COMPACT REPRESENTATIONS OF SIMPLICIAL MESHES IN TWO AND THREE DIMENSIONS*

Daniel Blandford
dkb1@cs.cmu.edu

Guy Blelloch
blelloch@cs.cmu.edu

David Cardoze
cardoze@cs.cmu.edu

Clemens Kadow
kadow@cmu.edu

Carnegie Mellon University

ABSTRACT

We describe data structures for representing simplicial meshes compactly while supporting online queries and updates efficiently. Our representation requires about a factor of four or 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.
We have used the representation in 2D and 3D incremental algorithms for Delaunay triangulation/tetrahedralization. The 3D algorithm can generate 100 Million tetrahedrons with 1.1 Gbytes 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 less memory.

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, 40, 2, 39, 1], these algorithms can be much more complicated than their main-memory counterparts, and are 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, 38, 31, 32, 37, 24, 22, 16]. In three dimensions, for example, these methods can compress a tetrahedral mesh to less than a byte per tetrahedron [37]—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 updates and queries 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 2 and 3 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 meshes [29] these operations all take constant time. The precise definition of our interface is described in Section 4.

* 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.

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.

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 9 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, 41] and the exact arithmetic predicates of Shewchuk [36] for all geometric tests. 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 can be summarized as follows.

- Using a memory footprint of 1 Gbyte we are able to generate a 2D mesh with 100 million triangles (.5 Gbytes for the mesh, .4 Gbytes for the vertex coordinates, and about .1 Gbytes for auxiliary data used by the algorithm). Compared to the Triangle code [35] (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.
- Using a memory footprint of 1.1 Gbyte we are able to generate a 3D mesh with 100 Million tetrahedrons (.86 Gbytes for the mesh, .17 Gbytes for the vertex coordinates, and .07 Gbytes for auxiliary data). Compared to the Pyramid code [34], our algorithm uses a factor of 3 less memory, and is about 10% faster.

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 [35], 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 [42], 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 [42], 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 on tetrahedrons or faces and edges. Again the simplest representation is to use a structure per tetrahedron. Each tetrahedron has 4 pointers to ad-

jacent tetrahedrons, and 4 to its corner vertices. Assuming no data this requires 8 pointers per tetrahedron. This representation is used by Pyramid [34] 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 [43], 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. 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 τ is a *face* of a simplex γ iff $\tau \subseteq \gamma$, and iff $\tau \neq \gamma$ we say that τ is a *proper* face of γ. 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 σ 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 σ under E, and let $|K| = \cup_{\sigma \in K} |\sigma|$. We say that $|K|$ is an embedding of K iff for all simplices σ and τ it holds that $|\sigma| \cap |\tau| = |\gamma|$ where γ 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). We also say that K is orientable iff $|K|$ is orientable. 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.

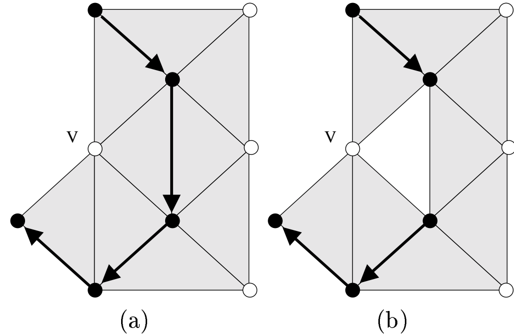

Figure 1: Example of a 2D manifold complex with boundary (a) and a pseudomanifold complex (b) along with the link of a vertex v. The link is a single path in (a) and two paths in (b).

For our purposes, a *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 pseudomanifold is orientable if its d-simplices can be given an orientation in such a way that when they meet at a $(d-1)$-simplex, they have consistent orientations. If K is a pseudomanifold of dimension d 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 an orientable pseudomanifold abstract simplicial complex.

4. INTERFACE

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

We start by describing an interface for *ordered simplices.* One can think of an ordered simplex as a simplex with a particular total order on its vertices. Intuitively the total order describes the way in which one is holding the simplex. Formally we define an *ordered*

simplex as a pair $s^k = (v_1, \ldots, v_{k+1}) : f$ where the first element is a sequence (ordering) of the vertices, and the second is a boolean flag. We make use of the flag f to determine whether the *orientation* for the simplex has been flipped. We also denote by $\overline{s^k}$ the *oriented simplex* corresponding to s^k. If s^k is an even permutation of the vertices $v_1, \ldots, v_k$ then $\overline{s^k}$ is the set of all such even permutations[1]. Otherwise it is the set of all the odd permutations.

