Hardware-Based Visibility Ordering

Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering and
Vis-Sort: Fast Visibility Ordering of 3-D Geometric Primitives
Overview

- Recent changes in hardware programmability
- k-Buffer: A fragment stream sorter
- Hardware-Assisted Visibility Sorting
- Dynamic Level-of-Detail
- Current research
- Vis-Sort
GPU: Recent Features

Render to texture

- Why?
  - Better performance
- Applications
  - Dynamic textures
  - Multi-pass algorithms
  - Image processing
GPU: Recent Features

Multiple Render Targets (MRTs)

- Write into multiple textures simultaneously
OpenGL Pixel Buffers (PBuffers)

- Enables off-screen rendering
- Contains its own depth, stencil, and aux buffers
- MRT support by rendering into Front and up to 3 AUX buffers
GPU: Recent Features

Disadvantages of PBuffers

- Each has its own OpenGL context
- Switching between PBuffers is expensive
- Cannot share buffers between PBuffers
- Pixel format selection
- Extensions only available on Windows
OpenGL Framebuffer Objects (FBOs)

- A collection of attachable textures
  - Color, depth, stencil, etc.
- Attached textures are source and destination for fragment shaders
- MRTs are available using multiple color attachments
Advantages of FBOs
- A single context
- Pixel format determined by texture format
- Share buffers between FBOs
- Easier to use than PBuffers
- Works on multiple platforms
GPU: Recent Features

Code

- PBuffers
  - RenderTexture 2.0 (Mark Harris)
- FBOs
  - Framebuffer Object Class (Aaron Lefohn)

www.gpgpu.org/developer
Object-Space Sorting

[Williams]
Sorting

Application
Object-Space Sorting
Rasterization
Image Space
Display
Image-Space Sorting

[Image: Diagram of image space sorting]

[Carpenter]
Sorting

Application

Object-Space Sorting

Rasterization

Image Space

Display
\textbf{k-Buffer}

- Fixed-size A-Buffer
- As a new pixel is inserted, another is removed
- Can efficiently sort a \textit{k}-Nearly Sorted Sequence (\textit{k}-NSS)

[Callahan et al. 05]
$k$-Buffer

input: 1 3 2 4 6 5

$k$-buffer: 

output: 


$k$-Buffer

input: 1 3 2 4 6 5

$k$-buffer: 1

output: 


$k$-Buffer

**input**

1 3 2 4 6 5

**$k$-buffer**

1 3

**output**
$k$-Buffer

Input: 1 3 2 4 6 5

$k$-Buffer: 3 2

Output: 1
\[ k \text{-Buffer} \]

<table>
<thead>
<tr>
<th>input</th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>5</th>
</tr>
</thead>
</table>

\[ k \text{-buffer} \]

<table>
<thead>
<tr>
<th>output</th>
<th>1</th>
<th>2</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
$k$-Buffer

- **input**: 1 3 2 4 6 5
- **$k$-buffer**: 4 6
- **output**: 1 2 3
A diagram labeled \( k \)-Buffer shows a sequence of input values 1, 3, 2, 4, 6, 5. The \( k \)-buffer contains the values 6 and 5, and the output sequence is 1, 2, 3, 4.
$k$-Buffer

**input**

\[
\begin{array}{ccccccc}
1 & 3 & 2 & 4 & 6 & 5 \\
\end{array}
\]

**$k$-buffer**

\[
\begin{array}{c}
6 \\
\end{array}
\]

**output**

\[
\begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 \\
\end{array}
\]
$k$-Buffer

**input**

```
1 3 2 4 6 5
```

**k-buffer**

```
[
]
```

**output**

```
1 2 3 4 5 6
```
Object-Space Sorting

- Performed on CPU
- Sort faces by center
- Least Significant Digit Radix Sort
- Handles floating-point numbers

```c
inline unsigned int float2fint (unsigned int f)
{
    return f ^ ((-(f >> 31)) | 0x80000000);
}
```

- Results: 15 million faces/sec
Image-Space Sorting

- Performed on GPU
- Uses $k$-Buffer as a fragment stream sorter
- Keeps $k$ entries per pixel, each entry contains a fragment’s scalar value and distance from the viewpoint $(v, d)$
- An incoming fragment replaces the entry that is closest to the eye (front-to-back compositing)
Hardware-Assisted Visibility Sorting

Sort in image-space and object-space
- Approximate sort in object-space
- Complete sort in image-space
### $k$-Buffer In Hardware

<table>
<thead>
<tr>
<th>Texture 1</th>
<th>$r_{\text{comp}}$</th>
<th>$g_{\text{comp}}$</th>
<th>$b_{\text{comp}}$</th>
<th>$a_{\text{comp}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texture 2</td>
<td>$v_1$</td>
<td>$d_1$</td>
<td>$v_2$</td>
<td>$d_2$</td>
</tr>
<tr>
<td>Texture 3</td>
<td>$v_3$</td>
<td>$d_3$</td>
<td>$v_4$</td>
<td>$d_4$</td>
</tr>
<tr>
<td>Texture 4</td>
<td>$v_5$</td>
<td>$d_5$</td>
<td>$v_6$</td>
<td>$d_6$</td>
</tr>
</tbody>
</table>
$k$-Buffer in Hardware
Details

