Per-Pixel Linked Lists with Direct3D 11

Nicolas Thibieroz
European ISV Relations
AMD
Why Linked Lists?

- Data structure useful for programming

- Very hard to implement efficiently with previous real-time graphics APIs

- DX11 allows efficient creation and parsing of linked lists

- Per-pixel linked lists
  A collection of linked lists enumerating all pixels belonging to the same screen position
Two-step process

1) Linked List Creation
   Store incoming fragments into linked lists

2) Rendering from Linked List
   Linked List traversal and processing of stored fragments
Creating Per-Pixel Linked Lists
PS5.0 and UAVs

- Uses a Pixel Shader 5.0 to store fragments into linked lists
  Not a Compute Shader 5.0!
- Uses atomic operations
- Two UAV buffers required
  - “Fragment & Link” buffer
  - “Start Offset” buffer

UAV = Unordered Access View
The “Fragment & Link” buffer contains data and link for all fragments to store.

Must be large enough to store all fragments.

Created with Counter support

D3D11_BUFFER_UAV_FLAG_COUNTER flag in UAV view

Declaration:

```c
struct FragmentAndLinkBuffer_STRUCT
{
    FragmentData_STRUCT FragmentData; // Fragment data
    uint uNext; // Link to next fragment
};

RWStructuredBuffer <FragmentAndLinkBuffer_STRUCT> FLBuffer;
```
Start Offset Buffer

- The “Start Offset” buffer contains the offset of the last fragment written at every pixel location.
- Screen-sized: \((\text{width} \times \text{height} \times \text{sizeof}(\text{UINT32}))\)
- Initialized to magic value (e.g. -1)
  Magic value indicates no more fragments are stored (i.e. end of the list)

Declaration:

```c
RWByteAddressBuffer StartOffsetBuffer;
```
Linked List Creation (1)

- No color Render Target bound!
  No rendering yet, just storing in L.L.
- Depth buffer bound if needed
  OIT will need it in a few slides
- UAVs bounds as input/output:
  StartOffsetBuffer (R/W)
  FragmentAndLinkBuffer (W)
### Linked List Creation (2a)

#### Start Offset Buffer

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<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>
<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>
<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>

#### Fragment and Link Buffer

Counter = 0
Linked List Creation (2b)

Viewport

Start Offset Buffer

Fragment and Link Buffer

Counter = 1
**Linked List Creation (2c)**

<table>
<thead>
<tr>
<th>Fragment and Link Buffer</th>
<th>Start Offset Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Counter = 3</td>
<td></td>
</tr>
</tbody>
</table>

Viewport
Linked List Creation (2d)

Viewport

Start Offset Buffer

Fragment and Link Buffer

Counter = 5
float PS_StoreFragments(PS_INPUT input) : SV_Target
{
    // Calculate fragment data (color, depth, etc.)
    FragmentData_STRUCT FragmentData = ComputeFragment();

    // Retrieve current pixel count and increase counter
    uint uPixelCount = FLBuffer.IncrementCounter();

    // Exchange offsets in StartOffsetBuffer
    uint vPos = uint(input.vPos);
    uint uStartOffsetAddress = 4 * (SCREEN_WIDTH*vPos.y) + vPos.x;
    uint uOldStartOffset;
    StartOffsetBuffer.InterlockedExchange(uStartOffsetAddress, uPixelCount, uOldStartOffset);

    // Add new fragment entry in Fragment & Link Buffer
    FragmentAndLinkBuffer_STRUCT Element;
    Element.FragmentData = FragmentData;
    Element.uNext = uOldStartOffset;
    FLBuffer[uPixelCount] = Element;
}
Traversing Per-Pixel Linked Lists
“Start Offset” Buffer and “Fragment & Link” Buffer now bound as SRV

Buffer<
int>
StartOffsetBufferSRV;

StructuredBuffer<
FragmentAndLinkBuffer_STRUCT>
FLBufferSRV;

Render a fullscreen quad

For each pixel, parse the linked list and retrieve fragments for this screen position

Process list of fragments as required
Depends on algorithm
 e.g. sorting, finding maximum, etc.

