EUROGRAPHICS 2011 / M. Chen and O. Deussen
(Guest Editors)

Volume 30 (2011), Number 2

SQuad: Compact Representation for Triangle Meshes

Topraj Gurung[1], Daniel Laney[2], Peter Lindstrom[2], Jarek Rossignac[1]

[1]Georgia Institute of Technology
[2]Lawrence Livermore National Laboratory

Abstract

The SQuad data structure represents the connectivity of a triangle mesh by its "S table" of about 2 rpt (integer references per triangle). Yet it allows for a simple implementation of expected constant-time, random-access operators for traversing the mesh, including in-order traversal of the triangles incident upon a vertex. SQuad is more compact than the Corner Table (CT), which stores 6 rpt, and than the recently proposed SOT, which stores 3 rpt. However, in-core access is generally faster in CT than in SQuad, and SQuad requires rebuilding the S table if the connectivity is altered. The storage reduction and memory coherence opportunities it offers may help to reduce the frequency of page faults and cache misses when accessing elements of a mesh that does not fit in memory. We provide the details of a simple algorithm that builds the S table and of an optimized implementation of the SQuad operators.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Data structures—Triangle Meshes

1. Introduction

We propose the *SQuad* data structure for representing and processing triangle meshes. It requires less storage than previous practical solutions and supports efficient operators for traversing the mesh that provide access to boundary, adjacent, and incident elements for the vertices, edges, and triangles of the mesh.

The need for space efficient mesh representations has resulted in a series of compression methods and lightweight mesh data structures. This previous work was motivated in part by the fact that the complexity (triangle count) of meshes grows continuously to meet the accuracy needs of applications. While modern workstations have powerful processors and ample memory for many tasks, research in compact mesh representations is of perennial importance because of two trends in compute hardware. The first is the rise of the handheld device, and the second is the move towards many-core architectures. In the former case, we have returned to the regime of limited memory resources; in the latter, the per-core memory size and bandwidth are limiting factors. The bottleneck on future massively multi-core architectures is shifting from the processing cost to data access cost, which is significantly increased by page faults and cache misses.

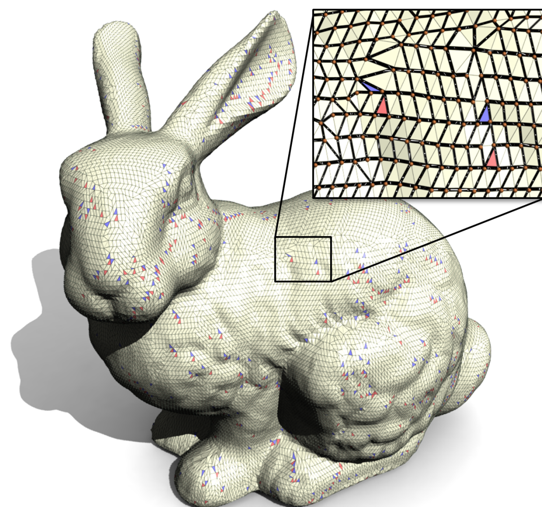


Figure 1: *SQuad pairs most triangles (here 97.3%) into quads and matches each vertex with a different quad or single triangle. By sorting the quads and single triangles to match the order of the vertices, one may represent the connectivity of the mesh by storing only 2.05 references per triangle. The rare single triangles are colored blue (unpaired) or red (unmatched).*

© 2011 The Author(s)
Journal compilation © 2011 The Eurographics Association and Blackwell Publishing Ltd. Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

In distributed applications, such as large-scale visualization and data analysis, simple pointer-free mesh representations are advantageous because they are straightforward to communicate between processors and to write to disk. Compact representations gain further advantage by requiring less off-processor and disk bandwidth for transfers.

Two complementary strategies may be used to attack the problem of reduced memory resources: (1) coherent data layout and access and (2) reduction of storage size. The former was explored elsewhere [YLPM05]. We address the latter here, and show that our solution meshes well with prior solutions to the former.

1.1. Contributions

We propose the SQuad representation, which reduces the storage requirements for the connectivity graph to about 2 rpt (integer references per triangle). We achieve this by matching pairs of adjacent triangles with one of their shared vertices, and by reordering the triangles (but not the vertices) so that part of the connectivity graph can be inferred at run time. We present a linear-cost construction algorithm and describe an optimized implementation of the operators that may be used to traverse the mesh efficiently using only its SQuad representation.

The ability to preserve the vertex order is a significant strength of our representation. For instance, it is important for stream processing, for enforcing data locality, for matrix computations, for further compression, and for applications that for other reasons impose a vertex order.

2. Background and overview

Most representation schemes for a triangle or polygonal mesh store an array G of points representing the location of its n_V vertices, which are identified by integers between 0 and $n_V - 1$. Hence $G[i]$ is the location of the vertex with integer ID i. The array G captures the "geometry" of the mesh. To specify how the surface represented by a triangle mesh interpolates these vertices, it suffices to store another array of $3n_T$ integer entries that associates 3 vertex references (integer IDs) with each one of the n_T triangles of the mesh. The vertex IDs of each triangle are listed in an order that corresponds to a consistent orientation of the mesh. This representation is sufficient for some applications, such as rendering or normal computation, but is prohibitively expensive for applications that require accessing neighboring elements in an orderly fashion, which requires a linear search.

2.1. Connectivity graphs

One may build a complete connectivity graph with nodes representing the vertices, edges, and faces of the mesh and links representing incidence, adjacency, and boundary relations. For example, each vertex may have access to its incident edges and triangles. An edge may have access to its two bounding vertices, its incident triangles, and its adjacent edges (which share a vertex with it). A triangle may have access to its bounding vertices and edges and also to its adjacent (neighbor) triangles. These references are often ordered to capture the arrangements of faces around an edge, of vertices and edges around a face, and of faces and edges around a vertex.

Such complete connectivity graphs require an exorbitant amount of storage. Hence, more compact representations have been proposed that only store a set of core links from which other links may be derived.

