SIGGRAPH2010

The People Behind the Pixels

Real-Time Order Independent Transparency and Indirect Illumination Using Direct3D 11

Jason Yang and Jay McKee

AMD
The future is fusion

…Continued from Last Year
Depth of Field using Summed Area Tables

Today's Overview

- Fast creation of linked lists of arbitrary size on the GPU using D3D11
- Integration into the standard graphics pipeline
  - Demonstrates compute from rasterized data
  - DirectCompute features in Pixel Shader
- Examples:
  - Order Independent Transparency (OIT)
  - Indirect Shadowing

Building data structures from the graphics pipeline

Background

- A-buffer – Carpenter '84
  - CPU side linked list per-pixel for anti-aliasing
- Fixed array per-pixel
  - F-buffer, stencil routed A-buffer, Z^3 buffer, and k-buffer, Slice map, bucket depth peeling
- Multi-pass
  - Depth peeling methods for transparency
- Recent
  - Freepipe, PreCalc [DX11 SDK]

Linked List Construction

- Two Buffers
  - Head pointer buffer
    - addresses/offsets
    - Initialized to end-of-list (EOL) value (e.g., -1)
  - Node buffer
    - arbitrary payload data + “next pointer”
- Each shader thread
  1. Retrieve and increment global counter value
  2. Atomic exchange into head pointer buffer
  3. Add new entry into the node buffer at location from step 1

Creating reverse linked list

Order Independent Transparency

Construction by Example

- Classical problem in computer graphics
- Correct rendering of semi-transparent geometry requires sorting – blending is an order dependent operation
- 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!

Order Independent Transparency with Per-Pixel Linked Lists

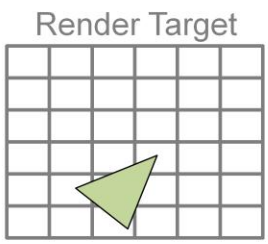
- Computes correct transparency
- Good performance
- Works with depth and stencil testing
- Works with and without MSAA
- Example of programmable blend

Algorithm Overview

0. Render opaque scene objects
1. Render transparent scene objects
2. Screen quad resolves and composites fragment lists

Step 0 – Render Opaque

- Render all opaque geometry normally



Algorithm Overview

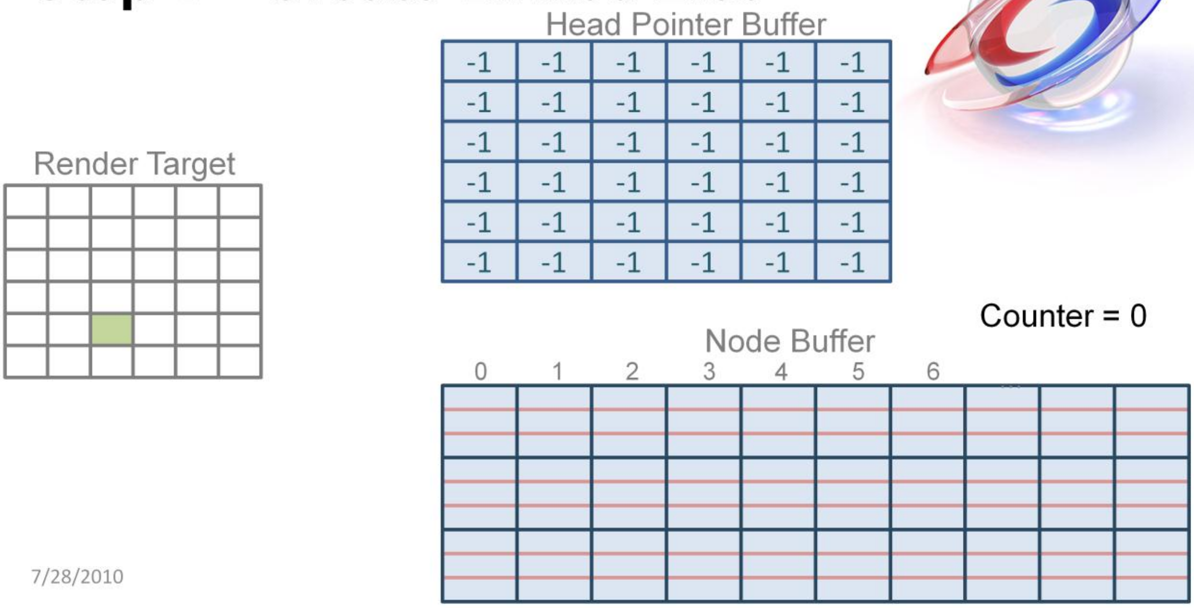
0. Render opaque scene objects
1. Render transparent scene objects
   - All fragments are stored using per-pixel linked lists
   - Store fragment's: color, alpha, & depth
2. Screen quad resolves and composites fragment lists

Setup

- Two buffers
  - Screen sized head pointer buffer
  - Node buffer – large enough to handle all fragments

- Render as usual
- Disable render target writes

Step 1 – Create Linked List



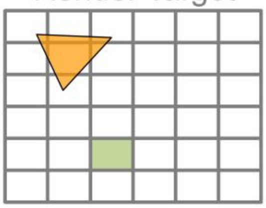
7/28/2010

Step 1 – Create Linked List

Head Pointer Buffer

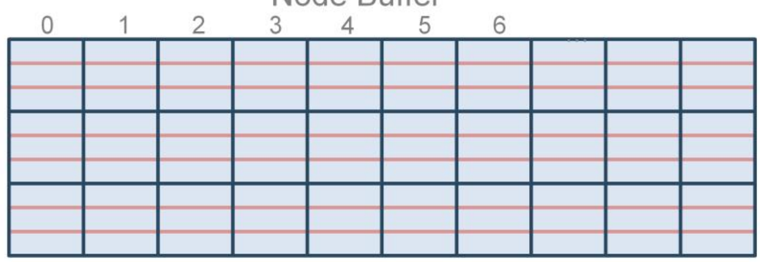
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1