- Fix incorrect texture coordinates caused by perspective-correct interpolation

Perspective Correct

Projecting vertices to find tex coords

Projecting tex coords in shader
Simultaneously reading and writing to a texture is undefined when fragments are rasterized in parallel.
Details

- Initialization and Termination
- Non-convex objects
Experiments

Environment
- 3.2 GHz Pentium 4
- 2048 MB RAM
- Windows XP
- ATI Radeon 9800 Pro

Results
- $k$-Buffer analysis
- Performance results
### $k$-Buffer Analysis

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Max A</th>
<th>Max $k$</th>
<th>$k &gt; 2$</th>
<th>$k &gt; 6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spx2</td>
<td>476</td>
<td>22</td>
<td>10,262</td>
<td>512</td>
</tr>
<tr>
<td>Torso</td>
<td>649</td>
<td>15</td>
<td>43,317</td>
<td>1,683</td>
</tr>
<tr>
<td>Fighter</td>
<td>904</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Accuracy Analysis
- $k$ depth required to render datasets
- Max values from 14 fixed viewpoints
$k$-Buffer Analysis

Distribution Analysis

- Shows the actual pixels that require large $k$ depths to render correctly

- $k \leq 2$: green
- $2 < k \leq 6$: yellow
- $k > 6$: red
Results

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Cells</th>
<th>$k = 2$ fps</th>
<th>$k = 2$ tets/s</th>
<th>$k = 6$ fps</th>
<th>$k = 6$ tets/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spx2</td>
<td>0.8 M</td>
<td>2.07</td>
<td>1712 K</td>
<td>1.7</td>
<td>1407 K</td>
</tr>
<tr>
<td>Torso</td>
<td>1.1 M</td>
<td>3.13</td>
<td>3390 K</td>
<td>1.86</td>
<td>1977 K</td>
</tr>
<tr>
<td>Fighter</td>
<td>1.4 M</td>
<td>2.41</td>
<td>3387 K</td>
<td>1.56</td>
<td>2190 K</td>
</tr>
</tbody>
</table>

GPU Sorting Performance

- $512^2$ viewport with a $128^3$ pre-integrated lookup table
Results

```
Dataset  | Cells  | CPU   | GPU  | Total  | Tets/s |
---------|--------|-------|------|--------|--------|
Spx2     | 0.8 M  | 160 ms| 368 ms| 528 ms | 1568 K |
Torso    | 1.1 M  | 210 ms| 390 ms| 600 ms | 1805 K |
Fighter  | 1.4 M  | 268 ms| 505 ms| 773 ms | 1816 K |
```

Total Performance

- CPU sorting + GPU sorting and compositing
- Pipeline optimization = \( \max(\text{CPU}, \text{GPU}) \)
Movie

Spx2
828K Tetrahedra
Movie

**Fighter**

1.40M Tetrahedra
Conclusion

- Introduced the $k$-buffer and an efficient GPU implementation
- Fastest volume renderer for unstructured data
- Can handle arbitrary non-convex meshes
- Requires minimal pre-processing of data
- Maximum data size is bounded by main memory
- Code is short and simple
- Requires no neighbor information
Dynamic Level-of-Detail

100% 2.0 fps
25% 5.3 fps
5% 16.1 fps

[Callahan et al. 05]
Time Varying Scalars

[Bernardon et al. 06]
iRun

[Vo et al. 06]
Depth Peeling

[Everitt 01, Bernardon et al. 05]
for (i=0; i<num_passes; i++)
{
    clear color buffer
    A = i % 2
    B = (i+1) % 2
    depth unit 0:
        if(i == 0)
            disable depth test
        else
            enable depth test
    bind buffer A
    disable depth writes
    set depth func to GREATER
    depth unit 1:
        bind buffer B
        clear depth buffer
        enable depth writes
        enable depth test
        set depth func to LESS
        render scene
    save color buffer RGBA as layer i
}
Nearly Sorted Sequences

- Knuth’s measure of disorder:
  - Given an input $I$, the measure of disorder is defined as the minimal number of elements that need to be removed so that the rest of the sequence is sorted.
  - Vis-Sort is $O(||Y|| n)$, where $Y$ is the set of disordered elements
Vis-Sort 1D

- Definitions:
  - \( I \) = the input list of elements for the sorting algorithm
  - \( S \) = the output list of elements in sorted order
  - \( M \) = the monotonically increasing sequence computed in each iteration, where \( M \subseteq I \)

