

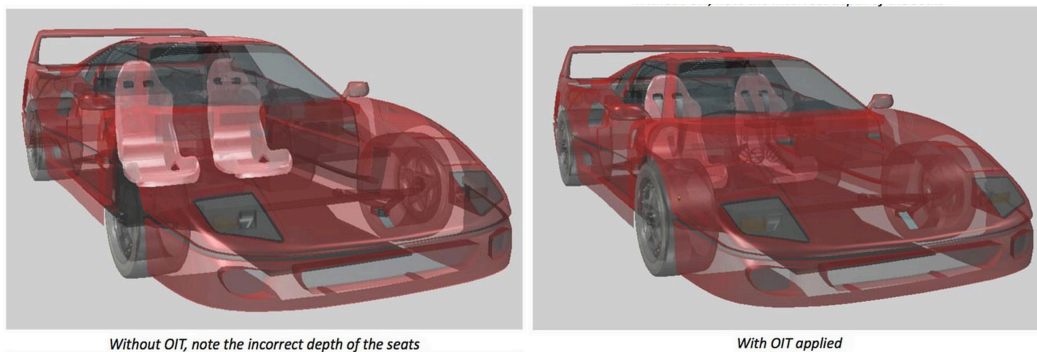
Spring 2015 CS 380 GPGPU Programming Final Project
Order Independent Transparency via Linked List Sorting on GPU

Gang Liao * ID: 133267

Friday 8th May, 2015

1 MOTIVATION

Transparency is a difficult effect to render accurately in pipeline architectures like OpenGL. Opaque objects can just cover the background. But for translucent objects, they must render by blending with the "background". Translucent objects must be rendered from far to near. It's very complex and complicated to render pixels from far to near. A possible solution to this issue is Order Independent Transparency technique (See Fig. 1).



*Extreme Computing Research Center, Department of Computer Science, King Abdullah University of Science and Technology (KAUST). Email: liao.gang@kaust.edu.sa

2 METHOD: LINKED LIST SORTING

Order Independent Transparency is drawing objects in any order that can get accurate results. In fragment shader, depth sorting can be done so that the programmer need not sort objects before rendering. There are a variety of techniques for doing this. The most common one is to keep a list of colors for each pixel, sort them by depth, and then blend them together in the fragment shader.

The algorithm can be expressed as three phases:

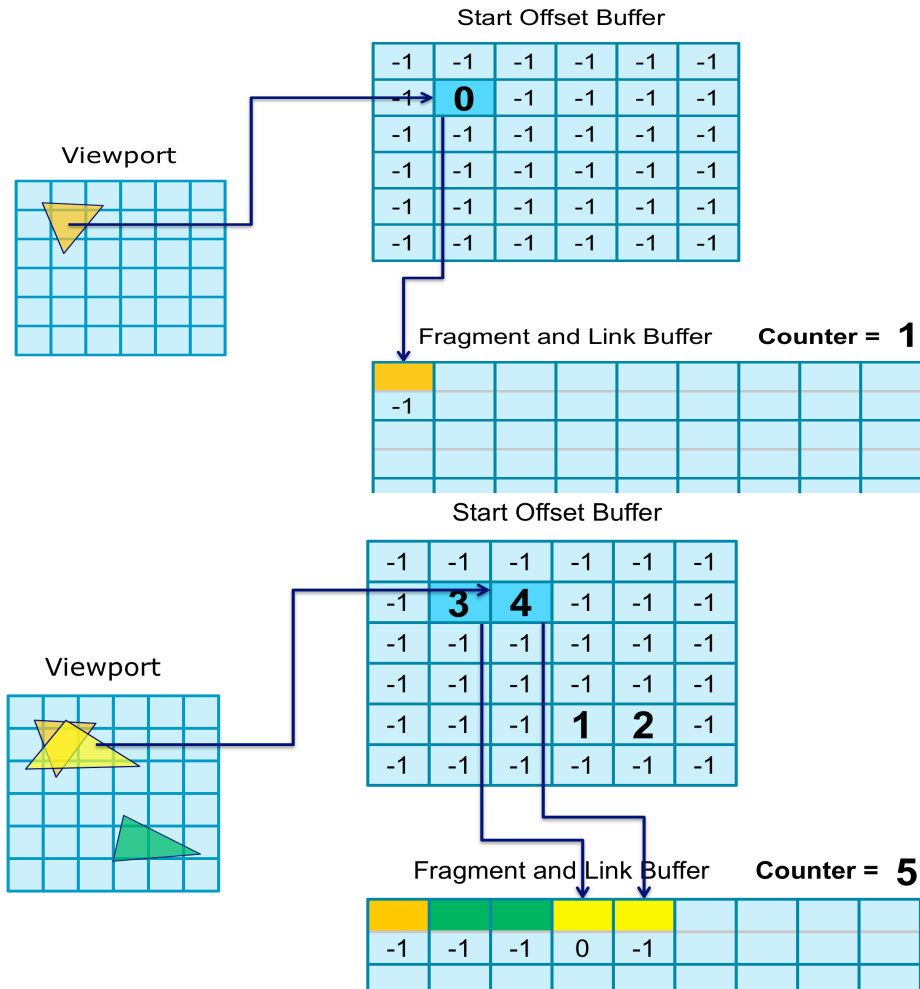
1. Render opaque scene objects.
2. Render transparent scene objects.
 - (a) All fragments are stored using per-pixel linked lists.
 - (b) Store fragments: color, alpha, & depth.
3. Screen quad resolves and composites fragment lists.
 - (a) each pixel in fragment shader sorts associated linked list (e.g., insertion sort, quick sort, etc.)
 - (b) blending fragments in sorted order with background
 - (c) output final fragment

