Zipper: A compact connectivity data structure for triangle meshes

Topraj Gurung
Georgia Institute of Technology

Mark Luffel
Georgia Institute of Technology

Peter Lindstrom
Lawrence Livermore National Laboratory

Jarek Rossignac
Georgia Institute of Technology

Abstract

We propose Zipper, a compact representation of incidence and adjacency for manifold triangle meshes with fixed connectivity. Zipper uses on average only 6 bits per triangle, can be constructed in linear space and time, and supports all standard random-access and mesh traversal operators in constant time. Similarly to the previously proposed LR (Laced Ring) approach, the Zipper construction reorders vertices and triangles along a nearly Hamiltonian cycle called the ring. The 4.4x storage reduction of Zipper over LR results from three contributions: (1) For most triangles, Zipper stores a 2-bit delta (plus three additional bits) rather than a full 32-bit reference. (2) Zipper modifies the ring to reduce the number of exceptional triangles. (3) Zipper encodes the remaining exceptional triangles using 2.5x less storage. In spite of these large savings in storage, we show that Zipper offers comparable performance to LR and other data structures in mesh processing applications. Zipper may also serve as a compact indexed format for rendering meshes, and hence is valuable even in applications that do not require adjacency information.

Key words: triangle meshes, mesh connectivity, Hamiltonian cycle, differential coding

1. Introduction

Zipper, introduced here, is a compact representation for manifold triangle meshes with fixed connectivity. It is a randomly-accessible-and-traversable (RAT) mesh representation: it provides **constant-time** retrieval of any triangle, vertex, or corner (equivalently half-edge) in the mesh, and also supports constant-time access to the consecutive vertices around a triangle and to the consecutive incident triangles around a vertex.

Popular polygon representations such as the Half-Edge representation use 48 bpt (bits per triangle) to represent the geometry (using three 32-bit floats per coordinate) and 528 bpt to represent the connectivity [1], as discussed in Sec. 2.2. The most compact existing triangle RAT, BELR (bit-efficient LR) [2], uses 26 bpt on average for connectivity. It reorders most vertices and triangles along a ring (a nearly Hamiltonian cycle) and stores one reference for each normal triangle and up to 15 references for each exceptional triangle.

Zipper uses a similar reordering, but represents most vertex indices in normal triangles using differential coding, which reduces storage for most triangles to only a 2-bit delta and 3 additional bits, and uses 2.5x less storage for exceptional triangles. It stores connectivity using only 6 bpt (on average over tested models), and hence provides a 4.4x improvement over BELR.

This storage decrease is remarkable, because it improves by 4.4x on the best results achieved over the past 40 years by experts across many research areas. By comparison, these earlier efforts have seen a decrease in storage of 17x from 432 bpt [3] to 26 bpt [2]. Zipper uses only an equivalent of a fifth of a 32-bit reference per triangle to store enough information to access (in constant time) not only the three vertices of each triangle, but also its three neighboring triangles, as well as one incident triangle for each vertex.

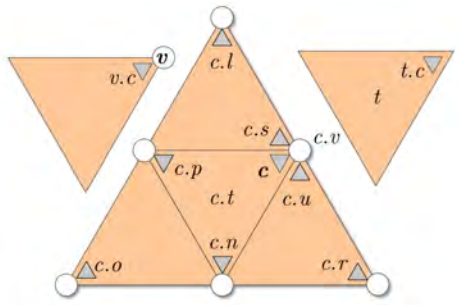


Fig. 1. The standard corner operators associate with corner c its vertex $c.v$, triangle $c.t$, and next $c.n$ and opposite $c.o$ corners. The other corner operators, previous $c.p$, swing $c.s$, unswing $c.u$, left $c.l$, and right $c.r$ are derived from $c.n$ and $c.o$. In addition, $v.c$ returns one corner of vertex v, and $t.c$ returns one corner of triangle t.

Our Zipper implementation of the standard mesh traversal operators not only has constant-time complexity, but is very fast (7–40 nanoseconds per operator). When executed on a small mesh, the basic operators for mesh access and traversal in Zipper are faster than those in BELR, but about 1.8x–3.6x slower than their counterparts in the optimized standard LR. However, the impact of this overhead on the performance of an application is often negligible when compared to other processing costs, and is more than compensated by performance gains resulting from fewer page faults when processing large meshes that do not fit in memory.

Zipper is significantly more compact than the commonly used triangle strip/fan or indexed formats. Hence, it is also a good candidate for applications (such as rendering) that do not require adjacency information.

2. Prior Art

2.1. *Corner operators*

Many mesh processing algorithms may be formulated using operators that access the next vertex around a triangle or the next triangle around a vertex. In this paper we use **triangle corners** [4], which uniquely identify a triangle and one incident vertex, to mark a specific spot on the connectivity graph of the mesh and to serve as a mesh traversal primitive. Hence, our mesh traversal operators manipulate corners, their vertices and their triangles. Note that an equivalent set of operators may be defined in terms of half-edges (also called edge-uses or darts) [1, 5], because one may trivially establish a mapping between corners and half-edges.

RAT mesh representations support the standard set of corner operators [6], defined below. The operators provide a simple mechanism to access all vertices, triangles, and corners in constant time from their IDs or from adjacent elements.

Given a corner c, the **standard corner operators** (see Fig. 1) are: the triangle $c.t$ of c, the vertex $c.v$ of c, the next corner $c.n$ in $c.t$, the opposite corner $c.o$ defined such that $c.n.v = c.o.n.n.v$ and $c.n.n.v = c.o.n.v$, a corner $t.c$ of triangle t, and a corner $v.c$ of vertex v. From these, we define a set of **derived corner operators**: the previous corner $c.p = c.n.n$ in $c.t$, the left $c.l = c.n.o$ and right $c.r = c.p.o$ neighboring corners of c, and the swing $c.s = c.l.n$ and unswing $c.u = c.r.p$ corners used to walk around $c.v$. In this paper we focus on efficiently encoding $c.v$ and $c.o$—the remaining operators can be inferred trivially (see [2]).

