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

Jason Yang and Jay McKee
...Continued from Last Year
Depth of Field using Summed Area Tables

7/28/2010

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

7/28/2010

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

**Head Pointer Buffer**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<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>-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>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<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>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

**Node Buffer**

```
0
1
2
3
4
5
6
```

Render Target

Counter = 0

7/28/2010
Step 1 – Create Linked List

Render Target

Head Pointer Buffer

Counter = 0

Node Buffer
Step 1 – Create Linked List

Render Target

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
```

Node Buffer

```
0  1  2  3  4  5  6
```

Counter = 1

7/28/2010
Step 1 – Create Linked List

Head Pointer Buffer

Counter = 1

Node Buffer

7/28/2010
Step 1 – Create Linked List

Head Pointer Buffer

Node Buffer

Culled due to existing scene geometry depth.

Counter = 3
Step 1 – Create Linked List

Render Target

Node Buffer

Counter = 5
Step 1 – Create Linked List

Render Target

Counter = 6

Node Buffer

0.87 0.89 0.90 0.65 0.65 0.71
-1 -1 -1 0 -1 3

7/28/2010
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

<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>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>2</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>
</tbody>
</table>

Node Buffer

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<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>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>3</td>
</tr>
</tbody>
</table>

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

7/28/2010
Step 2 – Render Fragments

Render Target

Head Pointer Buffer

Node Buffer

(1,1):
Fetch Head Pointer: 5
Fetch Node Data (5)
Walk the list and store in temp array
Insertion sort
Step 2 – Render Fragments

Head Pointer Buffer

Node Buffer

7/28/2010
Anti-Aliasing

- Store coverage information in the linked list
- Resolve on per-sample
  - Execute a shader at each sample location
  - Use MSAA hardware
- Resolve per-pixel
  - Execute a shader at each pixel location
  - Average all sample contributions within the shader

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

<table>
<thead>
<tr>
<th></th>
<th>Teapot</th>
<th>Dragon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List</td>
<td>743 fps</td>
<td>338 fps</td>
</tr>
<tr>
<td>Precalc</td>
<td>285 fps</td>
<td>143 fps</td>
</tr>
<tr>
<td>Depth Peeling</td>
<td>579 fps</td>
<td>45 fps</td>
</tr>
<tr>
<td>Bucket Depth Peeling</td>
<td>---</td>
<td>256 fps</td>
</tr>
<tr>
<td>Dual Depth Peeling</td>
<td>---</td>
<td>94 fps</td>
</tr>
</tbody>
</table>

Performance scaled to ATI Radeon HD 8770

---

7/28/2010

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

- 602K scene triangles
  - 254K transparent triangles
Worst case 370K fragments filling 40% of the frame
2ms to store the fragments
3.3ms 0->64 fps

Layers
Scaling

112 -> 60 fps -> 32fps
Indirect Illumination with Indirect Shadows using DirectX 11
Why Indirect Shadowing?

- Help perceive subtle dynamic changes occurring 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 dimly 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

Insert triangles of low LOD versions of blocker geometry into cells of 3D grid

eol = End of list (0xffffffff)
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

<table>
<thead>
<tr>
<th>Position</th>
<th>Color</th>
<th>Normal</th>
</tr>
</thead>
</table>
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.

\[
D = \frac{P_L - P_p}{|P_L - P_p|}
\]

\[
\text{Contribution}_{\text{VPL}} = \frac{\text{sat}(N_P \cdot D) \cdot \text{sat}(N_L \cdot (-D)) \cdot \text{Col}_{\text{VPL}} \cdot \text{Area}_{\text{VPL}}}{|P_L - P_p|^2}
\]
Problem!

- Too many samples per kernel will kill performance…but we need very large kernel to get good visual results.
- For decent results need $\geq 512 \times 512$ as well as big kernel $\geq 80 \times 80$
Solution:

• Don’t use the full kernel for each screen pixel.
• Instead, use dithered pattern of pixels which only considers 1 out of $N \times N$ 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.
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


Questions?