2.2. Half-edges

Half-edge data structures [Man88], also called "edge-use" or "oriented edge" structures, are based on tuples that associate each vertex in a mesh with one of its incident edges, and with one of the triangles incident upon that edge [Bri89]. A half-edge has a "starting" and an "ending" vertex.

2.3. Corner operators

A "corner" is a "vertex-use," i.e., the association of a triangle with one of its bounding vertices. Corners are equivalent to half-edges: each corner corresponds to the half-edge it faces within the same triangle. We use the corner terminology throughout the paper.

The following corner operators have been proposed by Rossignac et al. [RSS01], and are shown in Fig. 2:

- $v(c)$ returns the vertex of corner c.
- $t(c)$ returns the triangle of c.
- $s(c)$ returns the "swing" corner around $v(c)$.
- $n(c)$ returns the next corner in $t(c)$.
- $o(c)$ returns the opposite corner.
- $c(v)$ returns one corner of v so that $v(c(v)) = v$.
- $c(t)$ returns one corner of t so that $t(c(t)) = t$.

When the references v and t are represented as integers, we overload $c(v)$ and $c(t)$ and choose the appropriate function based on context. Because $s(c) = n(o(n(c)))$ and $o(c) = p(s(p(c)))$, only one of $s(c)$ or $o(c)$ needs to be stored. The corner operators provide access to corners in a clockwise order.

2.4. Corner Table

The *Corner Table* [RSS01] uses integers between 0 and $3n_T - 1$ to identify the corners of a mesh. It stores

© 2011 The Author(s)
Journal compilation © 2011 The Eurographics Association and Blackwell Publishing Ltd.

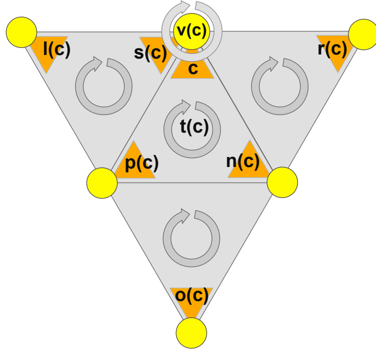


Figure 2: *From corner c, we access its triangle $t(c)$, vertex $v(c)$, next corner $n(c)$ around $t(c)$, swing corner $s(c)$ around $v(c)$, and opposite corner $o(c)$. Additional operators derived from these are left, $l(c) = o(n(c))$, right, $r(c) = o(p(c))$, and previous $p(c) = n(n(c))$.*

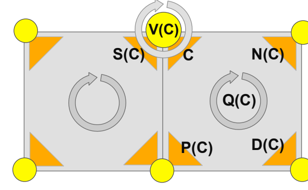


Figure 3: *From a corner C, we can access its vertex $V(C)$ and quad $Q(C)$, the next corner $N(C)$ in $Q(C)$, and the swing corner $S(C)$. For convenience, we also define $D(C)$ as $N(N(C))$ and $P(C)$ as $N(D(C))$.*

$v(c)$ as $V[c]$ in the table (i.e. array) V and $o(c)$ in the parallel table O. The corners of a triangle are stored as consecutive entries in these tables and are sorted according to the orientation of the triangle. Fig. 2 depicts the full set of corner operators, which can be implemented as follows:

$$\begin{array}{lll} v(c) = V[c] & t(c) = c/3 & o(c) = O[c] \\ n(c) = 3t(c) + ((c+1) \bmod 3) & & p(c) = n(n(c)) \\ l(c) = o(n(c)) & r(c) = o(p(c)) & s(c) = n(l(c)) \\ c(v) = C[v] & c(t) = 3t & \end{array}$$

The notation x/y denotes the quotient resulting from integer division.

The storage cost for the basic corner table is $6n_T$ integers for the V and O tables, which we report as 6 rpt (references per triangle). If direct access to the incident edges and triangles of a vertex v is needed, we may add a C table that for each vertex v stores a reference to one of its incident corners $c(v)$. Since in a mesh with small genus (relative to the number of triangles) there are about twice as many triangles as vertices, the total storage of the extended corner table would be 6.5 rpt. Note that these references require at most $\log_2(3n_T)$ bits each, which is usually less than the 32 bits that are allocated to each table entry in order to support fast random access. One may exploit a few of these bits as markers for corners or vertices.

2.5. Sorted Opposite Table (SOT)

The Corner Table has been extended by Lage et al. [LLLV05] to represent tetrahedral meshes. More recently, Gurung and Rossignac [GR09] have proposed the *Sorted Opposite Table* (SOT), which is a variation of this extension that eliminates the need to store $v(c)$. This is accomplished by matching each vertex with a different incident tetrahedron; by sorting the tetrahedra so that tetrahedron i in the first n_V tetrahedra is incident upon vertex i; and by ensuring that the corner of a vertex is listed as the first corner in tetrahedron i. In SOT, $v(c)$ is computed by visiting the corners around vertex v until the corner c matched with v is found, i.e., c is the first corner within its tetrahedron and satisfies $c < 4n_V$. From this corner c, the vertex ID is computed as $v = c/4$.

A more detailed description of the SOT representation for triangle meshes is given in [GR10], where it is shown that SOT requires only 3 rpt. The SQuad representation proposed here builds upon SOT and further reduces storage for triangle meshes to about 2 rpt, while providing the same functionality.

2.6. Quad meshes

One may easily extend the "triangle" Corner Table to a "quad" Corner Table for representing irregular quad meshes. To distinguish quad corners from triangle corners, we capitalize the names of quad corners and their operators. For efficiency, instead of storing the opposite corner in table O, we use an S table that stores the swing operator $S(C)$, as in [YL07]. The following primary quad corner operators (Fig. 3) may be used to traverse the quad mesh:

- $V(C)$ returns the vertex of corner C.
- $Q(C)$ returns the quad of corner C.
- $N(C)$ returns the next corner in $Q(C)$.
- $S(C)$ returns the "swing" corner around $V(C)$.
- $C(V)$ returns one corner so that $V(C(V)) = V$.
- $C(Q)$ returns one corner so that $Q(C(Q)) = Q$.

