VII 2

Order-Independent Transparency using Per-Pixel Linked Lists

Nicolas Thibieroz

2.1 Introduction

Order-independent transparency (OIT) has been an active area of research in real-time computer graphics for a number of years. The main area of research has focused on how to effect fast and efficient back-to-front sorting and rendering of translucent fragments to ensure visual correctness when order-dependent blending modes are employed. The complexity of the task is such that many real-time applications have chosen to forfeit this step altogether in favor of simpler and faster alternative methods such as sorting per object or per triangle, or simply falling back to order-independent blending modes (e.g., additive blending) that don't require any sorting [Thibieroz 08]. Different OIT techniques have previously been described (e.g., [Everitt 01], [Myers 07]) and although those techniques succeed in achieving correct ordering of translucent fragments, they usually come with performance, complexity, or compatibility shortcomings that make their use difficult for real-time gaming scenarios.

This chapter presents an OIT implementation relying on the new features of the DirectX 11 API from Microsoft.[1] An overview of the algorithm will be presented first, after which each phase of the method will be explained in detail. Sorting, blending, multisampling, anti-aliasing support, and optimizations will be treated in separate sections before concluding with remarks concerning the attractiveness and future of the technique.

2.2 Algorithm Overview

The OIT algorithm presented in this chapter shares some similarities with the A-buffer algorithm [Carpenter 84], whereby translucent fragments are stored in

[1] "Microsoft DirectX" is a registered trademark.

buffers for later rendering. Our method uses linked lists [Yang 10] to store a list of translucent fragments for each pixel. Therefore, every screen coordinate in the render viewport will contain an entry to a unique per-pixel linked list containing all translucent fragments at that particular location.

Prior to rendering translucent geometry, all opaque and transparent (alpha-tested) models are rendered onto the render target viewport as desired. Then the OIT algorithm can be invoked to render corrected-ordered translucent fragments.

The algorithm relies on a two-step process.

1. The first step is the creation of per-pixel linked lists whereby the translucent contribution to the scene is entirely captured into a pair of buffers containing the head pointers and linked lists nodes for all translucent pixels.

2. The second step is the traversal of per-pixel linked lists to fetch, sort, blend and finally render all pixels in the correct order onto the destination render viewport.

2.3 DirectX 11 Features Requisites

DirectX 11 has introduced new features that finally make it possible to create and parse concurrent linked lists on the GPU.

2.3.1 Unordered Access Views

Unordered access view (UAV) is a special type of resource view that can be bound as output to a pixel or compute shader to allow the programmer to write data at arbitrary locations. The algorithm uses a pair of UAVs to store per-pixel linked lists nodes and head pointers.

2.3.2 Atomic Operations

The creation of per-pixel linked lists also requires a way to avoid any contention when multiple pixel shader invocations perform memory operations into a buffer. Indeed such read/modify/write operations must be guaranteed to occur atomically for the algorithm to work as intended. DirectX 11 supports a set of `Interlocked*()` Shader Model 5.0 instructions that fulfill this purpose. Such atomic operation will be used to keep track of the head pointer address when creating the per-pixel linked lists.

2.3.3 Hardware Counter Support on UAV

A less well-known feature of DirectX 11 allows the creation of a built-in hardware counter on buffer UAVs. This counter is declared by specifying the `D3D11_BUFFER_UAV_FLAG_COUNTER` flag when generating a UAV for the intended buffer. The programmer is given control of the counter via the following two Shader Model 5.0 methods:

```
uint <Buffer>.IncrementCounter();
uint <Buffer>.DecrementCounter();
```

Hardware counter support is used to keep track of the offset at which to store the next linked list node.

While hardware counter support is not strictly required for the algorithm to work, it enables considerable performance improvement compared to manually keeping track of a counter via a single-element buffer UAV.

2.3.4 Early Depth/Stencil Rejection

Graphics APIs like OpenGL and Direct3D specify that the depth/stencil test be executed *after* the pixel shader stage in the graphics pipeline. A problem arises when the pixel shader outputs to UAVs because UAVs may be written into the shader even though the subsequent depth/stencil test actually discards the pixel, which may not be the intended behavior of the algorithm. Shader Model 5.0 allows the `[earlydepthstencil]` keyword to be declared in front of a pixel shader function to indicate that the depth/stencil test is to be explicitly performed *before* the pixel shader stage, allowing UAV writes to be carried out only *if the depth/ stencil test succeeds first*. This functionality is important for the algorithm presented in this chapter, since only *visible* translucent fragments need storing into the per-pixel linked lists.

2.3.5 SV_COVERAGE Pixel Shader Input

DirectX 11 allows `SV_COVERAGE` to be declared as an input to the pixel shader stage. `SV_COVERAGE` contains a bit mask of all samples that are covered by the current primitive. This information is used by this OIT technique when multisampling antialiasing (MSAA) is enabled.

2.3.6 Per-sample Pixel Shader Execution

DirectX 11 allows the pixel shader stage to execute *per sample* (as opposed to *per pixel*) when MSAA is enabled. This functionality will be exploited to allow MSAA support with our OIT technique.

2.4 Head Pointer and Nodes Buffers

The algorithm builds a reverse linked list for each pixel location in the target viewport. The linked list *head pointer* is the address of the first element in the linked list. The linked list *nodes* are the individual elements of the linked list that contain fragment data as well as the address of the next node.

2.4.1 Head Pointer Buffer

The algorithm allocates a screen-sized buffer of type `DXGI_FORMAT_R32_UINT` that contains the address of the head pointer for every 2D coordinate on the screen.

