GPU-Based Volume Rendering of Unstructured Grids

Module 2: Projected Tetrahedra + Polyhedral Cell Sorting

Cláudio T. Silva
University of Utah
Outline

• Introduction to Volume Rendering

• Polyhedral Cell Sorting

• Hardware-Assisted Techniques
Introduction to Volume Rendering
GPU-Based Volume Rendering of Unstructured Grids

Grid Types

- Regular
- Rectilinear
- Curvilinear
- Irregular
Volume Rendering vs Isosurfaces

- **Direct Volume Rendering**
  - Volume data → Image
  - Looks “inside” the data

- **Isosurfaces**
  - Volume data → Polygon model
  - Slice of the data
Isosurfaces

- For a query value \( q \), find and display the isosurface of \( q \):
  \[ C(q) = \{ p \mid F(p) = q \} \]
Volume Rendering at a High Level

(a) Sampling Phase
(b) Sorting Phase
GPU-Based Volume Rendering of Unstructured Grids
Rendering Unstructured Grids

Visibility Sort

Software

for each cell in order

Programmable Hardware

GPU find thickest cell distance

decompose to triangles

compute each triangle’s parameters

compute cell’s screen projection

model with millions of cells

PC (CPU)

GPU-Based Volume Rendering of Unstructured Grids
Visibility Sorting
Typical Rendering Pipeline

Sort Tetrahedra  ➔  Subdivide Tetrahedra into triangles  ➔  Render Triangles
GPU-Based Volume Rendering of Unstructured Grids

Existing Techniques

- Given mesh with $n$ cells, $b$ of them on the boundary.

<table>
<thead>
<tr>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort Tetrahedra</td>
<td>Subdivide Tetrahedra into triangles</td>
</tr>
</tbody>
</table>

- XMPVO $O(n+b^2)$
- BSP-XMPVO $O(n+bp)$
- SXMPVO $\sim O(n)$
- Shirley-Tuchman 90
  Generates $\sim 3.4n$ triangles
Existing Techniques

Sort Tetrahedra

Subdivide Tetrahedra into triangles

Render Triangles

CPU

GPU

XMPVO $O(n+b^2)$
BSP-XMPVO $O(n+bp)$
SXMPVO $\sim O(n)$

Wylie et al ‘02
Generates $\sim 3.4n$ triangles

Unfortunately, it is slower in practice because of some implementation overheads!
GPU-Based Volume Rendering of Unstructured Grids

**Process Flow**

1. **CPU**
   - Sort Tetrahedra

2. **GPU**
   - Subdivide Tetrahedra into triangles

3. **CPU**
   - Render Triangles

**Key Terms**

- HAVS
- SIBGRAPI 2005
- CPU
- GPU
- Sort Tetrahedra
- Subdivide Tetrahedra into triangles
- Render Triangles

**Additional Information**

- GPU-Based Volume Rendering of Unstructured Grids
GPU-Based Volume Rendering of Unstructured Grids

SIBGRAPI 2005

HAVS

Sort NOT Tetrahedra Triangles!

Render Triangles

CPU | GPU

GPU-Based Volume Rendering of Unstructured Grids
**Idea:** Define ordering relations by looking at shared faces.

- B < A
- A < C
- B < E
- C < E
- C < D
- E < F
- D < F

Viewing direction
MPVO Limitations

Missing relations!
MPVONC Rendering Errors

SXMVO
MPVONC
ERRORS

GPU-Based Volume Rendering of Unstructured Grids
GPU-Based Volume Rendering of Unstructured Grids

Depth-Sorting Algorithm History

- 1989-92: Max et al, VolVis90
  Convex case only!
  O(n^2)
- 1998: Williams et al, TVCG98
  O(n^2)
- 1999: Williams, TOG92
  O(n)

History

<table>
<thead>
<tr>
<th>Year</th>
<th>Rendering Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1989-92</td>
<td>1s</td>
</tr>
<tr>
<td></td>
<td>10s</td>
</tr>
<tr>
<td></td>
<td>100s</td>
</tr>
<tr>
<td></td>
<td>1000s</td>
</tr>
<tr>
<td></td>
<td>10000s</td>
</tr>
</tbody>
</table>

O(n^2)
Idea: Using ray shooting queries to complement ordering relations.

