Memory-Efficient Order-Independent Transparency with Dynamic Fragment Buffer

Marilena Maule*, João L. D. Comba* Rafael Torchelsen† Rui Bastos‡
*UFRGS
{mmaule,comba}@inf.ufrgs.br
†UFFS
rafael.torchelsen@uffs.edu.br
‡NVIDIA
rbastos@nvidia.com

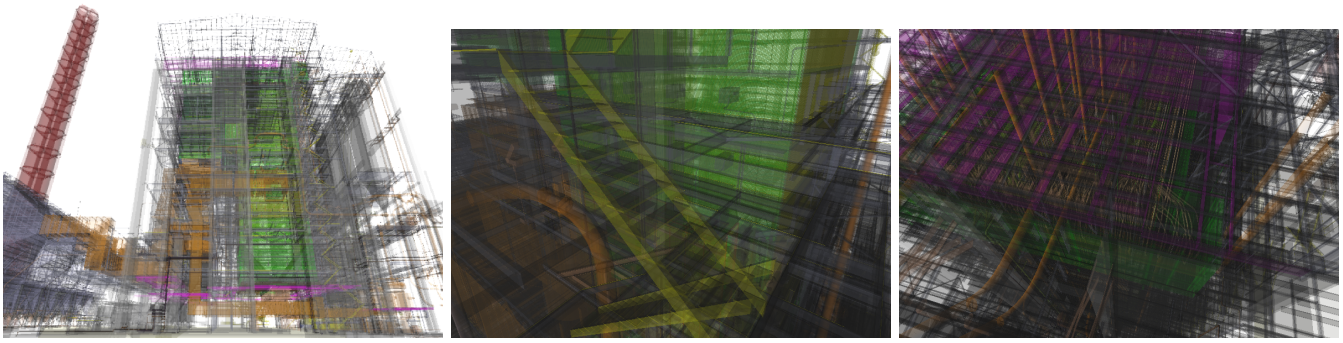

Fig. 1. Screenshots of the exterior and interior of the Power Plant model, with up to **588** transparent layers and **12,748,510** triangles rendered with the Dynamic Fragment Buffer technique at interactive frame rates.

***Abstract*—Order-independent transparency (OIT) rendering is computationally intensive due to required sorting and sufficient memory to store fragments before sorting. We present Dynamic Fragment Buffer, a revamped two-pass OIT rendering technique, which performs correct blending of a large number of transparent layers at interactive frame rates. Our approach self-adjusts memory allocation to handle a variable number of fragments per pixel without wasting memory. In this paper we perform a detailed analysis of several design decisions which lead to this technique. We present a collection of experiments that illustrate the advantages of our technique with respect to others OIT algorithms in the literature.**

***Keywords*-Order-Independent Transparency**

I. Introduction

The ability to render scenes with transparency effects is crucial to increase realism in computer-generated images. The transparency process relies on the composition, in visibility ordering, of multiple samples that capture the interaction of light against transparent materials of the scene. To enforce ordering for transparency computation, object-space sorting is a possibility. However, this can lead to artifacts in the composition when scenes have interpenetrating primitives.

Another possibility is to defer the visibility ordering to be performed at fragment level, leveraging the computational power of graphics cards. The class of algorithms that do fragment sorting is truly order independent (thus, the OIT) allows correct fragment blending. To accommodate for sorting at the fragment level, OIT algorithms must have a strategy to combine multiple samples (here simply called layers) for each pixel in depth-sorted order.

The most intuitive solution to the sorting problem, intimately attached to the transparency effect, is to store all fragments in an unbounded buffer, and sort them in a post-processing phase. The per-pixel storage can be done using sophisticated data structures, but their efficient implementation on the GPU can be tricky due to the multiple factors that influence performance (e.g. cache, memory locality, concurrent accesses and write updates, etc). This paper shows that the correct usage of memory is able to shift the limits, and handle scenes that the others cannot.