Despite the resource having the same dimensions as the render viewport, the declaration must employ the buffer type because atomic operations are not supported on Texture2D resources. Therefore an extra step will be required in the shader wherein a 2D screen-coordinate location is converted to a byte-linear address:

```
uint uLinearAddressInBytes = 4 * ( ScreenPos.y*RENDERWIDTH +
 ScreenPos.x );
```

The head pointer buffer is initialized to a "magic" value of 0xFFFFFFFF, indicating that the linked list is empty to start with. In effect, an address of 0xFFFFFFFF indicates that no more nodes are available (i.e., the end of the list has been reached).

The term *reverse* linked list is used because the head pointer is dynamically updated at construction time to receive the address of the latest linked list node written out at each pixel location. Once construction is complete, the head pointer value effectively contains the address of the *last* node written out, with this last node sequentially pointing to the nodes previously stored for the same pixel location.

2.4.2 Nodes Buffer

The nodes buffer stores the nodes of *all* per-pixel linked lists. We cannot allocate individual nodes buffers for every linked list since the render viewport dimensions guarantee that a huge number of them will be required (one for each pixel in the render viewport). Therefore the nodes buffer must be allocated with enough memory to accommodate all possible translucent fragments in the scene. It is the responsibility of the programmer to define this upper limit. A good heuristic to use is to base the allocation size on the render viewport dimensions multiplied by the average translucent overdraw expected. Certainly the size of the nodes buffer is likely to be very large, and may place an unreasonable burden on the video memory requirements of the OIT technique. Section 2.9 introduces a significant optimization to dramatically reduce the memory requirements of this algorithm.

The UAV for the nodes buffer is created with a built-in hardware counter initialized to zero as a way to keep track of how many fragments have been stored in the buffer.

2.4.3 Fragment Data

Each linked list node stores fragment data as well as the address of the next node. The address of the next node is stored as a `uint` while the type and size of the fragment data depends on the needs of the application. Typically the fragment structure includes fragment depth and color but other variables may be stored as needed (e.g., normal, blending mode id, etc.). Fragment depth is an essential component of the fragment structure since it will be required at linked list traversal time to correctly sort fragments prior to rendering.

It is important to point out that any additional structure member will increase the total size of the nodes buffer and therefore the total memory requirement of the OIT solution. It is therefore desirable to economize the size of the fragment structure. The implementation presented in this chapter packs the fragment color into a single `uint` type and uses the following node data structure for a total of 12 bytes per fragment:

```
struct NodeData_STRUCT
{
    uint uColor; // Fragment color packed as RGBA
    uint uDepth; // Fragment depth
    uint uNext;  // Address of next linked list node
};
```

2.5 Per-Pixel Linked List Creation

The algorithm presented does not use `DirectCompute`; instead, the storing of translucent fragments into linked lists is done via a pixel shader writing to the head pointer and nodes buffers UAVs. Earlier shader stages in the pipeline are enabled (vertex shader, but also hull shader, domain shader and geometry shader if needed) in order to turn incoming triangles into fragments that can be stored into per-pixel linked lists.

No color render target is bound at this stage although a depth buffer is still required to ensure that only translucent fragments that pass the depth/stencil test are stored in the per-pixel linked lists. Binding a depth buffer avoids the need to store translucent fragments that would result in being completely hidden by previously-rendered opaque or alpha-tested geometry.

2.5.1 Algorithm Description

A description of the algorithm used to build per-pixel linked lists follows.

- For each frame
    - *Clear head pointer buffer to 0xFFFFFFFF* (-1). This indicates that the per-pixel linked lists are all empty to start with.

- *For (each translucent fragment)*
    - *Compute and store fragment color and depth into node structure.* Calculate the fragment color as normal using lighting, texturing etc. and store it in node data structure. Pixel depth can be directly obtained from the z member of the `SV_POSITION` input.
    - *Retrieve current counter value of nodes buffer and increment counter.* The current counter value tells us how many nodes have been written into the nodes buffer. We will use this value as the offset at which to store the new node entry.
    - *Atomically exchange head pointer for this screen location with counter value.* The previous head pointer for the current screen coordinates is retrieved and set as the "next" member of the node structure. The node structure being prepared therefore points to the previous node that was written out at this pixel location. The new head pointer receives the current value of the counter as it will represent the latest node written out.
    - *Store node structure into node buffer at offset specified by counter value.* The node structure containing fragment data and next node pointer is written out to the nodes buffer at the offset specified by the current counter value. This is the latest node to be written out to the nodes buffer for these pixel coordinates; this is why the counter offset was also set as the new head pointer in the previous step.

Figure 2.1 illustrates the contents of the head pointer and nodes buffers after three translucent triangles go through the per-pixel linked list creation step.

The single pixel occupied by the orange triangle stores the current nodes buffer counter value (0) into location $[1, 1]$ in the Head Pointer Buffer and sets the previous Head Pointer value (-1) as the "next" node pointer in the node structure before writing it to the nodes buffer at offset 0.

The two pixels occupied by the green triangle are processed sequentially; they store the current nodes buffer counter values (1 and 2) into locations $[3, 4]$ and $[4, 4]$ in the head pointer buffer and set the previous head pointer values (-1 and -1) as the "next" node pointers in the two node structures before writing them to the nodes buffer at offset 1 and 2.

The left-most pixel of the yellow triangle is due to be rendered at the same location as the orange fragment already stored. The current counter value (3) replaces the previous value (0) in the head pointer buffer at location $[1, 1]$ and the previous value is now set as the "next" node pointer in the fragment node before writing it to offset 3 in the nodes buffer.

