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

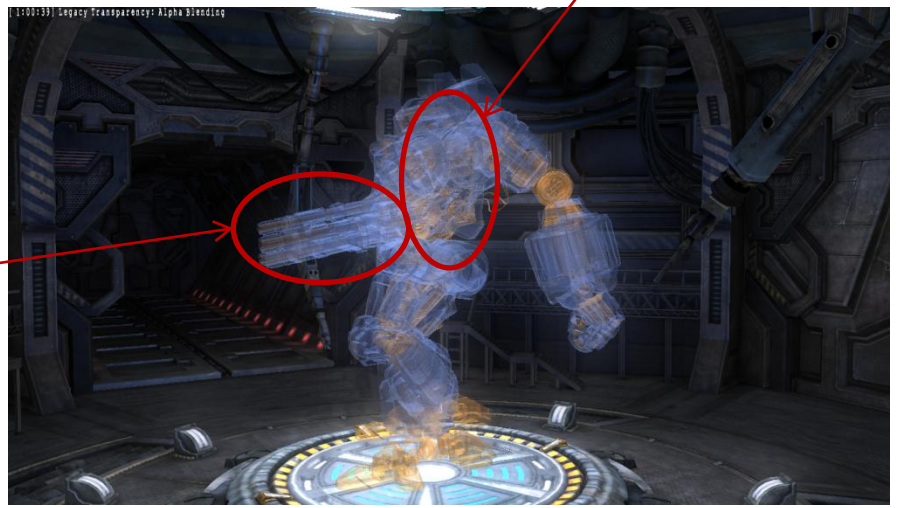
- Blending is an order dependent operation
- Back-to-Front: For src-alpha blending
- Front-to-Back: For dst-alpha blending

Example

With Sorting

Skeleton hidden

No Sorting

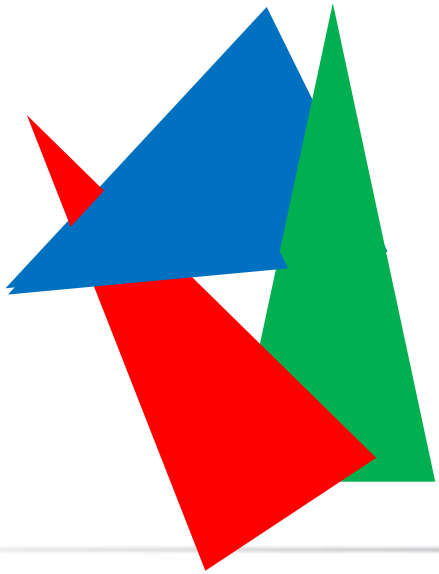


Arm appears in front of body

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)
- Liu, et al. *Efficient Depth Peeling via Bucket Sort*. HPG 2009.

