Order Independent Transparency with Non-local Opacity Modulation for 3D Meshes

Tomáš Pastýřík*
Supervised by: Ladislav Čmolík†

Department of Computer Graphics and Interaction
Czech Technical University in Prague, Czech Republic

Abstract

We are rendering semi-transparent 3D objects on GPU we can choose from a variety of order independent transparency (OIT) algorithms. The transparency of the objects can be modulated based on properties of the 3D objects such as curvature, distance from silhouette, distance from camera, etc. In this paper we focus on non-local opacity modulation where desired information needed for the modulation is a matter of global context and it is not known for current primitive directly. We introduce an algorithm to solve the Order Independent Transparency with non-local opacity modulation based on the Illustration Buffer. While the original Illustration Buffer is constructed from meshes of flow surfaces we focus on use with general 3D meshes.

We compare our algorithm with several OIT algorithms: depth peeling, dual depth peeling, and per pixel linked lists which provides us a deeper insight at what conditions is one algorithm better than another from the point of speed, memory consumption and effort needed to incorporate the transparency modulation based on a certain property of the 3D objects to the algorithm.

Keywords: order independent transparency, non-local opacity modulation, illustration buffer, comparison, depth peeling, dual depth peeling, per pixel linked lists, opengl

1 Introduction

We are rendering semitransparent 3D objects, the order of the rendered primitives is critical to correctly compute the final colour of each pixel. Considering only objects consisting of 3D meshes it is not trivial to determine the order of the primitives, e.g. triangles, given by distance from the camera. A group of methods that do not require the meshes to be sorted before the rendering process is called the *Order Independent Transparency (OIT)*.

When the general *OIT* problem is solved we can also consider opacity modulation techniques to enhance perception of object's inner structure. We can classify such techniques considering the knowledge of the information desired for the modulation as follows: *Local*, if information is known to a primitive - in case of this paper to a fragment - directly. This includes techniques based on the fragment lighting and shading, distance from the camera, or custom fragment properties. *Non-Local* is a complementary class to a local opacity modulation. It can be further divided to following cases: a) the information is retrievable from the direct neighbours along the surface or along the view ray. This e.g. includes modulation based on surface curvature or edges detection. b) the information is further than in case a). This e.g. includes modulation based on distance from important object features like silhouettes, etc.

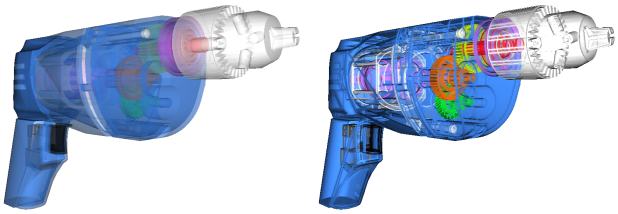

Figure 1: On the left all fragments are set 40% opacity. Non local opacity modulation (distance from silhouettes) is used on the right image to reveal inner structure of rendered drill.

Non-local opacity modulation is often needed to enable the user to see the required detail without losing the context. In this paper we introduce an OIT algorithm capable of both local and non-local opacity modulation. The algorithm is based on Illustration Buffer proposed by Carnecky et al. [3]. While they construct the Illustration Buffer from 3D meshes of the flow surfaces only our algorithm is designed for general 3D meshes. The main contributions of this paper include:

- The proposed OIT algorithm for 3D meshes along with the measurements of the algorithm stages and stages variants that provide better insight to the behaviour and performance bottlenecks of the algorithm.
- Comparison with existing algorithms solving OIT w.r.t. the speed, memory requirements, and ease of use.
- Our comparison also allows to decide which algorithm to use w.r.t. chosen the opacity modulation technique.

*mail@tomaspastyrik.cz
†cmolikl@fel.cvut.cz

2 State of the Art

Problem of the Order Independent Transparency (OIT) is well known in 3D scene rendering and there is no standard implementation included in either OpenGL or DirectX. It is a general problem consisting of rendering objects with a uniform or non-uniform alpha channel. To display such geometry correctly all fragments need to be blended in the correct order, thus sorting the fragments is often the key requirement for techniques solving OIT. In this section we briefly review algorithms that solve OIT using the rasterization proces. Please note that there are also *alpha blending approximations* [6][2] that perform approximative rendering in only one pass. Even though these approach are very fast, they only approximate the OIT problem and are not extendable to any methods considering non-local transparency and therefore these methods are not further examined in this text.

Depth Peeling presented in 2001 by C.Everitt [5] is based on multiple geometry passes, peeling just one layer of visible geometry per pass. It is in fact based on a shadow mapping technique, which helps to determine visibility between scene points and a certain light source. This algorithm process the scene by layers peeling one by one using two depth buffers per geometry pass.

While more advanced algorithms such as Dual Depth Peeling [2] blend these layers "on the fly" during the peeling passes, depth peeling algorithm [5] stores currently retrieved layer and performs another blending pass using full-screen quad, using OpenGL blending functions.