Their implementation is a trivial extension of their counterpart for triangle meshes, and hence is omitted here. Note that in a mesh of n_Q quads, the V and S tables each have $4n_Q$ entries. Hence, this representation uses 8 rpq (references per quad).

© 2011 The Author(s)
Journal compilation © 2011 The Eurographics Association and Blackwell Publishing Ltd.

2.7. Representing triangles with quads

The Euler formula $n_T = 2n_V - 4 + 4h$ for a manifold mesh with h handles and no boundary implies that n_T is even. If we arrange triangles into pairs so that each pair shares a common edge and hence forms a quad, we may use the above quad data structure to represent the connectivity of the triangle mesh. This approach translates into 4 rpt, since there are two triangles per quad. We need to add one reference per vertex (i.e. 0.5 rpt) for storing $C(V)$.

Hence, if we paired all the triangles of a mesh, we could encode its connectivity using 4.5 rpt [CADM06]. The pairing is always possible, since the dual of the mesh is a bridgeless trivalent graph [Pet91], and can be computed in $O(n_T \log^4 n_T)$ time [BBDL01]. Tarini et al. [TPC*10] present a method for converting a triangle mesh into a pure quad mesh. Unfortunately, this approach cannot be used directly for our purpose, unless we find a way to ensure that the pairing makes it possible to match each vertex with a different triangle.

3. SQuad overview

The *SQuad* representation proposed here combines the two ideas discussed above: (1) the use of a quad mesh to represent the connectivity of a triangle mesh, and (2) the sorting used in SOT.

SQuad requires only a table of swings (the S table) for the quad corners of the mesh to implement a complete set of adjacency queries for both the quad mesh and the triangle mesh. SQuad uses only 4 rpq, and if most triangles are paired, the storage approaches 2 rpt. More precisely, the storage required is $2+2f$ rpt, where f is the fraction of single triangles.

Our construction of SQuad involves the following sequence of steps.

1. In a depth-first traversal of the triangle adjacency graph [Ros99], we match each vertex with the triangle that visits it first and attempt to pair that triangle with one of its neighbors.
2. Then, we store each pair of triangles as a quad and, for regularity of representation, we also disguise the few unpaired triangles as quads by storing a sentinel value for the fourth unused corner.
3. Finally, we reorder these quads so that, after reordering, the ith quad is the one matched with the ith vertex.

We have tested SQuad on a benchmark of meshes of different complexities, ranging from a few thousand to about 55 million triangles, and report statistics on storage size and on construction and access time. In particular, we found that SQuad storage averages 2.072 rpt on these meshes.

4. Prior art

4.1. Compressed formats

Various triangle mesh compression schemes have been proposed (for example, see [TG98, Ros99]). Some of these schemes support progressive refinements [TGHL98] or streaming [ILS05]. Edgebreaker [Ros99, RSS01] encodes the connectivity of the Corner Table using one symbol per triangle drawn from a 5-symbol alphabet. Half of the triangles are assigned the same symbol, allowing a guaranteed compressed cost of 1.8 bpt (bits per triangle). In practice, pairing the symbols and using a simple Huffman table results in a compressed cost of about 1 bpt. The benefit of pairing triangles into quads to reduce storage has previously been addressed by King et al. for mesh compression [KRS99]. Unfortunately, the compressed format is not suitable for traversing and processing the mesh without the expensive decompression.

Several formats have been proposed that allow local decompression. These data structures do not support random access in the usual sense, since they require decompressing a contiguous portion or hierarchical structure of the mesh containing the vertex, triangle, or corner requested, which is then usually cached. Moreover, the code for constructing, decoding, and caching such data structures can be quite involved [YL07, CH09].

4.2. Succinct theoretical representations

Castelli Aleardi et al. [CADS06] propose a theoretical data structure for an optimal representation of the connectivity of planar triangle graphs of size n elements, which asymptotically matches the entropy of 1.62 bpt [Tut62]. They prove that their representation, which decomposes the mesh into tiny pieces of size $O(\log_2 n)$, could support constant time queries. The catalog of all connectivity graphs of the pieces may be constructed in $o(n)$ time and space, so that each tiny piece may be represented by an index into the catalog. To reduce the index size, they form small pieces that each group $O(\log_2 n)$ tiny pieces, and store the connectivity graph between small pieces. Extensions to higher genus surfaces and support for dynamic updates are discussed in [CADS08] and in [CAFL09]. Although this theoretical formulation was not implemented, the ideas upon which it is based have been explored independently [CADM06, Meb08].

4.3. Compact practical representations

Several data structures based on half-edges have been proposed for polyhedra: Baumgart's Winged-Edge and extensions based on it [Bau72, Wei85];

© 2011 The Author(s)
Journal compilation © 2011 The Eurographics Association and Blackwell Publishing Ltd.

Guibas and Stolfi's Quad-Edge [GS85], Mantyla's Half-Edge [Man88], and extensions to non-manifolds including Weiler's Radial-Edge [Wei88]. Although these general techniques can be used for triangle meshes, they require significantly more storage [Ros94] than representations designed to exploit the regularity of a triangle mesh. For example, the quad-edge stores 3 references per half-edge (one to a vertex and two to other half-edges), which amounts to 9 rpt.

Representations tailored for triangle meshes include the Corner Table and the SOT (see Section 2) and also Campagna's et al. Directed Edges [CKS98]. Kallmann and Thalmann's Star-Vertices [KT01] stores for each vertex a list of tuples, one per exiting half-edge, sorted around the vertex. The first entry in a tuple references the end-vertex of that half-edge. The second entry is the local index to a half-edge leaving that vertex. This representation requires 3.5 rpt. Blandford et al. [BBCK05] enhance Star-Vertices by using a variable-length encoding of relative vertex indices. By ordering vertices (building a k-d tree on the vertex positions, and assigning indices recursively) they reduce storage to about 5 bytes per triangle.