ATOMIC COUNTER AND BUFFERS

An atomic counter is used to keep track of the size of our linked list buffer. We also need to two buffers:

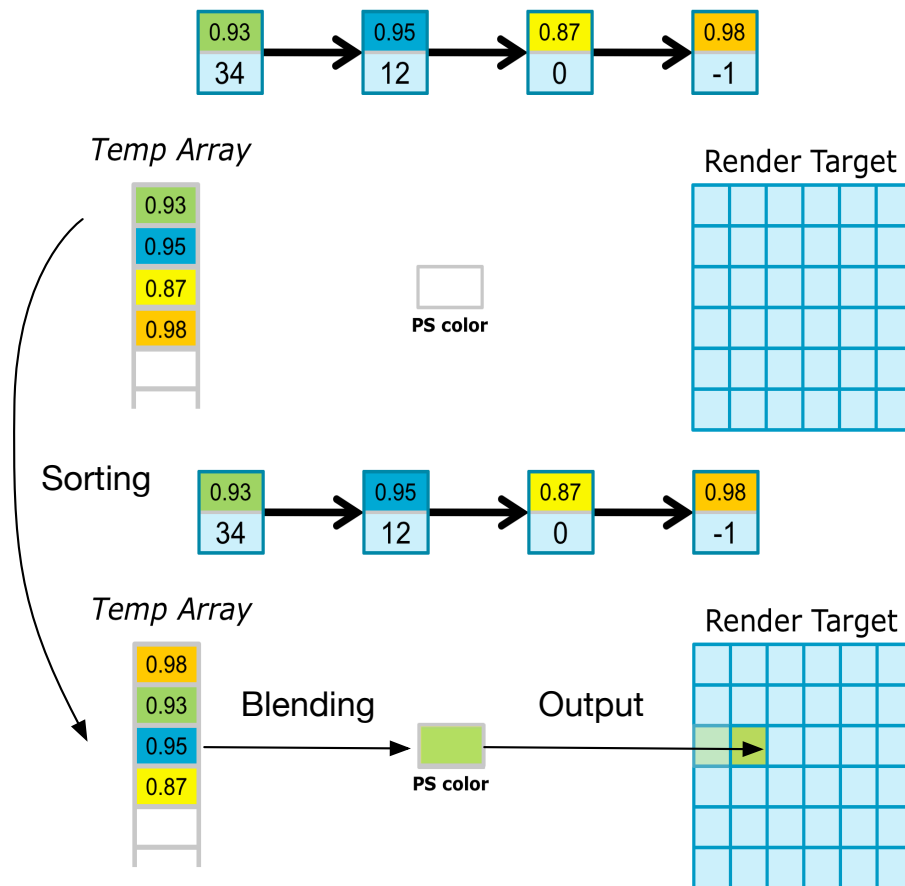
1. Screen sized head pointer buffer;
2. A buffer containing all of our linked lists - large enough to handle all fragments.