Early boundary graph representations of connectivity [1] use 32-bit memory pointers. Some of the more recent representations (for example [4]) assign consecutive positive integers to vertices, triangles, and corners, and use arrays to store sorted lists of references (32-bit integer indices), rather than pointers, which, for example, identify a triangle incident upon a vertex, the three vertices of a triangle, or the three opposite corners in adjacent triangles.

To simplify our algorithms and representation, as others have done in the past, we assume the mesh is manifold. Furthermore, our storage and performance statistics are only representative of meshes where the genus and the number of border edges are small relative to the number m of vertices. With these assumptions, there are roughly $n = 2m$ triangles and $3m$ edges. So, if a representation uses a total of nr references, we say that its storage cost is $32r$ bpt (bits per triangle) or r rpt (references per triangle).

To reduce connectivity storage, some representations reorder vertices, corners, or triangles. For example, ECT (the Extended Corner Table) [4, 6] encodes connectivity in 6.5 rpt, by assigning the three corners of a triangle consecutive numbers that are consistent with the orientation of the triangle. This assumption makes it unnecessary to store several of these look-up tables, because their content may be computed in constant time when needed: $c.t = \lfloor c/3 \rfloor$, $c.n = 3c.t + (c + 1 \bmod 3)$, $t.c = 3t$. Even with this improvement, connectivity accounts for up to 90% of the total storage when using 16-bit quantized coordinates to represent geometry.

2.2. *General representations*

Early representations use much storage, because they cater to more general (polygonal or higher-dimensional) meshes.

Brisson's cell-tuple structure [7] generalizes the quad-edge data structure of Guibas and Stolfi [8], which was restricted to 2-manifolds without boundaries, and the facet-edge data structure of Dobkin and Laszlo [9], which catered to subdivisions of 3-manifolds. It is restricted to subdivided manifolds with or without boundary. When applied to triangle meshes, the cell-tuple structure associates each triangle t with 6 groupings (n-tuples), each one corresponding to a choice of three entities (v, e, t): the triangle t, an edge e of t, and a vertex v of e. There are 6 groupings for each triangle because one has 3 choices for e and then 2 choices for v. With each grouping $g = (v, e, t)$, one stores a reference to triangle t and to vertex v, plus three references to adjacent groupings: $s_0(g)$ returns grouping (v', e, t), where

v' is the other vertex of e; $s_1(g)$ returns grouping (v, e', t), where e' is the other edge of t that is incident upon v; and $s_2(g)$ returns grouping (v, e, t'), where t' is the other triangle incident upon e. To support the standard operators, one also stores, for each triangle and for each vertex, a reference to one of its groupings. Hence, the total storage cost for connectivity is 31.5 rpt: 6 tuples per triangle that store 5 references each (vertex, triangle, and 3 swaps), plus a tuple reference for each vertex and triangle.

In the cell-tuple structure, groupings g, $s_0(g)$, $s_2(g)$, and $s_0(s_2(g))$ refer to the same edge. The popular Winged-Edge representation [3] combines them into a single edge, with which it associates references to its two bounding vertices, to its two incident triangles, and to the previous and next edge in each triangle. To be compatible with the RAT operators supported by other schemes, we also assume that it stores a reference to a winged-edge for each triangle and each vertex, so as to support the $v.c$, $t.c$, and $c.t$ operators in constant time. Adding these references pushes the extended winged-edge storage cost to 13.5 rpt.

The Half-Edge representation [1] associates with each half-edge a reference to the next, previous and opposite half-edge, together with a reference to a bounding vertex and incident face for a storage cost of 5 references per half-edge, or 15 rpt. Adding support for $t.c$ and $v.c$ for their half-edge counterpart yields a total cost of 16.5 rpt. The Surface-Mesh representation [10] uses half-edges, but reorders them so that opposite ones are consecutive, which eliminates one reference per half-edge. Surface-Mesh also does not store a reference to the previous half-edge in a triangle. Hence, its resulting storage cost is 10.5 rpt.

Star-vertices [11] stores for each vertex a radially sorted list of references to neighboring vertices. It also stores the reference to where that list starts. To make it compliant with our definition of RAT, we must add a reference from each triangle to one of its vertices or half-edges (equivalent to $t.c$) and a reference from each half-edge to its incident triangle. Hence the total cost includes an average of 6 references to neighboring vertices from each vertex (two per edge or equivalently 3 rpt), one reference per half-edge to an incident triangle (that amounts to 3 rpt), a reference per vertex to the start of the list (1 per vertex, or equivalently 0.5 rpt), and 1 rpt for $t.c$. Hence, the total storage for a RAT compatible star-vertices mesh is 7.5 rpt.

2.3. *Representations for triangle meshes*

Representations restricted to triangle meshes exploit the regularity of the connectivity (3 vertices per triangle and 3 neighbors per interior triangle) and reorder triangles, edges, and/or vertices.

The Directed Edge representation [12] is identical to the Corner Table [4], when considering a bijection between half-edges and corners. Both use 6.5 rpt when augmented with $v.c$ references to make them RAT compatible.

SOT [6] reorders triangles so that triangle i of the first m triangles corresponds to vertex i, where m is the number of vertices. Hence, there is no need to store $c.v$, since it may be recovered by swinging around v until the triangle $t = v < m$ is reached. The $v.c$ operator is also available implicitly. SOT stores only the adjacency table, and therefore requires only 3 rpt.

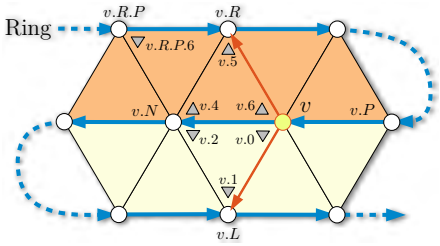


Fig. 2. With each ring vertex v, LR stores references $v.L$ and $v.R$ to the tips of the two triangles incident upon ring edge $(v, v.N)$.

Catalog-based representations [13] group triangles into facets and, instead of storing the internal connectivity of each facet, they store a reference to a catalog, in which the results of mesh operators are stored for look-up. The most compact version uses 3.83 rpt. SQuad [14] uses a very small catalog. It matches most vertices with an adjacent pair of incident triangles, making it possible to avoid storing one incidence reference and one adjacency reference for most triangles. Hence SQuad uses slightly more than 2 rpt, depending on the mesh.

