Geometry Processing on the GPU:
Parallel View-Dependent Refinement of Progressive Meshes

Speaker:
Pedro V. Sander
The Hong Kong University of Science and Technology

Joint work with Liang Hu and Hugues Hoppe
Overview

• Geometry processing on GPU
• Recent work: Vertex-level LOD on GPU
  • Challenging mesh “surgery” in parallel
  • Handle arbitrary meshes
  • Real-time performance
  • No CPU intervention
  • Use of a standard graphics API (DX10)
Related work

- Most methods for LOD on arbitrary meshes either:
  - On CPU (fine granular; per triangle)
  - On GPU (coarse granular; per cluster)

- View-dependent *vertex-granular* LOD on CPU:
  - Xia and Varshney [1996]
  - Hoppe [1997]

- Many others: Luebke et al. [2002] for survey
Basic operation: Edge collapse
Related work

- Hybrid approaches for arbitrary meshes (CPU+GPU), e.g.,
  - Cignoni et al. [2004]
  - Sander and Mitchell [2005]
  - Borgeat [2005]
Related work

- Terrain rendering
  - [Lindstrom et al. 1996], [Duchaineau et al. 1997], [Lindstrom and Pascucci 2002], [Levemberg 2002], [Cignoni et al. 2003], [Losasso and Hoppe 2004], ...

- Tessellation on GPU
  - Subdivision surfaces [Shiue et al. 2005]
  - NURBS patches [Guthe et al. 2005]
  - Procedural detail [Boubekeur and Schlick 2005], [Bokeloh and Wand 2006]
Related work

• DeCoro and Tatarchuk [2007]
  • Octree-based vertex clustering
  • Avoids precomputation of a vertex hierarchy
  • Less flexibility than full PM

• Ji et al. [2006]
  • LOD computations on GPU
  • Resampled to polycube map (not arbitrary mesh)
  • All vertices of original mesh must be processed during update
Talk overview

- Background: Geometry Shader
- Vertex hierarchy
- Data structures
- Runtime algorithm
- Amortized computation
- Videos
- Conclusion
Graphics Rendering Pipeline

Application → Vertex Shader → Rasterizer → Pixel Shader → Display
Graphics Rendering Pipeline

- Application
- Vertex Shader
- Geometry Shader
- Rasterizer
- Pixel Shader
- Memory/Display
The Geometry Shader

- Executed once per primitive (point/line/triangle)
- Parallel SIMD processing
- Two new features:
  - Variable data amplification
  - Zero or more output primitives
- Direct stream out to memory
- Skip rasterization, output to 1D buffer
Talk overview

• Background: Geometry Shader
• **Vertex hierarchy**
• Data structures
• Runtime GPU algorithm
• Amortized computation
• Videos
• Conclusion
Vertex hierarchy construction

- Using traditional triangle LOD techniques
- Preprocess:
  - Build PM vertex hierarchy [Xia and Varshney 1996, Hoppe 1997]
  - Half-edge collapse [Kobbelt et al. 1998]
Vertex hierarchy

$M^0$  
Base mesh

$M$  
Active mesh

Vertex frontier

$v_p$

$v$

$v_t$

$v_u$

$v_t^*$

Performing $spl_x$

Performing $col_x$

Leaf vertex $\in M^n$
Talk overview

- Background: Geometry Shader
- Vertex hierarchy
- **Data structures**
- Runtime GPU algorithm
- Amortized computation
- Videos
- Conclusion
### Data structure