Render Target

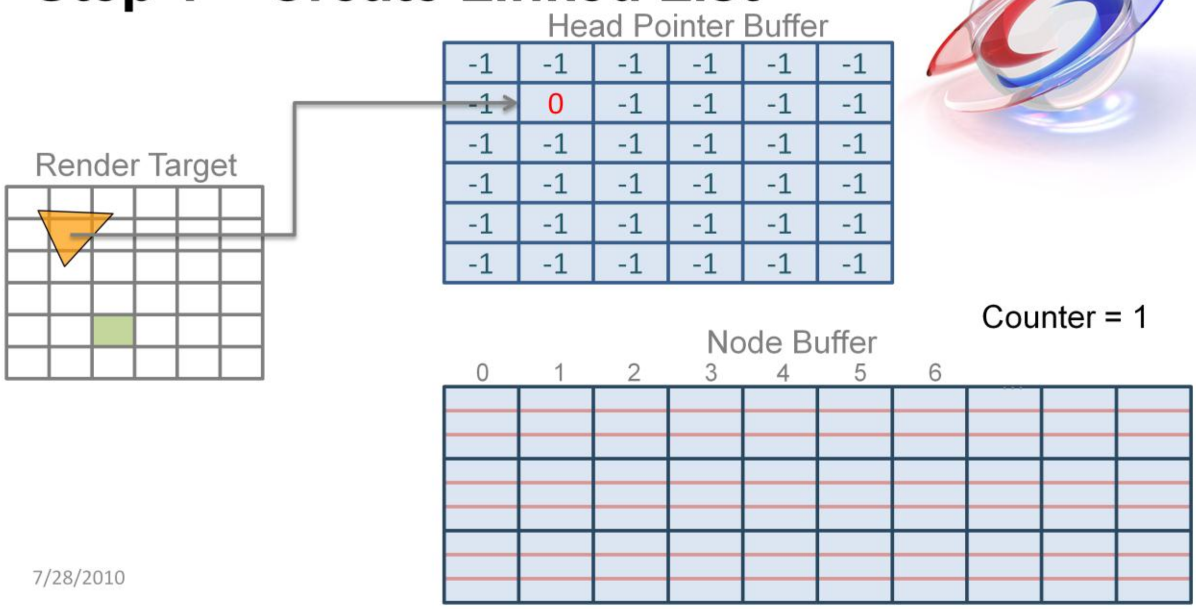


Counter = 0

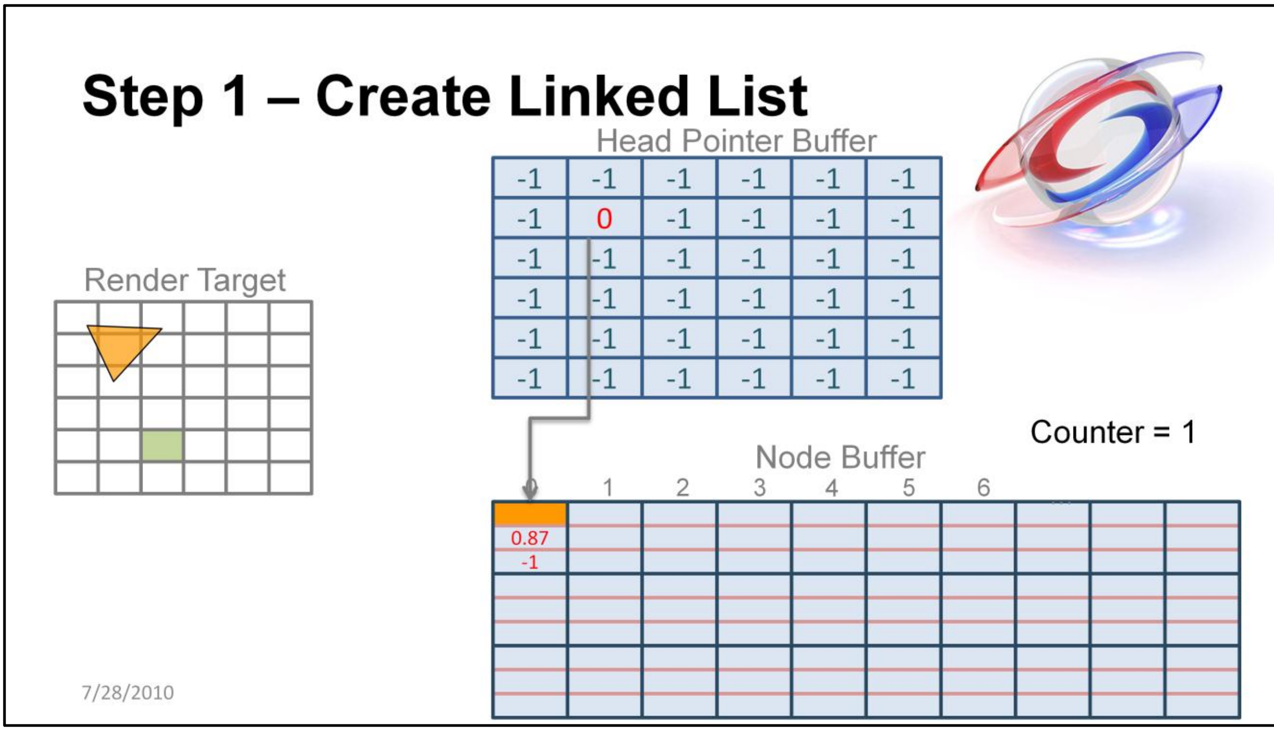
Node Buffer



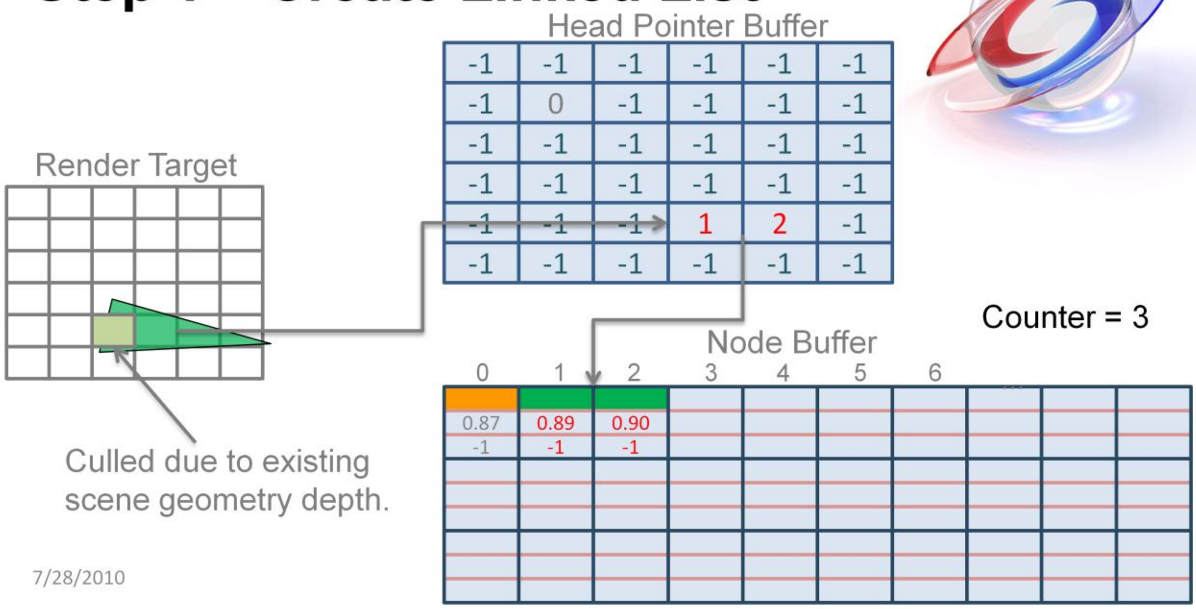
Step 1 – Create Linked List



7/28/2010

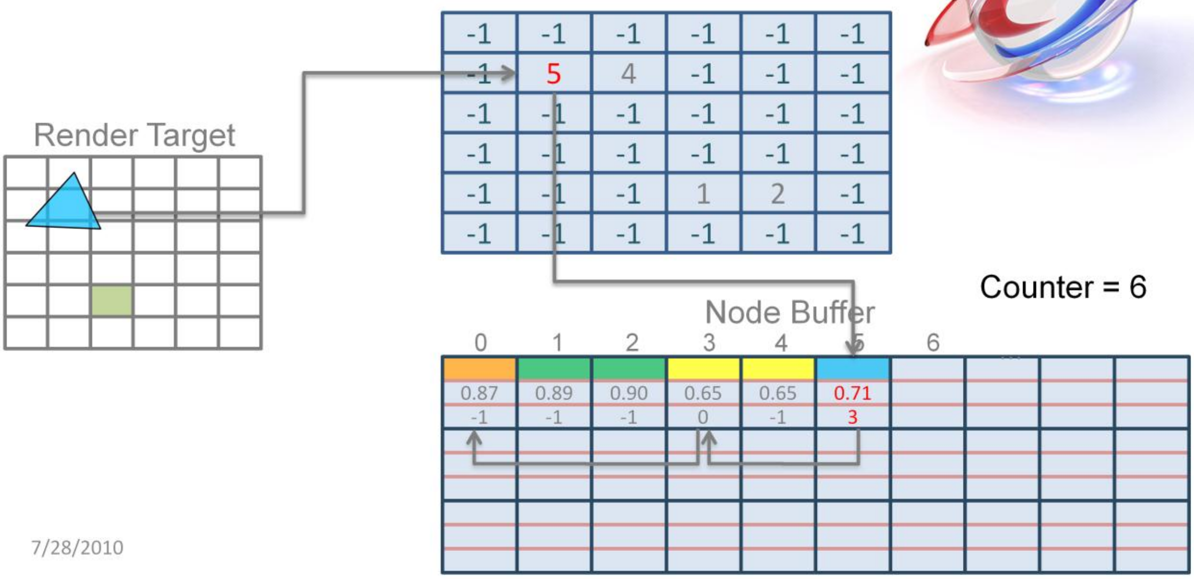

Step 1 – Create Linked List
Head Pointer Buffer
-1 -1 -1 -1 -1 -1
-1 0 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1
Render Target
Counter = 1
Node Buffer
0 1 2 3 4 5 6
0.87
-1
7/28/2010

Step 1 – Create Linked List



Step 1 – Create Linked List

Step 1 – Create Linked List



Node Buffer Counter

- Counter allocated in GPU memory (i.e. a buffer)
  - Atomic updates
  - Contention issues
- DX11 Append feature
  - Linear writes to a buffer
  - Implicit writes
    - Append()
  - Explicit writes
    - IncrementCounter()
    - Standard memory operations
  - Up to 60% faster than memory counters

Algorithm Overview

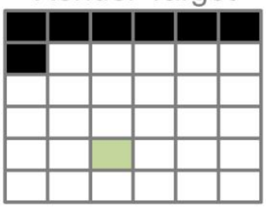
0. Render opaque scene objects
1. Render transparent scene objects
2. Screen quad resolves and composites fragment lists
   - Single pass
   - Pixel shader sorts associated linked list (e.g., insertion sort)
   - Composite fragments in sorted order with background
   - Output final fragment