Tripod [15] computes minimal Schnyder woods to orient the edges and group them into 3 sets, each defining a vertex spanning tree. It stores, for each vertex, the 3 references to its parent in each tree (3 outgoing edges) and also 3 references to incoming edges (each one radially following an outgoing edge). Making it compliant to support $t.c$ and to store a triangle ID with each half-edge brings the total cost to 7 rpt. A recently proposed variation [16] offers the option of reordering the vertices to further reduce storage to 2 rpt (not counting the cost of $t.c$ and references from edges to incident triangles).

3. LR

The Zipper approach presented here builds upon the recently introduced LR representation [2]. LR reorders the vertices and triangles of a mesh along a nearly Hamiltonian cycle called the ring. Given a ring vertex (a vertex visited by the ring) v, let $v.N$ and $v.P$ denote the next and previous vertex along the ring. $v.N$ and $v.P$ can be obtained by incrementing or decrementing v modulo m_r, where m_r is the number of ring vertices. LR associates the two triangles incident on a (directed) ring edge $e = (v, v.N)$ with vertex v. LR then stores, for each v, the (integer) references $v.L$ and $v.R$ to the tip vertices (those not on e) of the two triangles incident upon e. Fig. 2 shows a common arrangement.

The ring typically contains more than 99.99% of the vertices and is computed using the linear-time "Ring-Expander" algorithm. The few isolated vertices are handled as exceptions.

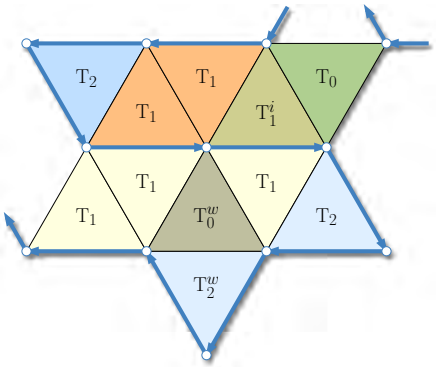


Fig. 3. The classification of triangles (T_0, T_1, T_2) in relation to the ring. A superscript w denotes wart triangles and a superscript i denotes triangles adjacent to one or more non-wart T_0 triangles.

3.1. *Ring-based classification of triangles*

The ring defines three types of triangles: T_1 (or **normal**) triangles are bounded by a single ring edge; T_2 (or **dead-end**) triangles have 2 ring edges; and T_0 (or **bifurcation**) triangles have none (see Fig. 3). Most triangles are of type T_1.

A T_2 triangle (which may be easily identified at runtime by testing whether $v.N.L = v$) is incident upon two consecutive ring edges, say $(v, v.N)$ and $(v.N, v.N.N)$. LR associates T_2 triangles with the second one of these edges.

LR identifies adjacent T_0/T_2 pairs and calls them **warts**. About 80% of T_0 triangles are warts. LR uses the notation T_0^w and T_2^w for triangles in warts. We use the term **split** to refer to a non-wart T_0 triangle that has 3 ring vertices and is not adjacent to another T_0 triangle, and the term **end** to refer to a non-wart T_2 triangle.

T_0 (bifurcation) triangles that are not part of a wart are stored in a separate Corner Table [4].

A small number of vertices are not part of the ring (typically about ∼0.005%). LR calls these vertices and their incident triangles **isolated**. Any isolated triangle is a T_0, because if it were bounded by a ring edge, the ring could be trivially expanded to include the isolated vertex.

3.2. *Adjacency*

With each vertex v on the ring LR associates two triangles whose corners are labeled $(v.0, v.1, \ldots, v.6)$ (see Fig. 2). LR does not need to store opposite adjacency references between corners $v.1$ and $v.5$ as this information can be inferred. The opposites of the other 4 corners $(v.0, v.2, v.4, v.6)$ may often be recovered using simple sequences of the ring operators: $v.L, v.R, v.P$, and $v.N$ (see Fig. 2 for an example where $v.6.o$ may be computed as $v.R.P.6$). When recovering these opposites, LR examines a small, fixed set of such sequences.

For T_0 triangles, there is no guaranteed constant-time recipe to recover their corners from opposite ones by using a constant length cascade of vertex operators. Hence, these references must be stored explicitly. Because these explicit references do not fit in the regular LR ring structure, an indirection (exception) is used to store them.

LR combines three main ideas:

(i) LR stores vertices in ring order, and hence references from a ring vertex v to the next $v.N$ or previous $v.P$ ring vertex are implicit. For most ring vertices, triangles $(v, v.L, v.N)$ and $(v, v.N, v.R)$ are implicitly defined and associated with IDs $2v$ and $2v + 1$.

(ii) LR stores the references $v.L$ and $v.R$ explicitly. Given that most vertices are in the ring, this amounts to storing one reference (32 bits) per triangle.

(iii) LR uses additional storage (up to 15 references per exceptional triangle) for representing the connectivity around the exception triangles that are not incident on any ring edge.

4. Zipper

Zipper uses the Ring-Expander algorithm from LR to build a ring and to renumber the vertices, triangles and corners. Zipper provides three improvements to LR:

(i) To reduce the number of exceptions, Zipper applies the Ring-Bender algorithm, as described in Sec. 5.

(ii) Zipper reduces by 2.5x the storage cost associated with exception triangles. It does so by inferring connectivity by locally traversing the ring.

(iii) Zipper avoids storing most of the $v.L$ and $v.R$ references explicitly. Instead it stores a pair of 3-bit codes for most ring vertices. These identify exceptional triangles and encode deltas rather than absolute references. To help resolve these references in constant time, Zipper stores two additional bits (amortized) per triangle.

We point out that BELR is also based on differential coding. Instead of storing $v.L$, BELR stores the difference $v.L - v$. Here, we propose a different and novel coding: we store $v.P.L - v.L$ (described in Sec. 4.1). This discovery of a new differential coding is a major factor in the large reduction in storage cost offered by Zipper over LR and BELR. These deltas are organized into fixed-sized blocks (described in Sec. 4.2).

Based on these improvements, Zipper reduces connectivity storage to about 6 bpt, which represents a 35x improvement over the standard ECT, a 5.8x improvement over LR, and a 4.4x improvement over BELR.