It works as follows (see Fig. 2):



LINKED LIST SORTING

When all fragments have been stored in the buffer, then we can sort linked list based on each pixel (see Fig. 3). You can use any kind of sorting algorithms you want to explore OIT technique.



3 IMPLEMENTATION

CONFIGURATION

I use the **glsprorgam** class from GLSL Cookbook to load, compile and link the vertex shader and fragment shader.

```

1 try {
2     prog.compileShader("shader/oit.vs");
3     prog.compileShader("shader/oit.fs");
4     prog.link();
5     prog.use();
6 } catch(GLSLProgramException &e) {
7     cerr << e.what() << endl;
8     exit( EXIT_FAILURE );
9 }

```

Several objects have been created to show the correctness of the final effects of order independent transparency.

```
1  cube = new VBOCube();
2  sphere = new VBOSphere(1.0f, 40, 40);
3  //plane = new VBOPlane(10.0f, 10.0f, 2.0f, 2.0f);
4  torus = new VBOTorus(0.7f, 0.3f, 30, 30);
5  ogre = new VBOMesh("../media/bs_ears.obj");
```

To guarantee the buffer is enough to cover all fragments, I set one buffer with very large memory which is one of limitations for OIT using linked lists.

Several objects have been created to show the correctness of the final effects of order independent transparency.

```
1  GLuint maxNodes = 20 * width * height;
2  // The size of a linked list node
3  GLint nodeSize = 5 * sizeof(GLfloat) + sizeof(GLuint);
```

The following code is to create atomic counter and buffers.

```
1  enum BufferNames {
2      COUNTER_BUFFER = 0,
3      LINKED_LIST_BUFFER
4  };
5  ...
6  // Our atomic counter
7  glBindBufferBase(GL_ATOMIC_COUNTER_BUFFER, 0, buffers[COUNTER_BUFFER
8      ]);
9
10  glBufferData(GL_ATOMIC_COUNTER_BUFFER, sizeof(GLuint), NULL,
11      GL_DYNAMIC_DRAW);
12
13  // The buffer for the head pointers, as an image texture
14  glGenTextures(1, &headPtrTex);
15  glBindTexture(GL_TEXTURE_2D, headPtrTex);
16  glTexStorage2D(GL_TEXTURE_2D, 1, GL_R32UI, width, height);
17  glBindImageTexture(0, headPtrTex, 0, GL_FALSE, 0, GL_READ_WRITE,
18      GL_R32UI);
19
20  // The buffer of linked lists
21  glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, buffers[
22      LINKED_LIST_BUFFER]);
23  glBufferData(GL_SHADER_STORAGE_BUFFER, maxNodes * nodeSize, NULL,
24      GL_DYNAMIC_DRAW);
```

FRAGMENT SHADER: CREATE LINKED LISTS

To increase the counter, we need to do this operation atomically. Due to the highly parallel nature of GPUs, fragments can be rendered in virtually any order. Every time we must allow one to modify the counter.

```
1 // Get the index of the next empty slot in the buffer
2 uint nodeId = atomicCounterIncrement(nextNodeCounter);
```

After that, we can change the value of the location was pointed by headPointers and the information of current node.

```
1 uint prevHead = imageAtomicExchange(headPointers, ivec2(gl_FragCoord.
    xy), nodeId);
2 nodes[nodeId].color = vec4(diffuse(), Kd.a);
3 nodes[nodeId].depth = gl_FragCoord.z;
4 nodes[nodeId].next = prevHead;
```