Step 2 – Render Fragments

Head Pointer 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



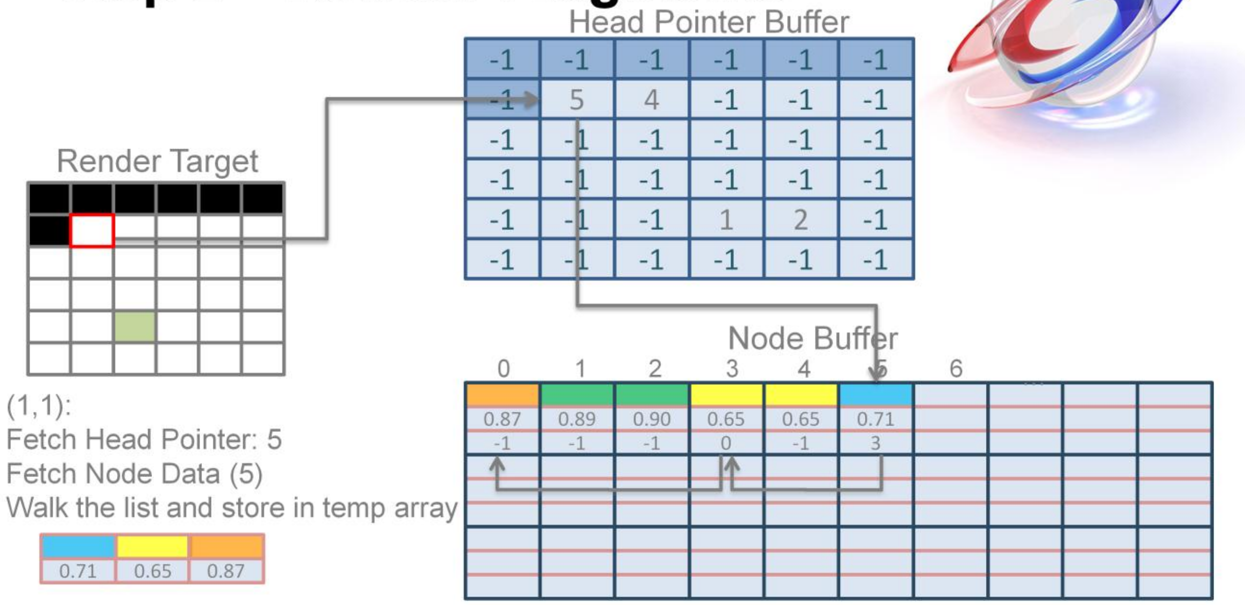
(0,0)->(1,1):
Fetch Head Pointer: -1
-1 indicates no fragment to render

Node 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				

7/28/2010

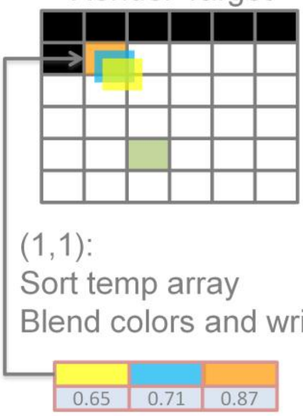
Step 2 – Render Fragments



Step 2 – Render Fragments

-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 temp array
Blend colors and write out

Node 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				

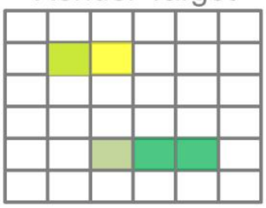
Insertion sort

Step 2 – Render Fragments

Head Pointer Buffer

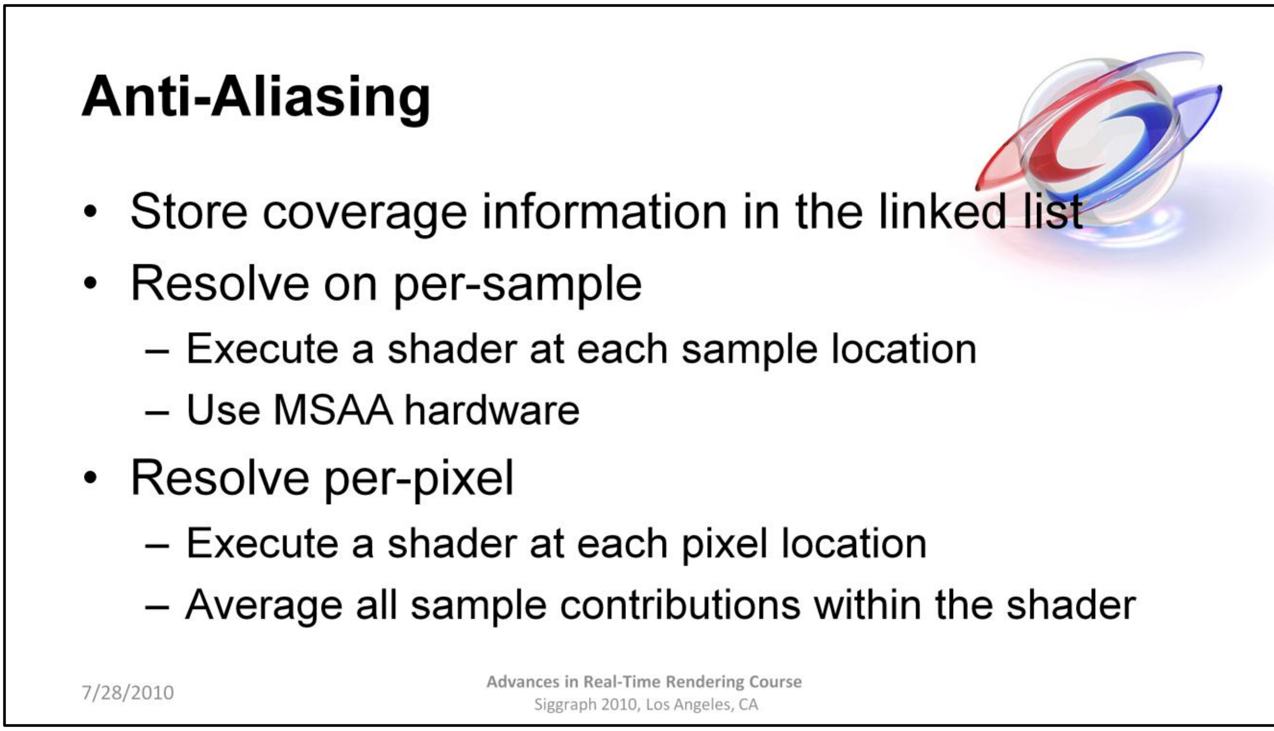
-1	-1	-1	-1	-1	-1
-1	5	4	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	1	2	-1
-1	-1	-1	-1	-1	-1