In Sec. 4.3, we describe how Zipper computes $c.o$ for different triangle types.

4.1. *Delta codes for vertices*

The most significant storage improvement in Zipper comes from the observation that, in most cases, $v.L$ can be recovered by subtracting a small integer decrement from $v.P.L$ (and similarly for $v.R$). Our tests indicate that in a typical mesh, 95% of the **deltas** $v.\Delta_L = v.P.L - v.L$ and

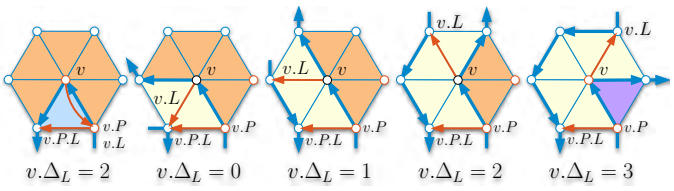


Fig. 4. The ring (blue) passes through a valence-six vertex v. The $v.L$ and $v.P.L$ references are shown as red arrows. In absence of incident T_0 triangles, only deltas in $\{0, 1, 2, 3\}$ are possible at v.

$v.\Delta_R = v.P.R - v.R$ are in the set $D = \{0, 1, 2, 3\}$. Hence, in these cases, Zipper stores only two bits per delta, instead of the 32-bit $v.L$ or $v.R$ references. Note that these delta values are typically non-negative, because the ring bounds "strips" of edge-adjacent triangles, and therefore $v.L$ and $v.R$ decrease as v increases (see Fig. 2).

To understand why most deltas are in D, consider the common case of a valence-6 vertex, as shown in Fig. 4. When the triangles incident on v are T_1 or T_2 (such triangles make up over 96% of the mesh), only the $0, 1, 2, 3$ deltas are possible. This observation also holds for vertices of lower valence.

Configurations where the delta is not in D are flagged as **exceptions**. For each exception we store a full 32-bit reference to the corresponding tip vertex. We refer to such tip vertices ($v.L$ or $v.R$) as **key vertices**.

As in LR, we identify warts (pairs of adjacent T_0/T_2 triangles). Because T_2 triangles have two ring edges and would thus be represented twice, we store the adjacent T_0 in place of the first (along the ring) T_2 copy. Such a wart pair, which we label T_0^w/T_2^w, is illustrated by triangles #95 and #97 in Fig. 8. Because not all T_2 triangles are adjacent to a T_0, we store a **wart bit** with each ring triangle to indicate whether it is a T_0^w.

Unlike in LR, we use a special encoding of T_2^w triangles $(v.P, v, v.N)$ to reduce the number of exceptions. Rather than storing $v.P$ (vertex #47 in Fig. 8) as the T_2^w tip vertex, which often would incur an exception, we set delta to zero and rely on the fact that we can always recover the tip vertex $v.P$ from v when the wart bit of the previous triangle is set. This encoding also ensures that the delta of the following triangle (for example triangle #99 in Fig. 8) is computed with respect to the T_0^w tip vertex (vertex #37 in Fig. 8). Furthermore, it avoids having to encode a second exception.

Adding a wart bit to the two-bit deltas results in a 3-bit encoding of each triangle. We reserve the 3-bit pattern 111 to mark exceptions, which would otherwise correspond to the rare case of $\Delta = 3$ in a T_0^w wart triangle.

4.2. *Blocks*

To recover $v.L$ and $v.R$ without summing all preceding deltas along the ring, we force an exception every 32 $v.L$ and $v.R$ references, and store these explicitly. We refer to such a sequence of 32 references as a **block**. Thus, computing $v.L$ or $v.R$ requires summing at most 31 deltas. To accelerate this key step, we have devised an efficient technique that computes the sum of deltas using bit-level operations, without executing a loop, as described in Sec. 6. Because the number of exceptions per block varies, each block stores a single index into an **exception table** (a dense array of key vertex references). The storage required for a block includes (1) a 32-bit reference for the first vertex of each block, (2) a sequence of 32 3-bit delta/wart codes, and (3) a 32-bit pointer into the exception table, resulting in a minimum of 160 bits per block (5 bpt). We chose a block size of 32 as a compromise: (1) we want the block to be large, so that we can amortize the cost of storing the $v.L$ and $v.R$ references at the beginning of each block and (2) we want to be able to compute the sum of the deltas efficiently.

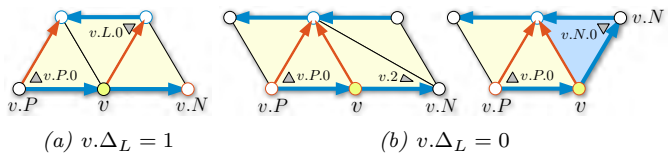


Fig. 5. Opposites can be efficiently computed for $\Delta \in \{0, 1\}$. For instance, $v.P.0.o = v.L.0$ whenever $v.\Delta_L = 1$.

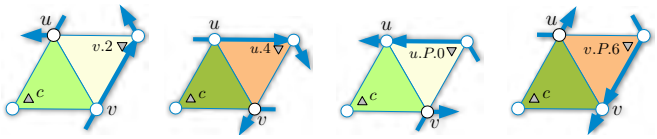
Whereas in LR each reference is stored as 32 bits, Zipper allows references to be stored in only 5 bits. References that generate exceptions require 36 bits. Consequently, in the best case we improve storage by 6.4x over LR, and in the worst case, when every reference is an exception, we introduce an overhead of $\frac{1}{8}$. In the best case, we improve storage by 5.2x over BELR, and in the worst case, we introduce an overhead of 38%.

4.3. *Computing opposites*

LR derives adjacency information (the $c.o$ references) by using reference-equality tests and combinations of $v.N$, $v.P$, $v.L$, and $v.R$ references. We explain here how we had to modify this approach to accommodate Zipper's more compact representation.