Our interface for *ordered simplices* consists of the following operations: **empty**, **up**, **down**, **flip**, and **rotate**$_j$. Let $s^k = (v_1, \ldots, v_{k+1}) : f$ be an ordered simplex. The **empty** operation creates an empty simplex with the given boolean flag: $\textbf{empty}(f) \to () : f$. The **up** operation adds a vertex v to s: $\textbf{up}(s^k, v) \to (v_1, \ldots, v_{k+1}, v) : f$. The **down** operation extracts the last vertex from s^k: $\textbf{down}(s^k) \to ((v_1, \ldots, v_k) : f) : v_{k+1}$. The **flip** operation flips the last two vertices and the value of the boolean flag: $\textbf{flip}(s^k) \to (v_1, \ldots, v_{k+1}, v_k) : (\text{not} f)$. Finally the **rotate**$_j$ operation performs a cyclic rotation by j of the vertices in s^k and flips the flag if the rotation changed the underlying orientation: $\textbf{rotate}_j(s^k) \to (v_{1+j}, v_{2+j}, \ldots, v_{k+1+j}) : g$, where g is set to f if the orientation didn't change and to not f otherwise, and all the indices are computed modulo $k+1$. We use **rotate** without subscript to denote **rotate**$_1$.

We now describe our interface simplicial meshes. The interface is rather minimal. It consists of three operations: **add**, **delete**, and **findUp**.

Let C be a d-dimensional simplicial mesh. The **add** operation takes C and a highest dimension ordered simplex s^d and returns a new mesh C' that results from adding $\overline{s^d}$ to C. The **delete** operation takes C and a highest dimension ordered simplex s^d and returns the mesh C' that results from removing $\overline{s^d}$ from C. The **findUp** takes an ordered simplex s^k, $0 \leq k \leq d$, and C. If s^k is not in C it returns **null**. Otherwise it returns an ordered simplex s^d such that $s^d \in C$, s^k is a prefix of s^d, and s^d and s^k have the same orientation flag. 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 C, an ordered simplex s^k, $0 \leq k \leq d$ and some user supplied data u. It associates u with $\overline{s^k}$ in C. The operation **findData** takes C and an ordered simplex s^k and returns the user data ud associated with $\overline{s^k}$ in C. If there is no associated data then **null** is returned.

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

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 [42], 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 at least two labels have to be either odd or even, this sampling of the edges guarantees that every triangle face 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 note that on average half the edges will appear in one order in E'. That means that every face will appear in the link of an average of 1.5 edges. Since there are twice as many faces as tetrahedrons, there are an average of 3 entries per tetrahedron. If the links are stored as lists this means a total of 6 pointers/tetrahedron (2 pointers/entry). 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 faces is therefore $6|T| + 3/2|E| + |V|$. For a typical 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. The additional space to support **findUp** on edges is 3 pointers per edge (v_1, v_2) not in E'— one for v_2, one for some v in its link, 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 are a face 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.

The technique for relabeling the vertices is 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.

Once the vertices are relabeled, the link of a vertex can be represented by concatenating the code for its

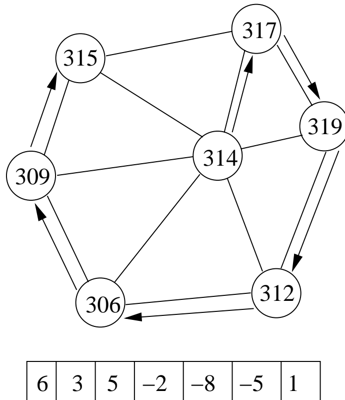

Figure 2: The neighborhood and corresponding difference code data for vertex 314. The first entry, 6, is the degree of the vertex. Other entries are the offsets of the neighbors.

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.

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 numbers 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 number, 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: $v' \leftarrow 2|v| + \mathtt{sign}(v)$.

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 eight-byte block per vertex. If a block overflows (that is, if the storage needed is greater than eight bytes), additional space is allocated from a separate pool of eight-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 $.75^{128} \simeq 10^{-16}$.

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. Fig-

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.

ure 3 shows the tradeoff between these factors for our Delaunay triangulation algorithm run on 2^{20} uniformly distributed points in the unit square. Although space is used most efficiently at a block size of 7, we chose a block size of 8 to decrease the number of hash computations needed.

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 as a linked list containing one node for each neighbor of the vertex. The linked lists are kept in a FIFO cache with a maximum capacity of 2000 nodes. Update operations may affect the linked lists while they are in the cache. The linked 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 to all of its representative out-edges. This is stored as a difference coded list of the corresponding neighbors. The code for each neighbor is followed by a code for the pointer to that edge. (The pointer is stored using the same hash trick as above to keep pointer sizes small.) Every representative edge has its own 8-byte block allocated from the memory pool, with the capability to allocate additional 8-byte blocks if needed.