[Govindaraju et al. 04]
**Vis-Sort 1D**

**1D Sort**
1. \( I = \) Unsorted input, \( S = \) \(
2. \) while (\( I \) is not empty)
   Begin Iteration:
3. \( \) do
4. \( M = \) ComputeIncSeq(\( I \))
5. \( \{I,S\} = \) ComputeRanks(\( I,M,S \))
6. \( \) end do
   End Iteration:
7. \( \) return \( S \)

**ComputeIncSeq(\( I \))**
First Pass:
1. \( min = \infty, M = \) \(
2. \) for each element \( x_i \in I, i = \) sizeof(\( I \)), \( \ldots, 1 \)
3. \( \) if \( x_i \leq min \)
4. \( \) add it to the beginning of \( M \)
5. \( if x_i \leq min, min = x_i \)
6. \( \) end for
7. \( \) return \( M \)

**ComputeRanks(\( I, M, S \))**
Second Pass:
1. \( min = \infty \)
2. \( T = I \)
3. for each element \( x_i \in I, i = 1, \ldots, \) sizeof(\( I \))
4. \( \) if \( x_i \in M \) and \( x_i < min \)
5. \( \) remove \( x_i \) from \( T \), and append it to the end of \( S \)
6. \( if x_i \in T \)
7. \( \) if \( x_i \leq min, min = x_i \)
8. \( \) end for
9. \( I = T \)
10. \( \) return \( \{I,S\} \)
Vis-Sort 1D

I  1 3 2 4 6 5

M

S

Max = ∞
Iter = 1
Vis-Sort 1D

I  

M

S

Max = 3
Iter = 1
Vis-Sort 1D

Max = ∞
Iter = 2
Vis-Sort 1D

Max = 6
Iter = 2
Vis-Sort 1D

Max = ∞
Iter = 3
Vis-Sort 1D

Max = 6
Iter = 3
Vis-Sort 3D

3D Sort
1 \( \textbf{M} = \{ \}, \ \textbf{I} = \text{Unsorted input}, \ \textbf{S} = \{ \} \)
2 \text{while}(\textbf{I} \text{ is not empty})
3 \text{ do}
4 \hspace{1em} \textbf{First Pass:}
5 \hspace{2em} \text{Clear the depth buffer}
6 \hspace{2em} \text{for each element } \textbf{x}_i \in \textbf{I}, \ i = \text{sizeof(\textbf{I})}, \ldots, 1
7 \hspace{3em} \text{if } (\textbf{x}_i \notin \textbf{M})
8 \hspace{4em} \text{Render } \textbf{x}_i \text{ using an occlusion query}
9 \hspace{4em} \text{if } \textbf{x}_i \text{ is fully visible, add it to the beginning of } \textbf{M}
10 \hspace{4em} \text{render } \textbf{x}_i
11 \hspace{2em} \text{end for}
12 \hspace{1em} \textbf{Second Pass:}
13 \hspace{2em} \textbf{T} = \textbf{I}, \ \textbf{M}_{\text{sub}} = \{ \}, \text{ clear the depth buffer}
14 \hspace{2em} \text{for each element } \textbf{x}_i \in \textbf{I}, \ i = 1, \ldots, \text{sizeof(\textbf{I})}
15 \hspace{3em} \text{if } (\textbf{x}_i \in \textbf{M})
16 \hspace{4em} \text{Render } \textbf{x}_i \text{ using an occlusion query}
17 \hspace{4em} \text{if } \textbf{x}_i \text{ is fully visible, remove } \textbf{x}_i \text{ from } \textbf{M} \text{ and } \textbf{T}, \text{ and}
18 \hspace{5em} \text{append it to the end of } \textbf{M}_{\text{sub}}
19 \hspace{3em} \text{if } (\textbf{x}_i \in \textbf{T}) \text{ render } \textbf{x}_i
20 \hspace{3em} \text{end for}
21 \hspace{1em} \textbf{I} = \textbf{T}, \ \textbf{S} = \textbf{S} \cup \textbf{M}_{\text{sub}}
22 \hspace{1em} \text{end do}
23 \hspace{1em} \text{return } \textbf{S}
Cycles

- If the size of I does not change after a few iterations, there is a cycle
- Solution: split triangles
Multi-Stage Ordering

- Sort objects (simplicial complexes) by bounding box with Vis-Sort
- Use local visibility sort for triangles in an object
Applications

- Order-Independent Transparency
- Compute back-to-front
- Use results of previous pass as NSS of next
Applications

- N-Body Collision Culling
  - Return a list of potentially intersecting objects
  - Refine this list from multiple viewpoints
Advantages

- Simple and applicable to all 3D models
- Works on dynamic scenes and exhibits almost linear time performance when there is a high degree of coherence present between successive input sequences
- It can be easily implemented using occlusion queries on current graphics processors
Limitations

- Input objects have to be non-overlapping with sorted order
- Cycles are not handled
- Image level comparisons of object order are not exact