UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS GRADUAÇÃO EM COMPUTAÇÃO

MARILENA MAULE

Survey on Order Independent Transparency Techniques

Individual Work.

João Luiz Dihl Comba
Advisor

Rui Bastos
Co-Advisor

Porto Alegre, July 2010.

SUMARY

ABBREVIATIONS

AA	*Anti-Aliasing*
DFS	*Depth-First Search*
FIFO	*First in-First out*
MRT	*Multiple Render Target*
MSAA	*Multi-Sample Anti-Aliasing*
OIT	*Order Independent Transparency*
RMW	*Read Modify Write*

SUMARY OF FIGURES

ABSTRACT

Order independent transparency is a desired effect in a large set of applications, such as medical visualization and computer games. This work presents a survey of methods with different approaches to solve the order independent transparency problem. These approaches try to solve one or more facets of the problem, which are memory consumption, processing time and accuracy.

Keywords: Order Independent Transparency, Anti Aliasing.

1 INTRODUCTION

Transparency is a physical property which allows light to pass through some materials allowing us to see beyond objects. This property is very important to correctly simulate the appearance of real objects. In computer graphics there is a vast range of algorithms to generate realistic scenes which incorporate transparency.

The Ray Tracing algorithm [SM03], for example, launches rays from the eye to the scene, simulating the reverse path made by the light ray. The first intersection of the ray with an object in the scene will define the pixel attributes. To evaluate these attributes physical models are used, however to consider transparency effects additional rays must be cast from the intersection point through the object, see Figure 1.

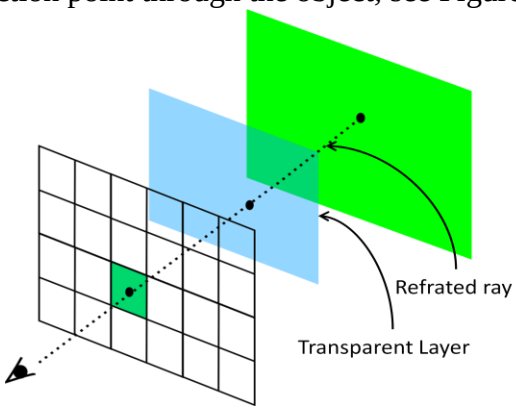

Figure 1.1 - Ray Tracing algorithm launches rays from the eye to the scene, with extra rays to evaluate transparency effect.

To combine the properties of each intersection point and properly evaluate the final color, values must to be computed in view direction proximity order, in other words, in the order in which they are intercepted by the ray.

Light interaction between distinct objects is expensive but easily handled by global illumination models. When dealing with local models, the techniques applies different approaches to guarantee the correct mix of the fragments. Most of them implies in some kind of ordering, with a high granularity level by sorting the fragments of each pixel, or in a less accurate manner by sorting the geometry. There are also other approaches that do not perform sorting.

Transparency is a pleasant effect and in some applications, such as medical visualization, is an imperative feature. In games, for example, there are many algorithms running at the same time and there is no much time for each one of them. Each new effect added to a game must to be meticulously planned to run in the little available amount of time. That is the reason why transparency is little used in

applications such games, the time or memory expend to compute transparency is high and still there is not a final solution in hardware.

The discussion about transparency begins with proposals to solve the hidden surface removal problem (in the mid-70). The first work to propose methods for compose transparent fragments was of Porter and Duff [PD84], in which are described techniques to combine fragments with different coverages over the pixel, and different transparencies represented in the alpha channel.

Initial works such the A-buffer [CARPENTER84] and the Depth Peeling [MAMMEN89] still inspire many algorithms. Most of them tries to take advantage of the growing programmability of graphics hardware, arising new challenges, such as, control the concurrent memory access in order to avoid read-modify hazards.

Some approaches, such as k-Buffer [CICS05] and the cell rasterization proposed by Liu et al. [LCD09], were first developed to solve other problems and later applied to rendering order independent transparency.

All techniques described in this work try to balance between accuracy, memory consumption and processing time. For generic scenes, it is very difficult to generate correct results in real time without extrapolate the amount of memory used.

Remaining sections will describe the most relevant solutions proposed to solve the transparency problem. A classification is proposed to group the methods that have similar approaches. The first approach described in Section 2 is the most common, it uses buffers to store auxiliary information until be able to process then or to acquire accuracy. The methods described in Section 3 perform many passes over the geometry, evaluating only the nearest (or the farter) fragments at each pass and combining them with the previous. Methods combining more than one approach are described in Section 4. Section 5 and 6 shows methods that do not perform sorting. In Section 5 they solve the fragments blending with depth based equations. And in Section 6 the alpha value is used as a probability to fill samples. The last described methods, in Section 7, uses point samples instead of triangle meshes to represent the scene.

2 BUFFER BASED METHODS

The buffer based methods use auxiliary storage to hold information while processing incoming fragments. Among these methods different approaches can be found. The A-buffer [CARPENTER84] is one of the first proposals to solve order independent transparency. It stores all fragments to evaluate the pixel in a post processing phase. Most of these methods have been (almost partially) inspired in the A-buffer, such as the FreePipe [LHLW10] approach.

The major part of these methods uses buffers with limited storage. The Z^3 [JC99] is a buffer with three entries for each pixel, used to reduce the errors while blending out of order fragments. The Stencil Routed [MB07] is a truncated A-buffer that performs depth peeling. And the FIFO buffers [MP01] [WITTENBRINK01] are unbounded buffers used iteratively to process incoming fragments. Other proposals, such as the k-Buffer [BCL07], also use buffers however with different approaches and are described in Section 4.

The main advantage of these methods is the performance. Although some of them focus in accuracy, the majority tries to avoid the expensive geometry passes, largely used by the methods in Section 3.

The main drawback is the memory consumption that may be unbounded in some cases.

2.1 The A-buffer

The A-buffer [CARPENTER84] is a robust method to handle hidden surface with anti-aliasing and order independent transparency. The technique is one of the pioneers to handle these problems.

The algorithm consists in a buffer with the size of the final image. Each position in this buffer corresponds to a pixel. A buffer entry may contain either a color, if that pixel has only one fragment, or a pointer to a fragment list with all fragments generated to that pixel.

Each fragment to be displayed is sampled by a mask, as shown in Figure 2.1, which represents its coverage over the final pixel. When rasterized, the fragments are append to the list of the correspondent pixel. If two consecutive fragments belong to the same object in the same pixel, they can be merged if they overlap in z, so their masks and areas are added, and only the resulting fragment is kept saving storage space.

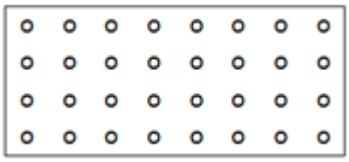

Figure 2.1 - Sampling mask used to represent partial coverage of the fragments over the pixel, and blend correctly the transparent pieces.

When the raster process is over the fragments are packed to determine the final pixel. To combine the fragments and generate the final pixel, it is used a search mask. Initially the search mask is empty, so it can be compared with the first fragment mask and its contribution is added. At each comparison the pixel mask is updated to represent the contributions already evaluated and be compared with the next fragment in the list, as shown in Figure 2.2.

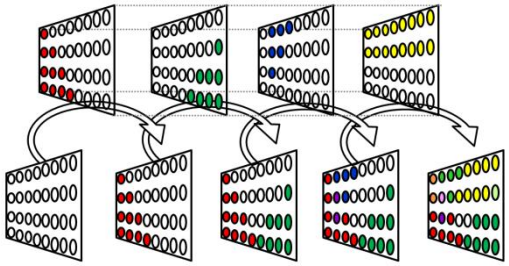

Figure 2.2 - The packing process starts with an empty search mask and traverses the sorted fragment list. For each fragment the mask is updated to represent the pixel area already covered. The samples of each fragment are added to the pixel if they pass through the mask. For transparent fragments, the mask is relaxed so they can be combined. The fragments list is depictured above the resulting mask for the iteration.

If the fragment is transparent, a recursive call searches the others fragments in the list with its mask to filter their color. For non intercepting fragments, when the outside and inside regions of the mask are evaluated then the colors are weighted blended by Equations 2.1 and 2.2. Where C_{in} is the inside color (the contribution of the fragment coverage), O_f is the fragment opacity, C_f is the fragment color and C_b is the accumulated color behind the fragment. α_{in} is the inside coverage, $\alpha_f$is the fragment coverage and $\alpha_b$is the accumulated coverage behind the fragment.

$$C_{in} = O_f\ C_f + \left(1 - O_f\right) C_b \tag{2.1}$$

$$\alpha_{in} = O_f\ \alpha_f + \left(1 - O_f\right) \alpha_b \tag{2.2}$$

When the inside and outside colors are evaluated, the final color of the pixel is computed by Equation 2.3. Where $C_r$is the returned color,

$$C_r = \frac{\alpha_{in} C_{in} + \alpha_{out} C_{out}}{\alpha_{in} + \alpha_{out}} \tag{2.3}$$

In case of interception between fragments, the blending is different. An intersection occurs when the fragments are from different objects and they overlap in z. To mix intercepting fragments the front most contribution is weighted by the Equation 2.4. Where Vis_{front} is the visibility estimation for the front most fragment, $Zmax_{next}$ is the major value of z for the fragment in back, $Zmax_{front}$ is the major value of z for the

fragment in front, $\Delta z_{next}$is the z range occupied by the second fragment and $\Delta z_{front}$is the z range value occupied by the front fragment.

$$Vis_{front} = \frac{Zmax_{next} - Zmin_{front}}{\Delta z_{front} + \Delta z_{next}} \quad (2.4)$$

The Figure 2.3 depicts an example of intercepting fragments.