FRAGMENT SHADER: SORTING

To make sure all fragments have been stored in the buffer, we have to use a barrier to wait until accomplishment.

```
1 glMemoryBarrier( GL_SHADER_STORAGE_BARRIER_BIT );
```

For each pixel, according to the coordinate, we can get the start index in the linked list buffer. If index is not equal to -1 (unsigned int: 0xffffffff), the nodes of each linked list can be stored into an array.

```
1 // Copy the linked list for this fragment into an array
2 while( n != 0xffffffff && count < MAX_FRAGMENTS ) {
3     frags[count] = nodes[n];
4     n = frags[count].next;
5     count++;
6 }
```

SORTING ALGORITHMS

To sort an array, insert sort, select sort, merge sort and shell sort all should be a good fit.

For **insert sort**,

```

1 // Sort the array by depth using insertion sort (largest
2 // to smallest).
3 for( uint i = 1; i < count; i++ )
4 {
5     NodeType toInsert = frags[i];
6     uint j = i;
7     while( j > 0 && toInsert.depth > frags[j-1].depth ) {
8         frags[j] = frags[j-1];
9         j--;
10    }
11    frags[j] = toInsert;
12 }

```

For select sort,

```

1 int max;
2 NodeType tempNode;
3 int j, i;
4 for( j = 0; j < count - 1; j++)
5 {
6     max = j;
7     for( i = j + 1; i < count; i++)
8     {
9         if(frags[i].depth > frags[max].depth)
10            max = i;
11    }
12    if(max != j)
13    {
14        tempNode = frags[j];
15        frags[j] = frags[max];
16        frags[max] = tempNode;
17    }
18 }

```

For bubble sort,

```

1 int j, i;
2 NodeType tempNode;
3 for(i = 0; i < count - 1; i++)
4 {
5     for(j = 0; j < count - i - 1; j++)
6     {

```

```

7         if (frags[j].depth < frags[j+1].depth)
8         {
9             tempNode = frags[j];
10            frags[j] = frags[j+1];
11            frags[j+1] = tempNode;
12        }
13    }
14 }

```

For **merge sort**, since shader doesn't allow recursive function, it's a little bit hard to implement the divide and conquer algorithm (merge sort).

```

1     int i, j1, j2, k;
2     int a, b, c;
3     int step = 1;
4     NodeType leftArray[MAX_FRAGMENTS/2]; //for merge sort
5
6     while (step <= count)
7     {
8         i = 0;
9         while (i < count - step)
10        {
11            a = i;
12            b = i + step;
13            c = (i + step + step) >= count ? count : (i + step + step);
14
15            for (k = 0; k < step; k++)
16                leftArray[k] = frags[a + k];
17
18            j1 = 0; j2 = 0;
19            for (k = a; k < c; k++)
20            {
21                if (b + j1 >= c || (j2 < step && leftArray[j2].depth >
22                    frags[b + j1].depth))
23                    frags[k] = leftArray[j2++];
24                else
25                    frags[k] = frags[b + j1++];
26            }
27            i += 2 * step;
28        }
29        step *= 2;
30    }

```


For shell sort,

```
1 //shell sort
2 int i, j;
3 NodeType tmp;
4 int inc = count / 2;
5 while (inc > 0)
6 {
7     for (i = inc; i < count; i++)
8     {
9         tmp = frags[i];
10        j = i;
11        while (j >= inc && frags[j - inc].depth < tmp.depth)
12        {
13            frags[j] = frags[j - inc];
14            j -= inc;
15        }
16        frags[j] = tmp;
17    }
18    inc = int(inc / 2.2 + 0.5);
19 }
```

BLENDING

When sorted array is done, traversing the array and combining the colors using alpha channel.

```
1 vec4 color = vec4(0.5, 0.5, 0.5, 1.0);
2 for( int i = 0; i < count; i++ )
3 {
4     color = mix( color, frags[i].color, frags[i].color.a);
5 }
6 // Output the final color
7 FragColor = color;
```

4 RESULT

For the dynamic effects of order independent transparency, you can browse it from my github page (<https://github.com/GangLiao/Order-Independent-Transparency-GPU>).