SRV = Shader Resource View
Rendering from Linked List

Start Offset Buffer

Fragment and Link Buffer

Render Target
Rendering from Linked List

Start Offset Buffer

<table>
<thead>
<tr>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>3</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>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>

Fragment and Link Buffer

<table>
<thead>
<tr>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>0</th>
<th>-1</th>
</tr>
</thead>
</table>

Render Target
Rendering from Linked List

Start Offset Buffer

-1 -1 -1 -1 -1 -1
-1 3 4 -1 -1 -1
-1 -1 -1 -1 -1 -1
-1 -1 -1 -1 1 2
-1 -1 -1 -1 -1 -1

Fragment and Link Buffer

-1 -1 -1 0 -1

Render Target

0
Rendering from Linked List

Start Offset Buffer

Fragment and Link Buffer
Rendering from Linked List

Start Offset Buffer

<table>
<thead>
<tr>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>3</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>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>

Fragment and Link Buffer

<table>
<thead>
<tr>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>0</th>
<th>-1</th>
</tr>
</thead>
</table>

Render Target
float4 PS_RenderFragments(PS_INPUT input) : SV_Target
{
    // Calculate UINT-aligned start offset buffer address
    uint vPos = uint(input.vPos);
    uint uStartOffsetAddress = SCREEN_WIDTH*vPos.y + vPos.x;
    // Fetch offset of first fragment for current pixel
    uint uOffset = StartOffsetBufferSRV.Load(uStartOffsetAddress);

    // Parse linked list for all fragments at this position
    float4 FinalColor = float4(0, 0, 0, 0);
    while (uOffset != 0xFFFFFFFF) // 0xFFFFFFFF is magic value
    {
        // Retrieve pixel at current offset
        Element = FLBufferSRV[uOffset];
        // Process pixel as required
        ProcessPixel(Element, FinalColor);
        // Retrieve next offset
        uOffset = Element.uNext;
    }
    return (FinalColor);
}
Order-Independent Transparency via Per-Pixel Linked Lists

Nicolas Thibieroz
European ISV Relations
AMD
Description

- Straight application of the linked list algorithm
- Stores transparent fragments into PPLL
- Rendering phase sorts pixels in a back-to-front order and blends them manually in a pixel shader
  - Blend mode can be unique per-pixel!
- Special case for MSAA support
Linked List Structure

- Optimize performance by reducing amount of data to write to/read from UAV
- E.g. uint instead of float4 for color
- Example data structure for OIT:

```c
struct FragmentAndLinkBuffer_STRUCT
{
    uint uPixelColor;    // Packed pixel color
    uint uDepth;         // Pixel depth
    uint uNext;          // Address of next link
};
```

- May also get away with packed color and depth into the same uint! (if same alpha)
  16 bits color (565) + 16 bits depth
  Performance/memory/quality trade-off
Visible Fragments Only!

- Use `[earlydepthstencil]` in front of Linked List creation pixel shader
- This ensures *only* transparent fragments that pass the depth test are stored i.e. Visible fragments!
- Allows performance savings *and* rendering correctness!

```
[earlydepthstencil]
float PS_StoreFragments(PS_INPUT input) : SV_Target
{
    
    ...
}
```
Sorting Pixels

- Sorting in place requires R/W access to Linked List
- Sparse memory accesses = slow!
- Better way is to copy all pixels into array of temp registers
  - Then do the sorting
- Temp array declaration means a hard limit on number of pixel per screen coordinates
  - Required trade-off for performance
Sorting and Blending

- Blend fragments back to front in PS
- Blending algorithm up to app
- Example: SRCALPHA-INVSRCALPHA
- Or unique per pixel! (stored in fragment data)
- Background passed as input texture
- Actual HW blending mode disabled
Sorting and Blending

- Blend fragments back to front in PS
- Blending algorithm up to app
- Example: SRCALPHA-INVSRCALPHA
- Or unique per pixel! (stored in fragment data)
- Background passed as input texture
- Actual HW blending mode disabled
Sorting and Blending

- Blend fragments back to front in PS
- Blending algorithm up to app
  - Example: SRCALPHA-INVSRCALPHA
  - Or unique per pixel! (stored in fragment data)
- Background passed as input texture
  - Actual HW blending mode disabled
Sorting and Blending