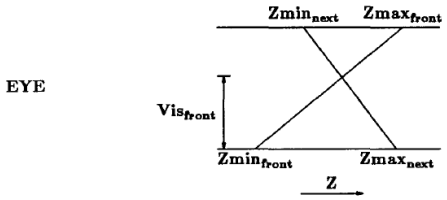

Figure 2.3 - Intercepting fragments from different objects. The contribution of each fragment is estimated by its depth values. Image from [CARPETTER84]

The A-buffer is a robust solution to order independent transparency, visibility and anti-aliasing. Its main disadvantage is the unbounded amount of memory used, followed by the random memory access imposed by the linked lists of fragments.

2.2 The Z³ Technique

The Z³ technique [JC99] tries to keep enough depth information to correct blend the fragments. It is based in keep three values to represent the depth of the fragment. A central z value with x and y slopes. With these values it is possible to correctly handle interpenetration between two fragments. The technique also keeps a fixed amount of slots to store fragments for blending. When occurs an overflow, the two nearest fragments are blended. The overage of the fragment also is considered, because changes in fragments with fewer samples introduce less error.

The three depth values kept for each fragment can be used to correctly generate z values for sub-samples of the fragment, and handle difficult cases such as those shown in Figure 2.4. Methods which keep only the center z value cannot distinguish most of the case of intercepting fragments, causing wrong blending and z fighting. The z average instead the central z improve accuracy in some cases, but it is still unable to solve many others. Use only the minimum and maximum values of z, for example, do not solve the upper right case in Figure 2.4 because it assumes that the fragments are always interpenetrating. The use of central z with x and y slopes improves the accuracy and can handles more cases of interpenetrating fragments.

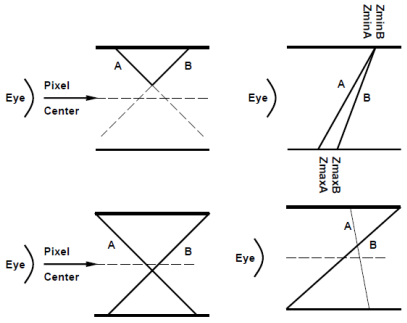

Figure 2.4 - Difficult cases to handle with incomplete z information. Image from [JC99].

To reduce the aliasing problem while saving memory storage, the idea is to compact information and *m* sampling points for each fragment, as shown in Figure 2.5. These samples are stored in a coverage *m*×*m* coverage mask that indicates which *m* samples are covered by the fragment. The *c* component is the average color of the samples covered by the fragment. The *z* value is the central depth, with other two values *Zdx* and *Zdy* containing *x* and *y* slopes respectively. The last stored value is the stencil bit *S*.

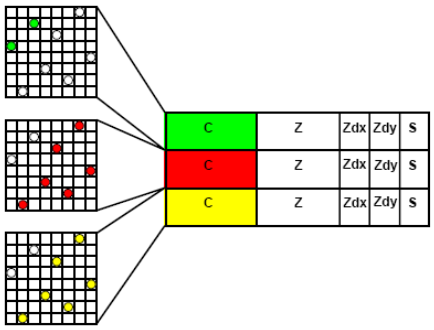

Figure 2.5 - A new structure to store the information of the fragments. Sampling mask, color *C*, central depth *z*, *x* and *y* slopes *dx* and dy, and the stencil bit *S*. Image from [JC99].

The quantity of fragments kept per pixel can vary to improve quality, or to reduce the total of memory used. The fragments are stored in fron-to-back order and when the amount of fragments generated is greater than the storage space available, the nearest fragments are combined. This approach not always produces correct results.

For each incoming fragment, is performed the occlusion test of its samples against the fragments stored in the Z^3 structure. This test is performed using the three values containing the depth information to reconstruct the samples depth. If any sample passes the test, the fragment contribution is stored. In the case that there are no more available entries, the fragment is blended with the entry with smallest depth difference pondered by the coverage of the fragment. Fragments with few samples have less contribution to the final color and the possible insertion of errors is little significant. For interpenetrating transparent fragments, is kept a swap vector with information about which samples must to be swapped in the color computation phase.

When the pixel color if finally computed, the method combines the fragments samples. For that, the correct z of each sample must to be evaluated before its contribution to be added to the final color. The alphas and colors of the samples are merged by Equations 2.5 and 2.6 respectively.

$$\alpha_{sample} = \alpha_{front} + \frac{(1-\alpha_{front})\alpha_{back}}{255} \tag{2.5}$$

$$C_{sample} = C_{front}\alpha_{front} + \frac{C_{back}(1-\alpha_{front})\alpha_{back}}{255} \tag{2.6}$$

The final color and opacity of the pixel is evaluated by blending the samples, which is done by Equations 2.7 and 2.8 for each channel. Where *#m* is the samples quantity.

$$\alpha_{merged} = \frac{\sum \alpha_{sample}}{\#m} \tag{2.7}$$

$$C_{merged} = \frac{\sum C_{sample}}{\alpha_{merged}\ \#m} \tag{2.8}$$

The Z^3 technique is a low cost approach to order independent transparency and anti-aliasing. It uses a fixed amount of memory to evaluate the pixels colors, and improve precision by storing more information about the fragment depth and coverage samples.

The main drawback is the incorrect blending of fragments when the available storage overflows.

2.3 FIFO Buffers

The first-in first-out methods store the fragments sequentially in the buffer in incoming ordering. The information in this buffer can be used in the next passes to process the fragments. The two methods described in this section are very similar, the main difference between them is that the F-buffer [MP01] do not write to the framebuffer until has finished the processing, while the R-buffer [WITTENBRINK01] enables it used during the processing.

2.3.1 The F-buffer

The fragment-stream-buffer [MP01] is a FIFO buffer that stores intermediate results (fragments information) between passes and fed its content into the input of the next fragment pass. The goal is to enable algorithms with only one geometry pass, by storing any kind of information that is required to evaluate pixels through multi-pass fragments processing.

As the values exit the rasterization stage they are sent to a FIFO buffer which guarantees no read-modify-write hazards by storing the data related to fragments instead of the x, y framebuffer location.

As A-buffer, this proposal drawback is the unbounded amount of memory that may be necessary, because it may store all incoming fragments.

The Figure 2.6 shows the F-buffer feedback to the fragment shader.

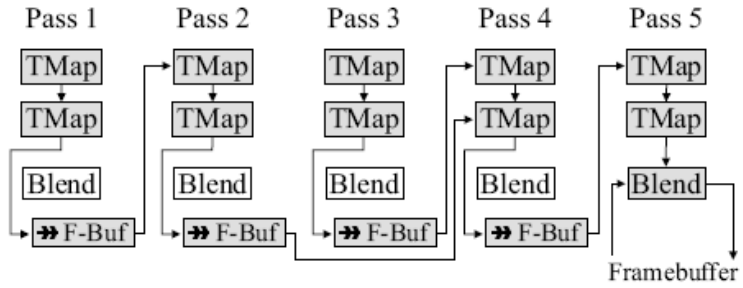

Figure 2.6 – More than one F-buffer can be used to hold information between fragment passes, avoiding read-modify-write hazards. Image from [MP01]

2.3.2 The R-buffer

The R-buffer [WITTENBRINK01] is an architectural proposal to implement Carpenter's A-buffer [CARPENTER84] with the data organized sequentially in memory, instead of the implementation with pointers (dynamic allocation).

The R-buffer main idea is the same as the F-Buffer. Although, the R-buffer presentation is better. It contains samples of how implement some effects using the

proposal, such as order independent transparency and anti-aliasing. These effects are compute by adding an R-Buffer and an extra Z-buffer to the classical pipeline.

Order independent transparency is implemented in a multi-phase scheme. In the first phase, the nearest opaque fragment is placed in the framebuffer, as a normal z-buffer stage. And all unoccluded transparent fragments are stored in the R-buffer. In the subsequent phases, the fragment shader traverses the buffer and composes the furthest fragment into the framebuffer (in a depth peeling fashion), until all fragments be processed. To perform these phases, are used two R-buffer. The fragment shader reads from the R-buffer containing the transparent fragments, compose the furthest in the framebuffer and write the others into the second R-buffer. In the next phase the orders of the buffers are swapped. The buffer containing the remaining fragments is read to find the furthest and the other is used to keep the fragments discarded in that phase. And so on until both buffers be empty.

Anti-aliasing: sub-pixel sample mask.