An efficient approach is to use a fixed-size array of layers per pixel, because it has low costs of memory management. However it might incur in inaccurate results if any pixel requires more layers than the pre-allocated array size. Furthermore, this approach wastes memory since the number of layers varies across pixels.

The use of linked lists is a classical solution for dynamic memory allocation without pre-defined size. However, current GPUs do not have dynamic allocation of memory, so, the emulation of per-pixel linked lists in GPU buffers suffers from performance penalties caused by heuristic of memory allocation in overflow cases.

Another approach to the problem uses a base+displacement

scheme that requires a two-step computation. In the first step, only the number of layers required for each pixel is computed. Once the number of layers is known, in the second step, per-pixel layers can be evaluated and stored compactly in consecutive memory addresses. Implementing this idea efficiently on the GPU is particularly challenging, as we can observe in the implementation described in the DirectX SDK 11 [1], which crashes with an untreated overflow for scenes with an average of more than 8 layers per pixel.

In this paper we present the Dynamic Fragment Buffer (DFB), an efficient implementation that shifts the memory bottleneck for storing transparent layers, enabling the correct rendering of a larger set of scenes with high number of layers and highly variable distribution of these layers. We follow the base+displacement idea, but with significant improvements in performance and memory management. For example, Figure 1 shows screenshots of the rendering at interactive rates of the Power Plant model, which can generate frames with several hundred of per-pixel layers.

The non-trivial impact of design decisions in the GPU implementation of these algorithms led us to design a collection of experiments, which validates the proposal of our algorithm, and allows it to be compared against competing strategies. All of them also perform visibility-correct blending of fragments. In summary, the contributions introduced in the paper are:

- Dynamic Fragment Buffer, a memory-economic technique able to allocate the exact amount of memory required to the multiple per-pixel layers in each frame;
- An efficient integration of our solution using CUDA with the OpenGL pipeline, which allows rendering scenes with multiple layers at interactive frames;
- A comprehensive evaluation and analysis of competing OIT strategies, using scenes of increasing complexity and different image resolutions.

II. Related Work

There are two main classes of OIT algorithms for handling transparency computation at the fragment level: **depth-peeling** solutions, originated with the Virtual Pixel Maps [2], and **buffer-based** solutions, inspired by the A-buffer [3]. Depth-peeling approaches combine layers with a reduced memory usage, but require several passes over the geometry. On the other hand, buffer-based methods can be implemented in a single rendering pass, at the expense of memory to store multiple layers. To correct evaluate transparency, all layers must be stored in the buffer.

In a single geometry step, the **A-buffer** first stores all fragments in per-pixel linked lists, followed by a second step where lists are sorted in visibility order and blended accordingly. The Per-Pixel Linked Lists (**PLL**) [5] describes a GPU implementation closest to the A-buffer. This technique takes advantage of atomic operations, recently available in shaders, to emulate a linked list inside a buffer in shared GPU memory. Each node in the linked list contains the fragment attributes and an index to the previously stored fragment of the same pixel. A variation of this algorithm using paged

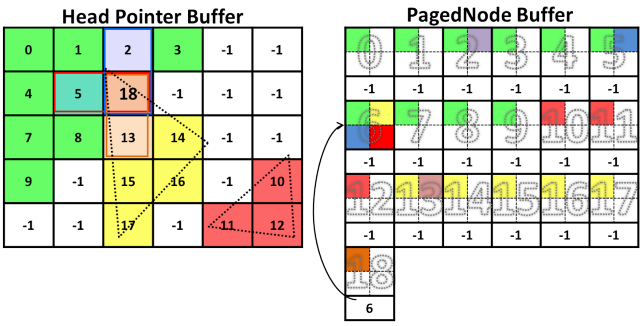

Fig. 2. Per-Pixel Paged Linked Lists: example showing three transparent triangles and three transparent lines, which are illustrated over the Head Pointer Buffer. The **Head Pointer Buffer** keeps a pointer to the PagedNode Buffer, which indexes the page of the last received fragment from the respective pixel. For example, the number 18 in the Head Pointer Buffer indicates that the last fragment for that pixel entry is stored in page 18 of the PagedNode Buffer. The **PagedNode Buffer** keeps fragments attributes in pages of size four per node, with one index to the next page and if there is no more pages, it indicates with -1. For example, in page 18, the number 6 indicates that the continuation of the list is in page 6. Image modified from [4].

per-pixel lists (**PPLL**) was described in [6]. The paging scheme, illustrated in Figure 2, stores more fragments per link, improving memory access efficiency.