#### Buffers | Elements | Memory
---|---|---

**Static structures**

<table>
<thead>
<tr>
<th><strong>VertexBuffer</strong></th>
<th><strong>Position</strong></th>
<th>12n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>TexCoord</strong></td>
<td>4n</td>
</tr>
<tr>
<td></td>
<td><strong>Normal</strong></td>
<td>4n</td>
</tr>
</tbody>
</table>

| **VertexTree**   | \(\{v_t, v_u, v_t^+\}\) | 12n |
|                  | \(v^0\)       | 8n  |

| **RefineCriteria** | \(\{\delta_v, r_v, \sin^2 \alpha_v\}\) | 4n |

| **OrigFaces**     | \(\{v_0, v_1, v_2\}\) | 18n |
|                  | \(v_{max}(v_{r_{max}})\) | 6n  |

| **BaseVStream**   | \(v\) | _† |
| **BaseFStream**   | \(f\) | _† |

**Dynamic structures**

| **VertexState** | \(v_{state}\) | \(n\) |
| **VertexStream** | \(v\) | \(4m \times 2\) |
| **IndexBuffer**  | \(\{v_0, v_1, v_2\}\) | \(24m \times 2\) |

**Total**

\[69n + 56m\]

\(n = |\text{fine mesh}|\)

\(m = |\text{active mesh}|\)

\(_† = |\text{base mesh}|\) (low complexity)
## Data structure

<table>
<thead>
<tr>
<th>Buffers</th>
<th>Elements</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Static structures</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>VertexBuffer</em></td>
<td>Position</td>
<td>$12n$</td>
</tr>
<tr>
<td></td>
<td>Texcoord</td>
<td>$4n$</td>
</tr>
<tr>
<td></td>
<td>Normal</td>
<td>$4n$</td>
</tr>
<tr>
<td><em>VertexTree</em></td>
<td>${v_t, v_u, v_t^*}$</td>
<td>$12n$</td>
</tr>
<tr>
<td></td>
<td>$v_p$</td>
<td>$8n$</td>
</tr>
<tr>
<td><em>RefineCriteria</em></td>
<td>${\delta_v, r_v, \sin^2 \alpha_v}$</td>
<td>$4n$</td>
</tr>
<tr>
<td><em>OrigFaces</em></td>
<td>${v_0, v_1, v_2}$</td>
<td>$18n$</td>
</tr>
<tr>
<td></td>
<td>$v_{l_{\max}}(v_{r_{\max}})$</td>
<td>$6n$</td>
</tr>
<tr>
<td><em>BaseVStream</em></td>
<td>$v$</td>
<td>_††</td>
</tr>
<tr>
<td><em>BaseFStream</em></td>
<td>$f$</td>
<td>_††</td>
</tr>
<tr>
<td><strong>Dynamic structures</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>VertexState</em></td>
<td>$v_{state}$</td>
<td>$n$</td>
</tr>
<tr>
<td><em>VertexStream</em></td>
<td>$v$</td>
<td>$4m \times 2$</td>
</tr>
<tr>
<td><em>IndexBuffer</em></td>
<td>${v_0, v_1, v_2}$</td>
<td>$24m \times 2$</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>$69n + 56m$</td>
</tr>
</tbody>
</table>

