |
| 11 | 30496 | 11974072 |

**Figure 3**: The number of extra blocks needed for $2^{20}$ vertices on a uniform distribution in 2D, and the total space required if we allocate 30% more blocks than are needed.

is allocated from a separate pool of seven-byte blocks. The last byte of the block stores a pointer to the next block in the sequence. Our representation uses a hashing technique to ensure that the pointer never needs to be larger than one byte. This requires a hash function that maps (address, $i$) pairs to addresses in the spare memory pool. Our representation tests values of $i$ in the range 0 to 127 until the result of the hash is an unused block. It then uses that value of $i$ as the pointer to the block. Under certain assumptions about the hash function, if the memory pool is at most 75% full, then the probability that this technique will fail is at most $.75^{128} \simeq 10^{-16}$.

If the vertices are labeled sparsely (so that new labels can be generated dynamically), our representation also makes use of a hash mapping between labels and vertex data blocks. One byte of memory is allocated per label; if the label is in use, this byte contains a hash pointer to the first data block for that vertex.

One bit is stored with each block to indicate whether the current block is the last in the sequence. For the first block this bit is stored with the degree of the vertex; for subsequent blocks it is stored as the eighth bit of the one-byte pointer to that block.

There is a tradeoff in the sizes of the blocks used. Large blocks are inefficient since they contain unused space; small blocks are inefficient since they require space for pointers to other blocks. In addition, there is a cost associated with computing hash pointers by searching for unused blocks in the memory pool. Figure 3 shows the tradeoff between these factors for our Delaunay triangulation algorithm run on $2^{20}$ uniformly distributed points in the unit square. We chose a block size of 7 since it gives the most efficient use of space.

To improve the efficiency of lookups our representation use a caching system. When a query or update is made, the blocks associated with the appropriate vertex are decoded. The information is represented

| Block Size | Blocks Allocated | Blocks Used $2^{10}$ | Blocks Used $2^{15}$ | Blocks Used $2^{20}$ |
|---|---|---|---|---|
| 2 | $0.55n$ | 59% | 67% | 70% |
| 4 | $1.3n$ | 90% | 90% | 88% |
| 6 | $1.55n$ | 90% | 90% | 87% |
| 8 | $1.3n$ | 78% | 73% | 75% |
| 10 | $1.8n$ | 30% | 51% | 63% |

**Figure 4**: The number of blocks of each size that are allocated for an $n$-vertex 3D mesh, and the percentage of blocks that were used for $n = 2^{10}$, $2^{15}$, and $2^{20}$.

in uncompressed form as a list with one vertex in the link per element of the list. The lists are kept in a FIFO cache with a maximum capacity of 2000 nodes. Update operations may affect the lists while they are in the cache. The lists are encoded back into blocks when they are flushed from the cache.

**3D Triangulation.** Our 3-dimensional structure is implemented as a slight generalization of our 2-dimensional structure. Recall that our 3D representation keeps a map from each vertex $v$ to all of its representative out-edges. This is stored as a difference coded list of the corresponding neighbors. The code for each neighbor $v'$ is followed by a code for the number of nibbles in the encoding of the representative edge $(v, v')$, and a pointer to the first block containing the data for that edge. (The pointer is stored using the same hash trick as above to keep pointer sizes small.) Every representative edge has its own block allocated from the memory pool, with the capability to allocate additional blocks if needed.

When an edge is queried, our representation loads only the list for one vertex and for the edge itself into the cache. It does not need to decompress the other edges adjoining that vertex.

Since the number of nibles needed per representative edge is quite variable, our data structure allocates from pools of 2, 4, 6, 8, or 10-byte blocks to reduce wasted space. The number of blocks in each pool was determined experimentally and is shown in Figure 4. The data structure ensures that each pool always has at least 10% free space; if a block cannot be allocated from a given pool, the data structure looks for a larger one. The initial block for each vertex comes from a separate array containing blocks of size 7.

**Dynamic point generation.** To support dynamic point generation we use an expanded label space. If a total of $n$ vertices are to be generated, we allow for $2n$ possible labels. Each label receives a one-byte hash pointer which, if the label is in use, points to the initial data block for the corresponding vertex. The initial vertices are spread evenly across the label space.