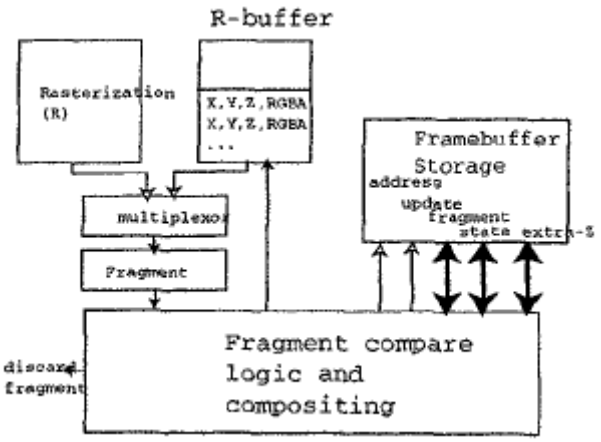

Figure 2.7 – R-buffer interface in hardware implementation. The stored information in both R-buffer and framebuffer can be acceded by the fragment shader. Image from [WITTENBRINK01].

2.4 Stencil Routed A-buffer

Myers and Bavoil [MB07] proposed a method to implement A-buffer using the stencil-buffer to choose fragments stored in a multi-sample texture. At each peeling pass, it can handle up to 8 fragments per pixel using the stencil buffer over the Multi-Sample Anti-Aliasing (MSAA) texture. If the scene has more than 8 transparent layers, then the method can detect an overflow in the stencil buffer and another depth peeling pass is performed. This implementation invalidates the use of the MSAA to perform anti-aliasing.

The MSAA performs shading at pixel level and stencil comparison at sample level. This stencil comparison is used to capture incoming fragments into the MSAA texture with z-test enabled. For a 4xMSAA, the stencil mask is initialized with:

1 2

3 4

With the stencil test set to be the equal operation, with the reference value set to 1, only one fragment will pass through the only position of the mask containing the value

1. After the first incoming fragment, with the stencil operation set to decrement its value, the mask becomes:

0 1

2 3

To test overflow, the mask initialization start at the value 2, and the reference value also becomes 2. After 4 fragments, the pixel stencil mask will be:

0 0

0 1

If more than 4 fragments tried to be written, then the lower right value will be 0, and a simple stencil zero-test can be used to determine if an overflow has occurred. In this case, a new geometry pass can be performed. To evaluate the pixel color, the samples are ordered by bitonic sort and blended.

This proposal cannot handle order independent transparency and anti-aliasing at the same time. However, for scenes with low complexity depth it can be very fast.

2.5 FreePipe

A recent work of Liu et al. [LHLW09b] proposed two approaches inspired in the A-buffer. Both are implemented in CUDA with their own rasterizer. The first approach is a fixed size array per pixel. The fragments lists are ordered by insertion sort via atomicMin operations to control the concurrent access of the threads. This technique waste memory because all pixels receive the same amount of storage and in most of the renderings the depth complexity is not that uniform. Moreover, the many atomiMin operations degrade the performance.

The second approach is more similar to the original A-buffer. It uses a list of fragments per pixel, and an counter index to avoid lots of atomic operations. The fragments are sorted and processed in a post processing stage.

Lately, Liu et al.[LHLW10] improved the proposal and made the FreePipe approach, a new graphic pipeline fully programmable to solve multi-fragments problems. The system is z-buffer based triangle rasterized implemented in CUDA.

In this proposal each triangle is independently processed by one thread (for big triangles, with large coverage in the framebuffer, the number of fragments per thread will be too high, this will degrades the performance reducing the parallelism).

The modelview-projection is allocated in constant memory, where can be accessed by all threads. (each thread access this information three times, they could have used shared memory).

As the conventional pipeline does, the FreePipe performs the basic operations. Each thread applies the transformations to the vertices, back face culling and frustum clipping, and generates the fragments. They also evaluates the fragments attributes as shading, writing them into its correspondent pixel location by atomic operations (probably concurrently, what cause the access to be serialized and consequently performance degradaion).

Depth and stencil tests are also done by atomic operations. a problem faced for the proposal is the simultaneously update of the depth and framebuffer, because framework used does not had support for 64b atomic operations.

The two proposed schemes are detailed below. The first is an improved version of the first proposal of use a fixed size array per pixel. And the second improves the A-buffer like approach.

Multi Depth Test Scheme

This scheme uses a fixed amount of memory to store up to N fragments per pixel in global memory. Via atomicMIN operations the incoming fragment is tested against the entries corresponding to its pixel location to find its corresponding z-sorted position.

The thread tries to store the fragment in the first position via atomicMIN, if the incoming depth is greater, the thread tries the next position, and so on until the fragment be stored or discarded in an overflow case. If the depth is less than the previously stored in some position of the array, the atomicMIN will store the new value and return the old one which the thread will try to store in the next position.

Each thread tries to store its depth value concurrently, but only one is allowed to access some position at each time while the others wait. For example, when the array is empty, the first thread wills successful store its value. Then the next thread gains access to the first position and its depth value is stored, the old value previously stored by the first thread will be returned by the atomicMIN, and the thread will store it in the next position. And so on until all the fragments be stored or discarded due to overflow of the array, as shown in Figure 2.8.

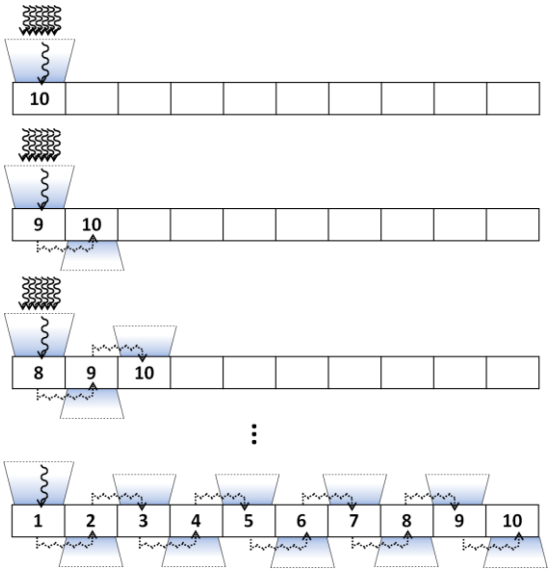

Figure 2.8 – Depth array updated concurrently by multiple threads via atomicMIN. The thread keeps trying to store the value until find an empty position or the end of the vector, in which case the value is discarded.

When the rasterization finish, all the fragments will be stored in sorted order by its z value. However, CUDA still do not provide critical sections, neither atomic operations for 64b. Then, to update the depth and the color buffer at the same time the authors proposed to compact the z information in the interval [0.5, 1] so it can be stored as an unsigned integer in 12b. Unfortunately this solution causes loss of precision, which may lead to artifacts due to wrong results in the z-test.

For the RGB value to be stored in 24b the alpha must be the same for all fragments. The color information is broken into two sets of 12b and stored separately in the low bits of two copies of the depth value. The color and the depth buffer are updated

separately by two atomicMIN operations. As the high bits contain the depth value, both the updates will successful store the correct sorted information.

This technique suffers loss of performance due to the high number of atomic operations in global memory.

A-Buffer Scheme

This scheme is directly inspired in Carpentter's A-buffer [CARPENTTER84]. Each pixel has a pointer to a list that can hold up to N fragments. Now, the fragment is a structure containing all the information needed, for example, depth and color.

For each pixel is also allocated a 32b counter to index the next available position in its array in which the incoming fragment can be stored. So, the threads firstly execute an atomicINC operation in the respective pixel counter, retaining the index to safely store the fragment, as shown in Figure 2.9.

In a post processing step the array of each pixel can be efficiently sorted and used. This solution eliminates many of the atomic operations needed to the first proposal.

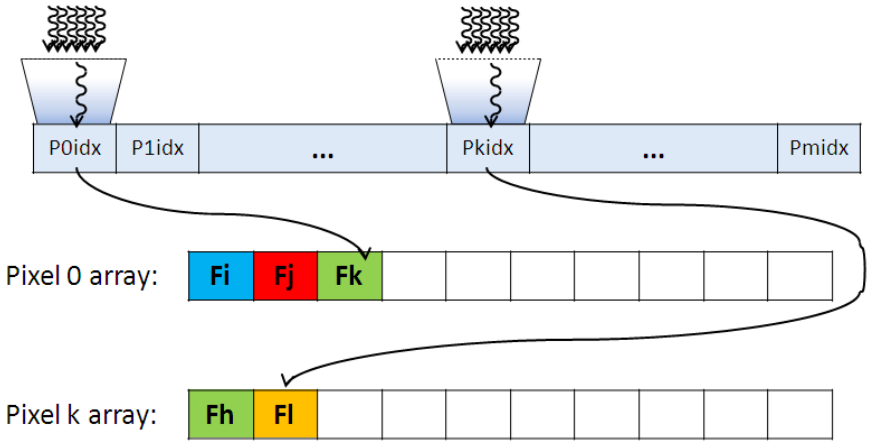

Figure 2.9 - Index counter consult and update. With the given index, the thread can safely write in the next free entry of the list.

The main advantage of the proposed solution is the control over the entire pipeline. The main disadvantages are the waste of memory cased by the uniform storage fixed to each pixel, and the series of atomic operations needed to update the buffers.