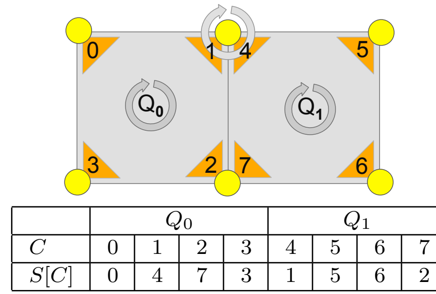
Snoeyink and Speckmann's Tripod [SS99] represents genus-0 meshes using 3 rpt. They orient all edges and partition them into three disjoint spanning trees. Each vertex (except the vertices of a seed triangle) has exactly one outgoing edge in each tree. They then associate with each vertex v six references to vertices w so that (v, w, u) is a triangle of the mesh and (v, u) an outgoing edge from v. Mesh traversal operators have constant cost, but their actions depend on triangle type, which is determined at runtime from the orientations of the edges of the triangle.

5. Representation and operators

Here, we provide the details of the SQuad data structure and of an efficient implementation of its operators. We begin by describing a complete quad mesh representation, and later elaborate on the additional operators needed to support triangle meshes.

5.1. Quad meshes

SQuad stores only the S table of swings of all the quad corners in the mesh. That is, the swing $S(C)$ of a quad corner C is defined as $S[C]$ by indexing the S table (see Fig. 4 for a simple example). The S table is divided into sets of four consecutive entries representing the four corners of a quad. The ID of the ith corner of quad Q is given by $C(Q, i) = 4Q + i$. This consecutive numbering of corners makes the implementation of $Q(C)$, $N(C)$, $C(V)$, and $C(Q)$ particularly straightforward, as these operators can be computed using multiplication, division, and modulo by 4.



	Q_0				Q_1			
C	0	1	2	3	4	5	6	7
$S[C]$	0	4	7	3	1	5	6	2

Figure 4: *S table for a mesh of two quads.*

What remains is the definition of $V(C)$. As discussed above, each vertex V is associated with exactly one quad Q and one of its corners C. In particular, we associate V with the zeroeth quad corner $C(Q, 0)$. Thus, we may determine if a corner C is matched with a vertex by examining its two least significant bits. If $C \bmod 4 = 0$, then C is matched with $V = C/4$. Otherwise, we swing around V using $S(C)$ until we find the corner matched with V.

The set of operators defined on a quad mesh can thus be implemented as follows:

$$C(V) = 4V$$

$$C(Q) = 4Q$$

$$V(C) = \begin{cases} C/4 & \text{if } C \bmod 4 = 0 \text{ and } C < 4n_V \\ V(S(C)) & \text{otherwise} \end{cases}$$

$$Q(C) = C/4$$

$$N(C) = 4Q(C) + ((C+1) \bmod 4)$$

$$S(C) = S[C]$$

The predicate $C < 4n_V$ is needed when the number of quads exceeds the number of vertices, as in this case some quads cannot be matched with a vertex. If instead the number of vertices exceeds the number of quads, some vertices cannot be matched with a quad. This case is not handled by our quad mesh representation. Fortunately, such unmatched vertices generally do not occur in triangle meshes, as $n_T \simeq 2n_V$.

5.2. Corner mapping

To support triangle meshes, we conceptually split each quad along one of its diagonals into two triangles. This splits two corners of each quad in half, while the other two quad corners each map to a single triangle corner (Fig. 5). Aside from this change, we use the same basic data structure, and simply map between triangle and quad corners when necessary. That is, the S table still stores swing pointers between *quad* corners.

© 2011 The Author(s)
Journal compilation © 2011 The Eurographics Association and Blackwell Publishing Ltd.

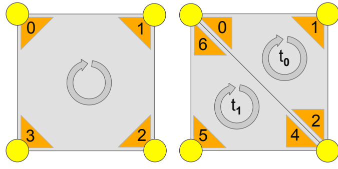


Figure 5: *Numbering of corners within a quad (left) and a triangle pair (right).*

The mapping $c \mapsto C$ from triangle to quad corners is not a bijection, since pairs of triangle corners may map to the same quad corner. Hence some care is needed in how this mapping is done. Fig. 5 depicts the mapping that we have chosen between the four quad corners and six triangle corners of a quad. The IDs of the triangle corners of quad Q are given by $c(Q,i) = 8Q + i$, where the corner offsets i are 0, 1, and 2 for the first triangle and 4, 5, and 6 for the second. We do not use offsets 3 or 7; this does not incur storage overhead because we do not store any per-triangle-corner information. While the choice of mapping is not unique, reserving eight offsets per quad enables an efficient implementation based on divisions and modulus operations by 4 and 8, as shown below.

In summary, the mappings (see also Fig. 5) from quad-corners to triangle-corners are: $0 \mapsto 0$, $1 \mapsto 1$, $2 \mapsto 4$, $3 \mapsto 5$; and from triangle-corners to quad-corners are: $0 \mapsto 0$, $1 \mapsto 1$, $2 \mapsto 2$, $4 \mapsto 2$, $5 \mapsto 3$, $6 \mapsto 0$. For quad corners associated with two triangle corners, the mapping $C \mapsto c$ is such that the second triangle corner is reached from the first by $s(c)$, which as will become apparent ensures that we can traverse all triangle corners around a vertex.

The mappings $C(c)$ and $c(C)$ may be implemented efficiently without lookup tables. We use the auxiliary functions $C(Q,i) = 4Q + i$, $c(Q,i) = 8Q + i$, $Q(C) = C/4$, and $Q(c) = c/8$ (note the overloaded pair $Q(C)$ and $Q(c)$), which allow us to convert from a triangle corner c to a quad corner C:

$$C(c) = C(Q(c), (c + ((c/2)\ \&\ 2)) \bmod 4)$$

where '&' indicates the bitwise AND operation.

Before defining the mapping from quad to triangle corners, we note that we may not always be able to pair triangles into quads. Single, unpaired triangles are still treated as quads, and we store for the last corner of a quad in the S table a special sentinel value, referred to as *null* in subsequent sections, to indicate that the second triangle of the quad does not exist. The predicate below determines whether the quad represents one or two triangles:

$$\mathit{isQuad}(Q) = (S[4Q+3] \neq \mathit{null})$$

We then compute the triangle corner c associated with a quad corner C as follows:

$$c(C) = \begin{cases} c(Q(C), (C \bmod 4) + (C\ \&\ 2)) & \text{if } \mathit{isQuad}(Q(C)) \\ c(Q(C), (C \bmod 4)) & \text{otherwise} \end{cases}$$

Although $C(c(C)) = C$, in general $c(C(c)) \neq c$.