**Incremental Delaunay:** We implemented a Delaunay triangulation algorithm in two and three dimensions using our compressed data structure. We employ the well-known Bowyer-Watson kernel [8, 42] to incrementally generate the mesh. During the course of the algorithm a Delaunay triangulation of the current pointset is maintained. An incremental step inserts a new vertex into the mesh by determining the elements that violate the Delaunay condition. Those elements form the Delaunay *cavity*. The faces that bound the cavity are called the *horizon*. The mesh is modified by removing the elements in the cavity and connecting the new vertex to the horizon.

Each *element* (simplex of highest dimension) in the mesh has an associated list of all points in its interior that have not yet been added to the mesh as described in [19]. At each incremental step all points on cavity elements have to be reassociated with new elements using lineside tests in 2D and planeside tests in 3D, which accounts for the dominant cost of the algorithm. We have carefully implemented the *bulldozing* idea described in [5] and extended it to three dimensions.

Our implementation does not require extra memory for the lists of points since at any time a point is either a vertex in the mesh or in one such list. The memory that will be used to store the vertex in the mesh can first be used as a list node.

The algorithm maintains a work queue of elements whose interiors contain points. When no elements contain points (*i.e.*, all have been added to the mesh), the algorithm terminates.

In this scenario all points are known at the beginning. We relabel the input points using cuts along coordinate directions as described above. The runtimes reported in the next section include this preprocessing step.

**Delaunay Refinement:** To test our implementation's performance for the case when new points are dynamically generated at runtime, we implemented a 2D Delaunay refinement code in the style of Ruppert [34]. We augment a Delaunay triangulation by adding circumcenters of badly shaped triangles while maintaining the Delaunay property. When the initial triangulation is built we walk through the mesh once and check the quality of each element, queuing the ones not satisfying a preset minimum angle bound. The same work queue used in the triangulation phase of the algorithm is used to store the list of triangles to be split.

| Distribution | # Pts | # Extra Blocks | Time(s) |
|---|---|---|---|
| uniform | $2^{18}$ | 70823 | 3.16 |
| normal | $2^{18}$ | 72239 | 3.52 |
| kuzmin | $2^{18}$ | 72917 | 4.36 |
| line | $2^{18}$ | 66297 | 3.64 |
| uniform | $2^{20}$ | 288255 | 13.25 |
| normal | $2^{20}$ | 292580 | 14.41 |
| kuzmin | $2^{20}$ | 292709 | 21.34 |
| line | $2^{20}$ | 276124 | 15.86 |

**Figure 5**: The number of extra 7-byte blocks needed to store triangular Delaunay meshes for various point distributions using our structure and the runtime of our 2D implementation.

Whenever a new point $p$ is generated the algorithm assigns as new label by considering the *horizon* vertices $H$ of the cavity created by $p$ and calculating the value $v$ that minimizes the sum of the log norms to $H$. It then finds the closest label to $v$ that is not yet used.

In the pure triangulation code, all vertices are known at the beginning, so we can store the point coordinates and the first level vertex arrays densely. In the refinement code we can only fill these arrays up to about 85% before the open address hashing takes prohibitively long. We also require extra memory for the additional map form the label space to the vertices.

## 7. EXPERIMENTS

We report experiments on a Pentium 4, 2.4GHz system, running RedHat Linux Kernel 2.4.18, GNU C/C++ compiler version 3.0.1. For all geometric operations (lineside, planeside, incircle, and insphere tests) we use Shewchuk's adaptive precision geometric predicates [37]. We use single-precision floating-point numbers to represent the coordinates.

**2D Delaunay:** We tested our 2D implementation on data drawn from several distributions to assess its memory needs for non-uniform data sets. We ran tests on the following distributions: Uniformly random, normal, kuzmin, and a line singularity. Details on these distributions can be found in [6]. In Figure 5 we report the number of extra (overflow) 7-byte blocks used to store Delaunay meshes of various point distributions and the runtime of our implementation. It can be seen that the runtime varies by about 40% while the number of extra blocks varies by about 10%. Furthermore the number of extra blocks used comes to only about 28% of the number of default blocks needed, which is one per vertex. In our experiments we set the number of extra blocks available to 35% of the number of default blocks. The extra blocks therefore fill