The general idea of the A-buffer algorithm using a base+displacement approach is implemented in DirectX SDK 11 [1]. The method consists of two geometry passes per frame. The first pass only counts how many fragments are generated per pixel. Once concluded, a prefix sum operation computes offsets, used in the second geometry pass to store fragments inside an entry in the shared buffer. A final shader sorts the fragments in visibility ordering, before blending results in the image buffer. The performance penalty of the two geometry steps can be reduced using vertex buffer objects. However, the prefix sum operation performed in shaders can be very inefficient, and turn the application impractical for scenes with a large number of transparent layers and large screen resolutions. The S-buffer proposed by Vasilakis and Fudos [7] addressed the prefix sum issue described above, and is similar to the CUDA Thrust implementation we used in our proposal.

Slightly different from the A-buffer, the **F-buffer** [8] and **R-buffer** [9] also store all fragments, but propose to use a different structure instead of a linked list. Both algorithms keep all fragments in a general buffer, with an attribute that explicitly encodes the pixel associated to them. Fragments are processed in multiple steps in visibility ordering. At each step, the frontmost fragment is processed to blend its contribution to the resulting image. Once processed, a fragment is removed from further processing. To overcome the unbounded memory requirement, techniques were proposed using buffers of fixed-sizes[10], [11], [12] and geometry preprocessing [13].

A simplification of the A-buffer is the use of a buffer of fixed-size, which is very efficient, simple to implement and has the advantage of not needing dynamic memory management. However, it may lead to artifacts due to overflow. Correct

transparency evaluation incurs in large memory requirements proportional to $w \times h \times l$, where w and h are the width and height of the screen, and l is the maximum number of layers in the scene. An example of fixed-sized A-buffer implementation is described in FreePipe [11]. For comparison purposes, we use the fixed-size buffer implementation described in [14], which is more efficient than FreePipe because it uses GLSL 4.0 instead of CUDA, thus taking advantage of the optimized hardware pipeline. We refer to this implementation as Buffer 3D (B3D).

The depth-peeling method consists in multiple passes over the geometry to extract visibility layers in depth-sorted order. It can be performed in back-to-front order, as initially proposed by Mammen, or in front-to-back order, as proposed by Everitt [15], or in both directions, as proposed in the dual depth-peeling algorithm [16]. These approaches are able to correctly evaluate transparency and have the advantage of low resources requirements, but the geometry passes might increase the processing time.

The Stochastic Transparency method [17] differs from the other methods by exploring probabilistic sampling to approximate blending without actually performing the costly sorting operation. However, it incurs in storage of multiple samples per pixel, furthermore, the final image requires post-processing to treat the resulting noise due to the limited number of samples.

In this paper, we focus our testing and analysis on techniques able to perform correct OIT rendering. We refer the work of Maule et al. [4] for a more complete review of OIT algorithms.

III. Dynamic Fragment Buffer

The Dynamic Fragment Buffer (**DFB**) was designed to render high quality images, composed by multiple transparent layers, at the best memory usage possible. It uses a base+displacement approach, because it provides compact storage of multiple fragments. Furthermore, most steps can be efficiently implemented by shader programs, leveraging the optimized graphic hardware pipeline.

Rendering of the opaque portion of geometry is more efficient when done on a separated geometry step, which sets up the framebuffer background and the depth buffer. In the following transparency rendering step, the depth buffer is tested to discard occluded fragments, but it is not updated. Rendering opaque and transparent fragments altogether is possible, but implies in wasting processing time and memory by shading and storing occluded fragments. When working with transparent textures, such as foliage rendering, it may be inevitable.

