Compression of Tetrahedral Meshes

Geometry Processing
CS 7960

Louis Bavoil
01/19/2006
Outline

• Corner Table
• Edgebreaker Efficiency
• Edgebreaker with Boundary
Corner Table

Two arrays of integers: V and O
Size of each array = 3|T|
Not used for storage, just to walk on a mesh

\[ c.v = V[c] \]
\[ c.o = O[c] \]
\[ c.n.v = V[3 \cdot c.t + ((c+1) \bmod 3)] \]
\[ c.p.v = V[3 \cdot c.t + ((c+2) \bmod 3)] \]
\[ c.l = c.p.o \]
\[ c.r = c.n.o \]
Building a Corner Table

• The O table may be derived from the V table using a hashing sort of
  \[ \text{min}(c.n.v, c.p.v), \text{max}(c.n.v, c.p.v, c), c \]  

• “Entries that correspond to opposite corners are consecutive in the list.”

Source: Helio Lopes, Jarek Rossignac, Alla Safonova, Andrzej Szymczak and Geovan Tavares.
“Edgebreaker: A Simple Compression Algorithm for Surfaces with Handles.”
Building a Corner Table

\[ \text{min}(c.n.v, c.p.v), \text{max}(c.n.v, c.p.v, c), c \]  

First two triangles:  
\[ 1, 2, 0 \quad [0, 2, 1] \quad [0, 1, 2] \quad [1, 2, 3] \quad [1, 3, 4] \quad [2, 3, 5] \]

Sorted list:  
\[ 0, 1, 2 \quad [0, 2, 1] \quad [1, 2, 0] \quad [1, 2, 3] \quad [1, 3, 4] \quad [2, 3, 5] \]

“Entries that correspond to opposite corners are consecutive in the list.”  
  – Only true for corners that have an opposite corner
Outline

• Corner Table
• **Edgebreaker Efficiency**
• Edgebreaker with Boundary
Edgebreaker

Red = already visited/marked
Black = not visited yet
Edgebreaker Efficiency

C = 0
S = 100
R = 101
L = 110
E = 111

c = number of bits of coded CLERS string
\[ c = |C| + 3(|S|+|L|+|R|+|E|) \]
Edgebreaker Efficiency

One triangle per CLERS op-code:
\[ |S| + |L| + |R| + |E| + |C| = |T| \]

One C op-code per interior vertex:
\[ |C| = |V_i| \]
Edgebreaker Efficiency

Euler formula for simple meshes:

\[ |T| - |E| + |V| = 2 - 2g - b = 1 \]

Num external edges = \(|V_E|\)
Num internal edges = \((3|T| - |V_E|)/2\)

\[ |T| - 2|V_I| = |V_E| - 2 \]
Edgebreaker Efficiency

Putting everything together:

\[ c = \text{number of bits of coded CLERS string} \]
\[ c = |C| + 3(|S|+|L|+|R|+|E|) \]
\[ |S|+|L|+|R|+|E|+|C| = |T| \]
\[ |C| = |V_I| \]
\[ |T|-2|V_I| = |V_E| - 2 \]
\[ \Rightarrow c = 2|T| + |V_E| - 2 \]

For simple meshes with short boundary (\(|V_E| \ll |T|\)):

\[ c \approx 2 \ |T| \ \Leftrightarrow \ 2 \ b/T \]
Outline

• Corner Table
• Edgebreaker Efficiency
• Edgebreaker with Boundary
Edgebreaker with Boundary

• Single boundary:
  – Store the boundary vertices first (boundary loop)
  – Start with an edge on the boundary

• Boundary loop:

  Assuming a simple mesh (~half sphere)

  Implementation:
  Double linked list of edges
  No half-edges needed
Edgebreaker with Boundary

Yellow = visited
Edgebreaker with Boundary

Yellow = visited
Edgebreaker with Boundary

• Question: Index of v2?
Edgebreaker with Boundary

- Question: Index of $v_2$?
- Edgebreaker: Compute offset from $v_1$
  - Original version does 2 passes
Outline

• Tetrahedral Meshes
• Related Work
• Streaming Meshes
• Streaming Compression of Tet Meshes
Tetrahedral Meshes

- Tetrahedron
  = Polygon with 4 vertices

- Tetrahedral Mesh
  - Unstructured (no grid)
  - Used for Finite Element Simulations
  - Interpolation of attributes at the vertices
    - Scalar fields (1 float / vertex)
    - Vector fields (3 floats / vertex)

http://tetgen.berlios.de/examples.dragon.html
Indexed meshes

• Most standard format for tet meshes:
  – Indexed mesh:
    
    // Vertices
    XYZS // position + scalar value
    XYZS
    ....
    