Dual Depth Peeling method by Bavoil [2] is a modification of the original Depth peeling algorithm allowing to peel two layers at once. In one pass it peels back and front layers simultaneously. Since this is not possible to do with the default depth buffer and GPU does not have multiple depth buffers to perform front to back and back to front rendering, custom min-max depth buffer has to be used.

To prevent peeling any fragments by both *front to back* and *back to front* directions, the algorithm uses mechanism of sliding window for two consecutive layers. While in the original depth peeling N geometry passes are necessary to process the scene, where N is the number of layers it created, Dual depth peeling performs $N/2+1$ geometry passes only. This algorithm however speeds up the rendering only if application is geometry (vertex) bounded.

Per Pixel Concurrent Linked Lists Another method is to store every fragment that belongs to one pixel in a linked list and sorting it by fragment's depth to determine the order of the fragments. Method [9] described below is very similar to *A-buffer* [4], it only achieves OIT by using linked lists constructed in memory of GPU. While the first GPU implementations of A-buffer presented by Meyers and Bavoil A-buffer [7] and Bavoil et al. [1] were able to store fixed amount of fragments per list, the method presented by Yang [9] is unbounded.

A GPU version of A-buffer can be constructed in two rendering passes. In the first pass we create a linked lists of fragments per pixel by rendering the geometry and storing all fragments of a pixel to the linked list. The linked list can be accessed by an index to the first fragment called the *pixel head*. The second pass consists of a full-screen quad rendering and sorting the linked lists. Traversing the sorted linked lists to compose final pixel color can be done at the end of the sorting pass thus no further passes are needed.

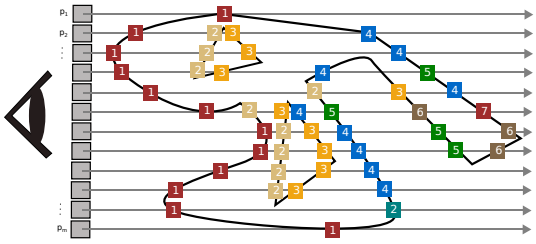

Figure 2: For the sake of simplicity *one row display* of pixels $p_i, i \in \{1,...,m\}$ where m is the number of pixels is shown. Fragments with the same number are in the case of *peeling* methods in the same layer. In case of *ray casting* terminology, numbers denote the order along the ray.

Illustration Buffer was presented by Carnecky et al. [3] and inspired by Yang [9]. The Illustration buffer data structure is motivated by several image enhancements that modulate opacity based on non-local information.

To provide the information about the surrounding shape of all fragments, A-buffer constructed in the GPU memory [9] is extended. While in A-buffer method fragments know their neighbours only along the viewing ray, Carnecky et al. present methods to find and connect also neighbours that belong to the surrounding pixels. For pixel with coordinates (x,y) new four neighbours are found in linked lists of pixels $(s+dx, y+dy)$ where $(dx,dy) \in \{(1,0),(-1,0),(0,1),(0,-1)\}$. After the neighbours are found the Illustration buffer can be used to traverse object surfaces to retrieve information about their shape, such as gradients, or distances to important features.

Structures created by our algorithm described in the next section are the same as used by Carnecky et al. [3]. We take advantage of the indexed geometry and propose a geometry motivated method to locate the neighbours which is faster and more precise than heuristics used by Carnecky et al. [3].

3 Proposed Algorithm

In this section we describe the Illustration buffer as well as our extension of the approach. The Illustration buffer requires the per pixel concurrent linked lists to be created first. However, in contrast to the *concurrent linked lists* we need to store much more information. In the following list are the buffers we need to construct and work with the Illustration buffer:

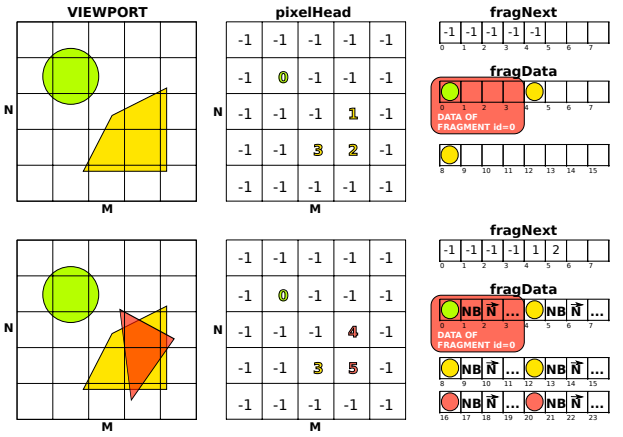

Figure 3: Shows the filling step of the algorithm. Data are spanned by four, giving a space for colour, four surrounding neighbours (NB), normal (N) and some other data, which depend on the target use of the buffer. Buffer pixelHead if of a same size as the viewport ($X \times Y$). This Figure does not consider the sorting of samples along the ray or search for the neighbouring fragments.