The algorithm for transparency rendering has four modules: (i) fragment counting, (ii) prefix sum, (iii) fragment storing, and (iv) sorting and blending. Fragment counting uses a geometry pass to render primitives without shading, in order to evaluate the number of fragments to be stored at each pixel. The prefix sum accumulates these counters to compute a base address, which points to the fragment list associated

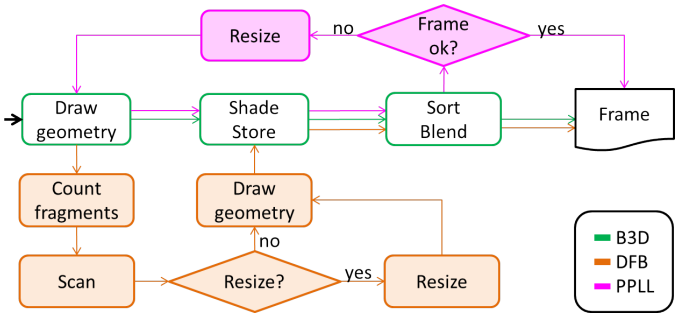

Fig. 3. Execution flow: an overview of similarities and differences among the Buffer 3D (B3D), Dynamic Fragment Buffer (DFB) and Paged per Pixel Linked Lists (PPLL).

to each pixel in the shared buffer. A second geometry pass is performed to compute and store fragments with shading. The final step consists of the sorting of all fragments of a given pixel, followed by blending. Figure 3 shows the execution flow of DFB, along with the competing strategies B3D and PPLL, with three modules in common: draw geometry, shade/store and sort/blend. We detail the DFB modules below.

A. Fragment Counting

The first step of the algorithm uses a geometry pass to count the number of fragments per pixel. For this purpose, the scene geometry is rendered without shading. The algorithm assumes that opaque geometry is already processed, so, transparent fragments are tested against the depth buffer without modifying its value, in order to discard occluded fragments.

We use a shader to count fragments, with each thread being responsible for updating the count in a *countingBuffer*. This is implemented using an atomic increment, in order to avoid read-after-write race conditions among concurrent threads. A 32-bit unsigned integer texture is used to represent the *countingBuffer*. The stencil buffer could also be used for this purpose, but its 8-bit representation would limit the scenes to 256 layers. This limitation would prevent rendering models such as the Power Plant, which has cases of more than 500 layers (Section IV). Since no fragments are rendered and the color buffers contents are not read in this stage, the color buffer is not cleared, which improves performance.

Conversely, the *countingBuffer* must be cleared, and the simplest solution would be calling a shader which would impose a memory synchronization barrier. However, this can be done more efficiently. Before the first frame, a shader is called to clear the *countingBuffers*. For the subsequent frames, instead of calling a new shader, we clear this buffer at the end of the last stage of the algorithm (sorting/blending stage).

B. Prefix Sum

In the second stage, we prepare the indexing structures to store all fragments consecutively in memory. For this purpose, for each pixel, we define a base index that points to the address in the shared buffer where the associated fragment list starts. This is done using a prefix sum over the list of per-pixel counting results, computed in the previous stage.

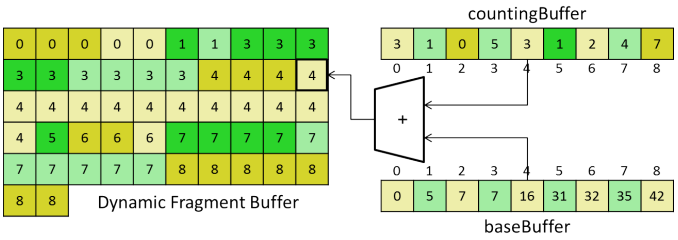

Fig. 4. DFB storage. The pixel-correspondent value from the *countingBuffer* s added to the associated index from the *baseBuffer*. This composed address stores the incoming fragment into a free position in the shared buffer.