$n = |\text{fine mesh}| \quad m = |\text{active mesh}|$

_†† = |\text{base mesh}| (low complexity)
Vertex hierarchy

$M^0$
Base mesh

$M$
Active mesh

$\nu^p$

$\nu$

$\{ f^v_l, f^v_r \}$

$\nu_t$

$\nu_u$

$\nu_t^*$

Performed $spl_x$

Performed $col_x$

Leaf vertex $\in M^n$
## Data structure

### Buffers

<table>
<thead>
<tr>
<th>Static structures</th>
<th>Elements</th>
<th>Cardinality</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VertexBuffer</strong></td>
<td>Position</td>
<td>L</td>
<td>12n</td>
</tr>
<tr>
<td></td>
<td>Texcoord</td>
<td>I</td>
<td>4n</td>
</tr>
<tr>
<td></td>
<td>Normal</td>
<td>LI</td>
<td>4n</td>
</tr>
<tr>
<td><strong>VertexTree</strong></td>
<td>( { v_p, v_u, v_t^* } )</td>
<td>I</td>
<td>12n</td>
</tr>
<tr>
<td>( v_t^p )</td>
<td></td>
<td>LI</td>
<td>8n</td>
</tr>
<tr>
<td><strong>RefineCriteria</strong></td>
<td>( { \delta_v, r_v, \sin^2 \alpha_v } )</td>
<td>I</td>
<td>4n</td>
</tr>
<tr>
<td><strong>OrigFaces</strong></td>
<td>( { v_0, v_1, v_2 } )</td>
<td>M( ^n )</td>
<td>18n</td>
</tr>
<tr>
<td>( v_{\text{max}}(v_{r\text{max}}) )</td>
<td></td>
<td>6n</td>
<td></td>
</tr>
<tr>
<td><strong>BaseVStream</strong></td>
<td>( v )</td>
<td>M(^0 )</td>
<td>_( ^\dagger )</td>
</tr>
<tr>
<td><strong>BaseFStream</strong></td>
<td>( f )</td>
<td>M(^0 )</td>
<td>_( ^\dagger )</td>
</tr>
</tbody>
</table>

### Dynamic structures

<table>
<thead>
<tr>
<th>Dynamic structures</th>
<th>Elements</th>
<th>Cardinality</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VertexState</strong></td>
<td>( v_{\text{state}} )</td>
<td>I</td>
<td>n( ^n )</td>
</tr>
<tr>
<td><strong>VertexStream</strong></td>
<td>( v )</td>
<td>M</td>
<td>4m ( \times 2 )</td>
</tr>
<tr>
<td><strong>IndexBuffer</strong></td>
<td>( { v_0, v_1, v_2 } )</td>
<td>M</td>
<td>24m ( \times 2 )</td>
</tr>
</tbody>
</table>

| Total             |             |             | 69n + 56m |

n = \( |\text{fine mesh}| \) \hspace{1cm} m = \( |\text{active mesh}| \)

_\( ^\dagger \) = \( |\text{base mesh}| \) (low complexity)
• Small memory footprint
• Active mesh $m$ significantly smaller than original mesh $n$

<table>
<thead>
<tr>
<th>View-dependent LOD scheme</th>
<th>Memory size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDPM [Hoppe 1997]</td>
<td>$216n$</td>
</tr>
<tr>
<td>SVDLOD [Hoppe 1998]</td>
<td>$88n + 100m$</td>
</tr>
<tr>
<td>MT [Floriani et al. 1998]</td>
<td>$75n$</td>
</tr>
<tr>
<td>VDT [El-Sana and Varshney 1999]</td>
<td>$90n$</td>
</tr>
<tr>
<td>FastMesh [Pajarola and DeCoro 2004]</td>
<td>$88n + 6m$</td>
</tr>
<tr>
<td>Our scheme</td>
<td>$69n + 56m$</td>
</tr>
</tbody>
</table>
Meshes
## Mesh sizes and static memory

<table>
<thead>
<tr>
<th>Model name</th>
<th>Total # faces</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lucy</td>
<td>2,000,000</td>
<td>65.8</td>
</tr>
<tr>
<td>Terrain</td>
<td>2,097,147</td>
<td>69.0</td>
</tr>
<tr>
<td>Dragon</td>
<td>7,218,906</td>
<td>237.5</td>
</tr>
<tr>
<td>Statue</td>
<td>10,000,000</td>
<td>329.0</td>
</tr>
</tbody>
</table>
Talk overview

- Background: Geometry Shader
- Vertex hierarchy
- Data structures
- **Runtime GPU algorithm**
- Amortized computation
- Videos
- Conclusion
Recall...