- **pixelHead** is a buffer of size $X \times Y$ where X, Y are the dimensions of the viewport. It stores ID of the first fragment in the linked list.
- **pixelCount** is also $X \times Y$ buffer storing lengths of the lists in each pixel. While not required when traversing the list and using special value for the end, this value is important for certain opacity modulation techniques.
- **fragNext** is a one dimensional buffer where the next index is stored for current fragment
- **fragData** stores all data we need to work with the Illustration buffer. For each fragment it stores the colour, indices of its four geodesic neighbours and optionally other data we need for non local transparency.
- **fragData2** is of the same layout as *fragData* and it is used for ping pong computational schemes
- **fragElements** stores 3 indices of the originating triangle per fragment, is used in our neighbours search

Figure 3 shows necessary structures for the *Illustration buffer* and how the data are stored when new element is rendered. To retrieve the next available free index for inserted fragment we need to use the global atomic counter `GL_ATOMIC_COUNTER_BUFFER`. We need a memory to store the indices of the fragment neighbours as well as custom fragment properties. Therefore we reserve several cells in the *fragData* buffer per fragment and use the spanning mechanism for retrieving the index to this buffer. The internal formats used by those buffers are: GL_RGBA32UI for the *fragData*, *fragData2* and *fragElements* buffers and GL_R32I for *fragNext*. The buffer *fragElements* consists of the vertex indices of the triangle it belongs to. The buffer *pixelHead* which stores pointers to first node of each per pixel linked lists is defined as GL_TEXTURE_2D with GL_R32I type of the viewport size.

3.1 Filling Step

Algorithm 1 shows the filling step of the algorithm when the geometry is rendered. Since we need to not only read but also write to the buffers in the later shader invocations, traditional GL_TEXTURE_2D cannot be used to store the data. Therefore we use the *ARB_shader_image_load_store* OpenGL extension. This extension brings functions *imageLoad(), imageStore()* and also many atomic operations *imageAtomic*()*. Packing of 4 floats $f, f \in [0,1]$ to one unsigned integer using bitwise shifts and GLSL built-in functions *floatBitsToUint* and *uintBitsToFloat* is used to reduce amount of used memory. In the beginning of each frame we reset the atomic counter and buffer *fragData* using GL_PIXEL_UNPACK_BUFFER to initial state. The lines 10 and 11 of Algorithm 1 have to be atomic to prevent read-write collisions.

3.2 Sorting

In our application two sorting methods are implemented. The first is sorting the linked list without using any auxiliary structures and in the second is used an array of fixed size for sorting. Both presented methods are invoked simply by rendering a full-screen quad with access to the algorithm buffers storing the linked lists.

Sorting the Linked List by Insertion sort can be done easily using two *fragNext* buffers. One to be filled initially and second that will be used for adding sorted fragments as shown in Algorithm 2.We can see this procedure as an analogy to two linked lists A,B. A is unsorted and B consists only from copy of head in A. Then we remove node *a* from the front of A and insert it to B. To be able to remove *a.next* from A and insert it to B, we would have to remember what was the original *a.next* since inserting the node *a* to B may change its next pointer. In single linked lists this could be solved also by copies of the nodes instead of their

Algorithm 1: Filling per pixel linked lists

```
   Data: Geometry to be rendered, atomic counter AC = 1
   Result: Unsorted concurrent linked lists
1  Render the geometry, Depth Test OFF
2  while fragments with (X,Y) to be processed do
3      index = AC
4      colour = shadeFragment()
5      if pixelHead(X,Y) == -1 then
6          pixelHead(X,Y) = index
7          fragData(index*span) = colour
8          fragNext(index*span) = -1 //next pointer is empty.
9      else
10         nextPointIndex = pixelHead(X,Y)
11         pixelHead(X,Y) = index
12         fragData(index*span) = colour
13         fragNext(index*span) = nextPointIndex
14     end
15     AC+ = 1 // increase the atomic counter
16     discard fragment // we do not want it to be seen yet.
17 end
```

removal from A. As mentioned, we solve this issue by two buffers for the next pointers. In Algoritm 2 we keep two next pointers for each node. Pointer *next* is the original next pointer and *nextSorted* guarantees that the depth of *nextSorted* is greater or equal to the current fragment.