Render Target



Node 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				



Sub-pixel intersections

Pros:

Slightly faster than per-sample execution

Can be done with a Compute Shader

Cons:

Destination Render Target is single sample

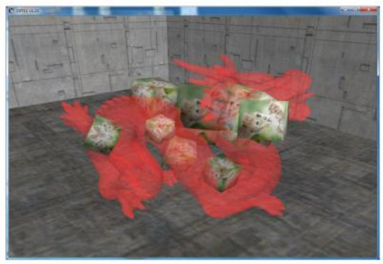
Depthstencil testing is not available for

early rejection

Performance Comparison

	Teapot	Dragon
Linked List	743 fps	338 fps
Precalc	285 fps	143 fps
Depth Peeling	579 fps	45 fps
Bucket Depth Peeling	---	256 fps
Dual Depth Peeling	---	94 fps

Performance scaled to ATI Radeon HD 5770



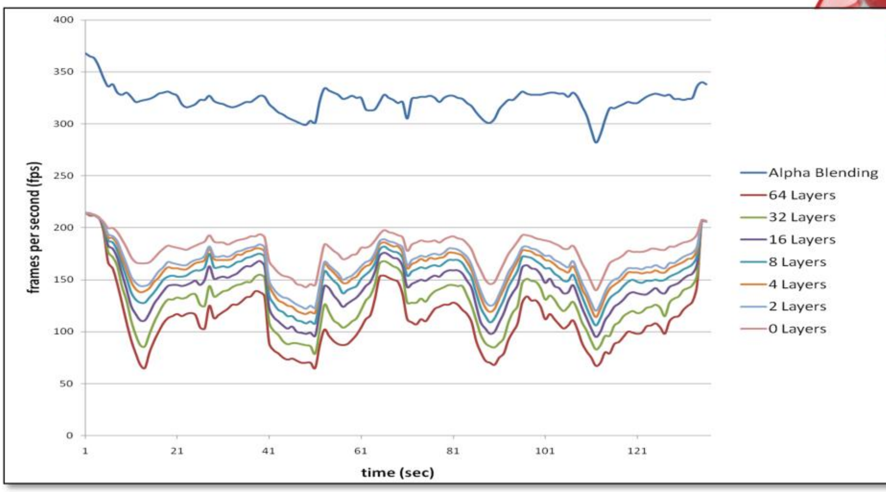
Mecha Demo

- 602K scene triangles
  - 254K transparent triangles

7/28/2010

Advances in Real-Time Rendering Course
Siggraph 2010, Los Angeles, CA

Layers

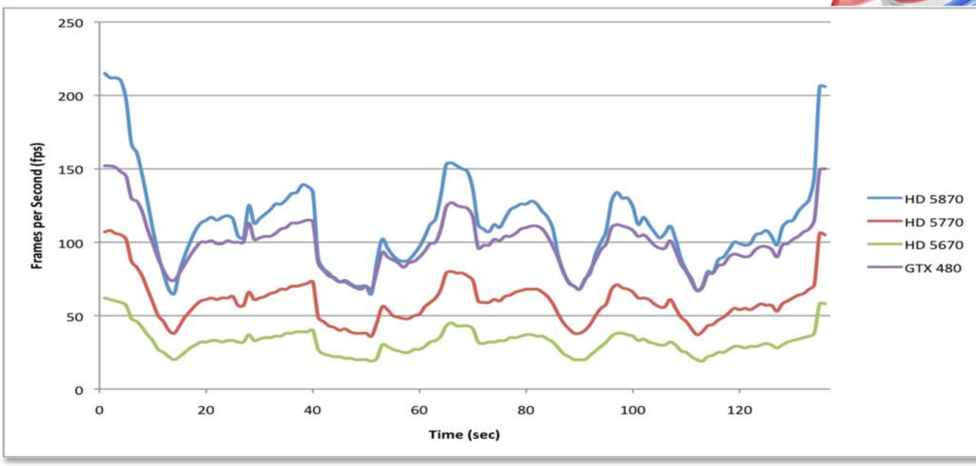


Worst case 370K fragments filling 40% of the frame

2ms to store the fragments

3.3ms 0->64 fps

Scaling



112 -> 60 fps -> 32fps

Indirect Illumination with Indirect Shadows using DirectX 11

Why Indirect Shadowing?

- Help perceive subtle dynamic changes occuring in a scene.
- Adds helpful cues for depth perception.
- Indirect light contribution on scene pixels more accurate.
- Especially important for visual experience and gameplay when environments are dimmly lit or action happens away from direct light.