The second pixel of the yellow triangle stores the current counter value (4) into location $[2, 1]$ and sets the previous value (-1) as the "next" node pointer before writing the node to the nodes buffer at offset 4.

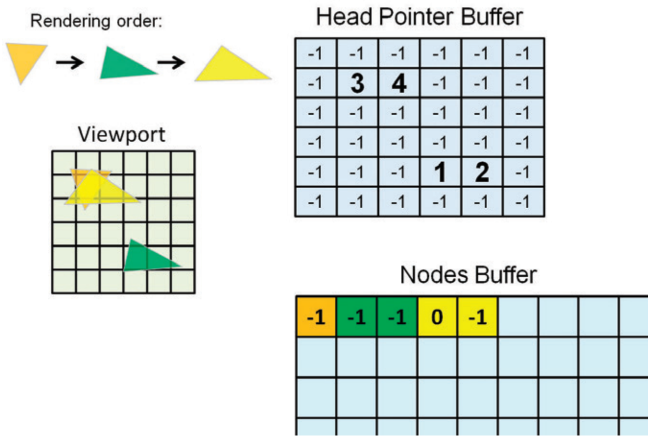

Figure 2.1. Head pointer and nodes buffers contents after rendering five translucent pixels (3 triangles) for a 6 × 6 render target viewport.

2.5.2 Pixel Shader Code

The pixel shader code for creating per-pixel linked lists can be found in Listing 2.1.

```
// Pixel shader input structure
struct PS_INPUT
{
    float3 vNormal : NORMAL;     // Pixel normal
    float2 vTex : TEXCOORD;      // Texture coordinates
    float4 vPos : SV_POSITION;  // Screen coordinates
};

// Node data structure
struct NodeData_STRUCT
{
    uint uColor; // Fragment color packed as RGBA
    uint uDepth; // Fragment depth
    uint uNext;  // Address of next linked list node
};

// UAV declarations
RWByteAddressBuffer HeadPointerBuffer : register(u1);
RWStructuredBuffer<NodeData_STRUCT> NodesBuffer : register(u2);

// Pixel shader for writing per-pixel linked lists
[earlydepthstencil]
float PS_StoreFragments(PS_INPUT input) : SV_Target
{
    NodeData_STRUCT Node;
    // Calculate fragment color from pixel input data
    Node.uColor = PackFloat4IntoUint(ComputeFragmentColor(input));

    // Store pixel depth in packed format
    Node.uDepth = PackDepthIntoUint( input.vPos.z );

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

    // Convert pixel 2D coordinates to byte linear address
    uint2 vScreenPos = uint(input.vPos.xy);
    uint uLinearAddressInBytes = 4 * ( vScreenPos.y*RENDERWIDTH +
                                       vScreenPos.x );

    // Exchange offsets in Head Pointer buffer
    // Node.uNext will receive the previous head pointer
    HeadPointerBuffer.InterlockedExchange(
        uLinearAddressInBytes, uPixelCount, Node.uNext);

    // Add new fragment entry in Nodes Buffer
    NodesBuffer[uPixelCount] = Node;

    // No RT bound so this will have no effect
    return float4(0,0,0,0);
}
```

Listing 2.1. Pixel shader for creating per-pixel linked lists.

2.6 Per-Pixel Linked Lists Traversal

Once the head pointer and nodes buffers have been filled with data, the second phase of the algorithm can proceed: traversing the per-pixel linked lists and rendering translucent pixels in the correct order.

The traversal code needs to be executed once per pixel; we therefore render a fullscreen quad (or triangle) covering the render viewport with a pixel shader that will parse all stored fragments for every pixel location. Each fragment from the corresponding per-pixel linked list will be fetched and then sorted with other fragments before a manual back-to-front blending operation will take place, starting with the color from the current render target.

2.6.1 Pixel Shader Inputs and Outputs

The head pointer and nodes buffers are now bound as pixel shader inputs, since the traversal step will exclusively be *reading* from them. We will also need a copy of the render target onto which opaque and alpha-tested geometry have previously been rendered. This copy is needed to start the manual blending operation in the pixel shader.

The current render target viewport (onto which opaque and alpha-tested geometry has previously been rendered) is set as output, and the depth buffer that was used to render previous geometry is bound with Z-Writes disabled (the use of the depth buffer for the traversal step is explained in the Optimizations section).

2.6.2 Algorithm Description

A description of the algorithm used to parse per-pixel linked lists and render from them follows.

- For (each frame)
  - *Copy current render target contents into a background texture.* This copy is required to perform the manual blending process.
  - *For (each pixel)*
    - *Fetch head pointer for corresponding pixel coordinates and set it as current node pointer.* The screen coordinates of the current pixel are converted to a linear address used to fetch the head pointer from the head pointer buffer. The head pointer is therefore the first node address pointing to the list of all translucent fragments occupying this pixel location.
    - *While (end of linked list not reached)*
      - *Fetch current fragment from nodes buffer using current node pointer.* The current node pointer indicates where the next fragment is stored in the nodes buffer for the current pixel location.
      - *Sort fragment with previously collected fragments in front-to-back order.* The current fragment is inserted into the last position of a temporary array. The fragment is then sorted on-the-fly with previous fragments stored in the array.
      - *Set current node pointer to "next" node pointer from current node.* The current fragment has been processed; we therefore proceed to the next fragment in the list by setting the current node pointer to the "next" pointer of the current node. Fragment parsing will stop once the "next" pointer equals 0xFFFFFFFF, as this value indicates the end of the list.
    - *The background color is fetched from the background texture and set as the current color.* Back-to-front blending requires the furthermost color value to be processed first, hence the need to obtain the current value of the render target to start the blending process.