Algorithm 2: sorting the linked lists directly

```
   Data: Buffers u_fragNext and u_fragNext2, u_fragData and u_pixelHead
   Result: Sorted next pointers in the u_fragNext2 and a head pointer in
           u_pixelHead.
1  int sortedSize = 1,int head = loadHead(x,y)
2  int newFrag = next(head); int current,int previous, float currentDepth
3  while sortedSize < totalCount do
4      if newFrag.depth < head.depth then
5          newFrag.nextSorted = head
6          head = new, head.depth = newFrag.depth
7          new = next(newFrag)
8          sortedSize++, continue
9      end
10     previous = head
11     current = head.nextSorted, int innerCounter = 0
12     while innerCounter <= sortedSize && current.depth < new.depth do
13         previous = current
14         current = current.nextSorted;
15         innerCounter++;
16     end
17     newFrag = new.next; sortedSize++;
18 end
```

Sorting in Array of Fixed Size The Number of accesses to the buffers is a bottleneck of the previous method. To eliminate the bottleneck we load all values and their next pointers to static arrays of fixed size (64 in our implementation). This array is then sorted and results are stored back to the buffers. Fragments are sorted using insertion sort as Yang [9] or selection sort as Carnecky et al. [3] (Even though selection sort should be more efficient due to number of writes over the insertion sort, no performance difference was found, presumably because of caching[12]). In comparison of the speed (Section 5) we use the selection sort.

3.3 Neighbours Location by Carnecky et al.

Let us assume we have already created the concurrent linked lists by Algorithm 1 and that the samples are already sorted along the viewing ray. This is essential to location of the neighbours.

However wanted neighbours differ greatly from the neighbours in the layers. Let us consider situation depicted in Figure 4 where we see the found neighbours and peeled layers. It shows the difference between *neighbours of P in peeled layer* and *neighbours of P on the surface* discussed later.

As shown in Figure 4 the goal is to find geodesic neighbours on the same surface and not of the same layer in the peeling point of view. We denote fragments of current linked list A as $f_i, i \in \{1,...,n\}$ where n is the number of fragments in A and neighbouring list B with fragments $f_j, j \in \{1,...,m\}$ where m is the number of fragments in B. To find the neighbour we need to traverse entire neighbouring list. Carnecky et al. use a simple heuristic measure ε of the surface continuity for two fragments as shown in

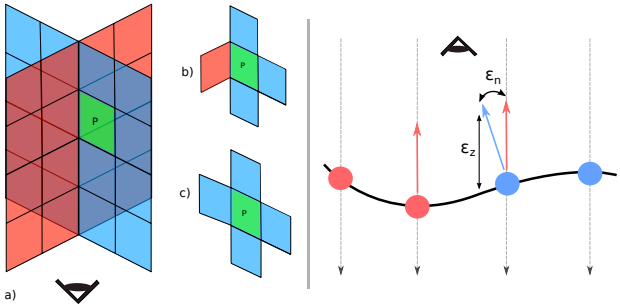

Figure 4: *Left*: Using a perspective we show two crossing planes **a)**. When peeling the first layer and querying the neighbouring pixels of P in such layer, we retrieve **b)**. When searching for neighbours of P in Illustrative buffer we want to find **c)**. *Right*: Geometrical meaning of ε_n and ε_z on the surface samples.

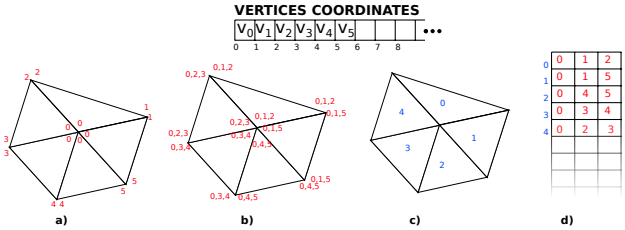

Figure 5: **a)** indexed geometry to prevent duplicate geometry to be sent to gpu. **b)** Every vertex knows indices of all vertices in given triangle. **c)** Every triangle has its own id. **d)** table maps IDs of the triangles (blue) to b) where every vertex knows all indices in its triangle.

Figure 4. They compute the ε_n as difference of fragments i, j normals n_i, n_j as:

$$\varepsilon_n(i,j) = 1 - n_i + n_j$$

The eye distance ε_z is computed using the radius of a rendered object bounding sphere r_{obj}, normal n_i, pixel coordinates x_i, eye distance z coordinate and finally the z_i gradient $\left(\frac{dz_i}{dx_i}\right)$:

$$\varepsilon_z(i,j) = \frac{1}{r_{obj}} \left[z_i + (x_j - x_i) \cdot \left(\frac{dz_i}{dx_i} \right) - z_j \right]$$

$$\varepsilon(i,j) = w_z \cdot \varepsilon_z(i,j) + w_n \cdot \varepsilon_n(i,j)$$

Given this heuristic measure ε will be small for probably neighbouring fragments of the same surface and large for fragments of different surfaces. For two neighbouring lists A, B and fragment $f_i \in A$ they first try to find the best neighbour candidate c_i for f_i in B and then they traverse A to find if there is better neighbour for c_i than f_i in A. Even though they use a component ID check to set $\varepsilon = \infty$ to exclude fragments of different components (and therefore surfaces), this is rather inefficient since for the location of one neighbour we have to traverse both the neighbouring and the original list.

3.4 Proposed Neighbours location

To overcome the inefficiency of the method [3] we propose a new method motivated by indexed geometry. Given two neighbouring lists A, B where list A is the current list and B is the list where neighbour is to be found, we propose auxiliary structure depicted in Figure 5.

