King Abdullah University of Science and Technology

Spring 2015 CS 380 GPGPU Programming Final Project

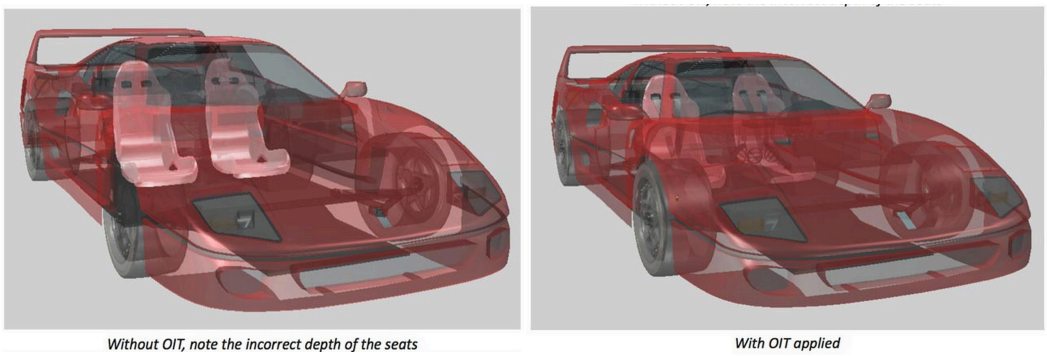
Order Independent Transparency via Linked List Sorting on GPU

Gang Liao * ID: 133267

Sunday 17th 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).



Without OIT, note the incorrect depth of the seats

With OIT applied

*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:

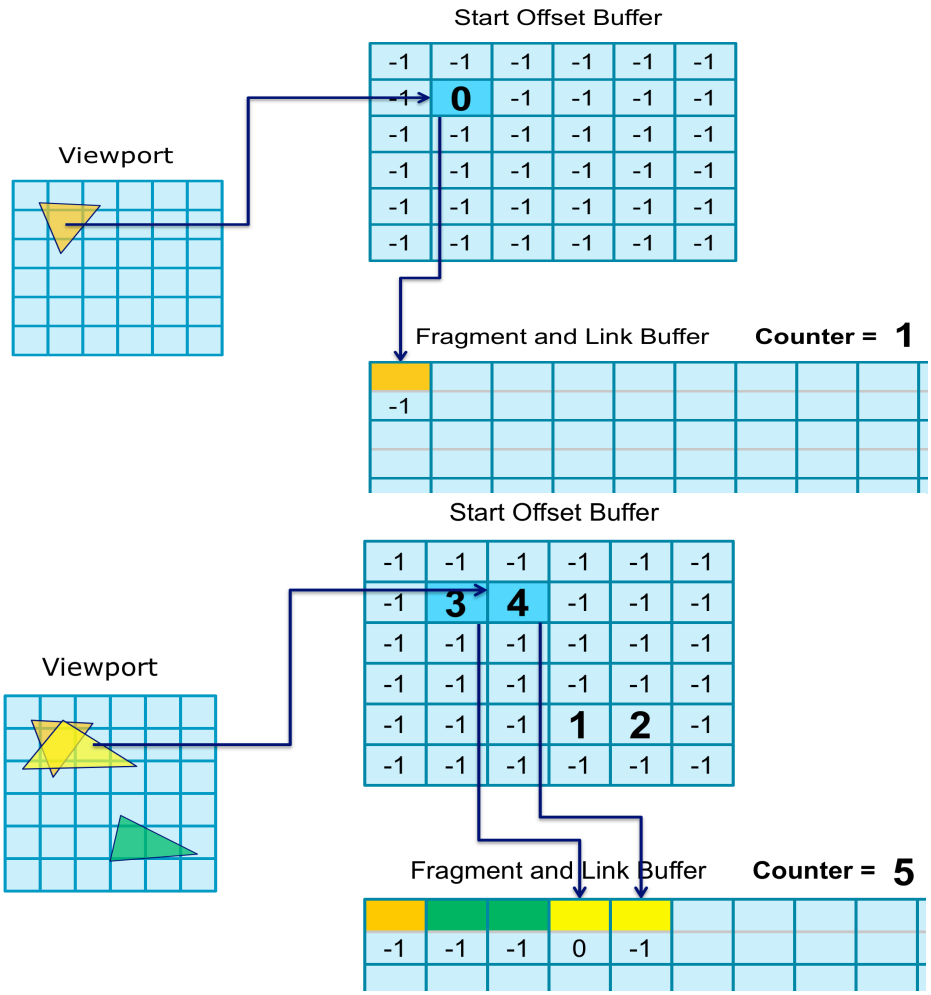
1. Render opaque scene objects.
2. Render transparent scene objects.
   (a) All fragments are stored using per-pixel linked lists.
   (b) Store fragments: color, alpha, & depth.
3. Screen quad resolves and composites fragment lists.
   (a) each pixel in fragment shader sorts associated linked list (e.g., 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:

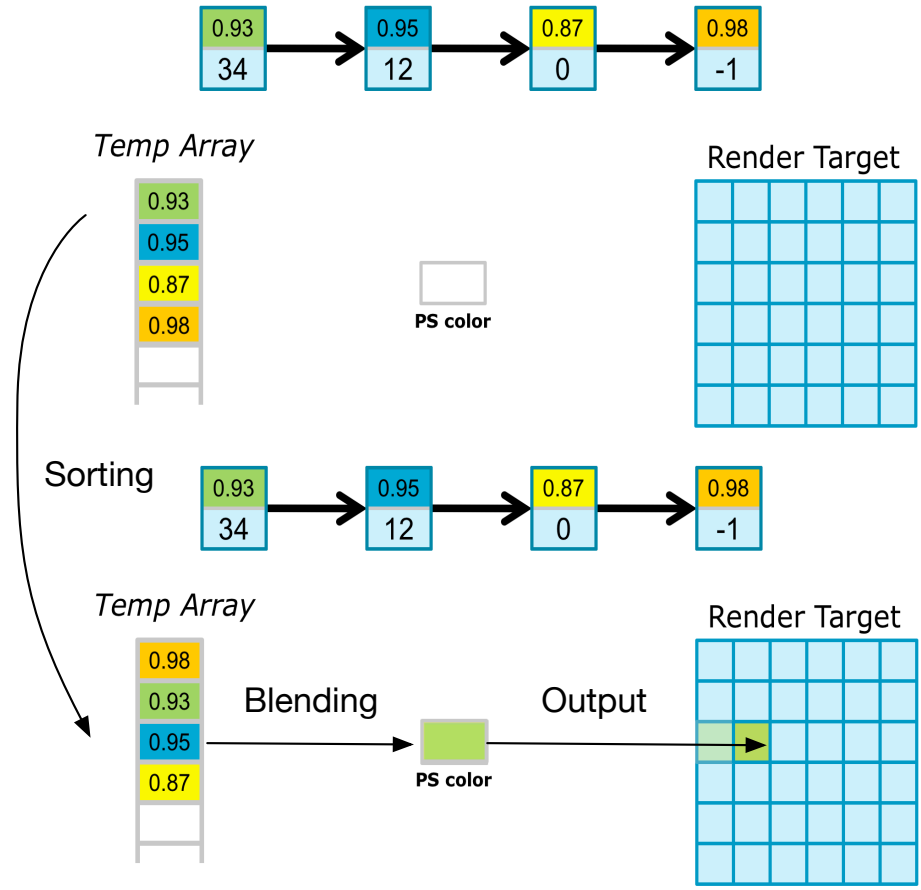
1. Screen sized head pointer buffer;
2. A buffer containing all of our linked lists - large enough to handle all fragments.

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

```
try {
    prog.compileShader("shader/oit.vs");
    prog.compileShader("shader/oit.fs");
    prog.link();
    prog.use();
  } catch(GLSLProgramException &e ) {
    cerr << e.what() << endl;
    exit( EXIT_FAILURE );
 }
```

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

```
cube = new VBOCube();
sphere = new VBOSphere(1.0f, 40, 40);
//plane = new VBOPlane(10.0f, 10.0f, 2.0f, 2.0f);
torus = new VBOTorus(0.7f, 0.3f, 30, 30);
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.

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

The following code is to create atomic counter and buffers.