    * *For (each fragment fetched from sorted list in reverse order)*
        - Perform manual back-to-front blending of current color with fragment color. The latest entry in the front-to-back sorted array is fetched, unpacked from a uint to a float4, and blended manually with the current color.
    * *The final blended color is returned from the pixel shader.* This color is the correctly-ordered translucent contribution for this pixel.

2.6.3 Sorting Fragments

The algorithm used to sort fragments for correct ordering can be arbitrary because it typically has no dependencies with the OIT algorithm described in this chapter.

The implementation presented here declares a temporary array of `uint2` format containing fragment color and depth. At parsing time each fragment extracted from the per-pixel linked list is directly stored as the last entry in the array, and sorted with previous array elements using a standard insertion-sort algorithm. This approach requires an array of a fixed size, which unfortunately limits the maximum number of fragments that can be processed with this technique. The alternative option of parsing and sorting the linked list "in place" resulted in much lower performance due to repeated accesses to memory, and was also subject to limitations of the DirectX Shader Compiler regarding loop terminating conditions being dependent on the result of a UAV fetch. For those reasons, sorting via a temporary array was chosen. In general the smaller the array the better the performance since temporary arrays consume precious general purpose registers (GPRs) that affect the performance of the shader. The programmer should therefore declare the array size as an estimate of the maximum translucent overdraw that can be expected in the scene.

2.6.4 Blending Fragments

The blending process takes place once all fragments for the current pixel coordinates have been parsed and sorted in the temporary array. The blending operation used in this implementation uses the same `SRCALPHA-INVSRCALPHA` blend mode for all translucent fragments. The blending process starts with the background color and then iteratively blends the current color with each fragment color in a back-to-front order. Because the blending is performed "manually" in the pixel shader, actual hardware color/alpha blending is disabled.

A different approach that would avoid the need for copying the render target contents to a texture prior to rendering translucent fragments would be to use underblending [Bavoil 08]. Underblending allows fragments to be blended in a front-to-back order, hence avoiding the need to access the background color as a texture (the background color will be blended with the result of the manually underblended result via actual hardware blending). However, this method imposes restrictions on the variety of per-fragment blend modes that can be used, and did not noticeably affect performance.

It is quite straightforward to modify the algorithm so that per-fragment blend modes are specified instead of adopting a blend mode common to all fragments. This modification allows translucent geometry of different types (particles, windows, smoke etc.) to be stored and processed together. In this case a bit field containing the blend mode id of each fragment is stored in the node structure (along with pixel color and depth) in the per-pixel linked list creation step. Only a few bits are required (this depends on how many different blend modes are specified— typically this shouldn't be more than a handful) and therefore the bit field could be appended to the existing color or depth member of the node structure by modifying the packing function accordingly. When the per-pixel linked lists are parsed for rendering, the blending part of the algorithm is modified so that a different code path (ideally based on pure arithmetic instructions to avoid the need for actual code divergence) is executed based on the fragment's blend mode id.

2.6.5 Pixel Shader Code

The pixel shader code for linked list traversal and rendering is given in in Listing 2.2.

```
// Pixel shader input structure for fullscreen quad rendering
struct PS_SIMPLE_INPUT
 {
    float2 vTex : TEXCOORD;     // Texture coordinates
    float4 vPos : SV_POSITION; // Screen coordinates
  };

// Fragment sorting array
#define MAX_SORTED_FRAGMENTS 18
static uint2 SortedFragments[MAX_SORTED_FRAGMENTS+1];

// SRV declarations
Buffer<uint> HeadPointerBufferSRV : register(t0);
StructuredBuffer<NodeData_STRUCT> NodesBufferSRV : register(t1);
Texture2D BackgroundTexture : register(t3);

// Pixel shader for parsing per-pixel linked lists
float4 PS_RenderFragments( PS_SIMPLE_INPUT input) : SV_Target
{

    // Convert pixel 2D coordinates to linear address
    uint2 vScreenPos = uint(input.vPos.xy);
    uint uLinearAddress = vScreenPos.y*RENDERWIDTH + vScreenPos.x;
```

```
// Fetch offset of first fragment for current pixel
uint uOffset = HeadPointerBufferSRV[uLinearAddress];

// Loop through each node stored for this pixel location
int nNumFragments = 0;
while (uOffset != 0xFFFFFFFF)
{

    // Retrieve fragment at current offset
    NodeData_STRUCT Node = NodesBufferSRV[uOffset];

    // Copy fragment color and depth into sorting array
    SortedFragments[nNumFragments] =
    uint2(Node.uColor, Node.uDepth);

    // Sort fragments front to back using insertion sorting
    int j = nNumFragments;
    [loop]while ( (j>0) &&
        (SortedFragments[max(j-1, 0)].y >
        SortedFragments[j].y) )
    {

        // Swap required
        int jminusone = max(j-1, 0);
        uint2 Tmp = SortedFragments[j];
        SortedFragments[j] = SortedFragments[jminusone];
        SortedFragments[jminusone] = Tmp;
        j--;
    }

    // Increase number of fragment if under the limit
    nNumFragments = min(nNumFragments+1,
    MAX_SORTED_FRAGMENTS);

    // Retrieve next offset
    uOffset = Element.uNext;
}

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

// Render sorted fragments using SRCALPHA-INVSRCALPHA
// blending
for (int k=nNumFragments-1; k>=0; k--)
{

    float4 vColor = UnpackUintIntoFloat4
    (SortedFragments[k].x);
    vCurrentColor.xyz = lerp(vCurrentColor.xyz, vColor.xyz,
                             vColor.w);
}
```

```
    // Return manually-blended color
    return vCurrentColor;
}
```

Listing 2.2. Pixel shader for parsing per-pixel linked lists.