We are using indexed geometry to lower the load of informations mapped to GPU memory. We extend the indices information so that every vertex knows indices of all vertices of the same triangle. This is shown in Figure 5 **b)**. However this would be against the very principle of indexed geometry since we would replicate a lot of data. This can be solved as shown in Figure 5 **c)** where every triangle has its unique *ID* attached and auxiliary table to map *IDs* to triangle indices as shown in Figure 5 **d)**.

For fragments $f \in A$ of coordinates x_f, y_f and $g \in B$ of coordinates x_g, y_g then apply following rules:

1. f and g are not neighbours if f and g do not share any indices of the triangle they are part of.
2. f and g are neighbours and fragments of the same triangle if f and g share exactly 3 indices.
3. f and g are neighbours and fragments of two neighbouring triangles if f and g share exactly 2 indices.
4. f and g are neighbours and fragments of two neighbouring triangles if f and g share exactly 1 indices. This situation can happen e.g. for triangles with $ID = 1, ID = 4$ in figure 5.

The algorithm for neighbour search is then simplified to only one cycle through the neighbouring list B and there is no need for the cycle through A afterwards.

3.4.1 Drawbacks

Even-though this method is geometry motivated there can be artifacts caused by the rasterization process. Such artifacts occur when rendered triangles are smaller than pixel and neighbouring fragments skip triangle(s). This error is shown in figure 6. With that knowledge we can higher the viewport resolution or lower the detail of the model to overcome this.

3.5 Memory consumption

Unfortunately, memory consumption is the biggest weakness of the Illustration Buffer and therefore of our algorithm as well. While in Depth Peeling and Dual Depth Peeling structures are of fixed size without any relation to the number of rendered fragments (except for the absolute size of the viewport, of course), structures *fragData* and *fragNext* of the Illustration Buffer are growing linearly based on the number of fragments.

4 Results

In this section we present the results and measurements of our algorithm. We have implemented both presented

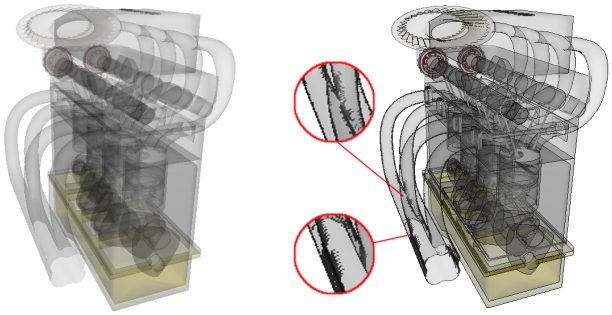

Figure 6: Every fragment has equal opacity in the left render of the engine. Right engine shows render in which fragment is fully opaque if the number of neighbours is less than four meaning it is part of the edge. Artifacts caused by the geometry detail and explained in the Section 3.4.1 are enlarged in the red circles.

Algorithm 3: proposed neighbour search

```
Data: two neighbouring lists A,B, current fragment f_i ∈ A, indices
      of f_i indicesOfF.
Result: Index to the linked list structure of f_i neighbour.
1  for b = 0; b < count(B); b++ do
2      fragB = B(b); indicesOfB = fragB.triangleIndices;
3      for k = 0; k < 3; k++ do
4          for l = 0; l < 3; l++ do
5              if indicesOfB[k] == indicesOfA[L] then
6                  return fragB.ID; // Neighbor has been found
                   since it shares at least one triangle index with f_i.
7              end
8          end
9      end
10 end
```

methods of the sorting and as the proposed neighbours search. Very precise OpenGL GPU queries are used to measure application rendering time in nanoseconds. For measurements of the total rendering time we use the `QElapsedTimer` from QT Framework. Twelve varied models are used to measure the Illustration buffer characteristics. Each model is tested in two positions - one general and one where the length of linked list is maximum possible. Note that in all measurements image resolution 600×600 is used if not stated otherwise. To see all the measurements conducted with all the tables and graphs, please see the master's thesis this paper originates from [8]. We have used GeForce GTX 660 with 2048 MB GDDR5 memory, 4×2GiB DIMM DDR and Intel Core™2 Duo CPU E6850 @ 3.00GHz for all measurements. Final renders using different non-local opacity modulations are shown in Figure 16.

4.1 Sorting

The graph in Figure 7 is very clear that the dynamic version of the sort is winning in all cases over sorting in static array. For the measurements we have used 3 arrays of size 64. One for IDs, second for the depths and third for the distances between layers. In case of our GPU it was more expensive to allocate such arrays than much bigger amount of texture reads and writes, which could differ on

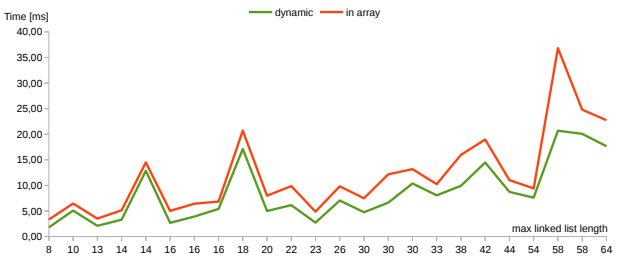

Figure 7: Speed comparison of the dynamic sort and sorting in static array. Bigger spikes are caused by the Ψ parameter which will be introduced when studying parameters that affect the performance of the algorithm the most.

hardware where invocation of the fragment shader for one pixel would have more memory available.