5.3. Triangle meshes

We are now ready to define the SQuad operators that enable efficient extraction of the triangle mesh connectivity information. We present their implementation first, and follow with details:

$$c(v) = 8v \qquad c(t) = 4t$$
$$v(c) = V(C(c)) \qquad t(c) = c/4$$
$$n(c) = \begin{cases} c-2 & \text{if } c \bmod 4 = 2 \\ c+1 & \text{otherwise} \end{cases}$$
$$s(c) = \begin{cases} c(Q(c), 6) & \text{if } c \bmod 8 = 0 \text{ and } \mathit{isQuad}(Q(c)) \\ c(Q(c), 2) & \text{if } c \bmod 8 = 4 \\ c(S(C(c))) & \text{otherwise} \end{cases}$$

Again, note the use of $c(v)$, $c(t)$, and $c(C)$ depending on context. The standard corner-based operators follow directly:

$$p(c) = n(n(c)) \qquad o(c) = p(s(p(c)))$$
$$l(c) = p(s(c)) \qquad r(c) = p(s(n(c)))$$

The majority of our operators have straightforward implementations. We note that $v(c)$ is computed efficiently by iteration over quad corners. This allows us to "skip over" triangle corners known not to be matched with v. Due to the consecutive ordering of corners, $n(c)$ like $N(C)$ is efficiently implemented using modular arithmetic. Finally, $s(c)$ computes the adjacent swing corner if it is within the same quad (the first two cases); otherwise it consults the S table. Our operators run in constant time, except $v(c)$, which runs in expected constant time but in time linear in the maximum degree in the worst case.

6. SQuad construction

Our linear-time construction of the SQuad data structure starts with the V table of a mesh. We compute the O table (see [GR10]) and then match vertices with triangles, pair most triangles into quads, reorder the quads and single triangles, and finally produce the S table. In this section we describe the triangle-vertex matching and triangle-triangle pairing operations, then describe a single-pass algorithm that combines the two operations. To avoid ambiguity, we use "matching" to refer to triangle-vertex associations and "pairing" to refer to triangle-triangle associations.

© 2011 The Author(s)
Journal compilation © 2011 The Eurographics Association and Blackwell Publishing Ltd.

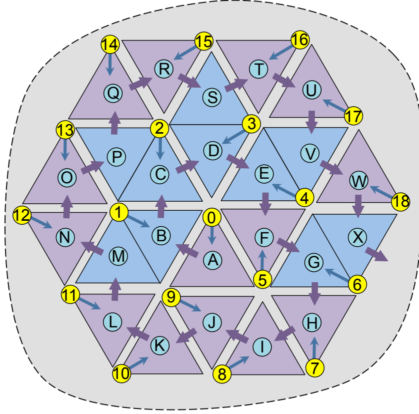


Figure 6: *We invade the mesh starting from triangle A. Purple arrows show the traversal order; blue arrows are triangle-vertex matches. Matches* $(0, A)$*,* $(5, F)$*, and* $(9, J)$ *are made at the beginning. Note that triangles* M*,* P*,* S*,* V*, and* X *are unmatched and thus paired (blue quads) with* B*,* C*,* D*,* E*, and* G*.*

```
MATCH-AND-PAIR(in : c, out : M, P)
 1  M[v(c)] ← t(c)                               match first 3 vertices
 2  M[v(n(c))] ← t(s(n(c)))
 3  M[v(p(c))] ← t(s(p(c)))
 4  P[t(s(n(c)))] ← t(s(n(c)))                   mark triangles as paired
 5  P[t(s(p(c)))] ← t(s(p(c)))
 6  T[t(c)] ← true                               mark t(c) as visited
 7  c ← l(c)
 8  stack.push(null)                             push sentinel value
 9  while stack ≠ ∅
10      T[t(c)] ← true
11      if M[v(c)] = null
12          M[v(c)] ← t(c)                       case C
13          if P[t(r(c))] = null and M[v(r(c))] ≠ null
14              P[t(c)] ← t(r(c))                pair t(c) and t(r(c))
15              P[t(r(c))] ← t(c)
16          else if P[t(l(c)))] = null
17              P[t(c)] ← t(l(c))                pair t(c) and t(l(c))
18              P[t(l(c))] ← t(c)
19          c ← r(c)
20      else
21          if T[t(l(c))] = true
22              if T[t(r(c))] = true
23                  c ← stack.pop()              case E
24              else
25                  c ← r(c)                     case L
26          else
27              if T[t(r(c))] = true
28                  c ← l(c)                     case R
29              else
30                  stack.push(l(c))             case S
31                  c ← r(c)
```

Table 1: *Matching & pairing in a single pass.*

6.1. Triangle-Vertex matching

Our algorithm computes vertex-triangle matches first, instead of directly matching quads with vertices. This is necessary because some meshes have more vertices than quads. For example, a quad mesh with genus zero has $n_V = n_Q + 2$ vertices (where $n_Q = n_T/2$), which shows that two vertices would remain unmatched.

Our matching procedure is described in [GR10] and illustrated in Fig. 6. We summarize it here for completeness. We start with a seed triangle and match its three vertices according to Fig. 6. We then invade the mesh [RSS01] by walking between edge-adjacent triangles. For the corner c not incident on the previous triangle, if its vertex $v(c)$ is unvisited, then we match it with triangle $t(c)$. This ensures that each vertex is visited and matched with a different triangle. Since $n_T \simeq 2n_V$, only about half of the triangles are matched. We observe that unmatched triangles tend to be uniformly distributed around the matched ones.