    // Indices into vertex list
    0 1 2 3 // first tet
    1 2 3 4
    ...
    ...
Indexed meshes

- Indexed mesh:
  - Vertices + Attributes
  - Indices

- Issues:
  - Non-streaming / Offline
    - Requires all vertices to be in main memory
  - Compression
    - Quantization of indices
    - Indices need $\log_2|V|$ bits
Outline

- Tetrahedral Meshes
- Related Work
- Streaming Meshes
- Streaming Compression of Tet Meshes
In-core Compression

• Grow & Fold

  – Similar to Topological Surgery
    • Builds a tetrahedral spanning tree
    • Boundary of spanning tree = triangle mesh
  – 7-8 bits per tet
In-core Compression

• Cut-border

  – Region growing process like Edgebreaker
    • Gate = triangle
    • Each step adds a tet
  – Use local-indexing for “connect” (like S op-code)
  – 2 bits per tet (state-of-the-art)
In-core Compression


• Can achieve 1.6 bpt
• Only works for meshes with Delaunay property
  – Empty-circumsphere criterion
• Reorders the tets in visibility order by sweeping a triangulated front through the mesh
• Adds tetrahedron into the front
In-core algorithms

Most mesh compression algorithms
1. Load the whole mesh into RAM (in-core)
2. Build mesh-traversal data structure

Issue:
(Mesh + Traversal Data Structure)
= Too large for main memory
→ Memory trashing → crash or low performance
Out-of-core algorithms

Compress the mesh piece by piece

J. Ho, K. Lee, aD. Kriegman. Compressing large polygonal models. VIS 2001

Use external memory data structure

Out-of-core algorithms

Compress the mesh piece by piece

J. Ho, K. Lee, aD. Kriegman. Compressing large polygonal models. VIS 2001

1. Cut large triangles meshes into pieces
2. Encore each piece & record how to stitch pieces
Out-of-core algorithms

Use external memory data structure


Mapping supports the topological adjacency queries of a given compressor
Outline

• Tetrahedral Meshes
• Related Work
• **Streaming Meshes**
• Streaming Compression of Tet Meshes
Streaming Meshes

- Different approach: Streaming


Mesh in any order → Stream of triangles → Compressor

- Native order
- Sorted / axis
- Sorted / z (space-filling curve)
- Breadth first traversal
Delay Buffer


Delay Buffer = buffer of N previous tets (green)
Streaming Meshes

M. Isenburg and P. Lindstrom. *Streaming meshes*. In Visualization’05 Proceedings

- Defines a streaming mesh format
  - Vertices and tets are interleaved
  - **Finalization**: Explicit information about when vertices are referenced for the last time
  - **Dynamic indexing**: Sliding window of vertex indices
Example of triangle mesh

<table>
<thead>
<tr>
<th>Standard Index Mesh</th>
<th>Streaming Mesh</th>
</tr>
</thead>
<tbody>
<tr>
<td>v 0.3 1.1 0.2</td>
<td>v 0.3 1.1 0.2</td>
</tr>
<tr>
<td>v 0.4 0.4 0.5</td>
<td>v 0.4 0.4 0.5</td>
</tr>
<tr>
<td>v 1.4 0.8 1.2</td>
<td>v 1.4 0.8 1.2</td>
</tr>
<tr>
<td>v 0.9 0.5 0.7</td>
<td>v 0.9 0.5 0.7</td>
</tr>
<tr>
<td>v 1.0 0.1 1.1</td>
<td>f 2 4 1</td>
</tr>
<tr>
<td>f 2 4 1</td>
<td>v 1.0 0.1 1.1</td>
</tr>
<tr>
<td>f 2 5 4</td>
<td>f -4 5 4</td>
</tr>
<tr>
<td>f 3 1 4</td>
<td>f 3 -5 4</td>
</tr>
<tr>
<td>f 4 5 3</td>
<td>f -2 -1 -3</td>
</tr>
</tbody>
</table>

1. Interleaving of vertices and faces
2. Negative index => Finalize Vertex
   Negative offset relative to last vertex
Definitions

• Front
  – Evolving set of active vertices
  – Partitions the mesh into finalized vertices and not yet encountered vertices

• Width
  – Maximal size of the front (num vertices)
  – Lower bound on the memory footprint
Definitions

• Front
  – Evolving set of active vertices
  – Partitions the mesh into finalized vertices and not yet encountered vertices

• Width
  – Maximal size of the front (num vertices)
  – Lower bound on the memory footprint

• Span
  – maximal index difference of vertex indices on the front
  – measures the longest duration a vertex remains active
  – at most log2(span) bits are needed for relative indexing of the vertices
Example

• Active Front:
  – \{2,4,1\}
  – ...