4.2 Illustration Buffer Performance

During the Illustration buffer creation we examine relations between number of vertices, rendered fragments, lengths of the linked lists storing the data along the ray, and rendering time on the GPU and rendering time combined with the CPU workload.

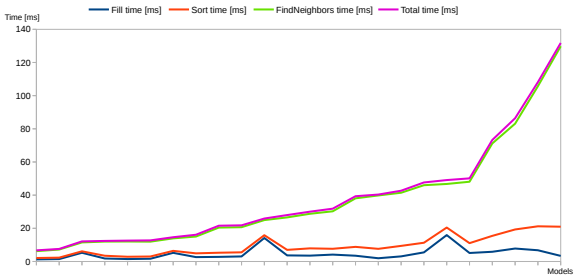

Figure 8: GPU Time in split stages of the Illustration buffer creation process. Models on the horizontal axis are sorted by the *Total* rendering time.

To be able to examine the relations fully,the creation process is split to several stages by the functionality. Figure 8 shows the performance of the separated algorithm stages. The time presented for each stage contain also times measured for all preceding stages. Figure 9 shows the relations between number of vertices, rendered fragments, lengths of the linked lists and their impact on the rendering time. We can see that another not mentioned parameter affects the rendering time significantly in Figure 9. Scenario that includes models GPU 2 and Suspension 2 is further shown in Figure 10.

Even though we process the scene on GPU, the process is not entirely parallel. All the parameters are higher for the Suspension 2 model than for GPU 2 model and yet the rendering time is greater for the GPU 2 model. The reason is that the number of long lists is much lower for the Suspension 2 model than for the GPU 2 model (see Figure 10. The dashed line in Figure 9 shows percentage coverage of the linked list lengths that are bigger than $\frac{2}{3}$ of

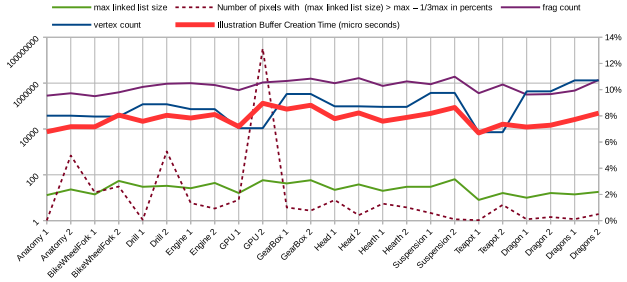

Figure 9: Secondary Y axis is used for the dashed line representing Ψ, primary Y axis (on the left) is then used for all other variables using logarithmic scale.

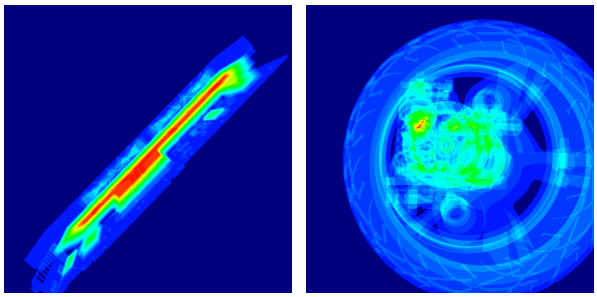

Figure 10: Heatmaps of the linked lists lengths of `GPU 2` and `Suspension 2`. Color represents the distribution of the linked lists lengths, blue is zero and red is maximal linked list length.

the longest linked list, please mind the secondary vertical axis. We denote this parameter as complexity coverage Ψ.

4.3 Memory Consumption

Using the presented structures we need 208 bytes per fragment and additional 96 bits per pixel. This gives us for the scene Suspension 2 with 1900548 fragments and 600x600 resolution memory requirements of approximately 397MB. For 800x800 resolution it requires 702MB and for 1200x1200 we would need 1.58GB of GPU memory. Given this amount of data we are very likely to face an overflow of the used buffers on memory bounded systems.

5 Comparison of OIT Algorithms

This section provides comparison of the speed of our algorithm with the remaining methods: *depth peeling, dual depth peeling* and *concurrent per pixel linked lists*.

We have used implementation of the peeling methods from the NVIDIA Graphics SDK 10, only our own GPU time measuring system was added to their implementation. Concurrent per pixel linked lists are on the other hand measured using our own implementation since it is a subproblem of the Illustration buffer construction. We have used GeForce GTX 660 with 2048 MB GDDR5 memory, 4×2GiB DIMM DDR and Intel Core™2 Duo CPU E6850 @ 3.00GHz.