```
enum BufferNames {
  COUNTER_BUFFER = 0,
  LINKED_LIST_BUFFER
};
...
 // Our atomic counter
 glBindBufferBase(GL_ATOMIC_COUNTER_BUFFER, 0, buffers[COUNTER_BUFFER
    ]);
 glBufferData(GL_ATOMIC_COUNTER_BUFFER, sizeof(GLuint), NULL,
    GL_DYNAMIC_DRAW);

 // The buffer for the head pointers, as an image texture
 glGenTextures(1, &headPtrTex);
 glBindTexture(GL_TEXTURE_2D, headPtrTex);
 glTexStorage2D(GL_TEXTURE_2D, 1, GL_R32UI, width, height);
 glBindImageTexture(0, headPtrTex, 0, GL_FALSE, 0, GL_READ_WRITE,
    GL_R32UI);

 // The buffer of linked lists
 glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, buffers[
    LINKED_LIST_BUFFER]);
 glBufferData(GL_SHADER_STORAGE_BUFFER, maxNodes * nodeSize, NULL,
    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.

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

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

```
uint prevHead = imageAtomicExchange(headPointers, ivec2(gl_FragCoord.
    xy), nodeIdx);
nodes[nodeIdx].color = vec4(diffuse(), Kd.a);
nodes[nodeIdx].depth = gl_FragCoord.z;
nodes[nodeIdx].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.

```
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.

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

Sorting Algorithms

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

For **insert sort**,

```
// Sort the array by depth using insertion sort (largest
// to smallest).
for( uint i = 1; i < count; i++ )
{
  NodeType toInsert = frags[i];
  uint j = i;
  while( j > 0 && toInsert.depth > frags[j-1].depth ) {
    frags[j] = frags[j-1];
    j--;
  }
  frags[j] = toInsert;
}
```

For **select sort**,

```
int max;
NodeType tempNode;
int j, i;
for(j = 0; j < count - 1; j++)
{
  max = j;
  for(  i = j + 1; i < count; i++)
  {
      if(frags[i].depth > frags[max].depth)
          max = i;
  }
  if(max != j)
  {
      tempNode = frags[j];
      frags[j] = frags[max];
      frags[max] = tempNode;
  }
}
```

For **bubble sort**,

```
int j, i;
NodeType tempNode;
for(i = 0; i < count - 1; i++)
{
  for(j = 0; j < count - i - 1; j++)
  {
```

```
        if(frags[j].depth < frags[j+1].depth)
        {
            tempNode = frags[j];
            frags[j] = frags[j+1];
            frags[j+1] = tempNode;
        }
    }
}
```

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).

```
    int i, j1, j2, k;
    int a, b, c;
    int step = 1;
    NodeType leftArray[MAX_FRAGMENTS/2]; //for merge sort

    while (step <= count)
    {
        i = 0;
        while (i < count - step)
        {
            a = i;
            b = i + step;
            c = (i + step + step) >= count ? count : (i + step + step);

            for (k = 0; k < step; k++)
                leftArray[k] = frags[a + k];

            j1 = 0;  j2 = 0;
            for (k = a; k < c; k++)
            {
                if (b + j1 >= c || (j2 < step && leftArray[j2].depth >
                    frags[b + j1].depth))
                    frags[k] = leftArray[j2++];
                else
                    frags[k] = frags[b + j1++];
            }
            i += 2 * step;
        }
        step *= 2;
    }
```

For **shell sort**,

```
//shell sort
int i, j;
NodeType tmp;
int inc = count / 2;
while (inc > 0)
{
    for (i = inc; i < count; i++)
    {
        tmp = frags[i];
        j = i;
        while (j >= inc && frags[j - inc].depth < tmp.depth)
        {
            frags[j] = frags[j - inc];
            j -= inc;
        }
        frags[j] = tmp;
    }
    inc = int(inc / 2.2 + 0.5);
}
```