Prefix sum is implemented efficiently on the GPU using a scan operation. Starting from zero, the scan iteratively accumulates the counting of fragments from previous pixels, and results are stored in a *baseBuffer* texture. After the scan ends, the per-pixel base index, pointing to a reserved list into the shared buffer, is stored in the *baseBuffer*. The last base combined with the last counter indicates the total number of transparent fragments generated for the entire frame.

We use the optimized parallel scan implementation provided by the **Thrust 1.5.1** [18] library, with additional cost of memory copies. Since Thrust is still being developed, the direct access of CUDA resources by Thrust operators is not available. However, we can copy a CUDA resource array, like the counting and base buffers, to a Thrust vector, where the scan operation can be performed quickly. Once the scan ends, data is copied back to the CUDA array, which consists of a pointer to the textures accessed in subsequent shaders. To end this step, the *countingBuffer* is efficiently cleared.

C. Storing Fragments

The second geometry pass is responsible for the generation of the fragments with shading, and its sequential storage in shared memory (which we call the *Dynamic Fragment Buffer*). The per-pixel base index (*baseBuffer*), computed in the previous phase, is used to address the fragment list in the shared buffer. Each base address is added to the incoming fragment count to define the address of the next free position to store the fragment. So, before start this step, the *countBuffer* must be cleared. In this step, the framebuffer is not used, and do not need to be cleared. As nothing is read from the shared buffer yet, it does not need to be cleared either. Figure 4 illustrates the addressing scheme for an incoming fragment.

D. Sorting and Blending Fragments

The last stage of the algorithm sorts and blends fragments from the shared buffer. The base indices are used to determine the start of the fragment list, and the counters state how many fragments need to be considered for each particular pixel.

A shader pass is used to first sort the pixel lists of fragments, followed by the colors compositions. We launch one thread for each pixel, which is responsible to perform an insertion sort in the pixel list and traverse it in front-to-back order, blending the fragments. The *baseBuffer* indicates were the pixel list start, and the *countingBuffer* indicates how many pixels the list contains. If the list is empty, no color is sent to the framebuffer.

As mentioned before, the *coutingBuffer* is cleared in this step, thus avoiding an extra clear shader call.

Resolution	B3D	DFB	PLL4	PLL1	DxDFB
500^2	1250	500	500	333	11
800^2	625	294	213	141	4
1100^2	345	170	114	74	0

TABLE I

Performance (in FPS) comparison for the decimated Bunny model (69K triangles). Small resolutions, in the first column, were used to be able to capture the performance of DxDFB.

IV. Experiments

Experiments were performed in a computer with an Intel i7 980 processor and a GeForce 580 (1.5 GB working memory) running Windows 7. We compare the DFB implementation against competing techniques using a collection of scenes with varying geometry and number of transparent layers (non-opaque) under different screen resolutions. All techniques were completely implemented using GLSL 4.0, only the DFB scan makes use of Thrust 1.5.1 with CUDA 4.0. Instead of rendering the result to the framebuffer, we used Frame Buffer Objects (FBO) to generate images with resolutions not supported by our monitors.

As discussed in the related work, a base+displacement algorithm closest to our proposal was described in the DirectX SDK11 [1]. We made minimal modifications to the original DirectX Deep Frame Buffer proposal (**DxDFB**), just to load 3D models and get performance measurement values by the *QueryPerformanceCounter* Windows function. The **B3D** and the **PPLL** implementations were instrumented with the same performance measurement code. For these techniques, and for the **DFB**, we used an insertion sort algorithm due to its efficiency for arrays with few elements. PPLL was tested in two versions: (i) four fragments per-page (**PPLL4**) and (ii) a single fragment per-page (**PLL**), emulating the original per-pixel linked lists technique.

To validate the algorithms under different scene configurations, we measured performance with a virtual walkthrough around the scene. The viewing changes allow the algorithms to be naturally tested in situations with varying image coverage, visible geometry and number of transparent layers. We evaluated synthetic scenes, variations of instances with simple geometric models, as well as a stress test with a complex CAD model. Since B3D uses a fixed per-pixel array, without overflow checking. So, we set its size to the maximum number of transparent layers (computed offline by a tracer program for each test configuration). For all tests, we guarantee the correct image generation.

