Overview

Per-pixel linked-list transparency is a simple way to implement order-independent transparency (OIT).

Here are the steps after opaque surfaces and the skybox are drawn:

1. Forward rendering of transparent surfaces.
    1. The final color is computed as usual.
    2. Instead of rendering to a render target, the final color and depth (and potentially other useful bits of information) are written to the fragment linked-list of the current pixel.
2. Post-process resolve
    1. Each pixel loads its linked-list into an array.
    2. The array is sorted by depth back-to-front.
    3. The fragments are blended in order against the opaque color.

Fragment storage

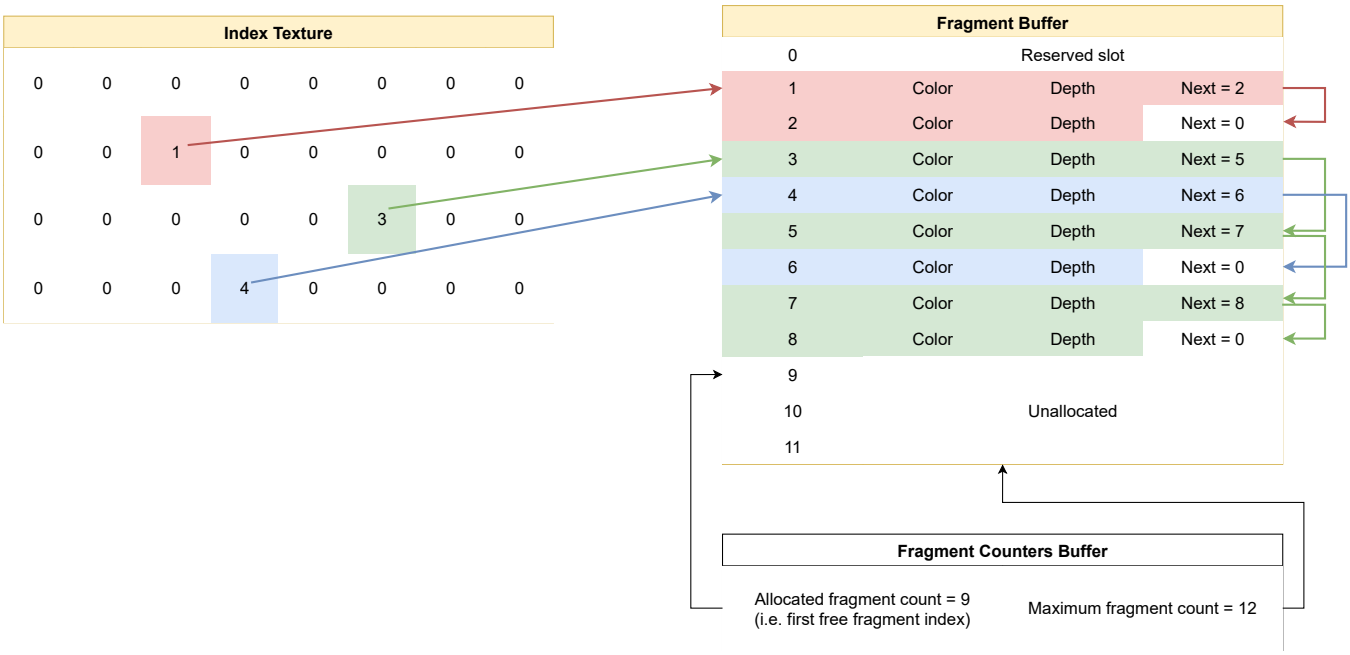
How is the per-pixel linked list stored?