3 DEPTH PEELING BASED METHODS

The depth peeling based methods are commonly multi-pass approaches. At each pass, the methods solve one or more layers of visibility. The main idea is to capture the nearest or the further fragment at each pass and compose it with an accumulated solution.

The first approach using depth peeling was proposed by Mammen [MAMMEN89]. His proposal composes the furthest fragments in the framebuffer by multi-passes over the geometry. The first pass takes all the opaque fragments and the next passes blends the transparent fragments.

Lately, Everitt [EVERITT01] proposed a solution based in the shadow maps algorithm. At each geometry pass an auxiliary z-buffer is updated and only the nearest fragments are composed. This approach enables an early stop when the peeled layers are enough to represent the desired information. Other similar approaches were proposed to peel more than one layer each time. A sample is the k-buffer that can be used to peel up to k layers each geometry pass.

Mammen and Everitt proposal are able to generate correct results because they pass through all the layers. The depth peeling via bucket sort [LHLW09] is an approximated approach. It tries to route each fragment to an unique location which guarantees the sorted order to post processing composition.

3.1 Virtual Pixel Maps

The Virtual Pixel Maps [MAMMEN89] allows the pixel to be anything of any size placed on virtual memory if needed. This paper is focused on describe how to applicate transparency and anti-aliasing with its solution.

The transparency is done by sorting the fragments in back to front order and accumulating the blending results in a multi-pass basis. The first to be evaluated is the closest to the opaque fragment, they are blended and its z is updated, becoming the new opaque fragment in the next pass. To do that, the first pass must to render all opaque objects in the opaque pixel map. At each pass the closest fragment is rendered in the sort pixel map, in the end of the pass, the opaque and the sort pixel maps are blended into the opaque map and its z is updated. The evaluating process is depictured in the Figure 3.1.

To handle transparency the pixel must contain an *opaque depth* used as a z-buffer, an *opaque color* used as a framebuffer, a *sort depth* used to store the current depth nearest to the opaque depth, a *sort color* used to keep the transparent fragment nearest to the opaque fragment, and an *alpha* used to blend the sorted color with the opaque one at the end of each geometry pass.

The multi-pass process stops when no more fragments are blended, or when the number of iterations exceeds a user-defined threshold.

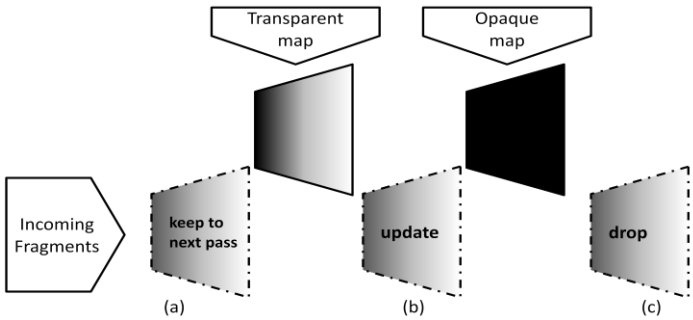

Figure 3.1 – Fragment evaluation cases. (a) I the incoming fragment is in front of the transparent depth map, it is kept to the next passes. (b) If the incoming fragment is further to the transparent depth map and nearest than the opaque depth map, it used to update the transparent map. (c) If the incoming fragment is further than the opaque layer, it is trivially discarded.

To handle anti-aliasing with transparency this method requires many passes over the geometry. The simple anti-aliasing is acquire by multi-pass weighted sampling, to obtain transparency, for each anti-aliasing (AA) sampling pass is done L passes to blend transparent fragments, where L is the number of transparent layers.

3.2 Interactive Order Independent Transparency

Inspired in the Shadow Mapping algorithm, the depth peeling technique proposed by Everitt [EVERITT01] uses an extra depth buffer to peel transparent layers.

The first geometry pass is done normally with the traditional z-buffer. The next passes uses the previous z-buffer values to find the n^{th}nearest fragments stored in an auxiliary framebuffer. At the end of the pass, the n^{th} layer is ready to be blended with the framebuffer.

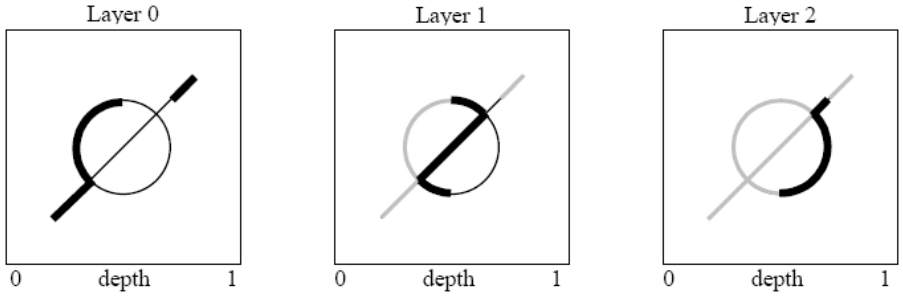

Figure 3.2 – The first geometry pass evaluate the front most layers. The second front most layer is evaluated in the second geometry pass, and so on. Image from [EVERITT01].

The Figure 3.2 shows the fragments evaluation sequence for interpenetrating shapes. The firsts to be evaluated are always the nearest to the eye and the next are blended according.

Lately, Bavoil and Myers [BM08] proposed a approach to peel two depth layers each geometry pass. The front most and the back most. The depth peeling algorithm is processed in 2 fronts: from front to back and from back to front simultaneously. Two sliding windows are used to capture the nearest and the further layers with min-max depth buffer, as shown in Figure 3.3.

Figure 3.3 - Advancing fronts. The algorithm captures the front and the back most layers at each geometry pass. Image from [BM08]

The report [BM08] also describes an approach to combine the fragments in a single geometry pass. This technique is described in Section 5.2.

3.3 Efficient Depth Peeling Via Bucket Sort

Liu et al. [LHLW09] proposed this technique to peel transparent layers sorting the fragments via bucket sort in the GPU. This approach enables to peel up to 32 layers at each geometry pass using multiple render targets MRT to store the buckets.

Each bucket can hold one fragment and is updated by MAX/MIN atomic operations. The depth range of each pixel is uniformly subdivided into intervals that will be mapped to one bucket. When fragments are routed to the same bucket there is a collision. To attenuate the artifacts caused by collisions multi-passes can be used over the geometry. Case else, on overflow the extra fragment is discarded. A post processing pass evaluate and merge the fragments stored in the buckets.

A two pass approach is used to alleviate the collisions of the fragments in the same bucket. The first pass renders all objects and evaluates a non-uniform depth range per pixel. This range is divided according to the fragments occupation.

The technique uses a fixed amount of memory per pixel WHk (W is the screen width, H is the screen height and k is the number of buckets).

The total of fragments per pixel can be up to 32 because the hardware enables up to 8 MRT, each one with 16 Bytes per pixel, and the method stores the fragment compacted into 32bits (4Bytes). So 8*16B/4B = 32 fragments per pixel.

The MAX blending operation compares the source and destination values of each MRT channel and keeps the greater. Each bucket it initialized with 0. The first MRT is updated to one, while the others are implicitly updated to 0, so the original value is always kept. When a collision happens, the MAX operation assures the bucket will keep the greater values and the others will be discarded.

During the rasterization the depth is normalized into [0,1] interval. The first geometry pass renders a bounding box of the scene into buckets intervals to obtain a depth estimative. Two consecutive buckets are bind into pairs. The depth range is subdivided into 16 subintervals [dk, dk+1) for k = [0,15], and $dk = zNear + k(\Delta z)/16$, where k is the bucket index.

The bucket k for a fragment is k = floor($16(df\text{-}zNear)/\Delta z$), where df is the depth value. The k^{th} pair update is (1-df, df) while the rest will be (0,0). When the first pass is over, the pairs will hold the min and max depth for that subinterval in correct depth order. If none fragment reaches some pair, the interval remains (0,0) and can be discarded.

In the next pass the intervals in the bucket array will be input as 8 MRT textures. The proposal for order independent transparency is pack the fragments into a 32b floating point, and unpack back to RGBA8 in a post processing step.

For non uniform depth scenes the collisions will be too frequent and must to be handled. The proposal to handle this situation is extend the algorithm to multi-pass approach, where a second geometry pass will receive the first result as input, as shown in Figure 3.4. This way, the second pass will store the second minimal and maximum depth values, then, both are passed to the post processing pass.

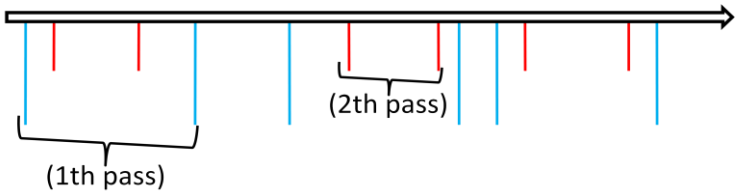

Figure 3.4 – For scenes with non uniform depth distribution the algorithm can be performed in two passes. The second pass capture the seconds nearest and further intervals.

