Order-Independent Transparency in DirectX11

Justin Hensley, Ph.D. | May 4, 2010
Senior Member Of Technical Staff
Advanced Micro Devices, Inc.
Motivation

Classical problem in computer graphics
Correct rendering of semi-transparent geometry requires sorting

- Blending is an order dependent operation
- Back-to-Front: For src-alpha blending
- Front-to-Back: For dst-alpha blending
Example

With Sorting

Arm appears in front of body

No Sorting

Skeleton hidden
Sorting Issues

Sometimes sorting triangles is enough but not always

- **Difficult to sort:** Multiple meshes interacting (many draw calls)
- **Impossible to sort:** Intersecting triangles (must sort fragments)

Try doing this in PowerPoint!
Before DirectX® 11

Until now, only real solution has been “depth peeling”
- Expensive (requires many passes over the scene)

Other software methods require concurrent read/write to the same buffer

New hardware
- A-buffer
- F-buffer
- K-buffer
OIT Using Fragment Linked Lists

Two-pass method using reverse linked list
Two UAV buffers required
Works with and without MSAA
Good performance
Correct transparency
Algorithm Overview

0. Render opaque scene objects

1. Render transparent scene objects
   a) All fragments are stored using per-pixel linked lists
   b) Store fragment’s: color, alpha, & depth

2. Screen quad resolves and composites fragment lists
   a) Pixel shader sorts associated linked list
   b) Composite fragments in sorted order with background
   c) Output final fragment
**Fragment and Link Buffer**

The “Fragment & Link” buffer contains data and links for all visible transparent fragments. Must be large enough to accommodate the maximum transparent overdraw allowed.

**RWStructuredBuffer UAV**

```csharp
struct Fragment_And_Link_Buffer_STRUCT
{
    uint uPixelColor;  // Fragment data
    float fDepth;      // Fragment depth
    uint uNext;        // Link to next fragment
};
```

UAV counter initialized to zero.
Start Offset Buffer

The “Start Offset” buffer contains the offset of the last fragment written for every pixel position

- Think of this as the “head” buffer pointing to the start of the linked list

Screen-sized (width * height * UINT32)

Initialized to magic value (e.g. -1)

- Magic value indicates no more fragments are stored

RWByteAddressBuffer UAV
Step 0 – Render Opaque

Render all opaque geometry normally
Step 1 – Create Linked List (1)

Render transparent geometry via a pixel shader
Color writes are disabled
UAV buffers set as input/output

\texttt{[earlydepthstencil]} is used to ensure only visible fragments are stored in linked list

- Depth testing is performed before the PS thus no fragment will be stored if depth test for this fragment fails
Step 1 – Create Linked List (2)

For every pixel:

- Calculate pixel data (color, depth etc.)
- Retrieve current pixel count from Fragment & Link UAV
  
  ```
  uint uPixelCount = FragmentAndLinkBuffer.IncrementCounter();
  ```

- Swap offsets in Start Offset UAV
  
  ```
  uint uOldStartOffset;
  StartOffsetBuffer.InterlockedExchange(
    PixelScreenSpacePositionLinearAddress, 
    uPixelCount, uOldStartOffset);
  ```

- Add new entry to Fragment & Link UAV
  
  ```
  Fragment_And_Link_Buffer_STRUCT Element;
  Element.uPixelColor = uPixelColor;
  Element.fDepth = fPixelDepth;
  Element.uNext = uOldStartOffset;
  FragmentAndLinkBuffer[uPixelCount] = Element;
  ```
Step 1 – Create Linked List (3a)

Render Target

Start Offset Buffer

Fragment and Link Buffer

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | ...
|---|---|---|---|---|---|---|---
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
Step 1 – Create Linked List (3b)

Render Target

Start Offset Buffer

Fragment and Link Buffer

0.87
-1
Step 1 – Create Linked List (3c)

Render Target

Culled due to existing scene geometry depth.