4.3.1. *From T_1 to T_1*

For T_1 and T_2 triangles, we do not store opposite corners $c.o$ explicitly, but compute them when needed by decoding the $v.L$ and $v.R$ references, and by using the implicit next $v.N = v + 1 \bmod m_r$ and previous $v.P = v - 1 \bmod m_r$ ring operators (where m_r is the number of ring vertices). An example of computing opposite corners is shown for $v.6$ in Fig. 2. When only T_1 and T_2 triangles are involved, we detect the local ring's configuration and return the appropriate corner. The four cases that need to be checked are detailed in LR [2]. We can often compute opposite corners by examining only the delta, i.e. without fully decoding $v.L$ or $v.R$. As shown in Fig. 5, this is possible whenever $\Delta \in \{0, 1\}$.



(a) $v.L = u$ (b) $u.R = v$ (c) $u.P.L = v$ (d) $v.P.R = u$

Fig. 6. The four cases for finding $c.o$ when $c.t$ is a T_0 triangle and $c.o.t$ is a T_1 or T_2 triangle. The expression for computing $c.o$ is shown inside either the orange or cream triangle.

4.3.2. *From T_0 to T_1*

Unlike in LR, which stores both vertices and opposites explicitly for all T_0 triangles, Zipper uses a different procedure for inferring opposites of T_0 corners, when those do not lie in another T_0, and stores only the three vertex references. Given the two vertices of the T_0 not incident on c (see Fig. 6), we use the $v.P$, $v.L$, and $v.R$ operators to navigate to the opposite corner. The four possible configurations are illustrated in Fig. 6.

Because in practice roughly half of the T_0 triangles are adjacent to three T_1 triangles, this elimination of three references per T_0 has a significant impact on storage.

4.3.3. *Hashing to a T_0*

When the opposite corner $c.o$ lies in a T_0, it is not possible to find it using only $v.L$, $v.R$, $v.N$, and $v.P$. When c lies in a T_1 (referred to as a T_1^i triangle), LR stores a special pointer into a table that holds both the tip vertex and the up to two unknown opposites of $c.t$. To avoid the cost of storing these exceptional references explicitly, we use an alternative data structure that stores only opposite corners for exceptional configurations. This data structure is used for all opposites that cannot be inferred from the rules above, i.e. for all corners $c.o$ that lie in a T_0.

For space efficiency, we use a d-ary cuckoo hash [17], which maps each key to one of d possible locations, out of which one is guaranteed to hold the hashed item. Aside from $O(N)$ insertion of N items and $O(1)$ lookups, cuckoo hashes have a desirable property in that, as d grows, the maximum allowable load factor f approaches one. In practice, $f = 97\%$ when $d = 4$ (the setting we used), thus only 3% is wasted on empty slots.

In Zipper, we use c as the key and store only the value $c.o$ in the hash. A typical hash lookup generates d possible candidates, which direct us to triangles $c.o.t$ in the T_0 table. Among the d candidates, we identify the one that contains the edge $e = (c.p.v, c.n.v)$ shared with $c.t$. If no such triangle is found, $c.o$ does not exist, indicating that e is a border edge. The cost of storing an explicit opposite reference is thus $\frac{32}{f}$, or about 33 bits when $d = 4$.

LR stores up to 15 references for each T_0 triangle (3 to its vertices, 3 to opposite corners, and 9 from adjacent triangles). Since, in a typical mesh, the percentage of T_0 triangles varies from 0.5% to 2.0%, using the LR approach to represent T_0 triangles adds between 2.5 and 10 bpt (a cost obtained by amortizing the storage cost of these exceptions over all triangles), and hence dominates the storage cost.

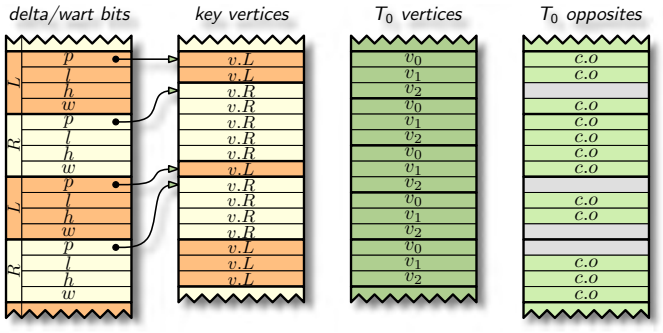


Fig. 7. Zipper storage. Delta/wart bits: For each run of 32 $v.L$ or $v.R$ references, we store a block consisting of four 32-bit integers that represent a key vertex pointer p and, for each vertex in the run, a low and high delta bit l and h, and a wart bit w, respectively. Key vertices: Pointer p points to the exception table containing the key vertices. T_0 vertices: vertex IDs for T_0 triangles. T_0 opposites: 4-ary cuckoo hash table containing opposite corner IDs for T_0 triangles.

With our improved scheme, we store an average of 6.0 references per T_0 triangle.

One attractive property of Zipper is that, unlike LR, it fully separates the representation of vertices and opposite corners. For those applications that do not require adjacency (e.g. rendering, transmission, etc.), the corner hash may be discarded without having to modify the Zipper incidence representation.

5. Ring-Bender

In addition to reducing storage for T_0 triangles, we may also reduce their frequency. We observe that every T_0 split triangle is connected to one or more T_2 triangles by a series of T_1 triangles that we call a **branch**. By applying the Ring-Bender algorithm, we iteratively shorten branches until the T_2 and T_0 become adjacent, and thus can be encoded as an inexpensive wart. We apply Ring-Bender after constructing the ring using Ring-Expander [2] and before delta encoding.

To remove the split triangles, we reroute the ring, starting from the vertex v shared by both ring edges in a T_2 triangle (highlighted in Fig. 9 and Fig. 10). In effect, this rerouting "flips" the T_2 to the other side of the ring, shortens the branch bounded by the T_2, and leaves v isolated. To bring v back into the ring, we flip one of its incident T_1 triangles. This procedure is performed iteratively until the branch is eliminated and a wart is created. Note that we allow a pair of triangles to be flipped so long as no new T_0 splits are introduced.

We show the result of a single step of Ring-Bender in Fig. 9. Notice that the T_0 triangle at the bottom is converted to a T_0^w, allowing us to encode it as a ring triangle. Ring-Bender may create as many as three warts in each step, but because Zipper encodes warts as if they were two T_1 triangles, using this solution does not increase the storage or execution cost.