The DxDFB implementation showed to be very limited, as shown in Table I, we could only evaluate it using a decimated version of the Stanford Bunny (with 69K triangles).DxDFB

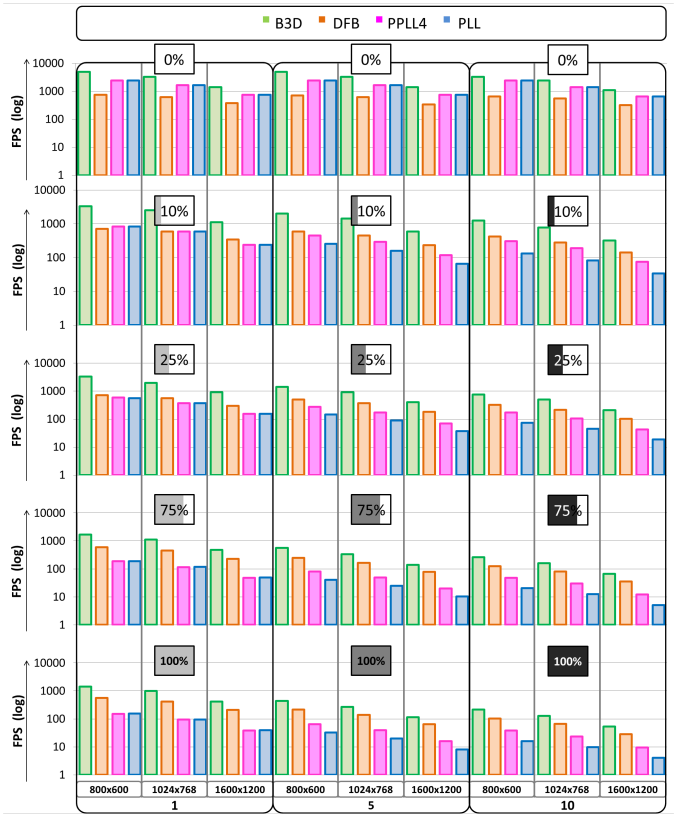

Fig. 5. FPS for synthetic scenes. Scenes are composed of 1, 5 and 10 quads, covering different percentages of the screen, for three image resolutions. Illustrations of the coverage are presented above each graph.

has the lowest FPS among all methods, and does not run at higher resolutions. DFB is much more competitive and faster than PLL and PPLL, while being the one to use the least memory of all. The B3D algorithm is the fastest among all methods due to its simplicity, and in this Bunny scene it is twice as fast as the others. However, the B3D method is severely limited by the amount of memory available, and has trouble dealing with scenes with uneven distribution of transparent layers across all the pixels (because it allocates the same number of layers, regardless of how many are needed).

Figure 5 shows the results of synthetic scenes used to evaluate the characteristics of the techniques under controlled parameters. The scenes are composed only by quadrilaterals to alleviate the geometry processing cost, and let us focus in the other aspects. We changed (i) the screen resolution, (ii) the number of transparent layers, by introducing more quadrilaterals, and (iii) the percentage of pixels covered on the screen.

Memory consumption of each technique depends on the attributes stored for each pixel, and can be calculated as follows: (i) B3D is proportional to $(width \times height) \times max\_layers$, (ii) PPLL, to $(width \times height \times percent) \times (\lceil \frac{max\_layers}{page\_size} \rceil \times page\_size)$, and (iii) DFB, to $(width \times height \times percent) \times max\_layers$. Even for 0% coverage, DFB still has to perform two geometry steps, thus decreasing performance. When more