Figures 11, 12, 13 and 14 show such comparison using parallel coordinates. In all presented graphs the render-

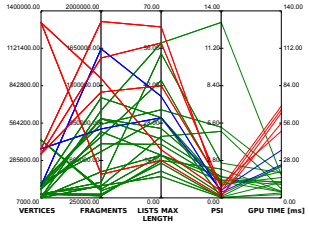

Figure 11:
Depth peeling

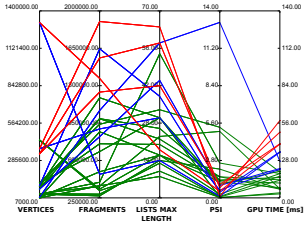

Figure 12:
Dual depth peeling

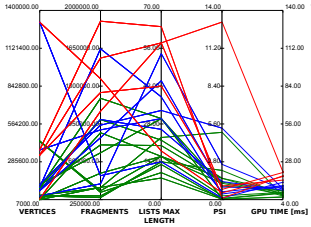

Figure 13:
Concurrent per pixel linked lists

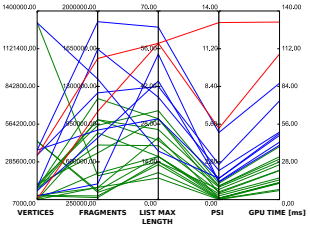

Figure 14:
Illustration buffer

ing time is divided to thirds, where times in the first third are green, in second blue and times of the worst third are coloured red.

We can see that the *linked lists* absolutely win in speed. Another observation is that only the *Illustration buffer* is really affected by the complexity coverage Ψ, which is caused by its *FindNeighbours* stage. We can also see that the overhead on the fragment shader is not big for first three methods and they are vertex bounded in most cases. This is logical for the *peeling methods* since we need to render the geometry in each peeling pass. It might be however surprising for the *per pixel linked lists*, where even though the sorting procedure must occur on all fragments, the overhead is small thus application stays vertex bounded. However this is completely different in the case of the *Illustration buffer* where processing of the *FindNeighbours* stage is vital for the final rendering time.

During our measurements, the speed comparison of the *dual depth peeling* and the *depth peeling* we have observed that the *dual depth peeling* is in fact much slower than the *depth peeling* algorithm. The authors [2] admit that the *dual depth peeling* may speed up performance by 2x for geometry bound applications. This issue should be however stressed and explained much more in the original article. Reason for this behaviour is that the workload of fragment shaders is only worth the computation if the work of the vertex shader is more demanding which is a case of vertex bounded applications.

6 Opacity Modulation

Opacity modulation is a desired feature of the OIT solving algorithms. Therefore this section briefly demonstrates the power of the proposed algorithm.

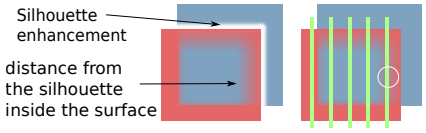

Figure 15: *Right:* Modulation by distance from the silhouettes, *Left:* Scene with the problematic situation in the white circle.

6.1 Local Opacity Modulation

When we are considering only the local opacity modulation using only information available to a fragment directly, all reviewed algorithms can be used. We only mention some of the techniques in this class for reader's convenience: based on the lighting intensity, custom per fragment data e.g. density, or by distance from defined plane/area (cut motivated). This class of modulation simply consists of all procedures that can be computed only from the global shader knowledge (uniform variables, constants, etc) and knowledge of the current fragment.

The *Depth peeling* is certainly the easiest one to be implemented but not the fastest. Dual depth peeling can speed up the rendering time only if application is vertex bounded as stated above. Implementing the per pixel linked lists is more challenging task but as we can see in the Section 5 it is certainly the fastest algorithm of all compared in this context.

6.2 Non-Local Opacity Modulation

The situation is much more challenging when the opacity of the current pixel depends on the context - on the state of its surroundings. While some opacity modulation techniques can be used quite easily using both peeling and linked lists mechanisms, some are not solvable by the peeling algorithms. Let us consider simple modulation by distance along the ray as in Equation 1. For two samples s with indices i, j where $i < j$ and therefore s_i closer to the camera, and user defined parameter $focusRegion$ we compute opacity α of sample s_i as shown in Equation 1. This is achievable by simply comparing values of last peeled and current sample in the *peeling methods* and by traversing the sorted linked list easily.

$$\alpha = saturate\left(\frac{|s_i.depth - s_j.depth|}{focusRegion}\right) \tag{1}$$

Let us now consider a modulation by distance from important features of the 3D mesh, in this case the silhouettes. If we consider the problematic situation from the Figure 15, this situation is not simply solvable by the peeling methods since neighbours of the peeled layers are not always neighbours of the same surface. We therefore cannot know if the red gradient inside the circle should continue or not since we cannot tell if the surface is continuous under the green surface or if it exists at all. This situation is however solvable easily by proposed algorithm since we have knowledge of the surface neighbours on the surface of the mesh and not only of the neighbours on the current layer.