We described in the previous paragraph the most common configuration. Our algorithm, in Listing 1, distin-

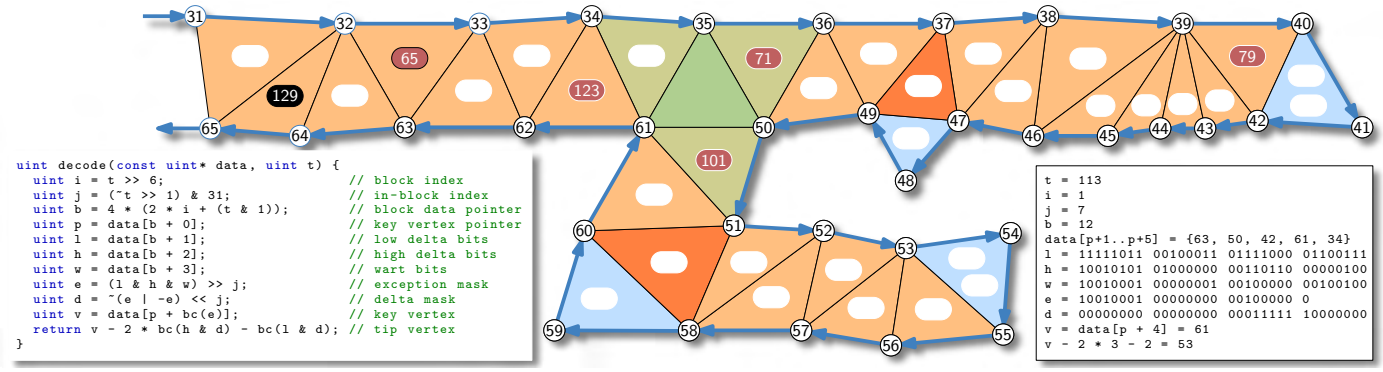
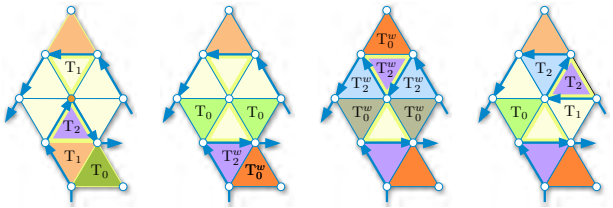
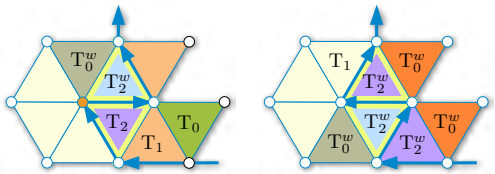


Fig. 8. Code for decoding $v.L$ or $v.R$ of a triangle (left), example block of 32 triangles $\{65, 67, \ldots, 127\}$ (center), and corresponding execution of the code (right) for triangle 113. The triangle numbers for exceptions (e.g. $65, 71, \ldots$) are marked red. The 128-bit fixed-size block data along with five 32-bit key vertices encode this block using 9 bpt.



(a) before *(b) flip 1* *(c) flip 2* *(d) bad flip 2*

Fig. 9. A single step of Ring-Bender results in exchanging a pair of triangles between the two sides of the ring (we say that we "flip" them). (a) We begin at the marked vertex of a T_2 triangle, (b) flip the T_2 to make a wart, and (c) make another flip to convert the two new T_0 triangles into warts. (d) If we flip the purple T_2 in the second step, we will fail to make warts of the T_0 triangles.



(a) before *(b) after*

Fig. 10. Ring-Bender zig-zag configuration corresponding to the clause on line 8 in Listing 1. Here, the flipped triangles are an adjacent T_2/T_2 pair, rather than the non-adjacent T_2/T_1 pair in Fig. 9.

guishes (line 8) the special situation when two T_2 triangles are adjacent (Fig. 10).

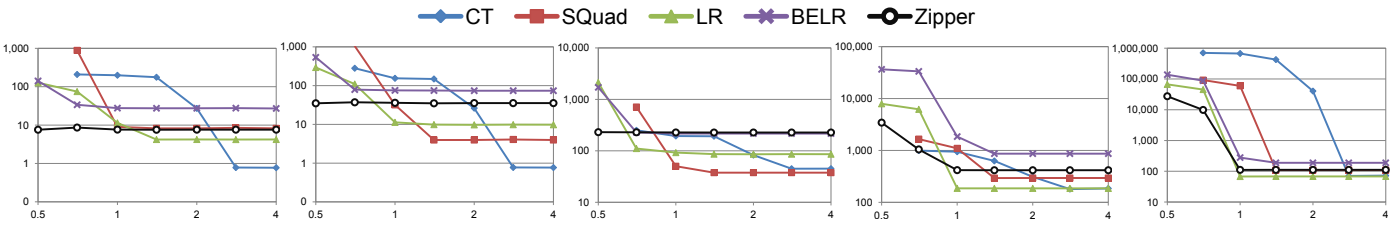
```
 1: do {
 2:   changed = false
 3:   for tri in triangles(mesh) {
 4:     if isT2(tri) {
 5:       tip = tipVertex(tri)
 6:       for neighbor in incidentTriangles(tip) {
 7:         if (isT1(neighbor) && !adjacent(neighbor, tri))
 8:         || (isT2(neighbor) && adjacent(neighbor, tri)) {
 9:           flipSides(tri)
10:           flipSides(neighbor)
11:           if anyT0(adjacentTriangles(tri)) {
12:             flipSides(tri)
13:             flipSides(neighbor)
14:           } else {
15:             changed = true
16:   } } } } }
17:} while changed
```

Listing 1. Ring-Bender code

6. Implementation details

The most important change to the implementation from LR is the computation of the $v.L$ and $v.R$ references. We explain here how we compute the references to the tip vertex for a given ring triangle t. C code for this computation and an accompanying example are presented in Fig. 8.