2.7 Multisampling Antialiasing Support

Multisampling antialiasing (MSAA) is supported by the OIT algorithm presented in this chapter via a couple of minor modifications to the technique. MSAA works by performing the depth test at multiple pixel sample locations and outputting the result of the pixel shader stage (evaluated at centroid location) to all samples that passed the depth test.

2.7.1 Per-pixel Linked Lists Creation

Directly storing translucent samples into per-pixel linked lists would rapidly become prohibitive from both a memory and performance perspective due to the sheer amount of samples to store. Instead the algorithm adapted to MSAA can simply store fragment data into per-pixel linked nodes as usual, but including *sample coverage* data in the node structure.

Sample coverage is an input provided by the pixel shader stage that specifies whether samples are covered by the input primitive. Sample coverage is a bit field containing as many useful bits as the number of samples, and is passed down to the pixel shader stage via the DirectX 11-specific SV_COVERAGE input (Figure 2.2 illustrates the concept of sample coverage on a single pixel.):

```
// Pixel shader input structure
struct PS_INPUT
{
    float3 vNormal : NORMAL;        // Pixel normal
    float2 vTex : TEXCOORD;         // Texture coordinates
    float4 vPos : SV_POSITION;      // Screen coordinates
    uint uCoverage : SV_COVERAGE;   // Pixel coverage
};
```

Only a few bits are required for sample coverage; we therefore pack it onto the depth member of the node structure using a 24:8 bit arrangement (24 bits for depth, 8 bits for sample coverage). This avoids the need for extra storage and leaves enough precision for encoding depth. The node structure thus becomes:

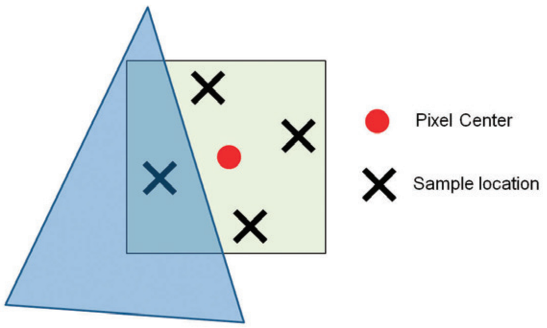

Figure 2.2. Sample coverage example on a single pixel. The blue triangle covers the third sample in a standard MSAA 4x arrangement. The input coverage to the pixel shader will therefore be equal to 0x04 (0100 in binary).

```
struct NodeData_STRUCT
{
    uint uColor;            // Fragment color packed as RGBA
    uint uDepthAndCoverage; // Fragment depth and coverage
    uint uNext;             // Address of next linked list node
};
```

The pixel shader to create per-pixel linked lists is modified so that depth *and* coverage are now packed together and stored in the node structure:

```
// Store pixel depth and coverage in packed format
Node.uDepthAndCoverage = PackDepthAndCoverageIntoUint(
    input.vPos.z, input.uCoverage );
```

2.7.2 Per-pixel Linked Lists Traversal

The traversal of per-pixel linked lists needs to occur at *sample frequency* when outputting into a multisampled render target. This is because fragments stored in the per-pixel linked lists may affect only some samples at a given pixel location. Therefore both fragment sorting and blending must be performed per sample to ensure that translucent geometry gets correctly antialiased.

As in the non-multisampled case, a fullscreen quad is rendered to ensure that every pixel location in the destination render target is covered; one notable difference is that the pixel shader input for this pass now declares the `SV_SAMPLEINDEX` system value in order to force the pixel shader to execute at sample frequency. This value specifies the index of the sample at which the pixel shader currently executes.

When parsing per-pixel linked lists for rendering, the sample coverage in the current node is compared with the index of the sample being shaded: if the index is included in the sample coverage, then this pixel node contributes to the current sample and is therefore copied to the temporary array for sorting and later blending.

One further modification to the blending portion of the code is that the background texture representing the scene data prior to any translucent contribution is multisampled, thus the declaration and the fetch instruction are modified accordingly.

After the fullscreen quad is rendered, the multisampled render target will contain the sample-accurate translucent contribution to the scene.