A < C
A < B
B < D
GPU-Based Volume Rendering of Unstructured Grids

SIBGRAPI 2005

Depth-Sorting Algorithm History

Rendering Time

1989-92
1994
1998
1999
2001

10000s
1000s
100s
10s
1s

O(n^2)
Stein et al, VolVis94

O(n^2)
Williams et al, TVCG98

O(n)
Williams, TOG92

Max et al, VolVis90

Convex case only!

General case →

Silva et al

XMPVO: O(b^2 + n)
Binary Space Partitioning Trees

1) Breaks geometry into cycle-free fragments
2) Provides a mechanism for computing visibility orders
Intuition: BSP for Visibility Ordering
Idea: Use BSP tree to replace ray shooting queries.
Idea: Use BSP tree to replace ray shooting queries
I.e., add extra ordering relation for BSP-tree

$H \rightarrow \{2, 3, 4, 5\}$
Idea: Use BSP tree to replace ray shooting queries

I.e., add extra ordering relation for BSP-tree

But Because of Cell Fragmentation the BSP-tree Does Not Catch all Necessary Ray Shooting Queries
Problem: G is partially projected, but we need to guarantee that F is projected after G.

Solution: Keep a list of partially projected cells.
BSP-XMPVO relations

- **MPVO dependencies** (<\textsubscript{adj}>)
  - Adjacency relation given by mesh

- **BSP dependencies** (<\textsubscript{bsp}>)
  - Each fragment c' on the boundary of C define a BSP-dependency for cell C

- **PPC dependency** (<\textsubscript{ppc}>)
  - If C' is partially projected and C' lies behind cell C, then we create a PPC dependency for C
GPU-Based Volume Rendering of Unstructured Grids

SIBGRAPI 2005

BSP-XMPVO Algorithm

BSP Traversal

See paper for proof of correctness!

Update Graph Dep.

Williams’ MPVO

Algorithm BSP-XMPVO_traversal(node, vp)
/* The algorithm projects in back-to-front order the part of the mesh S corresponding to BSP tree node node with respect to the viewpoint vp. */
1. if (node == NULL) then return;
2. if (vp is in front plane)
3. BSP-XMPVO_traversal(back(node));
4. BSP-XMPVO_update_dep(node);
5. BSP-XMPVO_traversal(front(node));
6. else
7. BSP-XMPVO_traversal(front(node));
8. BSP-XMPVO_update_dep(node);
9. BSP-XMPVO_traversal(back(node));

Algorithm BSP-XMPVO_update_dep(node)
/* Updates the dependency counters for the cells whose faces lie on node’s base plane. */
1. for (i = 0; i < numPPC; i++)
2. for (j = 0; j < numCutCells(node); j++)
3. Check_update_ppc_dep_count (C_i, C_j);
4. for (i = 0; i < numCutCells(node); i++)
5. Update_PPC (C_i);
6. for (i = 0; i < numCutCells(node); i++)
7. Decrem_bsp_dep_count(C_i);
8. if (num_inbound(C_i) == 0) and
9. (bsp_dep_count(C_i) == 0) and
10. (ppc_dep_count(C_i) == 0) and
11. enqueue(C_i);
12. MPVO_traverse();

Algorithm MPVO_traverse()
/* Modified MPVO traverse. */
1. while (dequeue(c) != false)
2. output(C);
3. for (i = 0; i < numFaces(C); i++)
4. if arrow(i, C) == INBOUND continue;
5. C_i = neighbor(C, i);
6. Decrem_num_inbound(C_i);
7. if (num_inbound(C_i) == 0) and
8. (bsp_dep_count(C_i) == 0) and
9. (ppc_dep_count(C_i) == 0)
10. enqueue(C_i);
<table>
<thead>
<tr>
<th>Year</th>
<th>Algorithm</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1989-92</td>
<td>Williams, TOG92</td>
<td>(O(n))</td>
</tr>
<tr>
<td>1994</td>
<td>Stein et al, VolVis94</td>
<td>(O(n^2))</td>
</tr>
<tr>
<td>1998</td>
<td>Williams et al, TVCG98</td>
<td>(O(n^2))</td>
</tr>
<tr>
<td>2001</td>
<td>Silva et al</td>
<td>(O(b^2 + n))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(O(n))</td>
</tr>
</tbody>
</table>