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

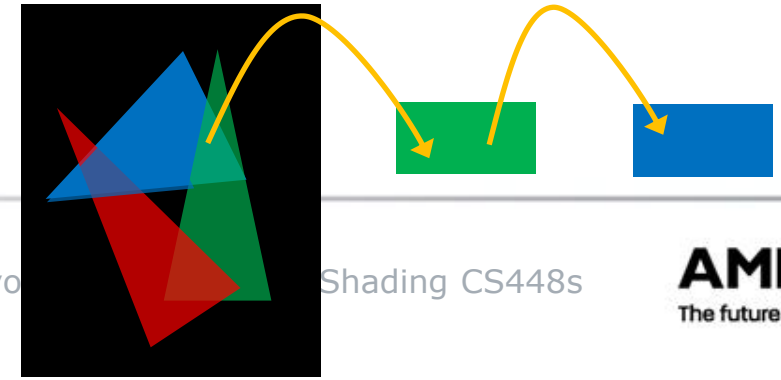
Works with and without MSAA

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

```
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)

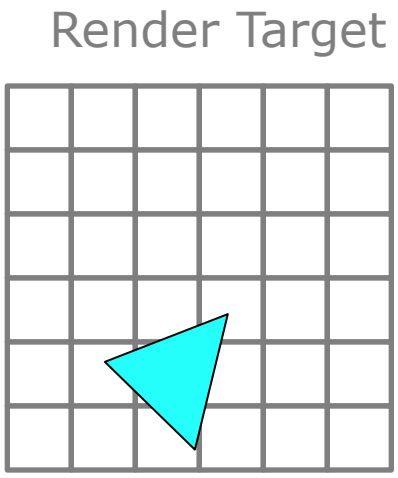
Initialized to magic value (e.g. -1)

- 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

`[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

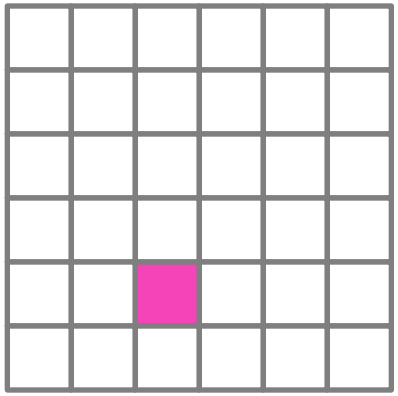
```
Fragment_And_Link_Buffer_STRUCT Element;

Element.uPixelColor                    = uPixelcolor;

Element.fDepth                         = fPixelDepth;

Element.uNext                          = uOldStartOffset;