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

Since each edge is stored in its own block, this representation requires many more blocks than the 2D representation. This is shown in Figure 4. The space-optimal block size is 5; this is lower than in the 2D case because of the large number of edges that take up about 5 bytes. Still, to decrease hashing overhead

Block Size	Blocks Needed	Total Space
4	8052433	66260088
5	5384387	65482812
6	3771063	65995447
7	2687996	67138778
8	1918235	68724518
9	1351559	70684974
10	944411	73245936
11	653667	76412890

Figure 4: The number of extra blocks needed for 2^{20} vertices on a uniform distribution in 3D, and the total space required if we allocate 30% more blocks than are needed. There were 3883291 representative edges in the complex.

we chose a block size of 8.

Incremental Delaunay: We implemented a Delaunay tessellation algorithm in two and three dimensions using our compressed data structure. We employ the well-known Bowyer-Watson kernel [8, 41] to incrementally generate the mesh. During the course of the algorithm a Delaunay tessellation 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 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 by means of 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 the beginning of the algorithm we relabel all input points as described above. The runtimes reported in the next section include the time for this preprocessing

Distribution	# Pts	# Extra Blocks	Time(s)
uniform	2^{18}	40985	3.01
normal	2^{18}	41540	3.40
kuzmin	2^{18}	41816	5.05
line	2^{18}	36798	3.67
uniform	2^{20}	164660	13.45
normal	2^{20}	168578	14.01
kuzmin	2^{20}	168593	22.18
line	2^{20}	155460	15.26

Figure 5: The number of extra 8-byte blocks needed to store triangular Delaunay complexes for various point distributions using our structure and the runtime of our 2D implementation.

phase.

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-, insphere tests) we use Shewchuk's adaptive precision geometric predicates [36]. 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) 8-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 extra blocks varies by about 10%. Furthermore the number of extra blocks used comes to only about 15% of the number of default blocks needed, which is one per vertex (*e.g.*, $41816/2^{18}$). In our experiments we set the number of extra blocks available to 25% of the number of default blocks. The extra blocks therefore fill to about 60% of capacity. Given this setting, the total space we require for the mesh is 1.25×8 bytes/vertex, which is 5 bytes/triangle.

Next, we compare runtime and memory usage of our implementation to Shewchuk's Triangle [35] code which is the most efficient we know of. 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 complex, point coordinates and the

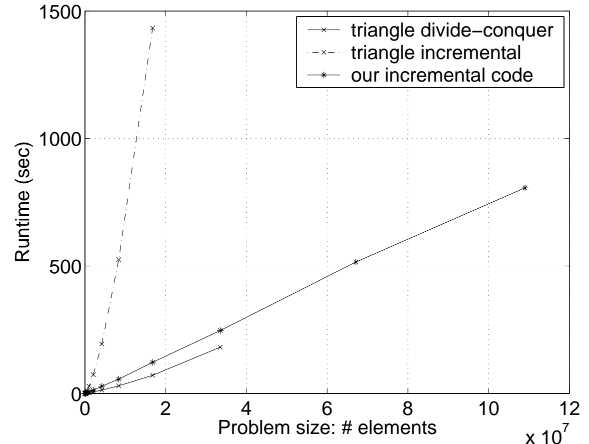

Figure 6: Runtime in 2D, uniformly random points

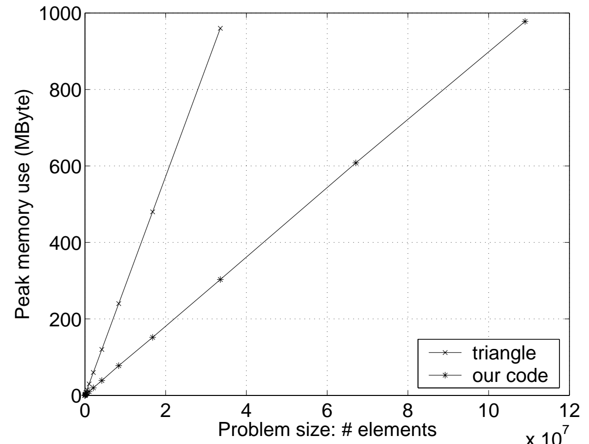

Figure 7: Memory use in 2D, uniformly random points

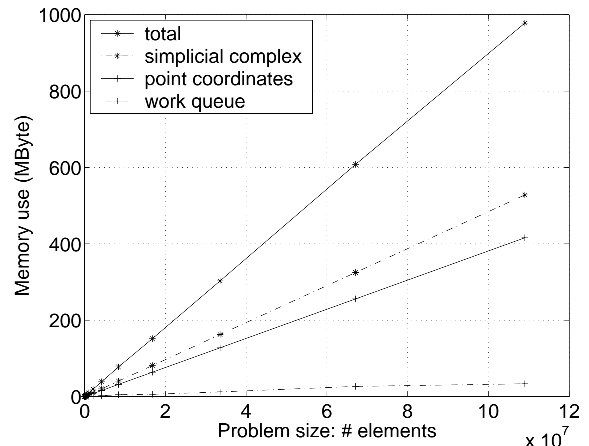

Figure 8: Breakdown of memory use in 2D, uniformly random points

Distribution	# Pts	# Extra Blocks	Time(s)
uniform	2^{16}	352448	10.496
normal	2^{16}	362115	10.995
kuzmin	2^{16}	361203	12.168
line	2^{16}	309478	9.386
uniform	2^{18}	1414620	46.570
normal	2^{18}	1469557	48.838
kuzmin	2^{18}	1464722	53.792
line	2^{18}	1295802	42.670

Figure 9: The number of extra 8-byte blocks needed to store tetrahedral Delaunay complexes for various point distributions using our structure and the runtime of our 3D implementation.

