GPU Construction and Transparent Rendering of Iso-Surfaces

Peter Kipfer
Rüdiger Westermann

tum.3D

computer graphics & visualization

Vision, Modeling and Visualization 2005, Erlangen
P. Kipfer – Computer Graphics and Visualization Group
Introduction

- Indirect volume visualization: display of an iso surface

- Fragment-based direct volume rendering with special transfer function for immediate display

- Geometry-based compute level set inside grid cell and store for further processing and display
Previous work

Marching tetrahedra for indirect volume rendering

- Works for all grids by splitting other element types
- Simple classification with less cases than marching cubes
- Only first order approximation of level set
- More elements

- Perform extraction on the GPU
- Avoids bus transfer bottleneck and exploits memory bandwidth and parallelism for interactive rendering
- Expect a speedup!
Previous GPU work

Implementation of the element classification

- in the vertex shader
  - compatible with CPU acceleration structures
  - send all element data all the times
  [Reck et al. 2004]

- in the fragment shader
  - more compute power and memory bandwidth
  - allows to store the surface vertices using OpenGL SuperBuffers
  - interpolation of vertex attributes very expensive
  - hardware restrictions (shader length)
  - no acceleration structures
  [Klein et al. 2004]
Marching Tetrahedra revisited

Classic approach: element-centric classification

1. mark element vertices wrt. iso value
2. lookup intersected edges according to marker
3. interpolate surface position along selected edges

- To avoid redundant interpolations, needs to store and access intermediate results
- On GPUs, vertex/fragment processing is independent
- Repeated classification and interpolation
Marching Tetrahedra revisited

New approach: edge-based processing

- Surface uniquely defined by edge intersections
- If vertices are sorted, element is implicitly classified by intersection status of edge $e_3$

$v_0 \leq v_1 \leq v_2 \leq v_3 \leq v_4$
Pass 1: Geometry (interpolation)

- Compute intersection along each edge and mark
  -1 iso value smaller than \(v_1\)
  \([0;1]\) valid intersection between \(v_1\) and \(v_2\)
  -2 iso value larger than \(v_2\)
Pass 1: Geometry (interpolation)

- Simple and short shader code

```cpp
edge = tex2D(Edges, TCoord[0]);
v0 = tex2D(Vertices, edge.xy);
v1 = tex2D(Vertices, edge.zw);

// we know that v0 has smaller scalar (stored in w comp.)
d = max(v1.w - v0.w, epsilon);
i = clamp((Iso - v0.w) / d);

result = lerp(v0, v1, i);

if (Iso > v1.w) result.w = -2;
else if (Iso < v0.w) result.w = -1;
else result.w = i;
```
Pass 2: Topology (global indices)

- Fetch global indices according to local index

  marker = -1 : [ 0 1 2 2 ] triangle-quad
  marker in [0;1] : [ 1 2 3 4 ] quad
  marker = -2 : [ 2 2 4 5 ] triangle-quad
Pass 2: Topology (global indices)

- Short and efficient shader code (17 ARB instr.)

```c
v = tex2D(InterpVtx, TCoord[0]);

if (v.w == -1)
    idx = [0, 1, 2, 2]; // iso smaller than values at edge3
else if (v.w == -2)
    idx = [2, 2, 4, 5]; // iso larger than values at edge3
else
    idx = [1, 2, 4, 3]; // flip last two for GL_QUAD draw

// get global edge indices of tet
map0 = tex2D(Map0, TCoord[0]*[2,1]);
map1 = tex2D(Map1, TCoord[0]*[2,1]);

res = map1.yyyy;
res = (idx < 5) ? map1.xxxx : res;
res = (idx < 4) ? map0.www : res;
res = (idx < 3) ? map0.zzzz : res;
res = (idx < 2) ? map0.yyyy : res;
res = (idx < 1) ? map0.xxxx : res;
```
Pass 3: Linearization (optional)

- Assemble linear arrays by indexed lookup into the interpolated vertices
- Create two array buffers for better aspect ratio
- Can write to multiple render targets if supported by shader output bandwidth

fixed point

floating point

global indices

linear data arrays

floating point
Properties

- Resulting geometry and topology buffer are in native OpenGL format
  - direct usability in any application that uses indexed drawing (*glDrawElements*)
  - optional third pass creates linear arrays for array-based drawing (*glDrawArrays*)
- Passes 1 and 3 can be carried out with any vertex attribute
  - interpolate any attribute on the surface (color, texture coordinates, normals, ...)
  - pass 2 (global indices) necessary only once
- Short shader code and good cache coherence
Acceleration structure

- Don't process non-contributing tetrahedra:
  - Global minimum scalar sort
  - Store min/max scalar value per row in an interval tree
  - Traverse tree for iso value to get contributing rows
  - Row-range valid for all passes
Processing

- **Pre-processing**
  - Sort tetrahedra, create canonical vertex ordering, split field into regions if too much tetrahedra for given index
  - Bit-width
  - Build interval tree

- **Extraction**
  - Traverse interval tree to determine active regions and quad sizes
  - Perform extraction passes
  - Draw surface
## Performance

<table>
<thead>
<tr>
<th>million tets per second in GPU memory</th>
<th>Interval tree disabled</th>
<th>Interval tree enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract (2 Pass)</td>
<td>65</td>
<td>83</td>
</tr>
<tr>
<td>Extract (3 Pass)</td>
<td>21</td>
<td>52</td>
</tr>
<tr>
<td>Extract + Render</td>
<td>15</td>
<td>42</td>
</tr>
<tr>
<td>ATI 9800Pro</td>
<td></td>
<td>ATI X800 XT</td>
</tr>
<tr>
<td>ATI X800 XT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Memory requirements

<table>
<thead>
<tr>
<th></th>
<th>element-based [Klein et al. 2004]</th>
<th>edge-based this approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex data</td>
<td>128 Bit</td>
<td>128 Bit</td>
</tr>
<tr>
<td>element data</td>
<td>224 Bit</td>
<td>128 Bit</td>
</tr>
<tr>
<td>texture read bandwidth</td>
<td>216 Bit</td>
<td>130 Bit*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* edge valence = 6</td>
</tr>
</tbody>
</table>
Demo

Linear interpolated vertex attributes

GPU sorted transparency

OpenGL extensions
third-party shader

cubic subdivision

Vision, Modeling and Visualization 2005, Erlangen
P. Kipfer – Computer Graphics and Visualization Group
Thanks for listening!

Questions?

Demo + Infos

http://wwwcg.in.tum.de