6.2. Triangle-Triangle pairing

Next, we try to pair each matched triangle with an unmatched one by traversing the triangles in the same order as before. For each matched triangle t, we check first its right, then left neighbors; if we find a neighbor that is neither matched nor paired, we pair it with t (Fig. 6). This simple procedure leaves very few unpaired triangles (see Fig. 1): on average 3.3% of the triangles remained unpaired in our benchmark meshes.

6.3. Combined matching and pairing

The matching and pairing process may be implemented in a variety of ways. A particularly elegant implementation uses a modified Edgebreaker [Ros99] traversal of the mesh, which performs the matching and pairing in a single pass, as shown in Fig. 6. Edgebreaker performs a depth-first traversal of the mesh triangles, and labels them using one of five symbols {'C', 'L', 'E', 'R', 'S'} depending on how the triangles are attached to the already visited triangles (see [Ros99] for further details). We include our matching algorithm here, to show its simplicity (see Table 1).

Our algorithm takes as input an arbitrary seed corner c, and outputs two tables M and P such that v is matched with triangle $M[v]$ and triangle t is paired with triangle $P[t]$. Thus, the tuple $\langle v, M[v], P[M[v]]\rangle$ forms a vertex-quad match from which the SQuad S table is easily constructed. All entries of M and P are assumed initialized to **null**. Our algorithm makes use of a temporary table T of booleans, which records for each triangle if it has been visited.

From the start corner c, we walk on the mesh while the stack is not empty. Each 'C' triangle $t(c)$ involves visiting a new (unmatched) vertex $v(c)$, which we match with $t(c)$. We then attempt to pair $t(c)$ with an unpaired neighbor. Because 'C' triangles generate vertex matches, and because each quad is matched with

© 2011 The Author(s)
Journal compilation © 2011 The Eurographics Association and Blackwell Publishing Ltd.

only one vertex, no two ‘C’ triangles may be paired; a condition we test for on line 13 by making sure that vertex $v(r(c))$ has already been matched (otherwise $t(r(c))$ is a ‘C’ triangle). A similar test on line 16 is not needed, since we must enter $t(l(c))$ from a triangle other than the current one, and hence $t(l(c))$ cannot be a ‘C’ triangle. This order of trying to pair right before left neighbors reduces slightly the ratio of single triangles, because it pairs more of the previously visited triangles. For ‘E’, ‘L’, ’R’, and ’S’ triangles, no matching or pairing occurs, and for those cases we simply follow the Edgebreaker traversal.

6.4. Quad reordering

We reorder quads (or single triangles) based on vertex matching, such that the ith quad (or i^{th} single triangle) and its first corner are matched with the ith vertex. Quads of unmatched triangles are placed at the end of the list. The original vertex order is thus left unmodified, while the triangle order is made “compatible” with the vertex order. This freedom in ordering may be exploited in applications that, for instance, require high locality of reference, or for further mesh compression for offline storage.

7. Topology extensions

So far, for simplicity and clarity, we have assumed that the mesh is a manifold without boundary. Here we explain how we have modified our representation to support any orientable, non-manifold mesh, as long as it can be represented as a pseudo-manifold [RC99] with boundary using a Corner Table.

We store for each vertex v a “bucket” of all corners incident upon v. We then perform the following steps:

1. We partition v’s bucket into disjoint subsets of corners. Each set has triangles that form edge-connected components.
2. We sort each set so that all consecutive corners (c_i, c_{i+1}) share an edge, i.e., $c_{i+1} = s(c_i)$.
3. For the last corner z of a set, we change $S[z]$, which may have been referencing the first corner or nothing (indicating that z was next to a border edge), to the first corner of the next set (or when z is the last corner in the last set, we set $S[z]$ to reference the first corner in the first set).

As a result, the links stored in S allow us to traverse all of the corners of a vertex v, even when v is non-manifold. However, to implement the true $s(c)$, we now need to know (1) whether $S[c]$ refers to the next set (i.e., when $t(c)$ and $t(s(c))$ are not edge-connected), and (2) whether $s(c)$ exists. We use the two least significant bits of $S[c]$ to record this information and modify the implementation of $s(c)$ accordingly.

8. Results

We here report on how effective our construction scheme is in matching and pairing mesh primitives, and compare the speed of accessing our SQuad data structure with the Corner Table and SOT. We use several familiar triangle meshes, shown in Table 2, as benchmarks for our evaluation. This table also lists the fraction of unmatched and unpaired triangles for these meshes in their SQuad representation. As is evident from the table, these fractions are usually small, allowing 2.15 references per triangle or less to represent all meshes. We note that the mesh regularity in terms of the fraction of valence-6 vertices is well correlated with the number of paired triangles. Because our benchmark meshes are all derived from regularly sampled range scans, we also evaluated the regularity of the Asian dragon and Thai statue after simplification by a factor of ten. This reduced the number of regular vertices in both meshes to about 32%, resulting in a SQuad representation of 2.14 rpt.

Our offline SQuad construction process is quite fast. For example, the 55 million triangle David mesh is constructed in only 20 seconds using 2.2 GB of RAM. This large mesh was used to evaluate the access speed of the CT, SOT, and SQuad data structures, which using floating-point geometry (equivalent to 1.5 rpt) require 1,700 MB, 950 MB, and 760 MB to store this mesh, respectively. We performed several experiments while varying the amount of total RAM (configured at boot time) on a MacbookPro with 2.66 GHz Intel Core i7 and 8 GB of 1067 MHz DDR3 memory. In addition to timing the operators $s(c)$ (for SQuad), $o(c)$ (for CT and SOT), and $v(c)$, we find it instructive to also compare higher-level tasks, such as vertex valence and normal computation, as well as a data-dependent contouring traversal. Our contouring task starts at the midpoint of a randomly chosen edge. It then traces a contour with the same z coordinate as the midpoint by walking from each triangle to the left or right neighbor stabbed by the contour. All computations, except contouring, access corners or vertices sequentially.