- Blend fragments back to front in PS
  - Blending algorithm up to app
  - Example: SRCALPHA-INVSRCALPHA
  - Or unique per pixel! (stored in fragment data)

- Background passed as input texture
  - Actual HW blending mode *disabled*

```
0.95  0.93  0.87  0.98
34    12    0     -1
```

Temp Array

```
0.95  0.93  0.87  0.98
34    12    0     -1
```

Render Target
Sorting and Blending

- Blend fragments back to front in PS
- Blending algorithm up to app
- Example: SRCALPHA-INVSRCALPHA
- Or unique per pixel! (stored in fragment data)
- Background passed as input texture
- Actual HW blending mode disabled
Storing Pixels for Sorting

(...)
Pixel Blending in PS

(...)

// Retrieve current color from background texture
float4 vCurrentColor = BackgroundTexture.Load(int3(vPos.xy, 0));

// Rendering pixels using SRCALPHA-INVSRCALPHA blending
for (int k = 0; k < nNumPixels; k++)
{
    // Retrieve next unblended furthermost pixel
    float4 vPixColor = UnpackFromUint(SortedPixels[k].x);

    // Manual blending between current fragment and previous one
    vCurrentColor.xyz = lerp(vCurrentColor.xyz, vPixColor.xyz, vPixColor.w);
}

// Return manually-blended color
return vCurrentColor;
OIT via Per-Pixel Linked Lists with MSAA Support
Sample Coverage

- Storing individual samples into Linked Lists requires a huge amount of memory... and performance will suffer!
- Solution is to store transparent pixels into PPLL as before
- But including sample coverage too! Requires as many bits as MSAA mode
- Declare SV_COVERAGE in PS structure

```c
struct PS_INPUT
{
    float3 vNormal : NORMAL;
    float2 vTex : TEXCOORD;
    float4 vPos : SV_POSITION;
    uint uCoverage : SV_COVERAGE;
}
```
Linked List Structure

- Almost unchanged from previously
- Depth is now packed into 24 bits
- 8 Bits are used to store coverage

```c
struct FragmentAndLinkBuffer_STRUCT {
    uint uPixelColor;       // Packed pixel color
    uint uDepthAndCoverage; // Depth + coverage
    uint uNext;             // Address of next link
};
```
Sample Coverage Example

Third sample is covered

\[ \text{uCoverage} = 0x04 \quad (0100 \text{ in binary}) \]

\[ \text{Element.uDepthAndCoverage} = \left( \text{In.vPos.z} \times (2^{24} - 1) \ll 8 \right) \mid \text{In.uCoverage}; \]
Rendering Samples (1)

- Rendering phase needs to be able to write individual samples
- Thus PS is run at sample frequency
  Can be done by declaring SV_SAMPLEINDEX in input structure
- Parse linked list and store pixels into temp array for later sorting
  Similar to non-MSAA case
  Difference is to only store sample if coverage matches sample index being rasterized
Rendering Samples (2)

```c
static uint2 SortedPixels[MAX_SORTED_PIXELS];

// Parse linked list for all pixels at this position
// and store them into temp array for later sorting
int nNumPixels=0;
while (uOffset!=0xFFFFFFFF)
{
    // Retrieve pixel at current offset
    Element=FLBufferSRV[uOffset];

    // Retrieve pixel coverage from linked list element
    uint uCoverage=UnpackCoverage(Element.uDepthAndCoverage);
    if ( uCoverage & (1<<In.uSampleIndex) )
    {
        // Coverage matches current sample so copy pixel
        SortedPixels[nNumPixels++]=Element;
    }
    // Retrieve next offset
    [flatten]uOffset = (nNumPixels>=MAX_SORTED_PIXELS) ? 0xFFFFFFFF : Element.uNext;
}
```
Q&A

Holger Gruen
nicolas.thibieroz@AMD.com

Nicolas Thibieroz
holger.gruen@AMD.com

Credits for the basic idea of how to implement PPLL under Direct3D 11 go to Jakub Klarowicz (Techland), Holger Gruen and Nicolas Thibieroz (AMD)