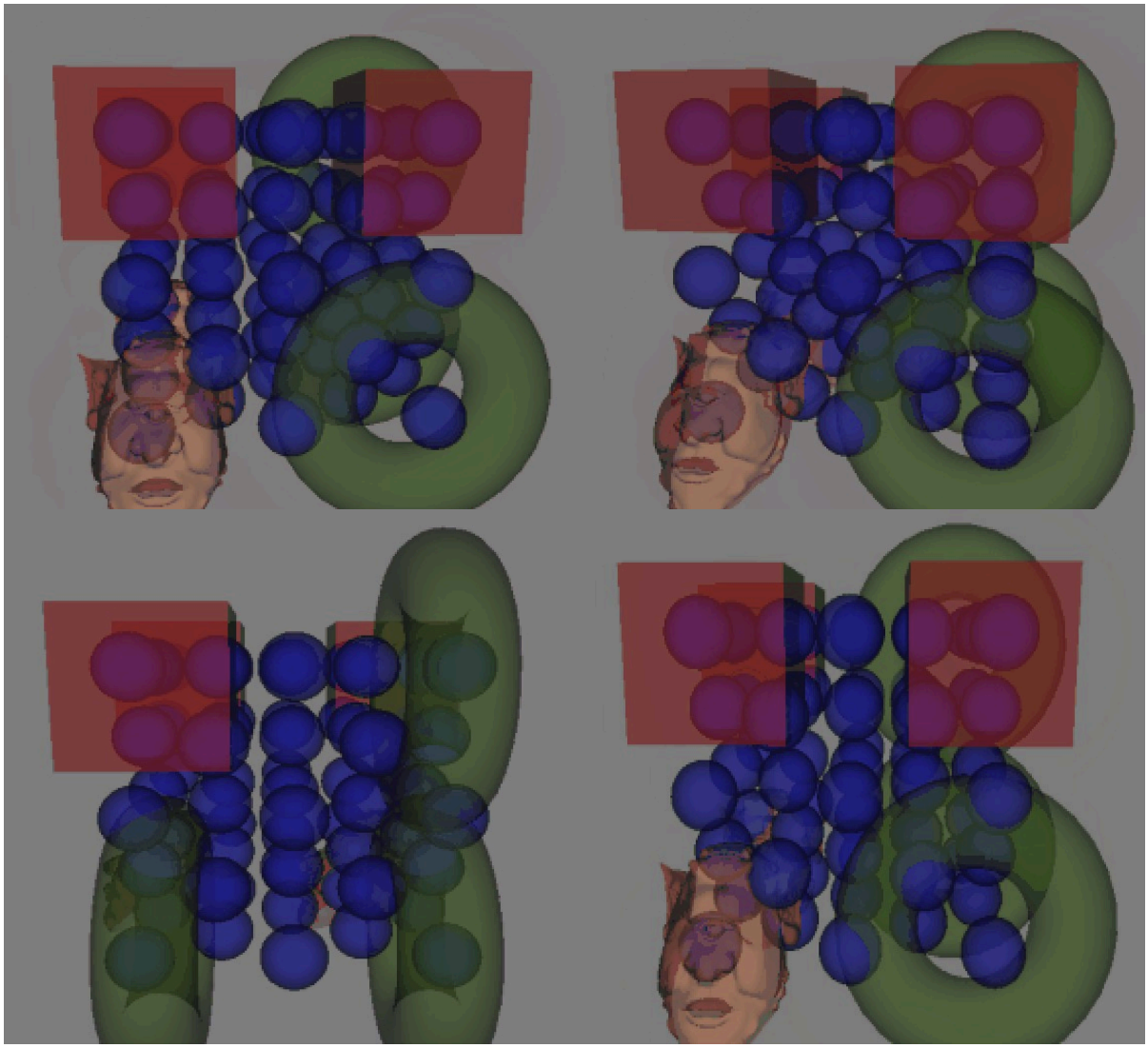
Blending

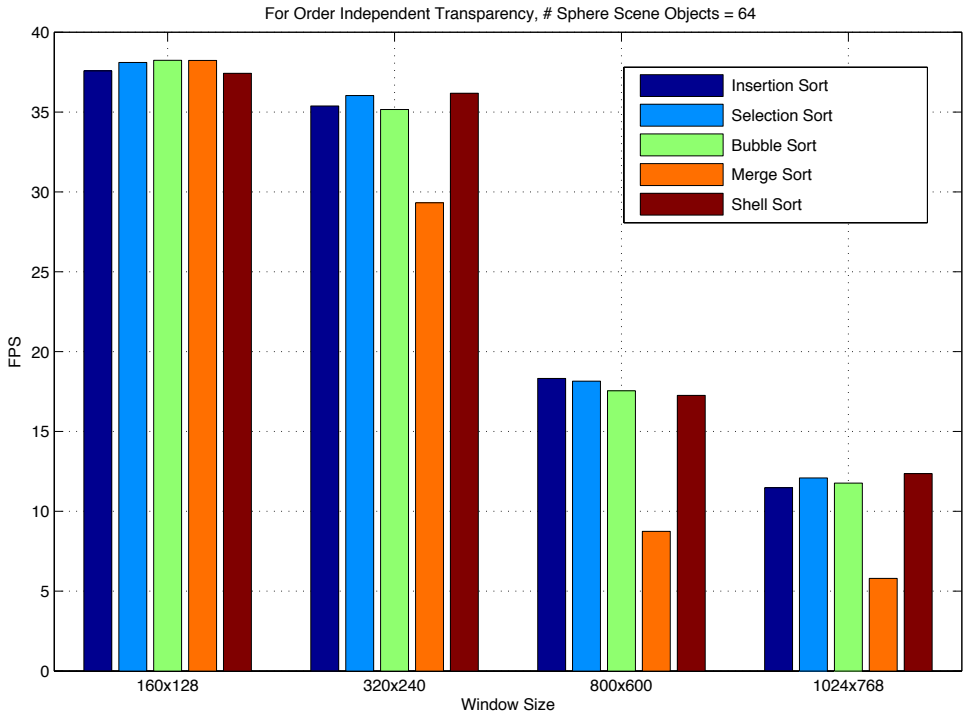
When sorted array is done, traversing the array and combining the colors using alpha channel.