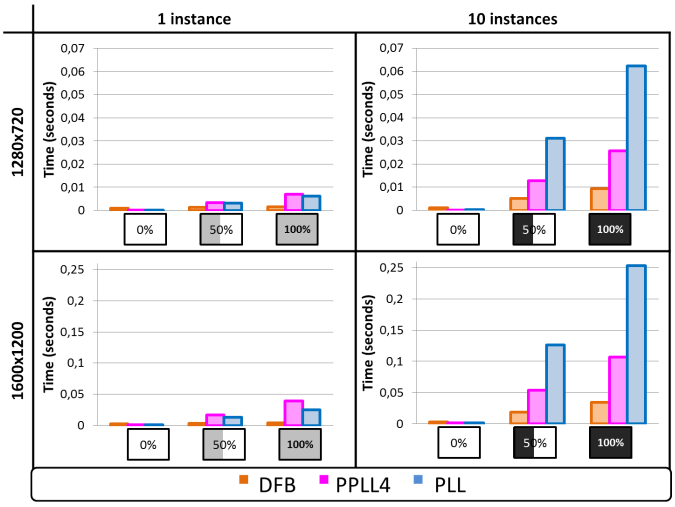

Fig. 6. Memory management costs for synthetic scenes: costs are given in seconds for scenes with 1 and 10 quads, with varying resolutions and screen coverage. Only methods with dynamic memory allocation are considered.

than 25% of the screen is covered, DFB is the second fastest technique. Note that B3D wastes memory in inverse proportion to the percentage covered (e.g., when 25% of the screen is covered, 75% of the memory reserved by B3D is wasted).

The same synthetic scenes are used to evaluate the isolated costs of memory management, associated to the techniques with dynamic memory allocation. Thus, B3D, which uses a fixed buffer, is not considered here. Figure 6 shows the time consumed by overflow checking and buffer resizing. It includes *counting* and *scan* stages, and the buffer resize when a future overflow is detected by the DFB. For the PPLL techniques, it includes overflow queries and the heuristic buffer resize (which simply doubles the size needed by the last frame, not taking into account that some pages are not completely full). When the screen resolution increases 25%, we observe the PPLLs performance decreases by an order of magnitude, while DFB costs increase at a slower rate.

Figure 7 presents a series of performance and memory measurements for the Stanford Dragon model. The memory consumption, given in colored boxes, represents the largest amount required by a given method to render all frames in the virtual walkthrough. We separate the results in three parts, corresponding to scenes with 1, 5 and 10 instances of the model. Scenes with more than one instance have them disposed side by side, and the camera navigates elliptically around them. Each scene is rendered with 3 different resolutions, for a total of 9 graphs. As expected, B3D is the fastest algorithm. However, it can not handle 4 out of the 9 tests, because the system does not have the required memory. Discarding B3D from the analysis, DFB has the best performance in 8 of the 9 tests, while having the lowest memory consumption.

We also performed experiments with a massive CAD model, the entire Power Plant, composed of **12,748,510** triangles. The camera visits the outside and interior of the model, varying

drastically the quantity of visible geometry and transparent layers during the tour. Because this model has a large number of triangles and transparent layers (up to 590), memory requirements are high and the B3D approach is unable to run correctly (it would require more than 4GB for fragment storage). This case also illustrates the situation where the stencil buffer cannot be used to count fragments, since it can only count up to 256 layers. All techniques with dynamic memory allocation can render this scene at barely interactive rates (between 4 and 8 FPS on average), but only at the low resolution of 800x600.

Figure 8 demonstrates a virtual walkthrough in the Power Plant model. At the top, some screenshots represent the model coverage on the screen and its relation to performance (at the bottom of the figure). More pixels covered leads to peaks of processing time, which is represented by the lines plotted in the front plane. In the background, a stacked chart presenting the count of pixels containing different number of layers. We can also observe the impact of the number of layers over the performance of the methods. We clearly observe the DFB time line smoother than the others. The peaks in PPLL4 and PLL plots are caused by the hick ups of memory reallocation. In the virtual walkthrough, they represent frozen frames. We observed a correlation between peaks and the increase of depth complexity in the frames. DFB not only has a consistent superior performance, but also requires less memory. This is because overflows are easier identified by the total amount of memory required by the frame, and DFB knows the exact amount of memory that need to be allocated in such cases.