2.7.3 Pixel Shader Code

The pixel shader code for parsing and rendering from per-pixel linked lists with MSAA enabled can be found in Listing 2.3. Code specific to MSAA is highlighted. Note that the shader can easily cater to both MSAA and non-MSAA cases by the use of a few well-placed `#define` statements.

```
// Pixel shader input structure for fullscreen quad rendering
// with MSAA enabled
struct PS_SIMPLE_INPUT
{
    float2 vTex : TEXCOORD;       // Texture coordinates
    float4 vPos : SV_POSITION;    // Screen coordinates
    uint uSample : SV_SAMPLEINDEX;    // Sample index
  };

// Fragment sorting array
#define MAX_SORTED_FRAGMENTS 18
static uint2 SortedFragments[MAX_SORTED_FRAGMENTS+1];

// SRV declarations
Buffer<uint> HeadPointerBufferSRV : register(t0);
StructuredBuffer<NodeData_STRUCT> NodesBufferSRV : register(t1);
Texture2DMS <float4, NUM_SAMPLES BackgroundTexture : register(t3);

// Pixel shader for parsing per-pixel linked lists with MSAA
float4 PS_RenderFragments( PS_SIMPLE_INPUT input) : SV_Target
{

    // Convert pixel 2D coordinates to linear address
    uint2 vScreenPos = uint(input.vPos.xy);
    uint uLinearAddress = vScreenPos.y*RENDERWIDTH
                             + vScreenPos.x;

    // Fetch offset of first fragment for current pixel
    uint uOffset = HeadPointerBufferSRV[uLinearAddress];
    // Loop through each node stored for this pixel location
    int nNumFragments = 0;
    while (uOffset != 0xFFFFFFFF)
    {

        // Retrieve fragment at current offset
        NodeData_STRUCT Node = NodesBufferSRV[uOffset];
        // Only include fragment in sorted list if coverage mask
        // includes the sample currently being rendered}
        uintuCoverage =
                 UnpackCoverageIntoUint(Node.uDepthAndCoverage);
       if ( uCoverage & (1<<input.uSample) )}
       {
             // Copy fragment color and depth into sorting array
             SortedFragments[nNumFragments] =
                  uint2(Node.uColor, Node.uDepth);

             // Sort fragments front to back using
             // insertion sorting
             int j = nNumFragments;
             [loop]while ( (j>0) &&
                  (SortedFragments[max(j-1, 0)].y >
                        SortedFragments[j].y) )
             {
                  // Swap required
                  int jminusone = max(j-1, 0);
                  uint2 Tmp = SortedFragments[j];
                  SortedFragments[j] = SortedFragments[jminusone];
                  SortedFragments[jminusone] = Tmp;
                  j--;
             }

             // Increase number of fragment if under the limit
             nNumFragments=min(nNumFragments+1,
                                    MAX_SORTED_FRAGMENTS);
        }

        // Retrieve next offset
        uOffset = Element.uNext;
    }

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

    // Render sorted fragments using SRCALPHA-INVSRCALPHA
    // blending
    for (int k=nNumFragments-1; k>=0; k--)
    {

        float4 vColor =
                 UnpackUintIntoFloat4(SortedFragments[k].x);
```

```
            vCurrentColor.xyz = lerp(vCurrentColor.xyz, vColor.xyz,
                                     vColor.w);
        }

        // Return manually-blended color
        return vCurrentColor;
}
```

Listing 2.3. Pixel Shader for parsing per-pixel linked lists when MSAA is enabled

2.8 Optimizations

2.8.1 Node Structure Packing

As previously mentioned in this chapter, the size of the node structure has a direct impact on the amount of memory declared for the nodes buffer. Incidentally, the smaller the size of the node structure, the better the performance, since fewer memory accesses will be performed. It therefore pays to aggressively pack data inside the node structure, even if it adds to the cost of packing/unpacking instructions in the shaders used.

The default node structure presented in previous paragraphs is three `uint` in size whereby one `uint` is used for packed RGBA color, one `uint` is used for depth and coverage, and the last `uint` is used for the next pointer. Some circumstances may allow further reduction of the structure for a performance/memory benefit; for instance, color and depth could be packed into a single `uint` (e.g., by encoding RGB color as 565 and depth as a 16-bit value (such a reduction in depth precision may need some scaling and biasing to avoid precision issues)). The "next" pointer could be encoded with 24 bits, leaving 8 bits for a combination of sample coverage and/or blend id. Such a scheme would reduce the node structure size to two `uint` (8 bytes), which is a desirable goal if the scene circumstances allow it.

2.8.2 Early Stencil Rejection in Parsing Phase

The second pass of the OIT algorithm can be accelerated by ensuring that only screen locations that have received at least one translucent fragment entry are processed. This would avoid the need for "empty" parsing of the per-pixel linked lists, improving performance in the process.

To achieve this goal, the linked lists creation phase is set up so that the stencil buffer is incremented for each fragment that passes the depth test. Once this phase is completed, the stencil buffer will contain translucent overdraw for each pixel in the scene, leaving the stencil clear value at 0 for pixel locations that haven't received any translucent fragment (i.e., for which the head pointer buffer is still 0xFFFFFFFF).

When per-pixel linked lists are parsed for rendering the stencil buffer is set up to pass if the stencil value is above 0. Early stencil rejection ensures that only pixel locations that have been touched by translucent fragments will be processed, saving on performance in the process.

2.8.3 MSAA: Fragment Resolve during Parsing

Executing the pixel shader at sample frequency is a significant performance cost compared to per-pixel execution. It is possible to replace per-sample execution for per-pixel execution for the fragment parsing and rendering pass in MSAA mode if fragments are "resolved" at the same time they are processed. Resolving fragments is a term that refers to the process of converting pixel samples into a single pixel color value that gets written onto a single-sample render target. The most common resolve function is a straight average of all samples belonging to a pixel but some variations exist (e.g., HDR-correct resolves).

To resolve fragments in the OIT rendering phase a *non-multisampled* render target has to be bound as an output. Doing so will prevent any further rendering operation requiring access to multisampled data, so it is important that this optimization is considered only if such access is no longer required. Should this condition be fulfilled, the performance improvements enabled by fragment resolving can be dramatic (up to a 50% increase was observed on a Radeon 5870 at 1024×768 with 4x MSAA) so this is certainly an optimization to consider. Because the render target bound is no longer multisampled the depth stencil buffer that was bound when storing fragments can no longer be used for early stencil rejection in the fragment parsing phase. Still, the performance boost obtained via fragment resolving outweighs the benefit of this previous optimization.

To restore per-pixel shader execution, the pixel shader input structure no longer declares `SV_SAMPLEINDEX`. Only the blending section of the per-pixel linked list parsing shader needs further code modification to enable fragment resolving. After fragments have been fetched and sorted in the temporary array, the algorithm needs to determine the blending contribution for each sample in the destination pixel. Hence the next node picked from the sorted list will add its blending contribution to a sample only if its pixel coverage includes the sample currently being processed. Once the blending contribution for all samples has been determined, the final average (resolve) operation takes place and the resulting color is output to the destination render target.