Start Offset Buffer

Fragment and Link Buffer

Culled due to existing scene geometry depth.
Step 1 – Create Linked List (3d)

Render Target

Start Offset Buffer

Fragment and Link Buffer

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | ...
|---|---|---|---|---|---|---|---
| 0.87 | 0.89 | 0.90 | 0.65 | 0.65 |     |     |     
| -1 | -1 | -1 | 0 | -1 |     |     |     

16 | Order-Independent Trans
Step 1 – Create Linked List (3d)

Start Offset Buffer

Fragment and Link Buffer

Render Target
Step 2 – Render Fragments (1)

Render a fullscreen triangle/quad via a PS

For each pixel:

- Fetch offset corresponding to current pixel location from “Start Offset” buffer
- While offset is not -1:
  - Fetch fragment from Fragment & Link buffer at current offset
  - Store fragment in temporary array
  - Retrieve new offset from fragment link
- Sort fragment temporary array back to front
- Perform “manual” blending of fragments in the pixel shader

Optimization: use depthstencil test to only fetch fragments for locations with at least one fragment
### Step 2 – Render Fragments (2a)

<table>
<thead>
<tr>
<th>Start Offset Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1 -1 -1 -1 -1 -1</td>
</tr>
<tr>
<td>-1  5  4 -1 -1 -1</td>
</tr>
<tr>
<td>-1 -1 -1 -1 -1 -1</td>
</tr>
<tr>
<td>-1 -1 -1 -1 1 -1</td>
</tr>
<tr>
<td>-1 -1 -1 -1 -1 -1</td>
</tr>
</tbody>
</table>

**Render Target**

(0,0)–>(1,1):
- Fetch Start Offset: -1
- -1 indicates no fragment to render
Step 2 – Render Fragments (2a)

(1,1):
Fetch Start Offset: 5
Fetch fragment in F&L(5)
Walk the list and store in temporary array
Step 2 – Render Fragments (2a)

(1,1):
Sort temporary array
Blend colors and write out

Render Target

Start Offset Buffer

Fragment and Link Buffer
Step 2 – Render Fragments (2a)

Start Offset Buffer

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>5</td>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Fragment and Link Buffer

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.87</td>
<td>0.89</td>
<td>0.90</td>
<td>0.65</td>
<td>0.65</td>
<td>0.71</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Demo
**MSAA Support**

Minor changes to the algorithm

**Step 1** - Sample coverage stored in fragment data (SV_COVERAGE)

**Step 2**

- Executed at *sample frequency* (declaring SV_SAMPLEINDEX)
- Store only covered fragments in temporary buffer

```c
if (Element.uCoverage & (1<<input.uSample) )
{
    // Store fragment into temporary array for later sorting
    // ...
}
```

**Pros:**

- Can use depthstencil for early rejection for pixels that don’t contain any fragments
- Destination Render Target stays multisampled
MSAA Support (2)

Step 2 (alternative method)
- Executed at *pixel frequency*
- Store fragments into temporary buffer
- Perform per-sample sorting of fragments
- Manual blending of samples
- Average (resolve) blended samples

Pros:
- Slightly faster than per-sample execution
- Can be done with a Compute Shader

Cons:
- Destination Render Target is single sample
- Depthstencil testing is not available for early rejection
UAV Counter Alternative

Create a 1x1 UAV, this “Global Counter” is used to allocate links in our linked lists. Initialized to 0.

Instead of IncrementCounter(), get the current value of Global Counter and increment it using InterlockedAdd().

All threads will be fighting for atomic access in global memory

- Use a NxN buffer with pixel sub-counters to improve performance

The global counter method is ~30% slower than using the UAV counter
Linked List Method Limitations

“Start Offset” UAV buffer cannot be Texture2D
  - Atomic operations require linear address
Compute Shader cannot write to MSAA RTs
  - Would allow CS to be used for rendering phase
Fragment sorting requires fixed-size array
  - Imposes a “max overdraw” limit to the algorithm
  - Should not be a problem if scene is known or do a “resize”

On-the-fly sorting at fragment storing phase runs into shader compiler limitations (breaking loops on UAV fetch result)
Future Work

Other potential applications

- Programmable blend
- Motion Blur
- Shadows

More complex data structures
For More Info

http://developer.amd.com

Demos:

HPG:

HIGH-PERFORMANCE GRAPHICS
SAARBRUCKEN, GERMANY  JUNE 25-27, 2010
Trademark Attribution

AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names used in this presentation are for identification purposes only and may be trademarks of their respective owners.

©2008 Advanced Micro Devices, Inc. All rights reserved.