Dynamic indexing = index into active front
Front values change as streaming goes

\begin{verbatim}
v 0.3  1.1  0.2
v 0.4  0.4  0.5
v 1.4  0.8  1.2
v 0.9  0.5  0.7
f  2  4  1
v  1.0  0.1  1.1
f  -4  5  4
f  3  -5  4
f  -2  -1  -3
\end{verbatim}
Mesh Reordering

Original Mesh Ordering
Beginning 2% of tet array
Num Vertices = 168,930
Width = 168,930

Breadth First Ordering
Num Vertices = 168,930
Width = 5,528 (3%)
Outline

• Tetrahedral Meshes
• Related Work
• Streaming Meshes
• **Streaming Compression of Tet Meshes**
Streaming Compression

Streaming Compression of Tetrahedral Volume Meshes, Isenburg, Lindstrom, Gumhold, Shewchuk, 2005

- **Main contributions:**
  - Scales to nearly arbitrary large input
  - Local instead of global reordering for connectivity compression
  - Detailed analysis of different geometry prediction rules

- **Parameters:**
  - Input mesh ordering
  - Size of delay buffer (num of tets)
Data Structures

• Active vertices
  – Until finalized

• Active half-edges
  – Until end vertex finalized
  – Or until incident face is processed

• Different definition of half-edge
  – Origin vertex
  – Next half-edge in the face
  ➔ no pointer to adjacent face
Connectivity Compression

• For each tet to compress:
  – How many faces and vertices are active?
    • Active faces = adjacent faces

  – Look at configuration of tet relative to the front
    • Encode {START, ADD, JOIN, FLIP, FILL}
Connectivity Compression

- **START**
  - 2,3,4 new vertices
  - no adjacent face

- **ADD**
  - 1 new vertex
  - 1 adjacent face

- **JOIN**
  - 0 new vertex
  - 1 adjacent faces

- **FLIP**
  - 2 adjacent faces

- **FILL**
  - 3 or 4 adjacent faces
Connectivity Compression

• Encode \{START, ADD, JOIN, FLIP, FILL\}
  – Use arithmetic encoder (more efficient than Huffman)
Connectivity Compression

• Encode \{START, ADD, JOIN, FLIP, FILL\}
  – Use arithmetic encoder (more efficient than Huffman)

• Two types of vertices:
  – New vertices
  – Active vertices (in the front)
Connectivity Compression

• Two types of vertices:
  – New vertices
    • Like Edgebreaker
    • Compress geometry
    • Write to separate stream
Connectivity Compression

• Two types of vertices:
  – New vertices
    • Like Edgebreaker
  – Active vertices (in the front)
    • Vertex shares face/edge/vertex with previous tet
      – Use an index into list of half-edges from one vertex
      – Use an index into list of intersection of half-edges from two vertices
    • Else direct dynamic indexing into list of active vertices
      – More expensive because index is larger than index into list of half-edges
Delay Buffer

- To avoid to have to use dynamic indexing
  - Favor tets that share faces/edges/vertices with current active elements

- Greedy strategy:
  - Local reordering
  - Small delay buffer from which compressor can pick the next tetrahedron to encode
Geometry Compression

• Quantization – Prediction - Encoding
  – 16 bits per vertex coordinate

• Prediction methods
  – Encode the vertex coordinates of an new vertex (ADD case) relative to the base triangle

• When prediction is not possible
  – No face available to take midpoint (START case)
  – Use delta coding (code difference) with previous vertex
Vertex Prediction

• How to encode the coordinates of a new vertex relative to base triangle?

Midpoint Rule:
Local coordinate system
Origin = Midpoint
Z axis // Base normal
Vertex Prediction

• Midpoint rule
  – Local coordinate system
  – Origin = Midpoint
  – Z axis // Base normal

Actual tip vertex

M = Predicted vertex
Vertex Prediction

• Flip rule
  – Extension of parallelogram rule for tets
  – Flip the tip vertex A of already known tet from the other side of the base through M
Vertex Prediction

• “Heightflip” rule
  – Combine Midpoint and Flip rules
  – Add a normal to the midpoint
  – Offset normal is scaled to the height of the tet on the other side
Vertex Prediction

• “Baseheight” rule
  – Same as heightflip rule
  – But estimate the height $h$ based on the area of the base triangle $\text{Area}$
    • $h' = 0.8 \times \sqrt{\text{Area} \times \text{Area}}$
Geometry Compression

• Encoding the residue/corrector
  – Good prediction => eliminates high-order bits
    • Int = 00000101 1010...

  – Hybrid coding:
    • Compress the highest eight non-zero bits with an entropy coder
    • Stores any remaining bit raw