FragmentAndLinkBuffer[uPixelCount] = Element;
```

Step 1 – Create Linked List (3a)

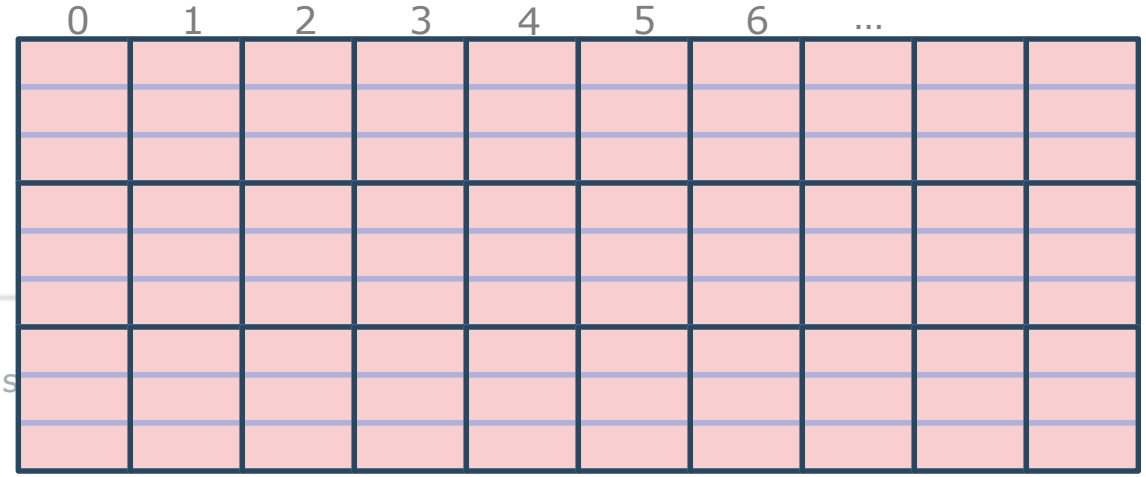
Start Offset Buffer

-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

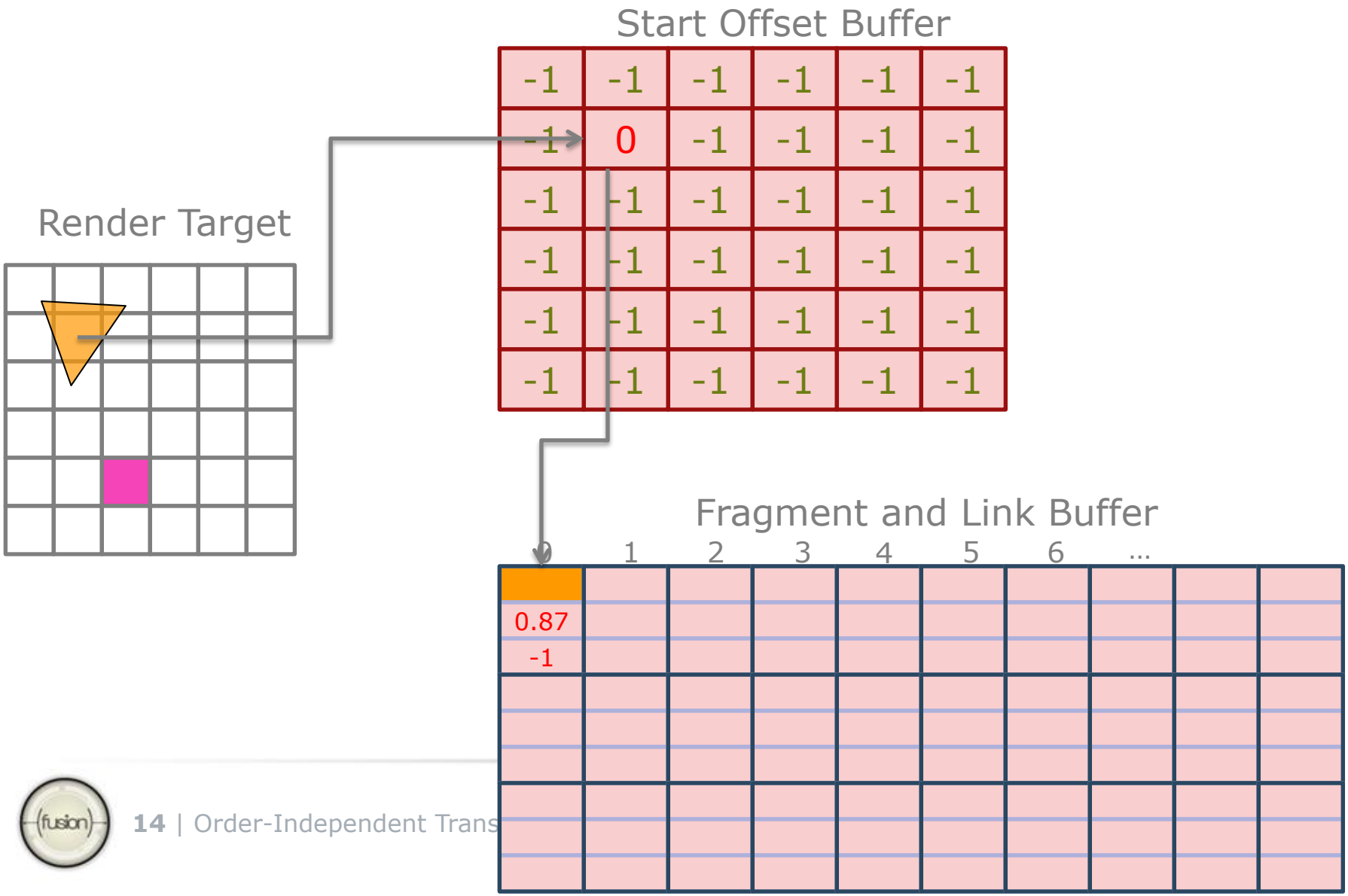
Render Target



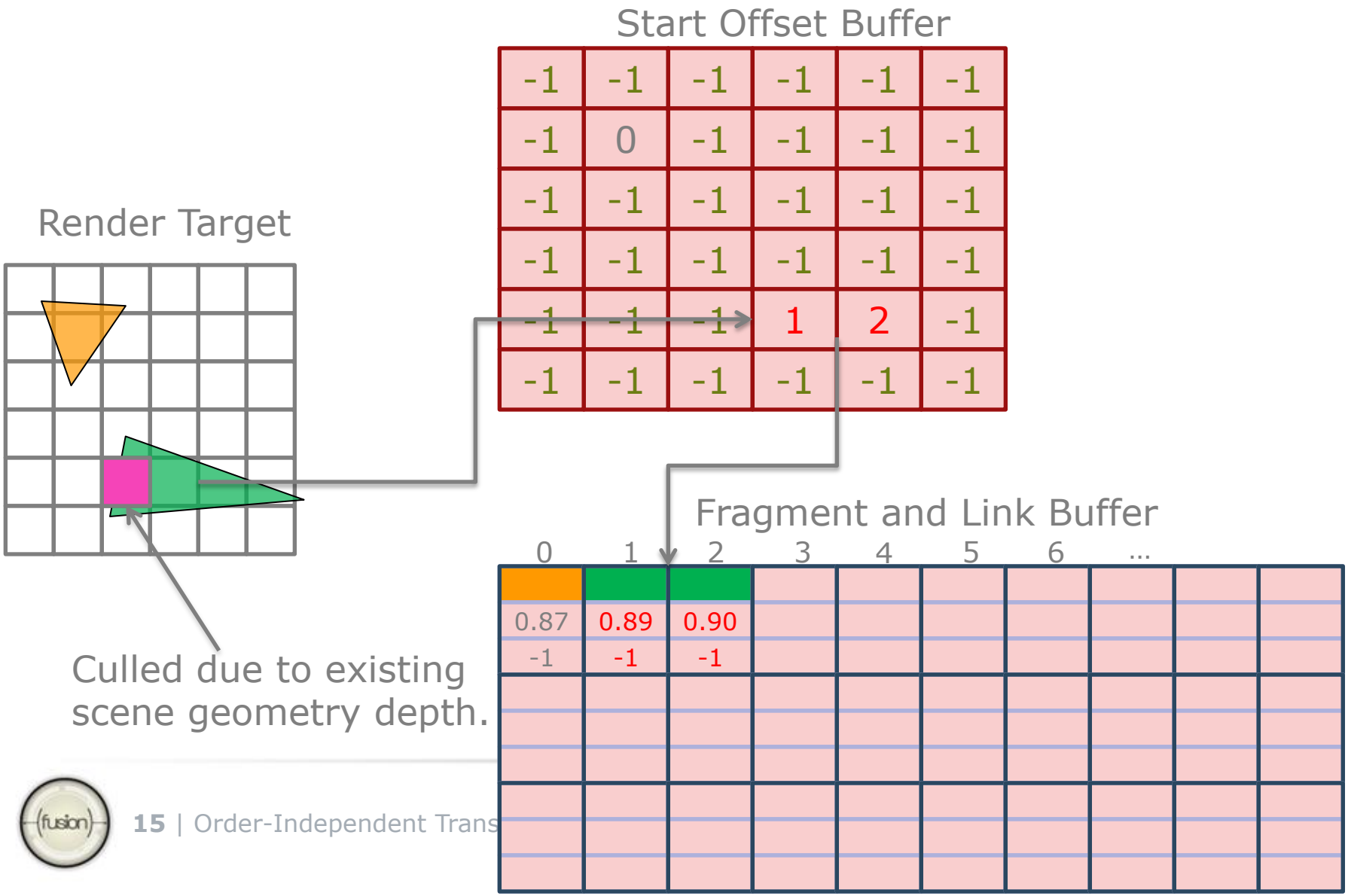
Fragment and Link Buffer



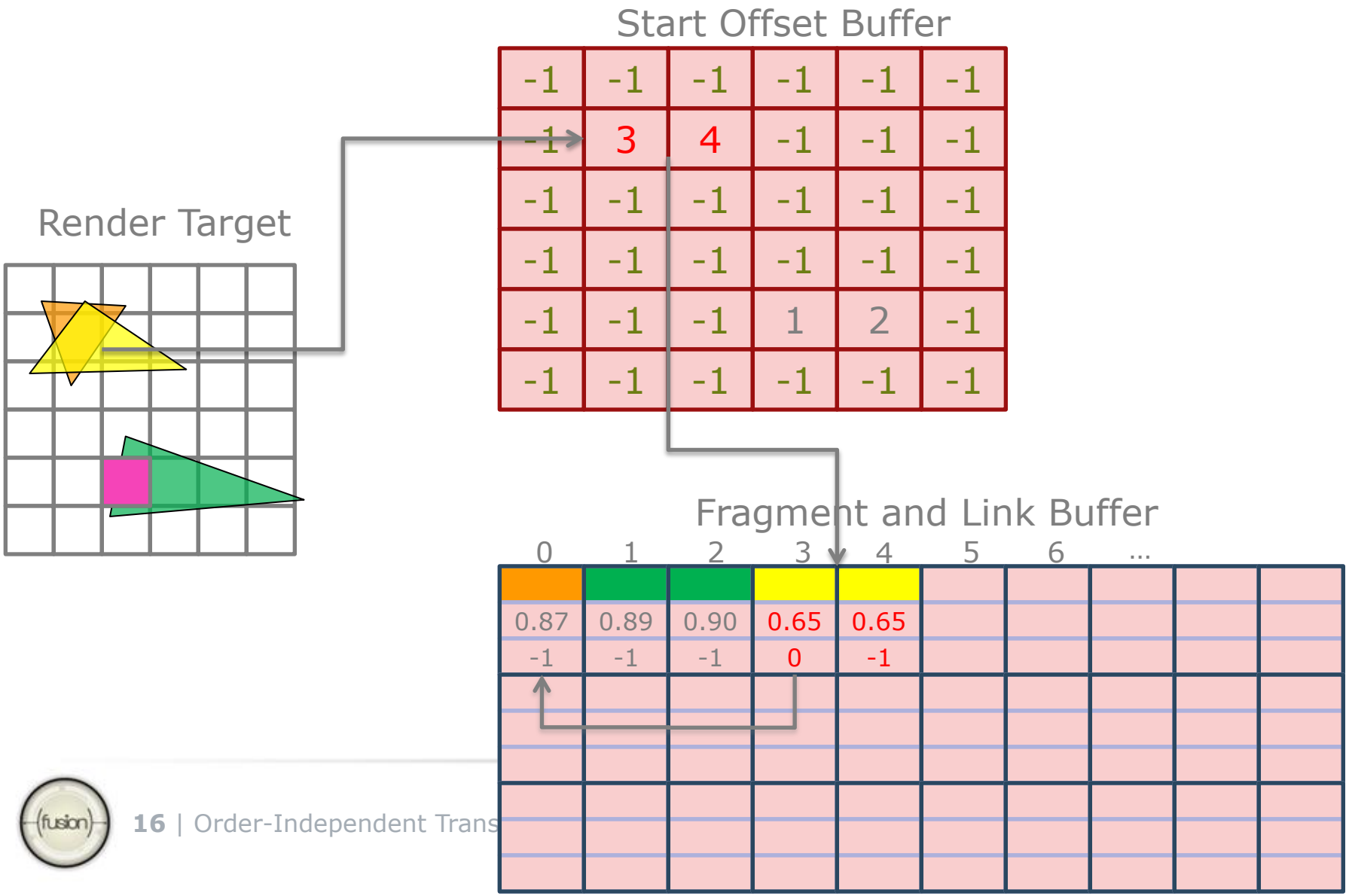
Step 1 – Create Linked List (3b)



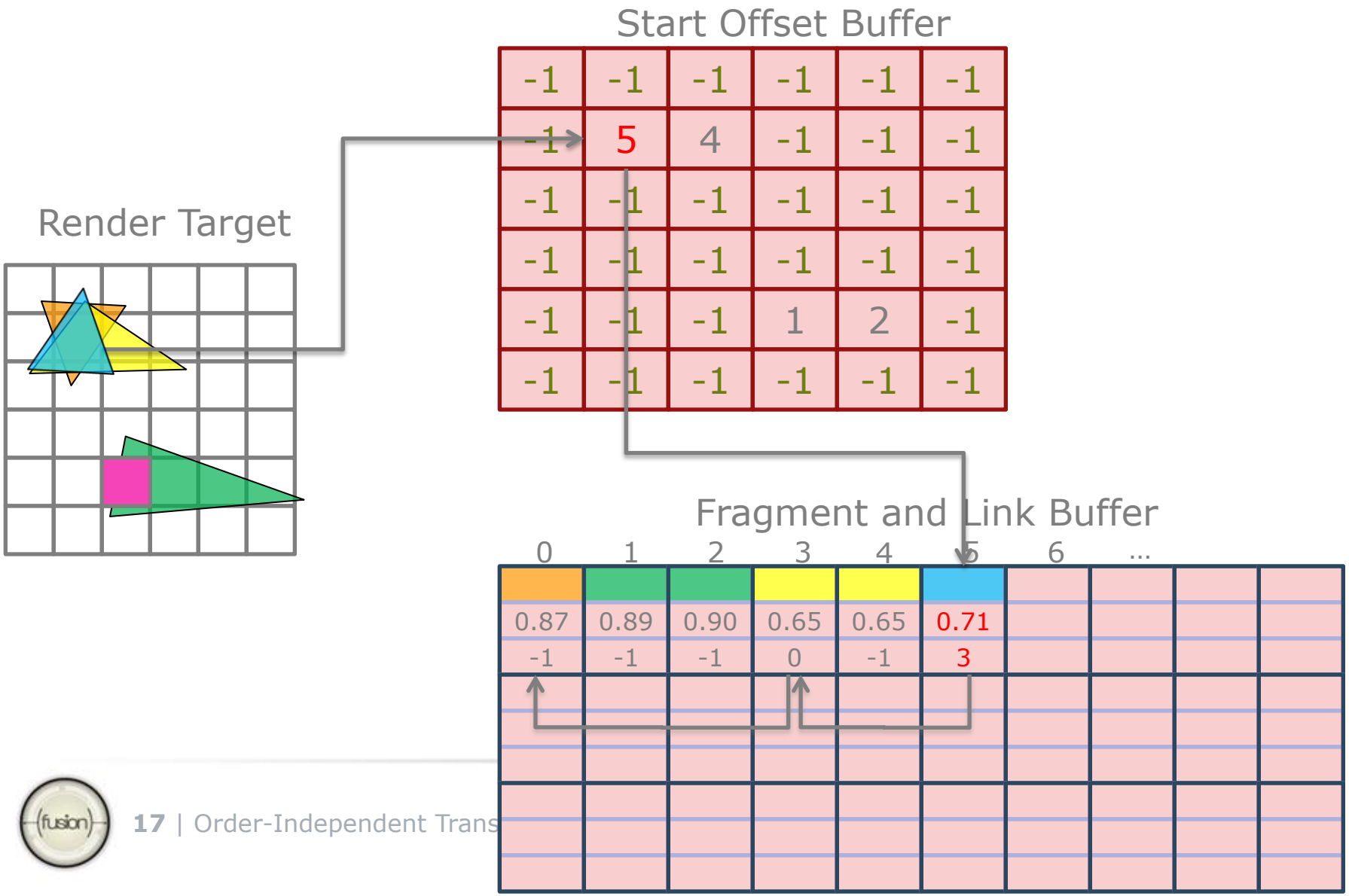