The adaptive bucket depth peeling was proposed to solve the problem with uniform sub-division of the depth range. While some buckets remain empty, others may overload, according with the depth distribution in the scene. The ideal sub-division is that in which each fragment falls in one bucket.

To create subintervals to better fit the depth density distribution, the idea is to create a depth histogram, an array with the subintervals density indicating the geometry distribution.

The implementation proposed uses 8 MRT in which each bit indicates the presence or not of an fragment in a small z interval, as shown in Figure 3.6.

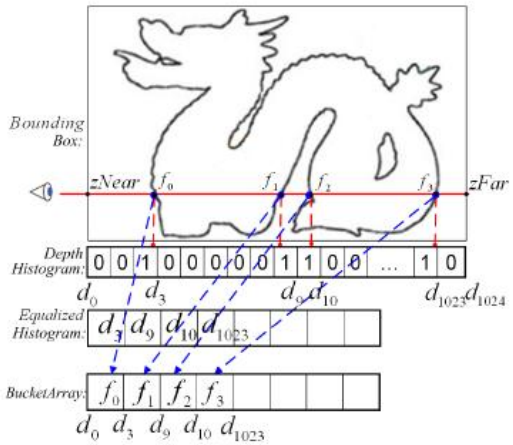

Figure 3.5 - An example of Adaptive Bucket Depth Peeling. The red dash arrows indicate the operations in the first geometry pass, and the blue dash arrows indicate the operations in the second geometry pass. Image from [LHLW09].

In a second pass the histogram is equalized and used in the third pass to evaluate the buckets intervals, generating non uniform subintervals. This solution eliminates collision for scenes with less than 32 depth layers and with depth distances superior to10^{-4}.

The main limitation of this technique is the lost of fragments (in z interval with high concentration of fragments) resulting in artifacts. And pack the color to peel more layers reduces the z precision.

For applications that require more attributes per fragment the total number of layers is reduced, or more geometry passes will be needed.

This method produces goods results for scenes with low depth complexity, but generates artifacts in the silhouettes due to the high fragments concentration.

4 HYBRID METHODS

Most of the methods, such as those presented early, handle the order independent transparency problem in image space by ordering the fragments with respect to its depth value. Other approaches, sort the geometric primitives in object-space with respect to its distance from the eye, and so, generate fragments in depth sorted order, ready to be blended.

The hybrid methods to compute order independent transparency use both approaches to handle the problem. Commonly, the GPU is used to perform the computations in image space. And the object space is handled by the CPU.

In this section, two hybrid methods are described. The first presented method sorts objects with respect to the view direction. The ordering between two objects is determined by the coverage of one over the other in their respective projections to the image-space.

The second method performs sorting in both object and image-space. The geometric primitives are partially sorted by the CPU, generating fragments in k-nearly sorted order (k-nearly notion is also described in section 4.2). The GPU sorts the remaining unsorted fragments to perform the blending operation.

4.1 Geometric Ordering With Image-Space Queries

Govindaraju et al. [GHLM05] proposes an algorithm for ordering of non overlapping geometric models with respect to the view direction. The objects (geometric models) are sorted in front-to-back order and then rasterized to generate consecutive fragments ready to be blended. The ordering relationship between the objects is determined by the occlusion of object O_1 over the object O_2.

The objects sorting algorithm is a 3D version of the 1D algorithm described below.

1D sorting algorithm

This section describes a sorting algorithm for ordering a set of scalar elements. An expansion of the idea used to sort 3D objects is described in the next section.

This algorithm is a multi-pass approach in which each iteration is performed in two passes. In the first pass, the input sequence **I** of scalar values is traversed in back-to-front order. Each element is compared with the current minimum value (denoted **min** for short, and initialized with ∞). If the element is less or equal to min, then it is added to the beginning of the monotonically increasing sequence **M,** and its value becomes the new min.

The second pass copies I into an auxiliary sequence **T** to preserve the original data. The elements of T are processed in front-to-back order. If the element belongs to M and is less or equal to min, then it is removed from T and from M, and is added at the tail of

the solution **S**. The min value is updated only when it is greater than an element of T that does not belong to M.

At the end of each iteration M will contain the unsorted elements. The remaining iterations take the previous M as input until M = { }, which means that the initial I is completely sorted.

3D Sorting

The 1D sorting algorithm described above is generalized to sort the geometric models in object-space through two constraints:

- The object O_i is in front of the object O_j if, and only if, the projection of $O_i$in image-space is fully visible with respect to the projection of O_j. This ordering relationship is denoted as $O_i \leq O_j$.
- If$O_i \leq O_j$, then $O_i$occurs before $O_j$in the output list.

These constraints are used to determine the comparison operations performed between the elements of I, T, M and with min.

In cases in which the scene has cycles between objects, the projections to the image space will overlap. The algorithm can detect this situation and switch to a mode that operates within triangles.

In the worst case, the complexity of this algorithm is $O(n^2)$. However, using the S of the previous frame as input for the next frame sorting, most of the objects will already be sorted due to the temporal coherence between frames. Causing the execution to fall in linear complexity.

4.2 The *k*-Buffer

Callahan et. al. [CICS05] presents a novel technique to efficiently render unstructured grids. The core of his technique is the k-buffer, lately generalized in [BCL07] to handle multi-fragments effects.

The k-buffer is a generalization of the fragment buffer to hold up to k fragments per pixel. It is used as a read-modify write (RMW) pool that interacts with each incoming fragment according with a function that describes the desired effect to be applied.

To perform depth peeling, the k-buffer can be used as a truncated A-buffer [CARPENTER84]. At each geometry pass it can peel up to k depth layers.

K-buffer used as a stream buffer enables interactive blending of its values with the incoming fragments. To correctly blend, these fragments must to be rasterized in k-nearly sorted order. Which means that none fragment can be more than k positions out of its sorted order.

Given a sequence S of real values, the exact sorted sequence ESS(S) is a tuple of integer values vi, where each vi is the position of the ith element of S in some order. The k-nearly sorted sequence k-NSS(S) is any sequence in which the ith element differs from vi less than k units.

A proposal to implement the k-buffer in the graphics pipeline is to place it in the fragments stage. However, many fragments are processed at the same time and must to be written to a reorder buffer before go to memory. This asynchronous write may cause RMW hazards in the k-buffer operations. . These hazards can be avoided by a fragment scheduling, which detects collisions and rearranges the fragments to be processed in an order without hazards. The main advantage of this proposal is the minimal changes

required to implement it within the graphics pipeline. And its main drawback is the parallelism reduction due to the reorders of the fragments.

The second proposed implementation uses the k-buffer values to process fragments in the blending stage. This solution does not require scheduling, but require changes in a fixed unit of the graphics pipeline.

An experimental implementation was created using a set of textures in OpenGL with MRT. Without hardware support, this implementation suffers from RMW hazards. To overcome these hazards, the authors proposed two heuristics:

- Sort primitives by centroid depth;
- Batch triangles and flush the pipeline.

K-buffer can be used to implement depth peeling capable of peel up to k layers per geometry pass, which benefit effects such as transparency and translucency.

A hybrid solution uses the Least Significant Digit Radix Sorting [SEDGEWICK 98] to sort the geometry in view direction by the centroids of the triangles in linear time. This heuristic do not guarantee the k-nearly sorted order, however produces good results. Scenes with geometry overlaps, or high variance in size or aspect ratio may result in artifacts for small k. To avoid these artifacts without using more memory, the triangles are subdivided to acquire better approximation from their centroid.

In isosurfaces and volumes rendering, the k-buffer is used to find the two nearest fragments from the eye, forming a ray segment used to compute the fragment color.

For large values for k the work is shifted to the GPU, while small k values require better sorting in the CPU, providing efficient load balance improving the performance.

5 SORT-INDEPENDENT METHODS

The methods describes in this sections do not sort primitives. They just propose equations to blend fragments in none sorted incoming order.

This kind of technique is less used because produces incorrect results.

5.1 Sort-Independent Alpha Blending

The method of weighted sum proposed by Meshkin [MESHKIN07] blends incoming fragments in arrival order by Equation 5.1.

$$C_{dst} = \sum(\alpha_{src}\, C_{src}) + C_{bg}\,(1 - \sum \alpha_{src}) \tag{5.1}$$

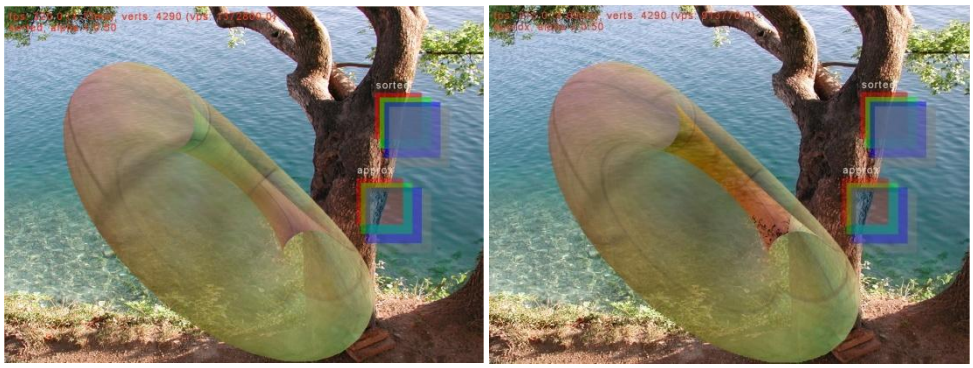

Figure 5.1 - Using α = 0.5. At left, the result using sorted fragments. At right, the result using weighted sums. Image from [MESHKIN07]

For low values of alpha, the error is acceptable. However, increasing the alpha value, the error becomes inacceptable, as show the Figure 5.1.