As in LR, we number left and right ring triangles interleaved: triangles on the left have even indices; those on the right are odd. For a given ring triangle t, we first identify the block $i = \lfloor \frac{t}{2\times 32} \rfloor$ and the index $j \in \{0, \ldots, 31\}$ within the block associated with t. We store the two delta bits and the single wart bit separately as three consecutive 32-bit words, such that bit j within each word is associated with triangle $31 - j$ within the block (i.e. the most significant bit corresponds to the first triangle). We then fetch and bitwise AND the delta and wart words to compute an exception mask e (recall that exceptions are assigned the 3-bit binary code 111). To determine the number of exceptions that precede t within the block, we first shift out any exceptions that follow t and then count the number of set bits remaining. Bit counting can be done in constant time using the SSE4 POPCNT assembly instruction, accessible via the `gcc __builtin_popcount()` function. (Current Intel, AMD, and nVIDIA processors have hardware support for the POPCNT assembly instruction.) The result is an index into the exception list where the most recent key vertex v is stored. We then form another mask d that has all bits set for triangles between t and the exception, i.e. d flags those deltas that require summation. This summation is again accomplished in constant time using bit counting of the low and high delta bits. The accumulated delta is then subtracted off from the key vertex. Each of these steps can be accomplished in constant time using no branches or loops. Our implementation of this process is extremely efficient: it compiles to only 33 assembly instructions.



(a) c.v *(b) c.o* *(c) valence* *(d) normal* *(e) contour*

Fig. 11. Per-element execution time (in nanoseconds) as a function of available main memory (GB) for various micro-benchmarks.

7. Results

We report in this section storage and performance results for our Zipper mesh data structure and compare with prior approaches. For these results we used a 2.66 GHz Intel Core i7 MacbookPro with 8 GB of 1067 MHz DDR3 memory. Our code was compiled using gcc 4.6.1 with the -O3 and -msse4 compiler flags. As benchmark meshes we used the collection from [2].

Our Ring-Bender technique, though simple, is quite effective at converting expensive T_0 triangles to cheap warts. Ring-Bender reduces the median fraction of T_0 triangles (relative to the total number of triangles) from 0.574% to 0.249%—a reduction of 2.3x—while increasing the ratio of warts from 2.25% to 3.24%. This reduces the storage cost by roughly 1 bpt on average.

The storage cost for Zipper can be expressed in terms of the number of ring triangles n_r, conditional exceptions n_e (i.e. not including the first key vertex in each block), T_0 triangles n_0, opposite corners n_c that cannot be inferred, total mesh triangles n, and the hash load factor f. Since the hash stores only T_0 corners, $n_c = 3n_0$. Thus the total cost is $\frac{32}{n}\left(5\lceil\frac{n_r}{32}\rceil + n_e + 3n_0 + \frac{1}{f}n_c\right) \simeq 5 + 32\frac{n_e+6n_0}{n}$ bits per triangle, where we have used the approximations $n \simeq n_r$ and $f \simeq 1$. We break the above sum down into the per-triangle storage cost for blocks (including the compulsory first key vertex in a block), conditional key vertices, and T_0 vertices and opposites.

In Table 1, we report these costs for our benchmark meshes, and compare them with the storage cost for the (extended) CT [4], SQuad [14], and LR and BELR [2] data structures, with BELR being the previously most storage efficient random-access data structure for triangle meshes. Similar to [2,14], Zipper storage increases with mesh irregularity (i.e. fewer valence-6 vertices). As seen in this table, the median storage needed for Zipper is 5.98 bpt, which is reduced to 5.74 bpt when adjacency is not needed. Standard LR (with adjacency) stores on average 34.6 bpt, which is a factor of 5.8 more than Zipper storage. BELR (with adjacency) stores on average 26.2 bpt, or 4.4x more than Zipper. Indeed, the storage cost of Zipper is only about 3.7x higher than the theoretical minimum of 1.62 bpt [18] achieved by state-of-the-art sequential mesh compressors that do not support random access [19].

We also compare the performance of Zipper and the above data structures (Fig. 11) as the amount of main memory available is varied, using the micro-benchmarks proposed in [2]. Our experiments were performed on the David mesh (55.5 million triangles). These results suggest that access speed is higher for the more verbose data structures when sufficient main memory is available, as accessing the more compact data structures involves more computation and nonsequential memory accesses. When the mesh fits in RAM, the Zipper *c.v* operator is 1.8x slower and the *c.o* operator is 3.6x slower than in LR, though the Zipper operators are 2–3 times faster than their BELR counterparts. Our naive version of Zipper, which loops over runs of vertices to sum up deltas, runs ten times slower than the optimized version reported here. As seen in Fig. 11, compactness becomes beneficial when less memory is available, as then page faults eventually dominate the access time. Being the most compact data structure by far, Zipper provides the highest performance under memory pressure.

Although these low-level kernels stress the performance of the mesh access operators, they may not be representative of actual mesh processing applications, which normally maintain additional data structures and perform more complex computations. As one example application, we implemented Dijkstra's (Euclidean) shortest path algorithm between mesh vertices, which requires access to geometry, the ability to mark visited vertices, and maintenance of a priority queue. We timed this (still rather simple) application on the Welsh dragon mesh (which fits in RAM), and found Zipper to be only 1.18x, 1.39x, and 1.42x slower than Surface-Mesh [10], CT, and LR, respectively. With a trend of decreasing memory per core and inter-core bandwidth, we expect that the substantial savings in storage afforded by our Zipper data structure will eventually lead to faster mesh access than using more verbose representations.

8. Conclusions

We present the Zipper data structure for representing triangle meshes. Zipper reduces the storage needed for the connectivity information to about 6 bits per triangle or equivalently to 0.19 rpt. This represents a 5.8x reduction of storage with respect to standard LR and a 4.4x storage reduction with respect to BELR (bit-efficient LR).