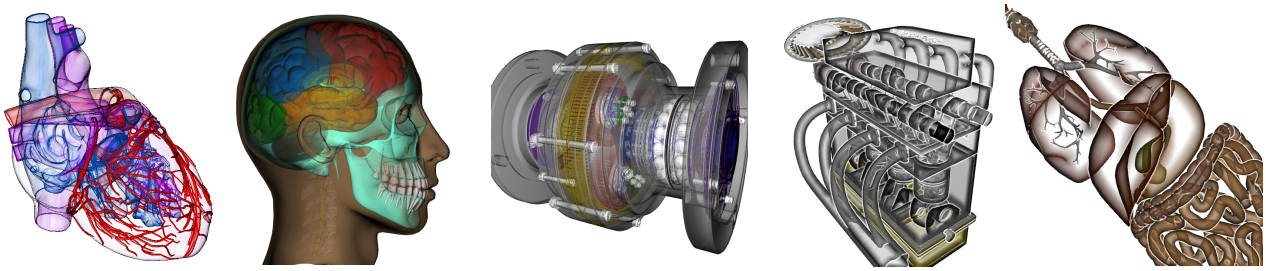

Figure 16: Final renders from our application using the Illustration buffer. These pictures were created by opacity modulation based on: surface curvature, distance between samples along the view ray, distance from surface silhouettes and silhouettes enhancements and their combinations.

7 Discussion of Results

The measurements of the OIT task alone shows clearly that methods based on the concurrent linked lists are much more efficient than the peeling methods. In case of the Illustration buffer the bottleneck is the neighbours location stage. To improve this process the future work should consider that in current form whole neighbouring list has to be traversed. We therefore suggest developing a heuristic based on the *interpolation search* optimized for the linked lists. If the approximate position of the desired neighbour is known, we can load only the next pointers to traverse to this node and omit loading its data.

Our measurements of the sorting methods shows that a bigger amount of read and write access to the OpenGL *image buffer* is more efficient than allocating an additional memory array for fragments to reduce number of *image buffer* accesses. This is a great motivation for developing dynamic algorithms that are unbounded. Proposed neighbours search method is faster and more precise than approach of Carnecky et al. [3], but it can generate artifacts as discussed above. Speeding up the neighbours search process by interpolation search might give us enough time to examine neighbours lists of distance greater than one to eliminate these artifacts.

The complexity coverage parameter Ψ impact should be further researched - how it affects the algorithm parallel computation since it demonstrates the complexity of GPU parallelization process and optimizations being done by the hardware.

8 Conclusions

In this paper we present an algorithm to solve the Order Independent Transparency for general 3D meshes that allows non-local opacity modulation. This algorithm is based on the Illustration Buffer. We discuss two methods of the fragment sorting as well as measurements of both methods. Novel geometry motivated technique to find geodesic neighbours of fragments is proposed. Our method is faster and more precise than heuristic proposed by Carnecky et al. [3]. However, further research is however required to eliminate presented artifacts.

To learn more about our algorithm we encourage the reader to read the thesis [8] this paper originates from. Figure 16 can serve as a motivation and demonstration of the Illustration buffer flexibility considering opacity modulation techniques.

The comparison of OIT algorithms provides insight into behaviour of the algorithms at different conditions and can be used to choose the right algorithm for the given conditions. While peeling methods are easier to implement, the best rendering times are achieved by implementing more complex per pixel linked lists.

References

[1] Louis Bavoil, Steven P Callahan, Aaron Lefohn, João LD Comba, and Cláudio T Silva. Multi-fragment effects on the gpu using the k-buffer. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 97–104. ACM, 2007.

[2] Louis Bavoil and Kevin Myers. Order independent transparency with dual depth peeling. Technical report, NVIDIA Corporation, 02 2008.

[3] Robert Carnecky, Raphael Fuchs, Stephanie Mehl, Yun Jang, and Ronald Peikert. Smart transparency for illustrative visualization of complex flow surfaces. *IEEE Trans. Vis. Comput. Graph.*, 19(5):838–851, 2013.

[4] Loren Carpenter. The a -buffer, an antialiased hidden surface method. *SIGGRAPH Comput. Graph.*, 18(3):103–108, January 1984.

[5] Cass Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, 06 2001.

[6] Houman Meshkin. Sort-independent alpha blending. 2007.

[7] Kevin Myers and Louis Bavoil. Stencil routed a-buffer. In *ACM SIGGRAPH*, volume 7, 2007.

[8] Tomáš Pastýřík. Visualization of inner structure of complex 3D objects based on opacity modulation. Master's thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, Prague, Czech Republic, 2015.

[9] Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-time concurrent linked list construction on the gpu. In *Proceedings of the 21st Eurographics Conference on Rendering*, EGSR'10, pages 1297–1304, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.