5.2 Weighted Average

The second technique proposed by Bavoil and Myers [BM08], named weighted average, is a single pass blending equation. All the fragments are rendered into an accumulation buffer and the depth complexity (n fragments per pixel). Then, for each pixel its average color is calculated with the proposed equations 5.2, 5.3, and 5.4.

$$C_{dst} = \sum(RGB)\alpha / \sum \alpha \tag{5.2}$$

$$C_{dst} = \sum(\alpha / N) \tag{5.3}$$

$$C_{dst} = C\alpha \frac{(1-(1-\alpha)^N)}{\alpha} + C_{bg}(1-\alpha)^N \tag{5.4}$$

The Figure 5.2 shows a comparison between a sort-dependent method and the equational methods presented.

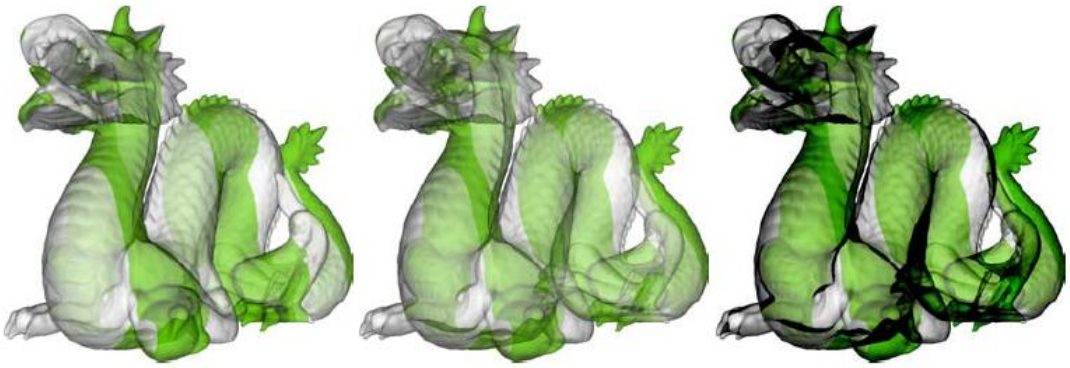

Figure 5.2 - Dual depth peeling (left). Weighted Average (middle). Weighted Sum (right). Image from [BM08].

6 STOCHASTIC AND PROBABILISTIC METHODS

This Section describes proposals that use probabilistic approaches to determine the fragment coverage over the pixel. The main drawback of these methods is the noise produced by random values used in the processing. To eliminate the noise is necessary use much more memory. The main advantage is the low processing time, because the fragments do not need to be sorted.

The "Stochastic Transparency", proposed by Enderton et al. [ESSL10] uses the alpha of the fragment as a stochastic value to fill samples of the pixel. And the "Silhouette opaque transparency rendering", proposed by Sen et al. [SCG03] uses sampling masks with random patterns to improve accuracy in the silhouettes evaluations of transparent objects.

6.1 Stochastic Transparency

Enderton et al. [ESSL10] use stochastic values to determine the fragment coverage over the samples of the pixel. This method implements a screen-door transparency with a random sub-pixel stipple pattern, where there are S samples for each pixel. A subset of the samples represents the fragment opacity, see Figure 6.1.

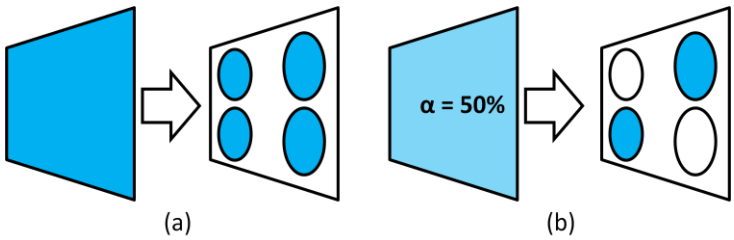

Figure 6.1 - (a) An opaque fragment covers all the samples. (b) A transparent fragment covers R of S samples, where α = R/S

Each sample has α probability to receive the fragment color. In average, the number of samples painted will be R, where α = R/S. For a fragment that is not in front, its α is considered as α = (1 - α_i), for all fragments i in front. However, without sorting the correct value of $\sum_i^n \alpha_i$ cannot be evaluated. Instead, the paper proposes a method to estimate an approximation for it.

This technique drawback is noise that can be attenuated increasing the number of samples per triangle. The problem known as *diminishing returns* says that is needed the quadruple number of samples to halve the error. The solution found was to use *stratified sampling*, where a group of *R* samples can be computed by $R_i = \lfloor \alpha_i S + \xi \rfloor$, where $\xi \in$

[0,1]. This approach enables to randomize the number of samples taken from a fragment which α value doesn't fit an exact subset of the total available samples.

In other words, a simple sampling evaluates the probability for each sample what may cause all to receive the color of a fragment with α = 0.2, for example. The stratified sampling chooses randomly a value that best fit the number of samples to be used.

The alpha correction proposal uses a rendering pass to evaluate the total alpha for each pixel by Equation 6.1.

$$\alpha_{total} = 1 - \prod_i^n 1 - \alpha_i \tag{6.1}$$

Then multiply the total average color by $\alpha_{total}/(\frac{R}{S})$. This correction factor is not exact, causing some layers to get higher errors (unfortunately, the error grows up for the nearest triangles and decreases for furthers).

To make the method more accurate, they propose one more step (*depth-based stochastic transparency*) that estimates the visibility for each fragment by considering its depth.

1. Render opaque primitives.
2. Render the accumulated α for each pixel in a buffer (one pass).
3. Render the z for all samples per pixel (one pass multi-sampled).
4. Accumulate colors by C = $\sum vis(zi)\,\alpha_i c_i$ where vis(z) = count(z $\leq$ zi)/S (one pass).
5. Multiply the result by the alpha correction.

The process is depictured in Figure 6.2.

The result is less noisy because it estimates the fragment visibility and not the α.

After all, they propose to consider the spatial occupation of the fragment over the samples, instead of all the samples being centered at the pixel, using MSAA supported by hardware.

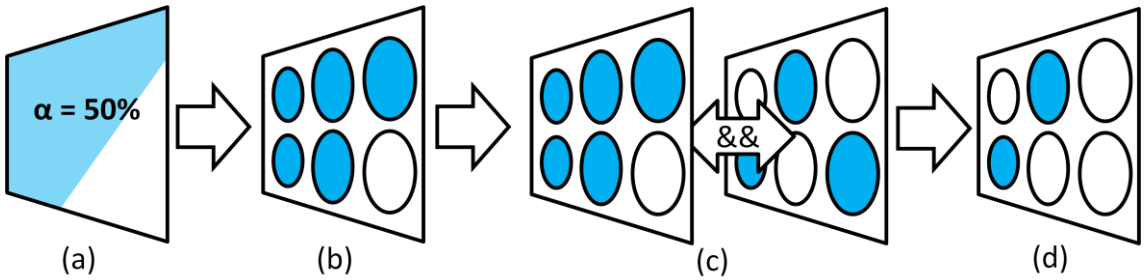

Figure 6.2 - (a) The shaded fragment covers a (b) Set of samples of the MSAA in which the z-test is performed. (c) The MSAA mask is merged with the Screen Door mask from transparency, resulting in (d) The final set of samples.

The disadvantage of the screen door mask is that the samples generation is always the same. While this proposal ensures that masks are uncorrelated between samples in the same pixel.

For correct render the edges of the transparent geometry, total alpha must be rendered with multi-sampling. For the rest, the filtered alpha is enough.

Finally, the algorithm becomes:

1. Render opaque and background multi-sampled in the z-buffer.
2. Render total alpha into a multi-sampled buffer (one pass).

3. Render the transparent primitives multi-samples into the opaque z-buffer using stochastic alpha-to-coverage (one pass).

4. Accumulating color by rendering in additive blending mode testing the z-buffer.

5. Dim 1-α_{total} from all background samples. Correct the accumulation buffer with the α_{total} and blend it with the opaque buffer.

The hardware used supports up to 8 samples per pixel so, to simulate more samples, the passes 1 and 2 do not need to be recomputed.

To accelerate the R of S samples choose, they store a lookup table with the result pre-computed.

This method can be applied to simulate hair, smoke, foliage, sheer cloth and in CAD softwares.

The advantage over depth peeling is the complexity, which impacts on large scenes. While DP is O(L) (L= number of transparent layers in the scene, in the worst case each primitive is an transparent layer, then O(L) = O(P²)) the stochastic transparency remains O(P). Over the specialized methods on smoke the advantage is generality.