**GPU-Based Volume Rendering of Unstructured Grids**
GPU-Based Volume Rendering of Unstructured Grids

SIBGRAPI 2005

Depth-Sorting Algorithm History

Rendering Time

1989-92
1994
1998
1999
2001

10000s
1000s
100s
10s
1s

Max et al, VolVis90
O(n)

Williams, TOG92
O(n)

Stein et al, VolVis94
O(n^2)

Williams et al, TVCG98
O(n^2)

Silva et al

XMPVO: \(O(b^2 + n)\)

BSP-XMPVO: \(O(n + |PPC| \times b)\)

SXMPVO: \(O(n)\)

Convex case only!

General case →
Outline

• Introduction to Volume Rendering
• Polyhedral Cell Sorting
• Hardware-Assisted Techniques
Shirley-Tuchman (ST) Algorithm

Class 1

Class 2

Class 3

Class 4
Wylie et al’s GPU-based ST

- Moves all of the following functions from the CPU to the GPU:
  - Transform to screen space
  - Determine projection class
  - Calculate thick vertex location
  - Determine depth at thick vertex
  - Compute color and opacity for thick vertex
  - Apply exponential attenuation texture
Each instance of a vertex shader program works independently on a single vertex in SIMD fashion

No support dynamic vertex creation or topology modification within the vertex program

No branching (at the time!)

No knowledge of neighboring vertices

Cannot change execution based on past information
Idea: Morph a Canonical Graph

Basis Graph

Isomorphic to all projection cases

Example later…
PT algorithm in Vertex Program

- Transform to screen space.
- Determine projection class (and permutation).
- Map the vertices to the basis graph.
- Calculate intersection point of line segments.
- Determine depth at thick vertex.
- Compute color and opacity for thick vertex (texture).
- Multiplex the result to correct output vertex.
PT algorithm in Vertex Program

- Transform to screen space. (Trivial)
- Determine projection class (and permutation).
- Map the vertices to the basis graph.
- Calculate intersection point of line segments.
- Determine depth at thick vertex.
- Compute color and opacity for thick vertex (texture)
- Multiplex the result to correct output vertex.
PT algorithm in Vertex Program

- Transform to screen space.
- Determine projection class (and permutation).
- Map the vertices to the basis graph.
- Calculate intersection point of line segments.
- Determine depth at thick vertex.
- Compute color and opacity for thick vertex (texture).
- Multiplex the result to correct output vertex.
Projection Classes

Class 1

Class 2

Class 3

Class 4
Projection Permutations

- Permutation Determination
  - 14 cases need at least 4 Boolean tests

- Definitions
  - vec1 = v1-v0
  - vec2 = v2-v0
  - vec3 = v3-v0
  - cross1 = vec1 x vec2
  - cross2 = vec1 x vec3
  - cross3 = vec2 x vec3

- Tests
  - test1 = (cross1*cross2 < 0)
  - test2 = (cross1*cross3 > 0)
  - test3 = (distance from v0 to middle vertex – distance from v0 to Intersection) > 0
  - test4 = (cross1 > 0)
PT algorithm in Vertex Program

- Transform to screen space.
- Determine projection class (and permutation).
- Map the vertices to the *basis graph*.
- Calculate intersection point of line segments.
- Determine depth at thick vertex.
- Compute color and opacity for thick vertex (texture)
- Multiplex the result to correct output vertex.
Isomorphic Property of Basis Graph

Object Space

Screen Space
(Case 5 projection)

Triangle Output

Map to Basis Graph
($V_0$ maps to both $V_0'$ and $V_3'$)

(Coincident points $V_0'$ and $V_3'$ create degenerate triangle)

GPU-Based Volume Rendering of Unstructured Grids
PT algorithm in Vertex Program

- Transform to screen space.
- Determine projection class (and permutation).
- Map the vertices to the *basis graph*.
- Calculate intersection point of line segments.
- Determine depth at thick vertex.
- Compute color and opacity for thick vertex (texture).
- Multiplex the result to correct output vertex.
In all cases the coordinates of the intersection point I are computed. (Intersection of lines computed ala Graphics Gems III p. 199-202). This intersection calculation gives us $\alpha$ and $\beta$ terms that are used for interpolation (depth, alpha, and color) later on.