- The *fragment buffer* contains all fragments of the scene view (i.e. a big array of fragments).
- The *counter buffer* has a fragment counter used to allocate fragments from the fragment buffer (using atomic increments).
- The 2D R32_UINT *index texture* stores the first index (i.e. the head "pointer") of each pixel's linked list.

```
struct OIT_Counter
{
    uint fragmentCount;    // number of allocated fragments
    uint maxFragmentCount; // size of the fragment buffer
    // extra: overflow counter, ...
};

struct OIT_Fragment
{
    uint packedColor;
    float viewDepth;        // linear depth, higher = further from the camera
    uint nextFragmentIndex; // index into the fragment buffer, 0 means end-of-list
    // extra: blend mode, material index, depth fade settings, ...
};

RWTexture2D<uint> indexTexture;
RWStructuredBuffer<OIT_Fragment> fragmentBuffer;
RWStructuredBuffer<OIT_Counter> counterBuffer;
```



Code

```
// step 1:
// - clear the index texture to 0 (index 0 means end-of-list)
// - set the fragment count in the counter buffer to 1 (since index 0 is reserved
for end-of-list)

// step 2: geometry pass
[earlydepthstencil]
void PSMain(VOut input)
{
    RWStructuredBuffer<OIT_Counter> counterBuffer =
ResourceDescriptorHeap[counterBufferIndex];

    // compute color and apply alpha-test as usual
    float4 finalColor = ShadePixel(input);

    // atomically allocate a fragment
    uint fragmentIndex;
    InterlockedAdd(counterBuffer[0].fragmentCount, 1, fragmentIndex);

    // was the allocation successful?
    if(fragmentIndex < counterBuffer[0].maxFragmentCount)
    {
        RWTexture2D<uint> indexTexture =
ResourceDescriptorHeap[indexTextureIndex];
        RWStructuredBuffer<OIT_Fragment> fragmentBuffer =
ResourceDescriptorHeap[fragmentBufferIndex];

        // atomically update the head of the linked-list
        uint prevFragmentIndex;
        InterlockedExchange(indexTexture[int2(input.position.xy)], fragmentIndex,
prevFragmentIndex);
```

```
        // write the fragment
        OIT_Fragment fragment;
        fragment.packedColor       = PackColor(finalColor);
        fragment.viewDepth         = input.depthVS;
        fragment.nextFragmentIndex = prevFragmentIndex;
        fragmentBuffer[fragmentIndex] = fragment;
    }
}

// step 3: full-screen resolve pass
void PSMain(VOut input)
{
    Texture2D<uint> indexTexture = ResourceDescriptorHeap[indexTextureIndex];
    StructuredBuffer<OIT_Fragment> fragmentBuffer =
ResourceDescriptorHeap[fragmentBufferIndex];
    Texture2D<float4> backgroundTexture =
ResourceDescriptorHeap[backgroundTextureIndex];

    // load this pixel's fragments into a local array
    OIT_Fragment fragments[OIT_MAX_FRAGMENTS_PER_PIXEL];
    uint fragmentIndex = indexTexture[int2(input.position.xy)];
    uint fragmentCount = 0;
    while(fragmentIndex != 0 && fragmentCount < OIT_MAX_FRAGMENTS_PER_PIXEL)
    {
        fragments[fragmentCount] = fragmentBuffer[fragmentIndex];
        fragmentIndex = fragments[fragmentCount].next;
        fragmentCount++;
    }

    // sort the fragments using an insertion sort
    // since we want back-to-front compositing,
    // we need the fragments in order of decreasing view depth
    for(uint i = 1; i < fragmentCount; i++)
    {
        OIT_Fragment insert = fragments[i];
        uint j = i;
        while(j > 0 && insert.depth > sorted[j - 1].depth)
        {
            fragments[j] = fragments[j - 1];
            j--;
        }
        fragments[j] = insert;
    }

    // blend the fragments against the background
    float4 color = backgroundTexture[int2(input.position.xy)]; // opaque/skybox
background color
    for(uint i = 0; i < fragmentCount; i++)
    {
        OIT_Fragment fragment = fragments[i];
        color = Blend(UnpackColor(fragment.color), color);
    }
```

```
    return color;
}
```

It's really not necessary to have the counters in their own buffer but it's simple and convenient for the code above.

Trade-offs

Good:

- Perfect OIT (unless you run out of memory)
- Very simple

Bad:

- Unbounded memory usage: missing fragments during resolve when running out of memory
- Highly variable performance due to lists having very different lengths

Reference

Real-Time Concurrent Linked List Construction on the GPU
by Jason C. Yang, Justin Hensley, Holger Grün, Nicolas Thibieroz