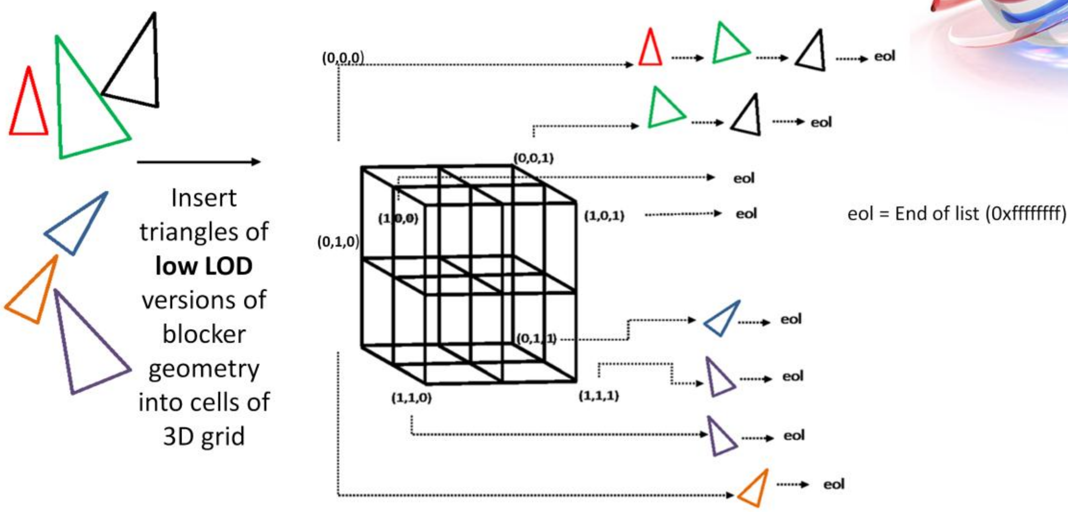
4 Phases:

1) Create 3D grid holding blocker geometry for indirect shadowing. (*use DX11 Compute Shader*)
2) Generate Reflective Shadow Maps (RSMs).
3) Indirect Light
4) Indirect Shadowing

PHASE #1

Create 3D grid containing blocker geometry for shadowing.

Create 3D grid for shadow blocker geometry



PHASE #2

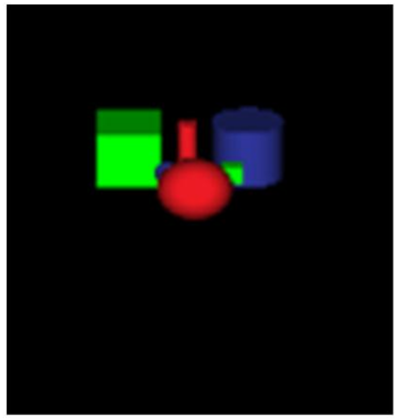
Generate RSMs

Reflective Shadow Map

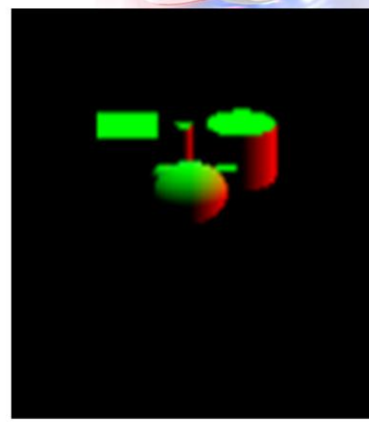
- RSM is like a standard shadow map but with added information such as color, normal, flux, etc.
- Pixels in RSM considered as point light sources for 1 bounce indirect light.
- Create 1 RSM for each light source you want to contribute indirect light.

RSM ~ G-Buffer for lights

Position



Color



Normal

PHASE #3

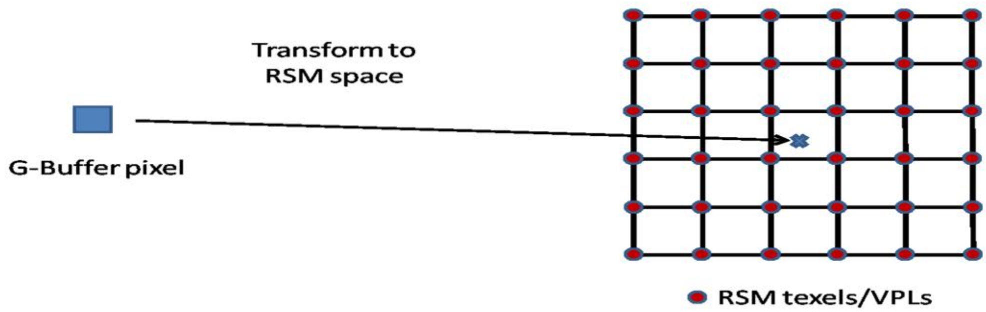
Indirect Light

Indirect Light

- At this point, assumed you have:
  - Main scene G-buffer with color, position, normal
  - Generated RSMs with color, position, normal
- Separate indirect light and indirect shadow phases so you can use different buffer sizes based on performance needs.
- In this example both phases use 1/4 size buffer.

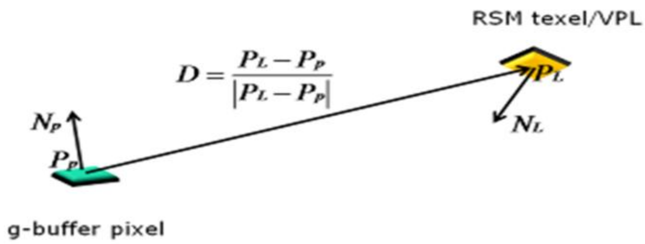
Full-screen quad. For each scene pixel:

- Transform scene pixel position and normal to RSM space



Indirect Light Accumulate

- For each scene pixel, loop through RSM kernel pixels, do standard lighting calculation between RSM kernel pixel and scene pixel and accumulate light.



$$Contribution_{VPL} = \frac{sat(N_P \cdot D) \cdot sat(N_L \cdot (-D))}{|P_L - P_P|^2} \cdot Col_{VPL} \cdot Area_{VPL}$$

Problem!