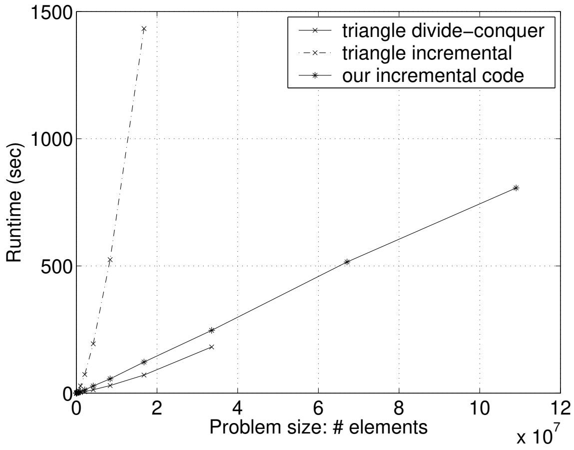


**Figure 6**: Runtime in 2D, uniformly random points

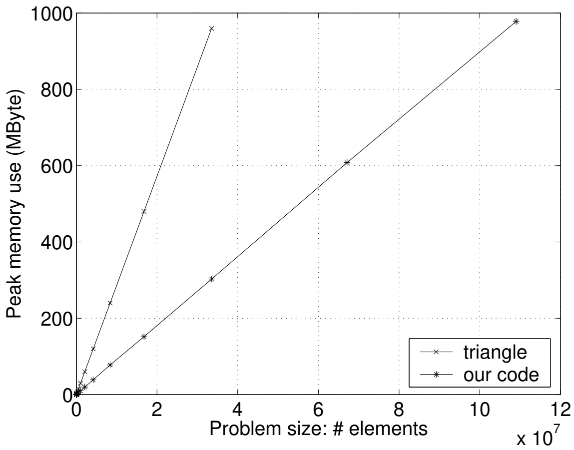


**Figure 7**: Memory use in 2D, uniformly random points

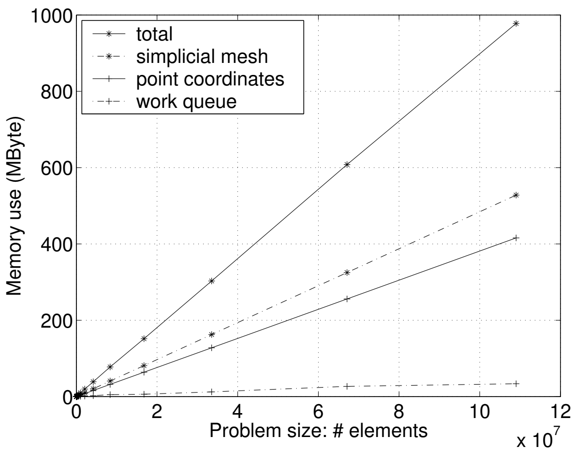


**Figure 8**: Breakdown of memory use in 2D, uniformly random points

| Distribution | # Pts | # Bytes used | Time(s) |
|---|---|---|---|
| uniform | $2^{16}$ | 2525309 | 9.26 |
| normal | $2^{16}$ | 2572659 | 9.38 |
| kuzmin | $2^{16}$ | 2571769 | 11.23 |
| line | $2^{16}$ | 2264465 | 8.77 |
| uniform | $2^{18}$ | 10135321 | 39.59 |
| normal | $2^{18}$ | 10463761 | 41.89 |
| kuzmin | $2^{18}$ | 10444195 | 45.04 |
| line | $2^{18}$ | 9372669 | 38.97 |

**Figure 9**: The number of bytes needed for occupied blocks to store tetrahedral Delaunay meshes for various point distributions and the runtime of our 3D implementation.

to about 80% of capacity. Given this setting, the total space we require for the mesh is $1.35 \times 7$ bytes/vertex, which is 4.725 bytes/triangle.

Next, we compare runtime and memory usage of our implementation to Shewchuk's Triangle [36] code which is the most efficient code reported by Boissonnat et. al. [7]. In Figure 6 we report the runtime of our (incremental) code vs. Triangle's divide-and-conquer and its incremental implementation. We report the total memory use of both codes in Figure 7 and break down our memory use for the simplicial mesh, point coordinates and the work queue in Figure 8. While using just about a third of the memory our code runs about 10% slower than Triangle's divide-and-conquer implementation and is about an order of magnitude faster than Triangle's incremental implementation. In our code 50% of the memory is used to represent the mesh, 40% to store the coordinates, and 10% for the work queue.

**3D Delaunay:** As in 2D we tested our 3D implementation on the same four point distributions. In our 3D structure we allocate memory blocks of different size. To compare the memory needs for various point distribution, we report the number of bytes used to store occupied blocks in Figure 9. As in 2D the runtimes differ, but the memory needed is nearly independent of the distribution.