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. Results are reported in Figure 9. As in 2D the runtimes differ, but the memory needed is independent of the distribution.

We compare our 3D implementation with uniform random data to Shewchuk's Pyramid code [34][2]. 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 80% of the total memory; point coordinates and work queue account for 15% and 5%, respectively.

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.

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

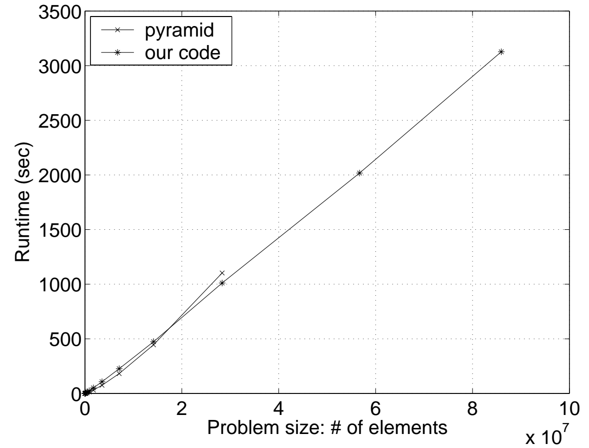

Figure 10: Runtime in 3D, uniformly random points

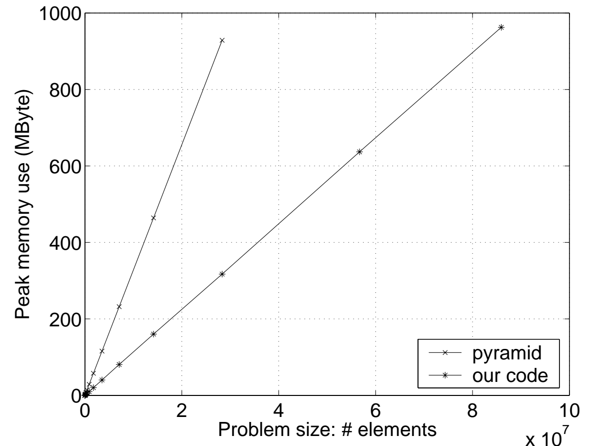

Figure 11: Memory use in 3D, uniformly random points

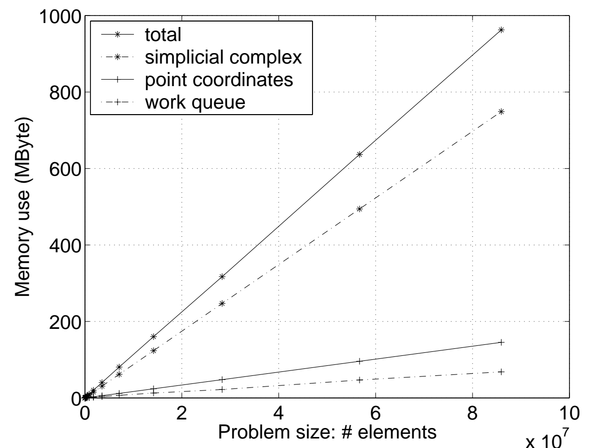

Figure 12: Breakdown of memory use in 3D, uniformly random points

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.

Some Limitations and Extensions. The effectiveness of our approach relies on the property that vertices that are close in the mesh are close in the ordering of labels. For the Delaunay code we generate the labeling using a quad/oct tree on the original set of points. In general, however, an algorithm might not have all the vertices when the mesh is constructed. For example in meshing code new points might be generated during the algorithm based on the current state of the mesh. In these situations we would want to generate each new label so that it is close to the label of its neighbors. One possible way to do this is to leave space in the initial labeling (e.g. the original n vertices are assigned labels $\{ki : i \in 1 \ldots n\}$), and to select an available label based on the labels of the neighbors.

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*, 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. Shewchuk. Pyramid mesh generator software. (`http://www.cs.berkeley.edu/~jrs/`). Personal Communication.

[35] 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.

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

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

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

[39] 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.

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

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

[42] 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.

[43] 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.

This research was sponsored in part by National Science Foundation (NSF) grant no. CCR-0122581.