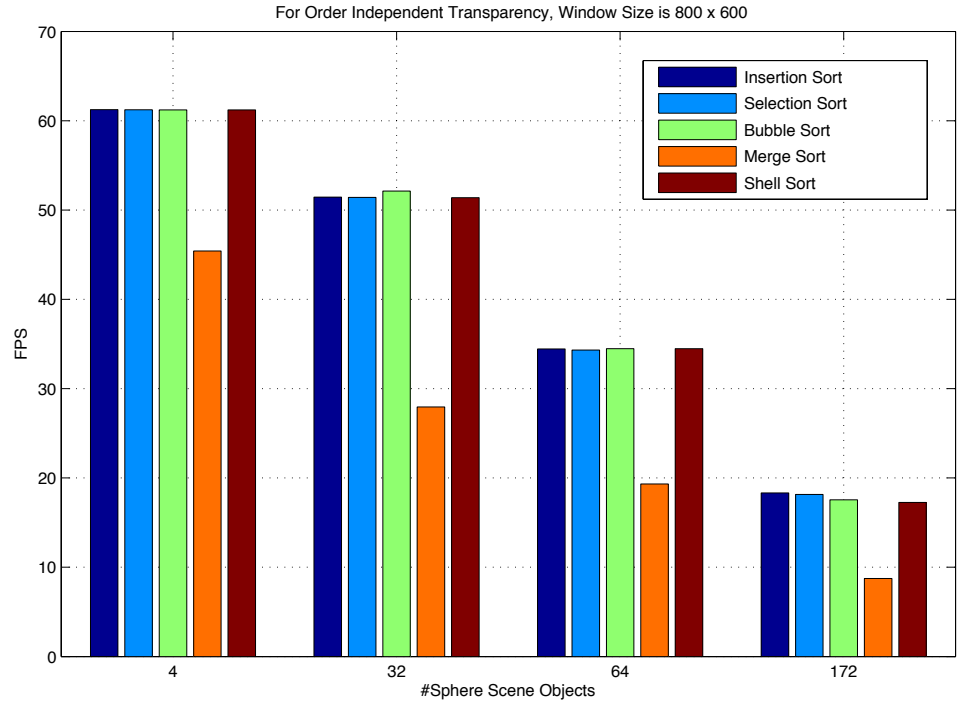
```
 vec4 color = vec4(0.5, 0.5, 0.5, 1.0);
  for( int i = 0; i < count; i++ )
  {
    color = mix( color, frags[i].color, frags[i].color.a);
  }
  // Output the final color
  FragColor = color;
```

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).




For Order Independent Transparency, # Sphere Scene Objects = 64
FPS
Window Size
160x128
320x240
800x600
1024x768
Insertion Sort
Selection Sort
Bubble Sort
Merge Sort
Shell Sort


For Order Independent Transparency, Window Size is 800 x 600
FPS
#Sphere Scene Objects
4
32
64
172
Insertion Sort
Selection Sort
Bubble Sort
Merge Sort
Shell Sort

References

[1] Christoph Kubisch, *Order Independent Transparency In OpenGL 4.x*, GTC 2014

[2] David Wolff, *OpenGL 4.0 Shading Language Cookbook*

[3] Jay McKee, *Real Time Concurrent Linked List Construction on the GPU*, AMD Technique Notes

[4] Nicolas Thibieroz, *Order-Independent Transparency using Per-Pixel Linked Lists*, EGSR 2010