We compare our 3D implementation with uniform random data to Shewchuk's Pyramid code [35][3]. Figures 10 and 11 show the runtime and the memory usage. Figure 12 breaks down the memory usage of our code.

In comparison our implementation runs slightly faster and uses only about one third of the memory. In 3D the representation of the mesh uses about 75% of the

[3] We note that the version of Pyramid we are using is a Beta release.

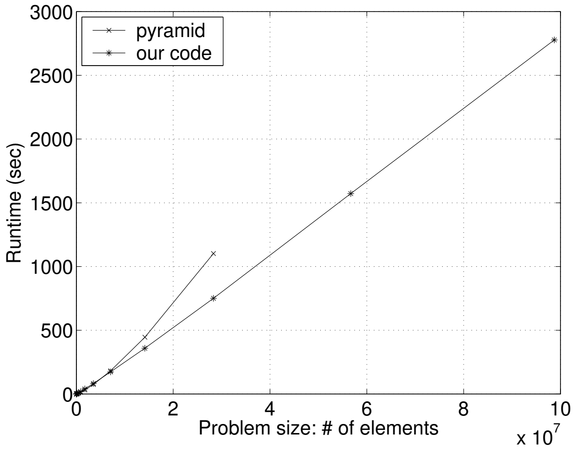


**Figure 10**: Runtime in 3D, uniformly random points

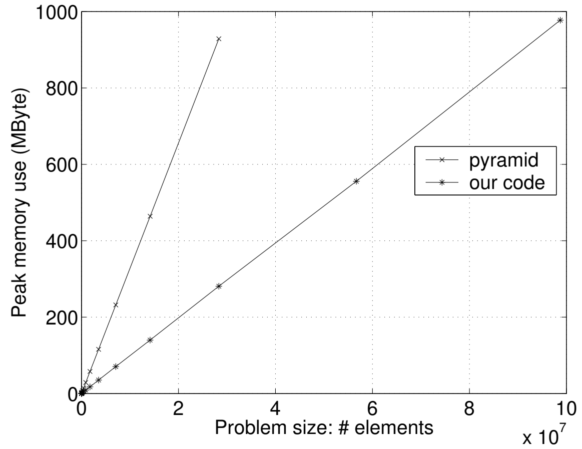


**Figure 11**: Memory use in 3D, uniformly random points

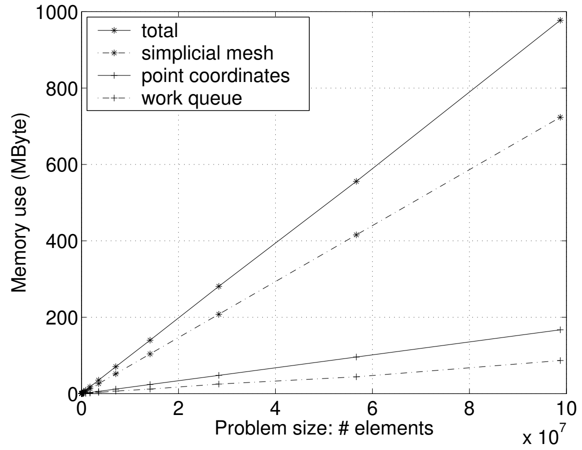


**Figure 12**: Breakdown of memory use in 3D, uniformly random points

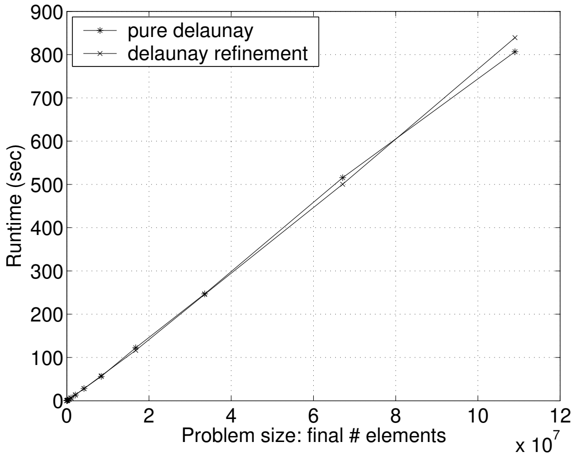


**Figure 13**: Runtime in 2D, pure Delaunay vs. Delaunay refinement

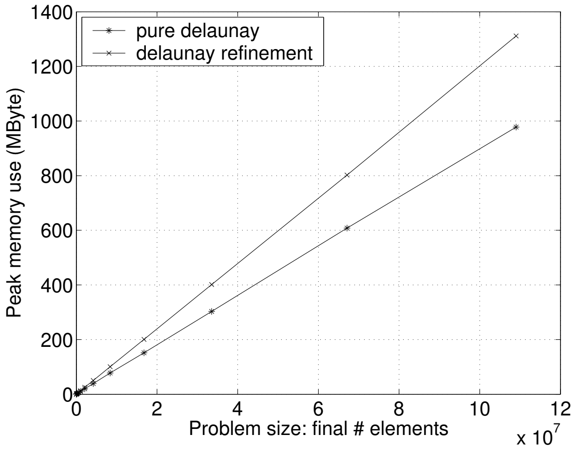


**Figure 14**: Memory use in 2D, pure Delaunay vs. Delaunay refinement

total memory; point coordinates and work queue account for 18% and 7%, respectively.

**2D Delaunay refinement:** We tested our 2D Delaunay refinement code and compare runtime and memory use to our pure 2D Delaunay code, see Figures 13 and 14. The Figures show problem size in terms of the final number of elements in the mesh. In the pure Delaunay code, all $n$ points are known initially, whereas in the refinement code only $n/2$ points are known initially, the other $n/2$ are generated and labeled on the fly as described in Section 6. We refine the mesh up to a minimum angle of $26.85°$.

The runtimes for the two versions are almost identical. We need about 30% more memory in the refinement code. Additional memory is needed for the map from labels to vertices and for slack in the point coordinate array and the first level vertex array needed for our hashing technique.

## 8. DISCUSSION

The representation we described can be used as an alternative to external memory (out-of-core) representations, when the mesh is within a factor of five or so of fitting in memory relative to a standard representation. Our representation has the advantage that it allows random access to the mesh without significant penalty, and can therefore be used as part of standard in-memory algorithms (or even code) by just exchanging the mesh interface.