The blending portion of the per-pixel linked list parsing shader that implements fragment resolve in MSAA mode can be found in Listing 2.4.

```
// Retrieve color of each sample in the background texture
float3 vCurrentColorSample[NUM_SAMPLES];
[unroll]for (uint uSample=0; uSample<NUM_SAMPLES; uSample++)
{
        vCurrentColorSample[uSample] =
            BackgroundTexture.Load(int3(input.vPos.xy, 0), uSample);
    }

    // Render fragments using SRCALPHA-INVSRCALPHA blending
    for (int k=nNumFragments-1; k>=0; k--)
    {
        // Retrieve fragment color
        float4 vFragmentColor=
                      UnpackUintIntoFloat4(SortedFragments[k].x);

        // Retrieve sample coverage
        uint uCoverage =
                      UnpackCoverageIntoUint(SortedFragments[k].y);

        // Loop through each sample
        [unroll]for (uint uSample=0; uSample<NUM_SAMPLES; uSample++)
        {
            // Determine if sample is covered by sample coverage
            float fIsSampleCovered= (uCoverage & (1<<uSample)) ?
                                           1.0 : 0.0;

            // Blend current color sample with fragment color
            // if covered. If the sample is not covered the color
            // will be unchanged
            vCurrentColorSample[uSample].xyz = lerp(
                    vCurrentColorSample[uSample].xyz,
                    vFragmentColor.xyz,
                    vFragmentColor.w * fIsSampleCovered);
        }
    }

    // Resolve samples into a single color
    float4 vCurrentColor = float4(0,0,0,1);
    [unroll]for (uint uSample=0; uSample<NUM_SAMPLES; uSample++)
    {
        vCurrentColor.xyz += vCurrentColorSample[uSample];
    }
    vCurrentColor.xyz *= (1.0/NUM_SAMPLES);

    // Return manually-blended color
    return vCurrentColor;
```

Listing 2.4. Blending and resolving fragments

2.9 Tiling

2.9.1 Tiling as a Memory Optimization

Tiling is a pure memory optimization that considerably reduces the amount of video memory required for the nodes buffer (and to a lesser extent the head

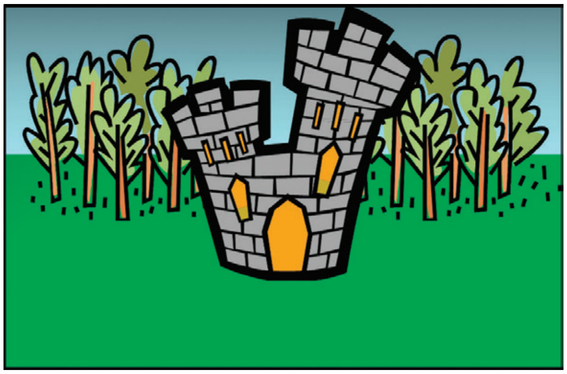

Figure 2.3. Opaque contents of render target prior to any OIT contribution.

pointer buffer). Without tiling, the memory occupied by both buffers can rapidly become huge when fullscreen render target resolutions are used. As an example a standard HD resolution of 1280×720 with an estimated average translucent overdraw of eight would occupy a total of $1280 \times 720 \times 8 \times$sizeof(node structure size) bytes for the nodes buffer only, which equates to more than 168 megabytes with a standard node structure containing 3 units (color, depth and next pointer).

Instead of allocating buffers for the full-size render target, a single, smaller rectangular region (the "tile") is used. This tile represents the extent of the

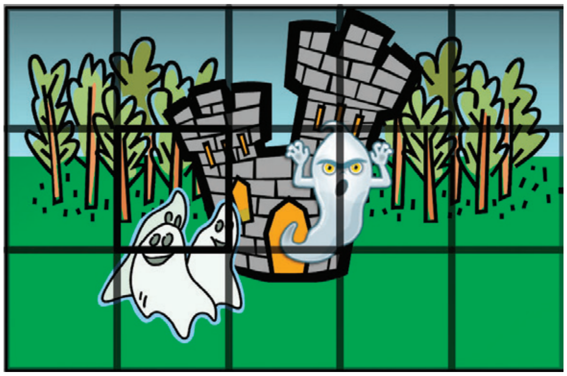

Figure 2.4. Translucent contribution to the scene is added to the render target via regular tiling. Each rectangular area stores fragments in the tile-sized head pointer and nodes buffers and then parses those buffers to add correctly ordered translucency information to the same rectangle area. In this example the tile size is 1/15 of the render target size, and a total of 15 rectangles are processed.

area being processed for OIT in a given pass. Since the tile is typically smaller than the render size, this means multiple passes are needed to calculate the full-size translucent contributions to the scene. The creation of per-pixel linked lists is therefore performed on a per-tile basis, after which the traversal phase fetches nodes from the tile-sized head pointer and nodes buffers to finally output the resulting color values onto the rectangular region corresponding to the tile being processed in the destination render target. As an example, a standard HD resolution of 1280×720 would take 15 passes with a tile size of 256×240 for the screen to be fully covered. Figure 2.3 shows a scene with translucent contributions yet to be factored in. Figure 2.4 shows the same scene with translucency rendered on top using a set of tile-sized rectangular regions covering the whole render area.

2.9.2 Transformation Overhead

Because multiple passes are required, any objects crossing tile boundaries has to be transformed more than once (once for each tile-sized screen region they contribute to), which can impact performance when objects with higher transformation costs (complex vertex skinning, advanced tessellation algorithms, use of long geometry shaders, etc.) are employed. Ideally, translucent geometry should be decomposed into subobjects to minimize transformation overlap between tile regions. Smaller objects such as particles will particularly benefit from the tiling approach, since overlap between tile regions is likely to be minimal (unless the tiles themselves are small), and boundaries between particles belonging to different rectangle regions are well defined.