- Too many samples per kernel will kill performance…but we need very large kernel to get good visual results.
- For decent results need >= 512x512 as well as big kernel >= 80x80

Solution:

- Don't use the full kernel for each screen pixel.
- Instead, use dithered pattern of pixels which only considers 1 out of NxN pixels each time in the light accumulation loop.
- Dithered pattern position uses scene pixel screen position modulo N.

Indirect Lighting

- However, the dithered pattern used to calculate indirect light falling on screen pixel still won't be smooth…
- Perform bilateral filter with up-sample to smooth things out and go to main scene image size.

PHASE #4

Indirect Shadowing

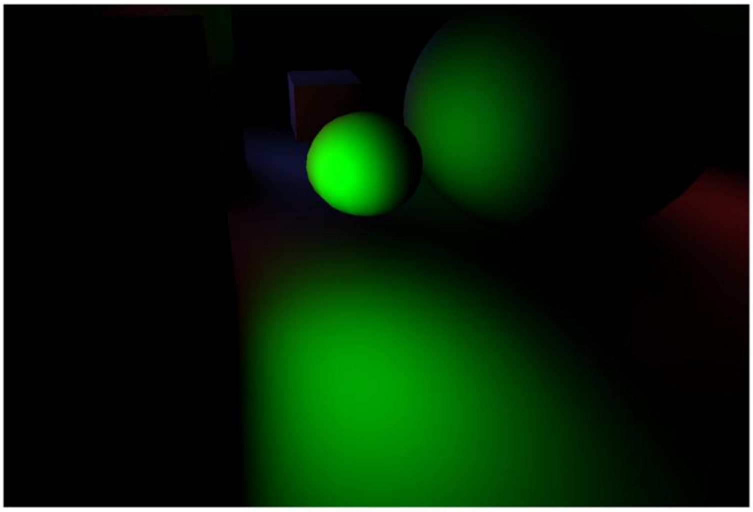
Indirect Shadowing

- Similar steps, full screen quad, transform scene pixel to RSM, but instead of lighting calculation…
- Accumulate the amount of ***blocked*** light between RSM kernel and scene pixel.

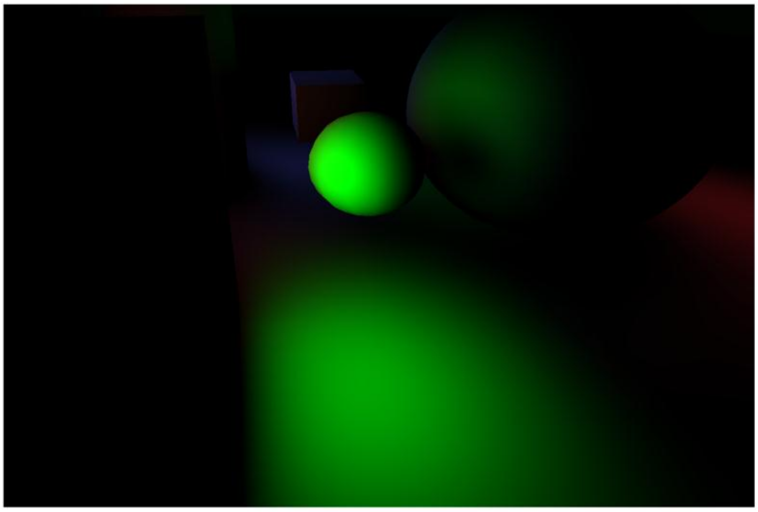
How do you estimate amount of blocked light?

- Trace N rays from scene pixel to RSM kernel pixels and check for blocking triangles from the 3D grid step.
- Accumulate indirect light from *blocked* RSM kernel pixels only!
- Apply bilateral filter and up-sample.
- SUBTRACT result from indirect light in previous step.