**In conjunction with external memory.** For very large problems our representation can be used in conjunction with external-memory techniques. Since in our representation the ordering of the vertices is designed to be local (it is based on the quad/oct tree decomposition), and the blocks of memory for vertices are laid out in this ordering, nearby vertices in the mesh will most likely appear on the same page. One problem is that if the data for a vertex overflows we now assign the overflow data to the extra blocks using a hash, which has no locality. To make sure that the overflow data has some spatial locality one could be more careful about assigning the extra blocks (e.g. preferentially within the same page as the original block). Based on this representation, algorithms that have a strong bias to accessing the mesh locally (e.g., see the recent work of Amenta, Choi and Rote [1]) will tend to have good spatial locality and work well with virtual memory when it does not fit into physical memory.

**Generalizations to $d$-dimensions.** The idea of storing the link of every $d-2$ dimensional simplex generalizes to arbitrary dimension. The compression technique also generalizes to arbitrary dimension, but is likely to be ineffective for large dimensions. This is because the size of the difference codes depends on the separator sizes [4], which in turn depends on the dimension. Choosing an effective way to select the representative subset of the $d-2$ dimensional simplices will depend on the dimension and would need to be considered to use our representation on dimensions greater than three. We have not done any experimentation to analyze the effectiveness of our techniques on dimensions greater than three, or to compare our representations to other representations.

## ACKNOWLEDGEMENTS

We are grateful to Jonathan Shewchuk for commenting on the paper and letting us use a pre-release version of Pyramid. This work was done as part of the Sangria [23] project, and several project members have contributed ideas.

## References

[1] N. Amenta, S. Choi, and G. Rote. Incremental constructions con BRIO. In *Proc. ACM Symposium on Computational Geometry*, pages 211–219, June 2003.

[2] L. Arge. External memory data structures. In *Proc. European Symposium on Algorithms*, pages 1–29, 2001.

[3] B. Baumgart. A polyhedron representation for computer vision. In *Proc. National Computer Conference*, pages 589–596, 1975.

[4] D. Blandford, G. Blelloch, and I. Kash. Compact representations of separable graphs. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[5] G. Blelloch, H. Burch, K. Crary, R. Harper, G. Miller, and N. Walkington. Persistent triangulations. *Journal of Functional Programming (JFP)*, 11(5), Sept. 2001.

[6] G. Blelloch, J. Hardwick, G. L. Miller, and D. Talmor. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica*, 24(3/4):243–269, 1999.

[7] J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, and M. Yvinec. Triangulations in CGAL. *Computational Geometry*, 22(1–3):5–19, 2002.