2.9.3 Tile Size

Tile size can be arbitrary; it may be a multiple of the render target dimensions, but this is not a strict requirement. Typically tile size will be dictated by the amount of free video memory available for OIT, since larger dimensions will lead once again to significant amounts of memory being consumed. There is also a direct correlation between tile size and performance, since the smaller the tile size, the higher the number of passes required to cover the render area. Thus, it is generally better to allocate a larger tile size if memory can be spared for this purpose.

2.9.4 Minimum Rectangle Optimization

It is not necessary to cover the *full* render target area with multipass tiling. Instead one needs to cover only the *minimum* rectangle area of the render target that will actually receive translucent contributions. This minimum rectangle area can be determined by using bounding volumes transformed to screen space in order to retrieve the 2D screen coordinates extents of all translucent geometry. Once the bounding volumes of all translucent meshes have been transformed,

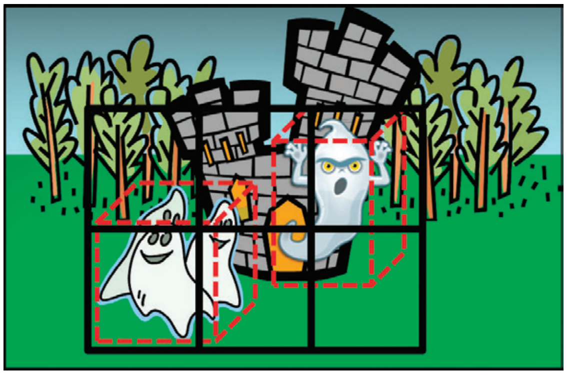

Figure 2.5. Translucent contribution to the scene is added to the render target via optimized tiling. Only tiles enclosing the bounding geometry of the translucent characters are processed. The bounding boxes of translucent geometry are transformed to screen space and the combined extents of the resulting coordinates define the minimum rectangle area that will be processed. This minimum rectangle area is covered by as many tile-sized rectangular regions as required (six in this example). Each of those regions performs fragment storing and rendering using a single pair of tile-sized head pointers and nodes buffers.

the minimum and maximum dimensions (in X and Y) of the combined set will define the rectangle area of translucent contributions. The minimum rectangle optimization typically allows a reduction in the number of tiles to process when parsing and rendering fragments from linked lists. In order to render a minimum number of tiles, it is desirable to ensure that the bounding geometry used is as tight as possible; for example, axis-aligned bounding boxes are likely to be less effective than arbitrary-aligned bounding boxes or a set of bounding volumes with a close fit to the meshes involved.

Because this optimization covers only a portion of the screen, the previous contents of the render target will need to be copied to the destination render target, at least for those regions that do not include translucent contribution. This copy can be a full-size copy performed before the OIT step, or stencil-based marking can be used to transfer only the rectangle regions that did not contain any translucency.

Figure 2.5 illustrates the optimized covering of tiles to cover only the 2D extents of translucent contributions to the scene.

2.10 Conclusion

The OIT algorithm presented in this chapter allows significant performance savings compared to other existing techniques. The technique is also robust, allowing different types of translucent materials to be used as well as supporting hardware multisampling antialiasing. Although video memory requirements can become unreasonable when larger viewports are employed, the use of tile optimizations allows a trade-off between memory and performance while still retaining the inherent advantages of this technique. The use of per-pixel linked lists can be adapted to techniques other than order-independent transparency, because they can store a variety of per-pixel data for arbitrary purposes: see Part III, Chapter 3 in this volume for an example.

2.11 Acknowledgments

I would like to thank Holger Gruen and Jakub Klarowicz for coming up with the original concept of creating per-pixel linked lists in a DirectX 11-class GPU environment.

Bibliography

[Bavoil 08] Louis Bavoil and Kevin Myers. "Order-Independent Transparency with Dual Depth Peeling." White paper available online at http://developer.download.nvidia.com/SDK/10/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf, 2008.

[Carpenter 84] L. Carpenter. "The A-Buffer, An Antialiased Hidden Surface Method." *Computer Graphics (SIGGRAPH)* 18:3 (1984), 103–108.

[Everitt 01] Cass Everitt. "Interactive Order-Independent Transparency." White paper available online at http://developer.nvidia.com/object/Interactive_Order_Transparency.html, 2001.

[Myers 07] Kevin Myers and Louis Bavoil. "Stencil Routed A-Buffer." *SIGGRAPH '07: ACM SIGGRAPH 2007 Sketches*, Article 21. New York: ACM, 2007.

[Thibieroz 08] Nicolas Thibieroz. "Robust Order-Independent Transparency via Reverse Depth Peeling," In *ShaderX*^{6}, edited by Wolfgang Engel, pp. 211–226. Hingham, MA; Charles River Media, 2008.

[Thibieroz 10] Nicolas Thibieroz and Holger Gruen. "OIT and Indirect Illumination using DX11 Linked Lists," *GDC 2010 Presentation from the Advanced D3D Day Tutorial.* Available online at http://developer.amd.com/documentation/presentations/Pages/default.aspx, 2010

[Yang 10] Jason Yang, Justin Hensley, Holger Gruen, and Nicolas Thibieroz. "Real-Time Concurrent Linked List Construction on the GPU. *Computer Graphics Forum, Eurographics Symposium on Rendering 2010* 29:4 (2010), 1297–1304.