- **VertexStream** encodes both:
  - Active faces
    - $f_l$ and $f_r$ of all vertices in VertexStream
    - (+ Faces in BaseFStream)
  - Active vertices
    - Children of vertices in VertexStream that are *collapsed*
Algorithm overview

• **UpdateVertexState**
  • Check for splits/collapses on GS, update state accordingly

• **UpdateVertexStream**
  • Based on recently updated state

• **UpdateIndexBuffer**
  • Write the final index buffer given updated vertex state and stream
1. UpdateVertexState

- **Input**: \textit{VertexStream}
- **GS**: Check if stays split
  - Criteria from Hoppe [1998]
  - Also update children and grandchildren state to “splitting” if they should split now
- **Output**: \textit{VertexState} texture
  - Use \textit{point primitives} to scatter updates based on ID
Addressing dependencies

- Problem: The split dependencies will propagate

- Hard to do in parallel
  - Not efficient to maintain single update stack
  - GS Output limit

- How to still process splits independently?
Cascaded update

- **Option 1:**
  - During split, also tag neighbors that need to be split by setting their state to *splitting*.
  - Wait until a later iteration.
  - Problem: Lag.

- **Option 2:**
  - Tag neighbors that need to be split as above.
  - Do *not* wait. Split current vertex immediately.
  - Result:
    - Significantly faster.
    - May result in temporary foldovers.
    - Hardly observed very fast updates.
Addressing dependencies

• Problem: Conflict if
  • A vertex $v$ wants a neighbor $v'$ to split AND $v'$ doesn’t think it needs to split

• Solution
  • When processing $v$, also set $v'$ to split regardless of VD criteria test
2. UpdateVertexStream

- **Input:**
  - `VertexStream`
  - `BaseVStream`

- **GS:** If not in collapsed state, output vertex
  - Output splitting descendants as well

- **Output:** Updated `VertexStream`
  - Uses `stream-out` to output vertices
3. UpdateIndexBuffer

- **Input:**
  - VertexStream
  - BaseFStream
- **GS:** Write active mesh triangle faces
- The two faces stored in each vertex of VertexStream
- Active vertex indices must be retrieved from frontier
- Also process base mesh faces from BaseFStream
- **Output:** IndexBuffer (ready to render!)
Update time

- Roughly linear on **active** mesh
- Updates/collapses cause additional processing
Talk overview

• Background: Geometry Shader
• Vertex hierarchy
• Data structures
• Runtime GPU algorithm
• **Amortized computation**
• Videos
• Conclusion
Amortized computation

- Three major update steps
  - May be costly
  - Require different amount of time

- Objective:
  - Partition across multiple frames
  - Maintain bound on frame time (e.g., 33.3ms)
Amortized computation

• Solution 1: Naive amortization
  • One step per iteration
  • Each step may still take longer than desired frame time
**Amortized computation**

- **Solution 2: Per-step amortization**
  - Partition step based on time from previous iteration
    - # partitions based on time step took in previous frame
  - Still does not load-balance across steps (+ lag)
• Solution 3: Full amortization
  • Predict each step cost from previous iteration
  • Execute fraction of next step that fits within budget
  • Achieves much more stable framerates
Talk overview

• Background: Geometry Shader
• Vertex hierarchy
• Data structures
• Runtime GPU algorithm
• Amortized computation
• Videos
• Conclusion
Lucy (side view)
Dragon (camera view)
Dragon (side view)
Future work

- Indexed triangle strips
- Improved vertex cache locality
- Geomorphs for smoother transitions
- Geometry amplification costs

- Other APIs? (e.g., CUDA, CAL)
  - Use shared memory to maintain dependency stacks
  - Scattered reads and writes to same buffer
Summary

• View-Dependent PM entirely on GPU
  • Vertex-granular
  • Handles arbitrary meshes
  • Real-time performance
  • Entirely on GPU – just a set of draw calls
    • Easier to incorporate in rendering engines
  • Using a standard graphics API

• Hope the insights from parallel mesh “surgery” are useful
More information

• I3D 2009 paper:
  • Liang Hu, Pedro V. Sander, Hugues Hoppe. Parallel View-Dependent Refinement of Progressive Meshes.

• Available online on my page next week:
  • http://www.cse.ust.hk/~psander/
  • or Google my name (“Pedro Sander”)