[8] A. Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24:162–166, 1981.

[9] E. Brisson. Representing geometric structures in d dimensions: Topology and order. In *Proc. ACM Symposium on Computational Geometry*, pages 218–227, 1989.

[10] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. In *Proc. ACM Symposium on Computational Geometry*, pages 259–268, June 1998.

[11] M. Deering. Geometry compression. In *Proc. SIGGRAPH*, pages 13–20, 1995.

[12] F. K. H. A. Dehne, D. Hutchinson, A. Maheshwari, and W. Dittrich. Reducing I/O complexity by simulating coarse grained parallel algorithms. In *Proc. IPPS/SPDP*, pages 14–20, 1999.

[13] D. Dobkin and M. Laszlo. Primitives for the manipulation of three-dimensional subdivisions. *Algorithmica*, 4(1):3–32, 1989.

[14] H. Edelsbrunner. *Geometry and Topology of Mesh Generation.* Cambridge Univ. Press, England, 2001.

[15] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, March 1975.

[16] P.-M. Gandoin and O. Devillers. Progressive and lossless compression of arbitrary simplicial complexes. In *Proc. SIGGRAPH*, 2002.

[17] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 714–723, Nov. 1993.

[18] X. Gu. Harvard graphics archive—mesh library. `http://www.cs.deas.harvard.edu/~xgu/mesh/`.

[19] L. Guibas, D. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(4):381–413, 1992.

[20] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(3):74–123, 1985.

[21] S. Gumhold and W. Strasser. Real time compression of triangle mesh connectivity. In *Proc. SIGGRAPH*, pages 133–140, 1998.

[22] M. Isenburg and J. Snoeyink. Face fixer: Compressing polygon meshes with properties. In *Proc. SIGGRAPH*, pages 263–270, 2000.

[23] J. F. Antaki et. al. Sangria project. `http://www.cs.cmu.edu/~sangria`, 2002.

[24] Z. Karni and C. Gotsman. Spectral compression of mesh geometry. In *Proc. SIGGRAPH*, pages 279–286, 2000.

[25] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry – Theory and Applications*, 13:65–90, 1999.

[26] P. Lienhardt. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *International Journal of Computational Geometry and Applications*, 4(3):275–324, 1994.

[27] S. McMains, J. M. Hellerstein, and C. H. Squin. Out-of-core build of a topological data structure from polygon soup. In *Proc. Symposium on Solid Modeling and Applications*, pages 171–182, June 2001.

[28] K. Mehlhorn and S. Naher. *LEDA: A platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[29] G. L. Miller, S.-H. Teng, W. P. Thurston, and S. A. Vavasis. Separators for sphere-packings and nearest neighbor graphs. *Journal of the ACM*, 44:1–29, 1997.

[30] D. Muller and F. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7:217–236, 1978.

[31] R. Pajarola, J. Rossignac, and A. Szymczak. Implant sprays: Compression of progressive tetrahedral mesh connectivity. In *Proc. Visualization 99*, pages 299–306, 1999.

[32] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, 1999.

[33] J. Rotman. *An Introduction to Algebraic Topology, Graduate Texts in Mathematics.* Springer Verlag, 1988.

[34] J. Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1993.

[35] J. Shewchuk. Pyramid mesh generator software. (`http://www.cs.berkeley.edu/~jrs/`). Personal Communication.

[36] J. R. Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In *Proc. First Workshop on Applied Computational Geometry*, pages 124–133, Philadelphia, PA, May 1996.

[37] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18(3):305–368, 1997.

[38] A. Szymczaka and J. Rossignac. Grow & Fold: compressing the connectivity of tetrahedral meshes. *Computer-Aided Design*, 32:527–537, 2000.

[39] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, 1998.

[40] T. Tu, D. R. O'Hallaron, and J. C. Lopez. Etree - a database-oriented method for generating large octree meshes. In *Proc. International Meshing Roundtable*, pages 127–138, Sept. 2002.

[41] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[42] D. F. Watson. Computing the n-dimensional Delaunay tesselation with application to Voronoi polytopes. *The Computer Journal*, 24:167–172, 1981.

[43] K. Weiler. Edge based data structures for solid modeling in a curved surface environment. *IEEE Computer Graphics and Applications*, 5(1):21–40, Jan. 1985.

[44] K. Weiler. The radial edge structure: A topological representation for non-manifold geometric boundary modeling. In *Geometric Modeling for CAD Applications*, pages 3–36. North-Holland, 1988.