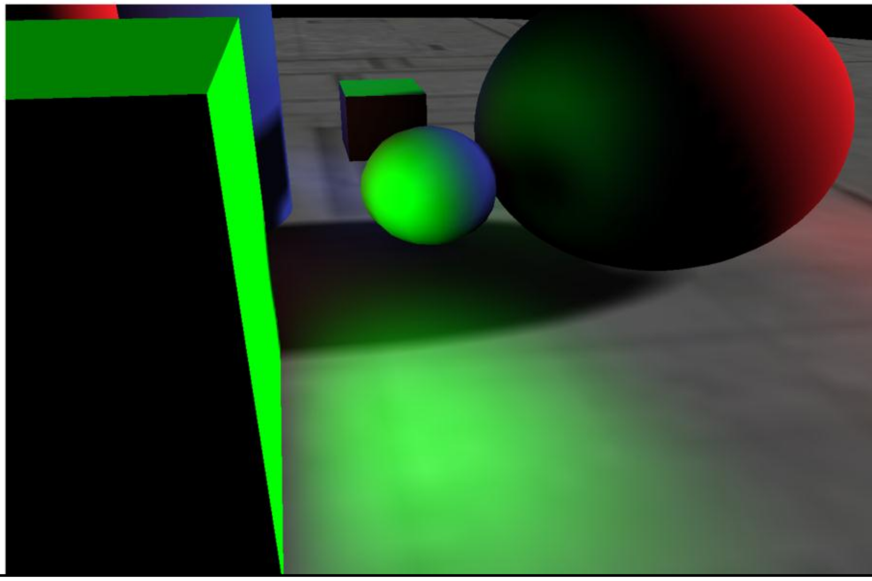
Indirect Light



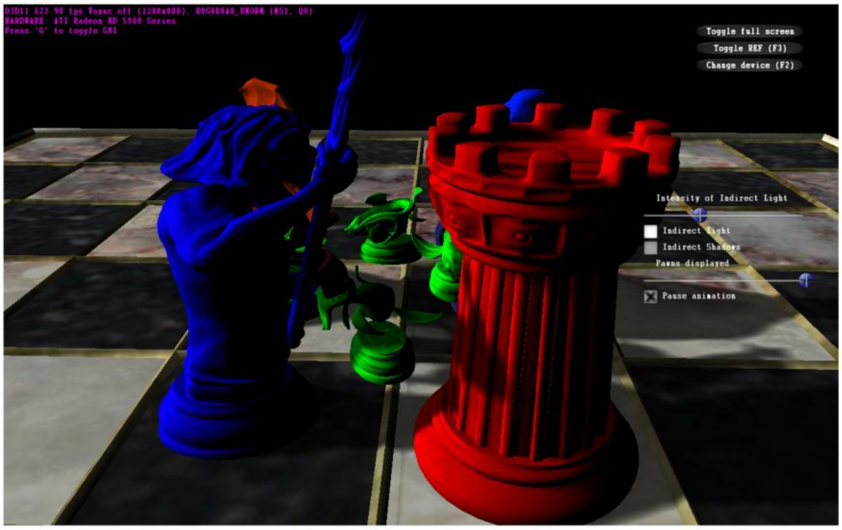
After Indirect Shadowing



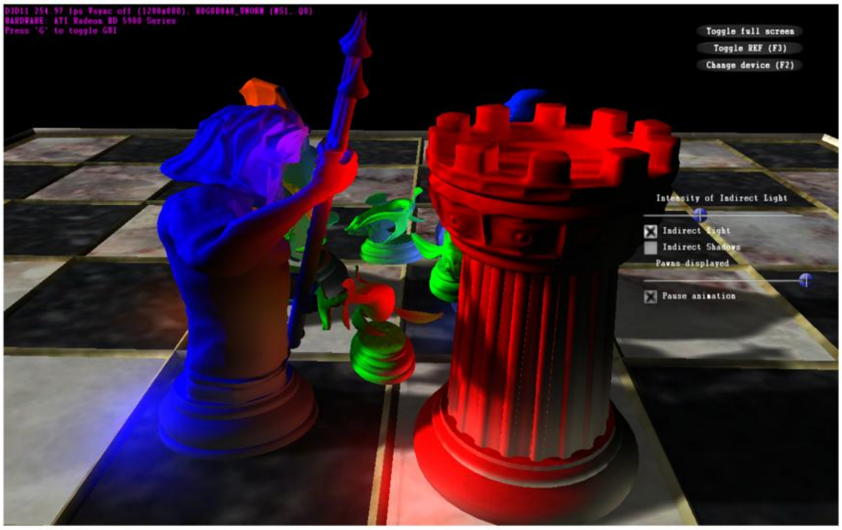
Full Scene



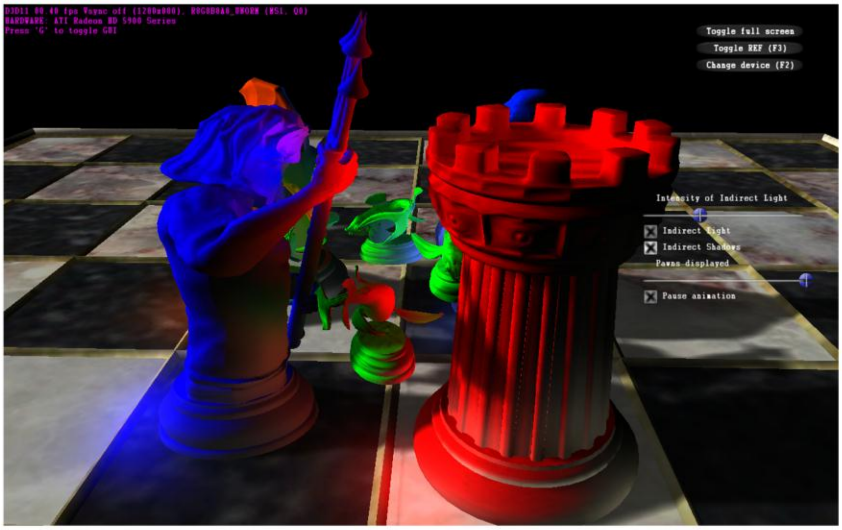
No Indirect Lighting



With Indirect Lighting



Indirect Lighting + Shadowing



Demo Time

Summary:

- Fairly simple implementation. All but the 3D grid phase is probably in your pipeline today.
- Fully dynamic. No pre-generated data required.
- Offers a “playground” to experiment with ray-casting and per-pixel data structures in DX11.
- 70-110 fps on AMD HD5970
  - 12800x800 -- 9 shadow rays per pixel
  - 32x32x32 grid. -- ~6000 blocker triangles per frame

Thanks

- Holger Grün, Nicolas Thibieroz, Justin Hensley, Abe Wiley, Dan Roeger, David Hoff, and Tom Frisinger – AMD
- Chris Oat – Rockstar New England
- Jakub Klarowicz – Techland

References

- Yang J., Hensley J., Grün H., Thibieroz N.: Real-Time Concurrent Linked List Construction on the GPU. In Rendering Techniques 2010: Eurographics Symposium on Rendering (2010), vol. 29, Eurographics.
- Grün H., Thibieroz N.: OIT and Indirect Illumination using DX11 Linked Lists. In Proceedings of Game Developers Conference 2010 (Mar. 2010). http://developer.amd.com/gpu_assets/OIT%20and%20Indirect%20Illumination%20using%20DX11%20Linked%20Lists_forweb.ppsx

Questions?

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