Step 1 – Create Linked List (3c)



Step 1 – Create Linked List (3d)



Step 1 – Create Linked List (3d)



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

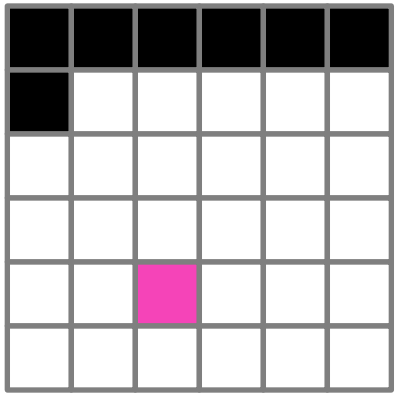
Optimization: use depthstencil test to only fetch fragments for locations with at least one fragment

Step 2 – Render Fragments (2a)

Start Offset Buffer

-1	-1	-1	-1	-1	-1
-1	5	4	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	1	2	-1
-1	-1	-1	-1	-1	-1

Render Target

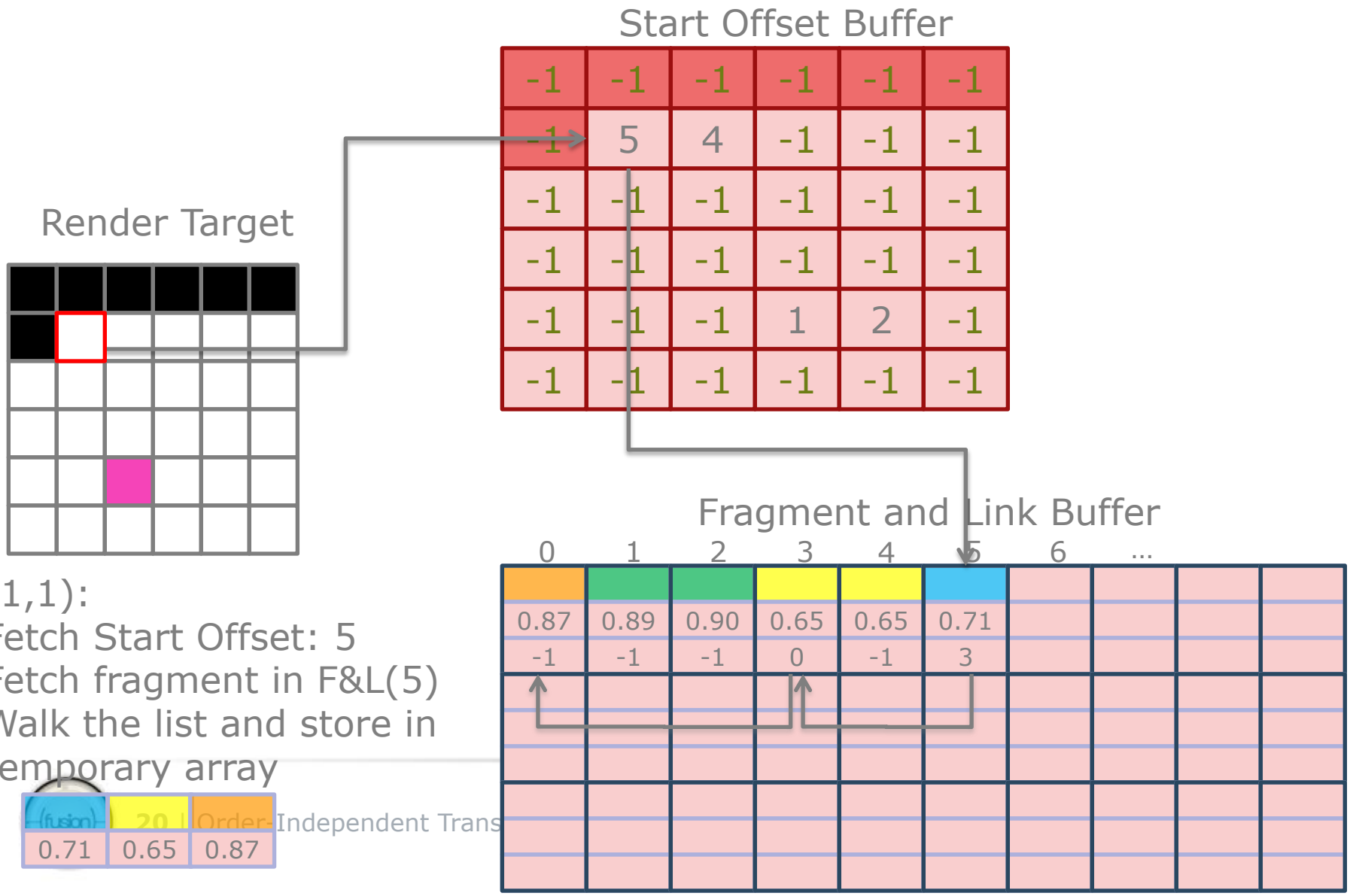


(0,0)->(1,1):
Fetch Start Offset: -1
-1 indicates no fragment to render

Fragment and Link Buffer

0	1	2	3	4	5	6	…
0.87	0.89	0.90	0.65	0.65	0.71		
-1	-1	-1	0	-1	3		

Step 2 – Render Fragments (2a)

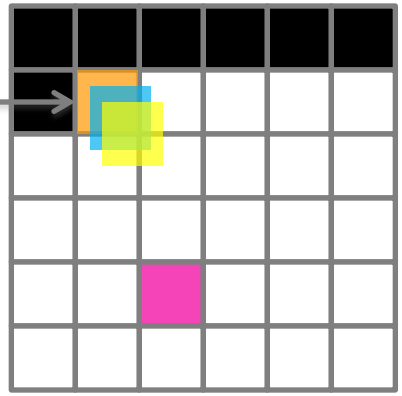


Step 2 – Render Fragments (2a)

Start Offset Buffer

-1	-1	-1	-1	-1	-1