The timing results are presented in Fig. 7. With the operating system using about 400 MB, far less memory than the total RAM indicated in these graphs was available for processing. When the whole mesh fits in RAM, the additional memory accesses and computations for SQuad increase the execution time of $s(c)$ by 5X and $v(c)$ by 10X over CT. With SQuad being more memory efficient than CT, this relationship is reversed as memory is reduced and parts of the Corner Table are stored out-of-core. One can show that the expected number of memory accesses to the S table in SQuad is $\frac{5}{3}$ and $\frac{3}{2}$ for $s(c)$ and $v(c)$, respectively, compared to one access per operation for CT. $v(c)$ in SOT

© 2011 The Author(s)
Journal compilation © 2011 The Eurographics Association and Blackwell Publishing Ltd.

Mesh	n_T	Val 6	Un-matched	Matched unpaired	rpt
bunny	69,451	75.1%	1.2%	1.5%	2.054
rocker arm	80,354	65.2%	1.3%	1.3%	2.054
horse	96,966	66.5%	1.1%	1.1%	2.046
dinosaur	112,384	57.9%	1.8%	1.8%	2.072
armadillo	345,944	52.6%	1.7%	1.7%	2.069
hand	654,666	53.4%	2.4%	2.4%	2.096
buddha	1,087K	32.1%	3.8%	3.7%	2.150
blade	1,760K	62.4%	2.0%	1.9%	2.078
welsh dragon	2,210K	86.7%	0.7%	0.7%	2.027
asian dragon	7,219K	89.1%	0.7%	0.7%	2.026
thai statue	10.0M	44.4%	2.8%	2.8%	2.111
david	55.5M	51.6%	1.9%	2.1%	2.082

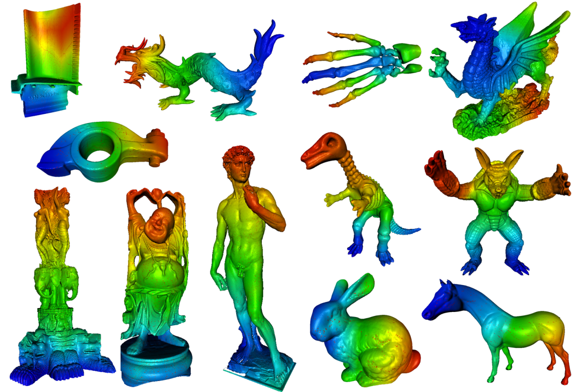
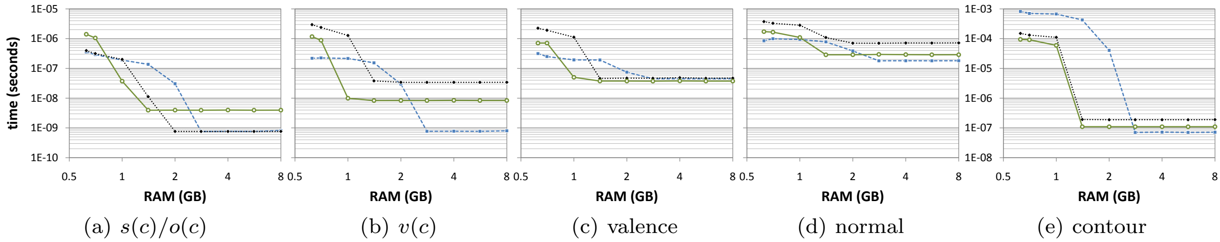


Table 2: *Mesh statistics (number of triangles and regular vertices) and SQuad representation (unmatched triangles, matched unpaired triangles, and references per triangle). All meshes have zero genus except hand (6), rocker arm (1), buddha (104), blade (163), and thai statue (3). The mesh color coding illustrates the ordering of triangles.*



(a) $s(c)/o(c)$ (b) $v(c)$ (c) valence (d) normal (e) contour

Figure 7: *Average per-element execution time of various operators on the David mesh for CT (dashed blue), SOT (dotted black), and SQuad (solid green). For $s(c)/o(c)$, CT and SOT compute $o(c)$ while SQuad computes $s(c)$. "valence" computes the number of triangles incident on a vertex; "normal" computes and averages face normals incident on a vertex; while "contour" traverses a loop of triangles intersected by a horizontal plane.*

is much slower than in SQuad, because SQuad does fewer swings (since it swings quads), because swinging in SOT (to find the matched corner) requires both $o()$ and $n()$, and because SQuad avoids modulo 3 when checking if a corner is matched. (The SOT code has not been optimized to avoid such divisions.) For the same reason, SQuad executes $n(c)$ faster than CT and SOT: 2.7 ns vs. 3.4 ns. The timings of $o(c)$ for CT and SOT and of $v(c)$ for CT are similar, since all three are costs of a simple table look-up. The cost of $s(c)$ for SQuad involves converting a triangle corner to a quad corner, a table look-up, and a reverse conversion. This overhead is amortized in the higher-level tasks, for which SQuad yields comparable or, when memory is scarce, better performance than CT.

The practical implications of these performance improvements on the total execution time depend of course heavily on the application's memory access pattern and on the hardware architecture. With per-core memory bandwidth expected to decrease dramatically on future architectures, we anticipate that the data locality afforded by SQuad combined with its small memory footprint will be of increasing importance.

9. Conclusions

We propose a new representation for the connectivity of triangle meshes that only requires about 2 references per triangle and yet provides constant-time access to neighboring elements in the connectivity graph. Our data structure pairs two adjacent triangles with a shared vertex, and then reorders the triangles to follow the user-specified vertex order. It increases the cost of some low-level queries on small meshes, but reduces disk access for large meshes, which may drastically improve performance for some applications.

Although designed for in-core random access, our compact SQuad representation is also suited for out-of-core stream processing. By merging geometry (G table) with connectivity (S table), we obtain a complete binary streaming mesh representation. Future research will focus on how to convert an indexed mesh to such a representation without having to traverse the whole mesh and reorder all triangles.

© 2011 The Author(s)
Journal compilation © 2011 The Eurographics Association and Blackwell Publishing Ltd.

Acknowledgements

This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, and was partially supported by NSF grant 0811485.

References

[Bau72] BAUMGART B. G.: *Winged edge polyhedron representation*. Tech. Rep. CS-TR-72-320, Stanford University, 1972.