Zipper combines the LR approach with three improvements: differential coding, inference of adjacency information, and a method for reducing the number of costly split triangles. We have developed a simple and efficient (linear-

mesh	n	%v6	delta frequency (%)					Zipper storage cost (bpt)					storage cost (bpt)				ratio to Zipper			
			Δ_0	Δ_1	Δ_2	Δ_3	ex.	block	key	T_0 v	T_0 o	total	CT	SQuad	LR	BELR	CT	SQuad	LR	BELR
bunny	69,451	75.1	23.8	51.8	17.7	2.3	4.5	5.022	0.426	0.231	0.238	5.92	208	65.73	33.98	20.27	35.13	11.10	5.74	3.42
rocker	80,354	65.2	26.0	48.4	18.7	2.9	4.0	5.006	0.286	0.161	0.167	5.62	208	65.73	33.76	18.37	37.01	11.70	6.01	3.27
horse	96,966	66.5	23.7	52.3	17.6	2.4	4.0	5.006	0.279	0.179	0.186	5.65	208	65.48	33.76	21.59	36.81	11.59	5.98	3.82
dinosaur	112,384	57.9	29.3	43.3	18.8	3.7	4.9	5.006	0.572	0.344	0.356	6.28	208	66.30	35.39	26.21	33.12	10.56	5.64	4.17
armadillo	345,944	52.6	30.3	41.2	20.2	4.0	4.3	5.002	0.381	0.198	0.205	5.79	208	66.21	34.37	26.12	35.92	11.43	5.94	4.51
hand	654,666	53.4	32.5	38.0	19.6	4.4	5.5	5.000	0.754	0.416	0.430	6.60	208	67.07	37.25	33.86	31.52	10.16	5.64	5.13
buddha	1,087,716	32.1	39.1	26.9	16.4	5.1	12.6	4.997	3.016	2.635	2.720	13.37	208	68.80	50.66	45.26	15.56	5.15	3.79	3.39
welsh	2,210,378	86.7	21.7	54.9	18.6	1.3	3.4	5.000	0.094	0.058	0.059	5.21	208	64.86	32.64	26.12	39.92	12.45	6.26	5.01
thai	10,000,000	44.4	35.4	33.5	18.7	5.3	7.1	5.000	1.264	0.839	0.866	7.97	208	67.55	40.32	34.35	26.10	8.48	5.06	4.31
david	55,514,795	51.6	28.2	45.1	18.0	3.8	4.8	5.010	0.533	0.247	0.254	6.04	208	66.62	34.85	28.89	34.44	11.03	5.77	4.78
median		55.7	28.7	44.2	18.6	3.8	4.6	5.004	0.480	0.239	0.246	5.98	208	66.26	34.61	26.16	34.79	11.07	5.75	4.24
mean		58.5	29.0	43.5	18.4	3.5	5.5	5.005	0.760	0.531	0.548	6.84	208	66.44	36.70	28.10	32.55	10.36	5.58	4.18

Table 1
For each mesh we indicate its triangle count n and percentage of valence-6 vertices; the percentage of delta values 0–3 and exceptions; the Zipper storage cost for fixed-size block data, conditional key vertices, T_0 vertex references, T_0 opposite references; and the total Zipper, CT, SQuad, LR and BELR storage cost (in bits per triangle) and ratio relative to Zipper.

time and space complexity) Zipper construction algorithm and an efficient implementation of constant-time random access and traversal operators, which are between 1.8 and 3.6 times slower than their counterparts in LR. Our mesh traversal and query operators on Zipper are faster than operators on BELR.

We show that in some configurations, this storage reduction (when compared to LR) leads to fewer page faults and cache misses, and that the associated performance improvements exceed the performance loss of the operators.

Finally, we suggest Zipper as a more compact alternative to previously proposed formats for triangle meshes and that Zipper may be useful even in applications that do not require adjacency information, such as rendering large models on mobile devices.

References

[1] M. Mantyla, Introduction to Solid Modeling, Computer Science Press, 1988.

[2] T. Gurung, M. Luffel, P. Lindstrom, J. Rossignac, LR: Compact connectivity representation for triangle meshes, ACM Transactions on Graphics 30 (4) (2011) 67:1–67:8.

[3] B. G. Baumgart, Winged edge polyhedron representation, Tech. Rep. CS-TR-72-320, Stanford University (1972).

[4] J. Rossignac, A. Safonova, A. Szymczak, 3D compression made simple: Edgebreaker on a corner-table, in: International Conference on Shape Modeling & Applications, 2001, pp. 278–283.

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

[6] T. Gurung, J. Rossignac, SOT: Compact representation for triangle and tetrahedral meshes, Tech. Rep. GT-IC-10-01, Georgia Institute of Technology (2010).

[7] E. Brisson, Representing geometric structures in d dimensions: Topology and order, in: Proceedings of the fifth annual Symposium on Computational geometry, 1989, pp. 218–227.

[8] L. Guibas, J. Stolfi, Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams, ACM Transactions on Graphics 4 (2) (1985) 74–123.

[9] D. P. Dobkin, M. J. Laszlo, Primitives for the manipulation of three-dimensional subdivisions, in: Proceedings of the third annual Symposium on Computational geometry, 1987, pp. 86–99.

[10] D. Sieger, M. Botsch, Design, implementation, and evaluation of the surface_mesh data structure, in: International Meshing Roundtable, 2011, pp. 533–550.

[11] M. Kallmann, D. Thalmann, Star-vertices: a compact representation for planar meshes with adjacency information, Journal of Graphics Tools 6 (1) (2001) 7–18.

[12] S. Campagna, L. Kobbelt, H.-P. Seidel, Directed edges—a scalable representation for triangle meshes, Journal of Graphics Tools 3 (4) (1998) 1–11.

[13] L. Castelli Aleardi, O. Devillers, Catalog based representation of 2D triangulation, International Journal of Computational Geometry & Applications 21 (4) (2011) 393–402.

[14] T. Gurung, D. Laney, P. Lindstrom, J. Rossignac, SQuad: Compact representation for triangle meshes, Computer Graphics Forum 30 (2) (2011) 355–364.

[15] J. Snoeyink, B. Speckmann, Tripod: A minimalist data structure for embedded triangulations, in: Computational Graph Theory and Combinatorics, 1999.

[16] L. Castelli Aleardi, O. Devillers, Explicit array-based compact data structures for triangulations, Tech. Rep. 00623762, INRIA (2011).

[17] D. Fotakis, R. Pagh, P. Sanders, P. Spirakis, Space efficient hash tables with worst case constant access time, Lecture Notes in Computer Science 2607 (2003) 271–283.

[18] W. Tutte, A census of planar triangulations, Canadian Journal of Mathematics 14 (1962) 21–38.

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