-1	5	4	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	1	2	-1
-1	-1	-1	-1	-1	-1

Render Target



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

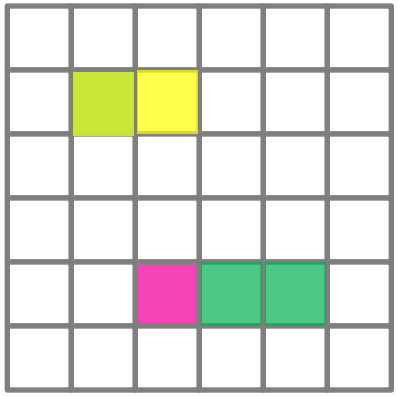
Fragment and Link Buffer

0	1	2	3	4	5	6	…
0.87	0.89	0.90	0.65	0.65	0.71		
-1	-1	-1	0	-1	3		

(fusion)	21	Order-Independent Trans
0.65	0.71	0.87

Step 2 – Render Fragments (2a)

Render Target



Start Offset Buffer

-1	-1	-1	-1	-1	-1
-1	5	4	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	1	2	-1
-1	-1	-1	-1	-1	-1

Fragment and Link Buffer

0	1	2	3	4	5	6	…
0.87	0.89	0.90	0.65	0.65	0.71		
-1	-1	-1	0	-1	3		

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

```
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:
http://developer.amd.com/samples/demos/pages/ATIRadeonHD5800SeriesRealTimeDemos.aspx

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.