[BBCK05] BLANDFORD D. K., BLELLOCH G. E., CARDOZE D. E., KADOW C.: Compact representations of simplicial meshes in two and three dimensions. *International Journal of Computational Geometry and Applications 15*, 1 (2005), 3–24.

[BBDL01] BIEDL T. C., BOSE P., DEMAINE E. D., LUBIW A.: Efficient algorithms for Petersen's matching theorem. *Journal of Algorithms 38*, 1 (2001), 110–134.

[Bri89] BRISSON E.: Representing geometric structures in d dimensions: Topology and order. In *ACM Symposium on Computational Geometry* (1989), pp. 218–227.

[CADM06] CASTELLI ALEARDI L., DEVILLERS O., MEBARKI A.: 2D triangulation representation using stable catalogs. In *Canadian Conference on Computational Geometry* (2006), pp. 71–74.

[CADS06] CASTELLI ALEARDI L., DEVILLERS O., SCHAEFFER G.: Optimal succinct representations of planar maps. In *ACM Symposium on Computational Geometry* (2006), pp. 309–318.

[CADS08] CASTELLI ALEARDI L., DEVILLERS O., SCHAEFFER G.: Succinct representations of planar maps. *Theoretical Computer Science 408*, 2–3 (2008), 174–187.

[CAFL09] CASTELLI ALEARDI L., FUSY E., LEWINER T.: Schnyder woods for higher genus triangulated surfaces, with applications to encoding. *Discrete & Computational Geometry 42*, 3 (2009), 489–516.

[CH09] COURBET C., HUDELOT C.: Random accessible hierarchical mesh compression for interactive visualization. In *Symposium on Geometry Processing* (2009), pp. 1311–1318.

[CKS98] CAMPAGNA S., KOBBELT L., SEIDEL H.-P.: Directed edges—a scalable representation for triangle meshes. *Journal of Graphics Tools 3*, 4 (1998), 1–11.

[GR09] GURUNG T., ROSSIGNAC J.: SOT: Compact representation for tetrahedral meshes. In *SIAM/ACM Geometric and Physical Modeling* (2009), pp. 79–88.

[GR10] GURUNG T., ROSSIGNAC J.: *SOT: Compact Representation for Triangle and Tetrahedral Meshes*. Tech. Rep. GT-IC-10-01, Georgia Institute of Technology, 2010.

[GS85] GUIBAS L., STOLFI J.: Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics 4*, 2 (1985), 74–123.

[ILS05] ISENBURG M., LINDSTROM P., SNOEYINK J.: Streaming compression of triangle meshes. In *Symposium on Geometry Processing* (2005), pp. 111–118.

[KRS99] KING D., ROSSIGNAC J., SZYMCZAK A.: *Connectivity Compression for Irregular Quadrilateral Meshes*. Tech. Rep. GIT-GVU-99-36, Georgia Institute of Technology, 1999.

[KT01] KALLMANN M., THALMANN D.: Star-vertices: a compact representation for planar meshes with adjacency information. *Journal of Graphics Tools 6*, 1 (2001), 7–18.

[LLLV05] LAGE M., LEWINER T., LOPES H., VELHO L.: CHF: A scalable topological data structure for tetrahedral meshes. In *Brazilian Symposium on Computer Graphics and Image Processing* (2005), pp. 349–356.

[Man88] MANTYLA M.: *Introduction to Solid Modeling*. W. H. Freeman & Company, 1988.

[Meb08] MEBARKI A.: *Implantation de structures de données compactes pour les triangulations*. PhD thesis, Université de Nice-Sophia Antipolis, 2008.

[Pet91] PETERSEN J.: Die theorie der regulären graphs. *Acta Mathematica 15*, 1 (1891), 193–200.

[RC99] ROSSIGNAC J., CARDOZE D.: Matchmaker: Manifold BReps for non-manifold r-sets. In *ACM Symposium on Solid Modeling and Applications* (1999), pp. 31–41.

[Ros94] ROSSIGNAC J.: Through the cracks of the solid modeling milestone. In *From object modelling to advanced visualization*. Springer Verlag, 1994, pp. 1–75.

[Ros99] ROSSIGNAC J.: Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics 5*, 1 (1999), 47–61.

[RSS01] ROSSIGNAC J., SAFONOVA A., SZYMCZAK A.: 3D compression made simple: Edgebreaker on a corner-table. In *International Conference on Shape Modeling & Applications* (2001), pp. 278–283.

[SS99] SNOEYINK J., SPECKMANN B.: Tripod: A minimalist data structure for embedded triangulations. In *Computational Graph Theory and Combinatorics* (1999).

[TG98] TOUMA C., GOTSMAN C.: Triangle mesh compression. In *Graphics Interface* (1998), pp. 26–34.

[TGHL98] TAUBIN G., GUÉZIEC A., HORN W., LAZARUS F.: Progressive forest split compression. In *ACM SIGGRAPH* (1998), pp. 123–132.

[TPC*10] TARINI M., PIETRONI N., CIGNONI P., PANOZZO D., PUPPO E.: Practical quad mesh simplification. *Computer Graphics Forum 29*, 2 (2010), 407–418.

[Tut62] TUTTE W.: A census of planar triangulations. *Canadian Journal of Mathematics 14*, 1 (1962), 21–38.

[Wei85] WEILER K.: Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics & Applications 5*, 1 (1985), 21–40.

[Wei88] WEILER K.: The radial-edge data structure: A topological representation for non-manifold geometric boundary modeling. In *Geometric Modeling for CAD Applications* (1988), pp. 3–36.

[YL07] YOON S.-E., LINDSTROM P.: Random-accessible compressed triangle meshes. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1536–1543.

[YLPM05] YOON S.-E., LINDSTROM P., PASCUCCI V., MANOCHA D.: Cache-oblivious mesh layouts. *ACM Transactions on Graphics 24*, 3 (2005), 886–893.

© 2011 The Author(s)
Journal compilation © 2011 The Eurographics Association and Blackwell Publishing Ltd.