The main limitation of this method is noise that changes each frame. Under certain conditions, the noise reduction may cause stripe of noisier pixels in the boundaries.

6.2 Silhouette-Opaque Transparency Rendering

Sen et al. [SCG03] proposed a method that focuses in correctly render the opacity in the silhouettes of the objects. Transparency depends of the amount of material the light passes through. It causes the silhouettes to look more opaque.

The method uses random alpha masks patterns to sample in object space with super-sampling in a single pass. It uniformly samples $m<n$ primitives from the object, where n is total amount of primitives belonged to that object, and m is the effective amount of samples taken.

For the size of the primitives be approximately equal to the pixel size, it is estimated by the distance of the primitive to the view point.

The point-samples primitives for a triangle are generated on the fly by evaluating the triangle coverage over the projection plane, as shown in Figure 6.3. The primitives chosen to be rendered are proportional to the alpha of the triangle in super-sampling. After, the framebuffer is filled to its normal size.

The primitive's generation is done by rotating the triangle to be parallel to the projection plane and evaluated normally. So, at silhouettes, the density of primitives per pixel is higher.

This technique cannot be used with large models because the size of the primitives becomes very small.

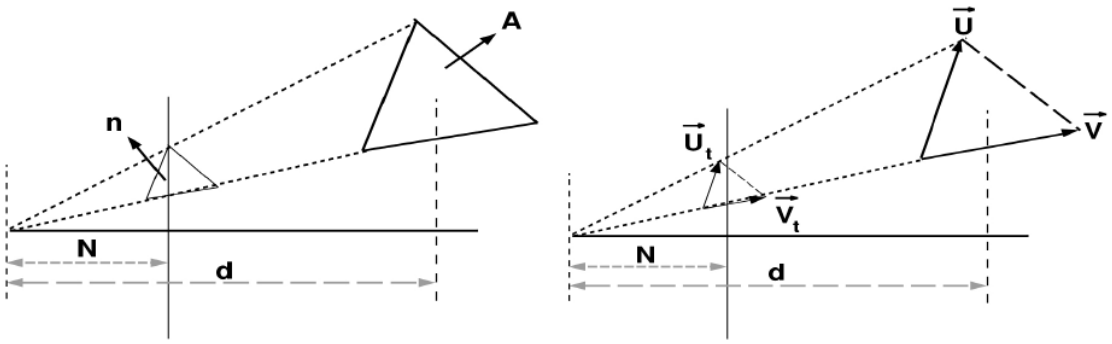

Figure 6.3 - The left image shows the diagram for computing the number of samples in a triangle in the software implementation. The right image shows the diagram used in the hardware assisted method. Image from [SCG03].

A hardware implementation is proposed to improve the performance. The idea is to use the texture mapping to parameterize the sampling mask. For each alpha value a mask is generated in a pre-processing step. The triangles are mapped to its corresponding alpha-mask. The point samples are generated by the transparent texels of the mask. The remaining passes are the same as in the software implementation.

The hardware implementation suffers with limitations in the texture filters available, which make the silhouettes not to be visible.

The main drawback of this method is the scene scale limitation imposed by the way with which the point samples are generated. In second place the noise due to the probabilistic sampling.

The main advantage is the fast rendering in a single geometry pass.

7 POINT BASED RENDERING METHODS

This section presents methods that does not use the triangle based rendering, in which the triangles are projected into a projection plane and rasterized.

The first presented method is a point-based approach proposed by Zhang and Pajarola [ZP06] to improve performance and quality by grouping primitives.

The second method proposed by Liu et al. [LCD09] is used to isosurfaces rendering. The main idea is to subdivide the space in cells and escape the empty cells. To improve the performance, they proposed a technique to order the tiled cells which enables order independent transparency.

7.1 GPU-Accelerated Transparent Point-Based Rendering

The method proposed by Zhang and Pajarola [ZP06] is a technique to improve quality for point-based rendering with transparency. They use graph coloring to create minimal overlapping groups. Between points in the same group the overlap is minimal, so the PBR-interpolation cam be omitted at first. The alpha-blending is performed separately for each group, and the PBR-interpolation is acquired by merging the images of each group.

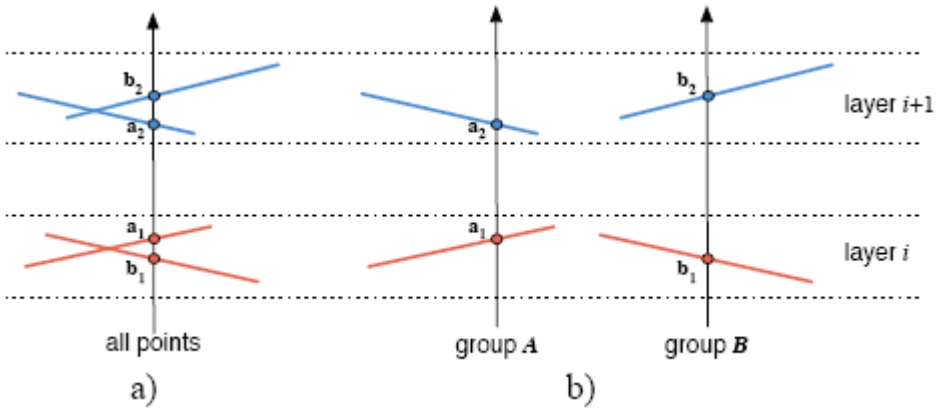

Figure 7.1 - a) Rendering of transparent points must distinguish per fragment blending and per point interpolations as well as transparent alpha-blending. b) We divide points into groups; where a1, b1 are alpha blended for transparency with a2, b2 within their groups respectively, and then PBR-interpolation is performed in a post-pass. Image from [ZP06].

7.2 Fast Isosurface Rendering on a GPU by Cell Rasterization

The goal of the method proposed by Liu et al. [LCD09] is to efficiently render many isosurfaces layers. The view-dependent sorting algorithm was developed aiming to take advantage from the early z-culling hardware feature.

To evaluate the isosurface, the space is discretized into a cell grid. The grid commonly has many empty cells which do not contribute to the final image. So, the active cells (those which contribute to the image generation) are extracted from the grid into a HistoPyrmid texture [ZTTS06].

Each cell is rendered independently via glPoints. The vertex shader computes the cyclical projection of the point in screen space. To do the correct projection, cyclical not elliptical, the authors propose a new technique that projects the circle based on the major axis of the ellipse, represented by the points in with the view-ray tangents the bounding sphere of the point, as shown in Figure 7.2.

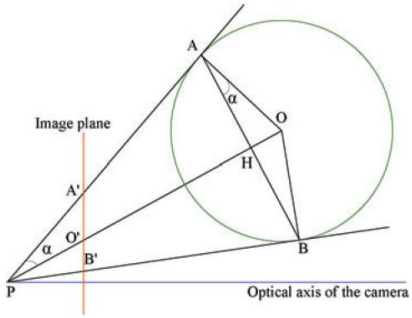

Figure 7.2 - The computation of a sphere's perspective projection on the CrossPlane. The green circle is the intersection of the CrossPlane and the bounding sphere with centre O and radius R = |→OA| = |→OB|.
Image from [LCD09].

With this procedure the circle in screen space will correspond to the projection of the bounding sphere in the image plane.

The fragment shader computes the ray intersections with the cells to generate the fragment. This intersection can be computer with the bounding sphere of the point but to transparency effects is better use the cell cube because the ray-segments produced no overlapping with neighbors.

Each ray may intercept the cell in two points which define the ray-segment, see Figure 7.3. The approximated interception point with the isosurface is found interactively searching the ray segment to generate the fragment.

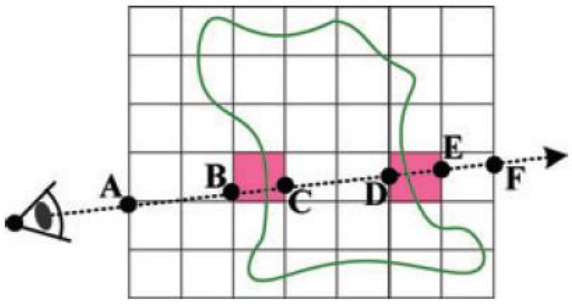

Figure 7.3 - For opaque isosurface rendering (early ray exiting due to first-hit can be applied), the ray length of our method is |→BC|, while the ray length of raycasting is →AC. For transparent rendering (early ray exiting can not be applied), the ray length of our method is | →BC| + |→DE|, while the ray length of raycasting is |→AF|. In this illustration, ray-cube intersection is being used by our method.
Image from [LCD09].

To accelerate the technique using early z-culling, the authors developed a view-dependent sorting algorithm that works in the HistoPyramid data, ordering the cells in front-to-back order. In addition to the performance gain with early z-culling, this ordering can provide order independent transparency.

The base of the HistoPyramid is a tiled 2D texture created from 3D data. Tilling is performed along the major axis of the view direction, which is the one with the smallest angle with the view direction.

HistoPyramid is created in GPU and stored in quadtree order. Sorting is performed in CPU over the l_r level data from the HistoPyramid. This level have only one element for each slice, and can be computed as $l_r = 1 + \log\sqrt{L/4}$, where L is the number of slices.