- **Class 2:**
  
  ```c
  float thickness = fabs(z1-z2);
  ```

- **Class 1:**
  
  ```c
  // Extra computation for class 1
  if (!test3) thickness /= alpha;
  ```
PT algorithm in Vertex Program

- Transform to screen space.
- Determine projection class (and permutation).
- Map the vertices to the basis graph.
- Calculate intersection point of line segments.
- Determine depth at thick vertex.
- **Compute color and opacity for thick vertex (texture)**
- Multiplex the result to correct output vertex.
Color and Opacity Calculation

- Use same $\alpha$ and $\beta$ terms to interpolate color and opacity along the line segments to give the front and back face intersection terms $C_F$, $C_B$ and $\tau_F$, $\tau_B$.

- Thick vertex color $(C_F + C_B) / 2$ *

- The extinction coefficient $\tau$ is $(\tau_F + \tau_B) / 2$.

- $\tau$ and the thickness $l$, are then used as lookups into a 2D texture map defined as $1 - \exp(-\tau l)$. [Stein et al. 1994].

* approximate color from Shirley and Tuchman.
PT algorithm in Vertex Program

- Transform to screen space.
- Determine projection class (and permutation).
- Map the vertices to the *basis graph*.
- Calculate intersection point of line segments.
- Determine depth at thick vertex.
- Compute color and opacity for thick vertex (texture).
- *Multiplex the result to correct output vertex.*
Multiplex input to output

- Use a lookup table (loaded in the parameter registers)
- and an index based on the 4 tests to determine the
- output vertex.

// Which vertex to copy to output (using lookup table)
lookup_index = test1*8 + test2*4 + test3*2 + test4;
output_vertex = lookup_table[call_index][lookup_index];
// Load up the 4 vertices
glVertexAttrib3fvNV(1, nodes[0]->getXYZ());
glVertexAttrib3fvNV(2, nodes[1]->getXYZ());
glVertexAttrib3fvNV(3, nodes[2]->getXYZ());
glVertexAttrib3fvNV(4, nodes[3]->getXYZ());

// Load up color for the vertices
glVertexAttrib4fvNV(5, colorvectors[0]);
glVertexAttrib4fvNV(6, colorvectors[1]);
glVertexAttrib4fvNV(7, colorvectors[2]);
glVertexAttrib4fvNV(8, colorvectors[3]);

// Writing to v[0] here invokes the vertex program.
gBegin(GL_TRIANGLE_FAN);
    glVertexAttrib3sNV(0, 0, 1, 0);
    glVertexAttrib3sNV(0, 1, 0, 1);
    glVertexAttrib3sNV(0, 2, 0, 1);
    glVertexAttrib3sNV(0, 3, 0, 1);
    glVertexAttrib3sNV(0, 4, 0, 1);
    glVertexAttrib3sNV(0, 1, 0, 1);
End();

These calls could easily be wrapped up into a glTetraExt() call.
Remarks

• Wylie et al’s technique can be easily extended to other computations, e.g., isosurface or isoline generation (this was a homework exercise in my graphics class last Spring)

• For isosurfaces, one can send two triangles (four vertices in a strip), since for a tetrahedral cell, the isosurface going through it has at most two triangles (see Comba’s talk later today)
More Remarks

- Cell sorting is complicated, and error prone! (In our opinion, SXMPVO is probably easier to implement, and most robust technique)
- Cell projection is more stable, but still fairly complicated.
- Code available from http://www.cs.utah.edu/~csilva
Acknowledgements

- DOE VIEWS, DOE MICS, SNL, and LLNL, NSF, ARO, NIH, IBM, U of Utah.

- Joao Comba, Ricardo Farias, Brian Wylie, and others for slides and help with presentation