The sorting algorithm results in two vectors with length L. The first contain the amount of active cells in the *k* previous slices, and the second contains the number of active cells for the k^{th} slice. These vectors are used to index the HistoPyramid texture and draw the active cells in view-dependent ordering.

To perform order independent transparency, this method requires high volume resolution because cells in the same slice are not sorted.

The main drawback of this method is the high memory consumption.

In a latter work, Liu et al. [LCD10] extended this proposal by using a 16b mask to route the fragments into layered 2D textures as shown in Figure 7.1.

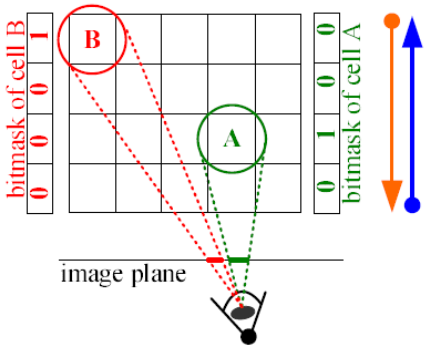

Figure 7.4 - If the camera is aligned with the major axis (blue arrow), then Seqcell for cell A and B are 1 and 3, respectively; while if the camera direction is in the negative direction of the major axis (orange arrow), then Seqcell for cell A and B are 2 and 0, respectively, before flipping, while they are 1 and 3 after flipping. So finally the bitmasks for cell A and B are 0x2 and 0x8, respectively. Here TotalSlice is 4.
Image from [LCD10]

The implementation uses glLayer to set the texture in which to draw according to the sequence number of the active cells extracted from the HistoPyramid. The output is the bitmask used to route the cell its layer in sorted order.

In the raycasting pass the bitmasks in the layered textures are used to reconstruct the depth intervals to define valid ray segments, which are used to escape empty spaces.

This method requires different parameterizations for different datasets to adjust the cells granularity.

8 CONCLUSIONS

This work presented different methods to handle transparency, defined as the amount of light passing through primitives and captured by the rendering camera. Each method tries to solve an aspect of the problem. Some methods focus in reduce the processing time and/or memory consumption, and others improve rendering quality

Most of them use buffers to store partial results until be able to properly compose the fragments. The main drawback of these methods is the memory consumption and bandwidth, because to evaluate a single pixel they need to store as much as possible fragments that fall in that pixel. For scenes with high depth complexity, the amount of memory needed to generate good results can compromise the execution of the entire application.

Methods based in multi-pass depth peeling are straightforward and simple to understand and implement. They produce correct results if pass through all depth layers, and also can be stopped after a satisfied number of layers. However, for large scenes the multi-pass approach do not acquire interactive frame rate. Some methods propose techniques to reduce the number of geometry passes by using extra memory to peel more than one layer each pass.

The major part of the methods sorts the primitives at fragment level. This work described two methods which differ from others by processing in both domains, image and object space. In special the k-buffer that can balance between performance, accuracy and memory consumption.

Methods that do not sort fragments are very fast, computing transparency in a single geometry pass without extra memory. However, the results present big errors to high alpha values.

Few methods described here use the alpha as a probabilistic value to fill samples of the pixel. They also use extra memory and often invalidate the anti-aliasing application.

Also were described methods using point-based rendering instead of triangle meshes. The approaches are very different from the others, most applicable to rendering of isosurfaces.

The ideal method handles order independent transparency in a constant amount of memory, with correct results in linear time. This ideal method must be implemented in hardware and render interactive transparent scenes with high depth complexity at interactive frame rates. However, it is not possible to acquire all the aimed features. Only the primitives sorting is $O(n \log n)$, or $O(n)$ using extra memory. And using relaxed sorting not guarantee correct results.

9 REFERENCES

[AMN03] AILA, T., MIETTINEN, V., AND NORDLUND, P. 2003. Delay streams for graphics hardware. ACM Trans. Graph. 22, 3, 792–800.

[BCL07] BAVOIL, L., CALLAHAN, S. P., LEFOHN, A., COMBA, JO A. L. D., AND SILVA, C. T. 2007. Multi-fragment effects on the GPU using the k-buffer. In I3D ’07: Proceedings of the 2007 symposium on Interactive 3D graphics and games, ACM, New York, NY, USA, 97–104.

[BM08] BAVOIL, L. MYERS, K. 2008. Order Independent Transparency with Dual Depth Peeling. Technical report, NVIDIA Corporation.

[CICS05] CALLAHAN, S. P., IKITS, M., COMBA, J. L. D., AND SILVA, C. T. 2005. Hardware-assisted visibility sorting for unstructured volume rendering. IEEE Transactions on Visualization and Computer Graphics 11, 3, 285–295.

[CARPENTER84] CARPENTER, L. 1984. The A-buffer, an anti-aliased hidden surface method. In SIGGRAPH ’84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques, ACM, New York, NY, USA, 103–108.

[ESSL10] ENDERTON, E., SINTORN, E., SHIRLEY, P., AND LUEBKE, D. 2010. Stochastic transparency. In I3D ’10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, ACM, New York, NY, USA, 157–164.

[EVERITT01] EVERITT, C. 2001. Interactive Order-Independent Transparency. Technical report, NVIDIA Corporation.

[GHLM05] GOVINDARAJU, N. K., HENSON, M., LIN, M. C., AND MANOCHA, D. 2005. Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In I3D ’05: Proceedings of the 2005 symposium on Interactive 3D graphics and games, ACM, New York, NY, USA, 49–56.

[JC99] JOUPPI, N. P., AND CHANG, C.-F. 1999. Z^3: an economical hardware technique for high-quality anti-aliasing and transparency. In HWWS ’99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, ACM, New York, NY, USA, 85–93.

[LCD09] LIU, B., CLAPWORTHY, G., AND DONG, F. 2009. Fast isosurface rendering on a GPU by cell rasterization. Comput. Graph. Forum 28, 8, 2151–2164.

[LCD10] LIU, B., CLAPWORTHY, G., AND DONG, F. 2010. Multi-layer Depth Peeling by Single-Pass Hardware Rasterisation for Faster Isosurface Raytracing on a GPU. EuroVis.

[LHLW09] LIU, F., HUANG, M.-C., LIU, X.-H., AND WU, E.-H. 2009. Efficient depth peeling via bucket sort. In HPG '09: Proceedings of the Conference on High Performance Graphics 2009, ACM, New York, NY, USA, 51–57.

[LHLW09b] LIU, F., HUANG, M.-C., LIU, X.-H., AND WU, E.-H. 2009. Single pass depth peeling via CUDA rasterizer. In SIGGRAPH '09: SIGGRAPH 2009: Talks, ACM, New York, NY, USA, 1–1.

[LHLW10] LIU, F., HUANG, M.-C., LIU, X.-H., AND WU, E.-H. 2010. FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, ACM, New York, NY, USA, 75–82.

[MAMMEN89] MAMMEN, A. 1989. Transparency and anti-aliasing algorithms implemented with the virtual pixel maps technique. IEEE Comput. Graph. Appl. 9, 4, 43–55.

[MESHKIN07] MESHKIN, H. 2007. Sort-Independent Alpha Blending. Perpetual Entertainment, GDC Session. Available at: http://www.houmany.com/data/GDC2007_Meshkin_Houman_SortIndependentAlphaBlending.ppt. Last Access: 21 Jul. 2010.

[MP01] MARK, W. R., AND PROUDFOOT, K. 2001. The F-buffer: a rasterization-order fifo buffer for multi-pass rendering. In HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, ACM, New York, NY, USA, 57–64.

[MB07] MYERS, K., AND BAVOIL, L. 2007. Stencil Routed A-buffer. In SIGGRAPH '07: ACM SIGGRAPH 2007 sketches, ACM, New York, NY, USA, 21.

[PD84] PORTER, T., AND DUFF, T. 1984. Compositing Digital Images. In SIGGRAPH Comput. Graph., ACM, New York, NY, USA, 253–259.

[SEDGEWICK 98] SEDGEWICK, R. 1998. Algorithms In C, third ed. Addison-Wesley, 298–301 and 403–437.

[SCG03] SEN, O., CHEMUDUGUNTA, C., AND GOPI, M. 2003. Silhouette opaque transparency rendering. In Computer Graphics and Imaging, 153–158.

[SM03] SHIRLEY, P. & MORLEY, R. K. Realistic Ray Tracing A. K. Peters, Ltd., 2003.

[WITTENBRINK01] WITTENBRINK, C. M. 2001. R-buffer: a pointerless A-buffer hardware architecture. In HWWS ’01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, ACM, New York, NY, USA, 73–80.

[ZP06] ZHANG, Y., AND PAJAROLA, R. 2006. Gpu-accelerated transparent point-based rendering. In SIGGRAPH ’06: ACM SIGGRAPH 2006 Sketches, ACM, New York, NY, USA, 178.

[ZTTS06] ZIEGLER G., TEVS A., THEOBALT C., SEIDEL H.- P.: On-the-fly point clouds through histogram pyramids. In VMV06 (2006), pp. 137–144.