V. Discussion

In this section we discuss the results obtained above with regards to the geometry costs, impact of varying image resolutions, costs of memory allocation, and image quality.

A. Performance Impact due to Geometry Processing

Scenes with a large number of triangles and few transparent layers configure the worst situation for the DFB technique due to its two geometry passes. The impact of the two passes can be clearly seen in Figure 7 for the lowest resolution with 10 instances. For the PPLL techniques, the amount of geometry influences the most when a frame must be rebuilt due to overflow. B3D, which uses a single geometry pass, is less affected than the others with respect to geometry costs.

B. Performance Impact due to Image Resolutions

The scan operation is the costliest step of the DFB, and its performance is directly dependent of the screen resolution. In our tests, the Thrust *exclusive_scan* showed to be data dependent, which means that its performance changed with the number of transparent layers in the frame. Due to the fixed array allocation, the B3D approach required more memory to keep all fragments. This was especially important for high depth-complexity scenes, where B3D was unable to handle large resolutions without exceeding the memory limits.

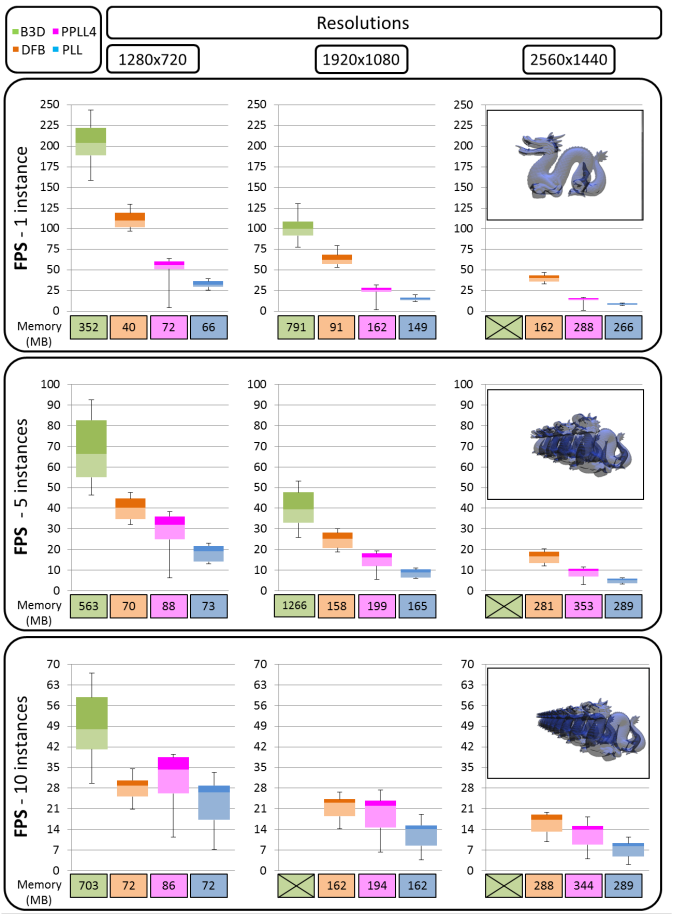

Fig. 7. FPS and memory consumption for the Stanford Dragon model. Tests with scenes composed of 1, 5, and 10 instances of the model and three image resolutions. One model has 871,414 triangles, with up to 20 transparent layers. Five instances can go up to 28 layers, and 10 instances up to 46 layers. Performance graphs have different scales to better present the differences among techniques when the number of instances changes. Boxplots display performance variations among all frames in the virtual walkthrough, with 25, 50 and 75 percentiles, and upper and lower errors.

PPLL also decreases performance when image resolution increases, but the causes are different and the performance loss is greater. To test for overflow, PPLL must wait until the GPU finishes processing the current frame, causing a pipeline stall. When more pixels are covered, the heuristic re-allocation causes the frame to be rebuilt many times, always waiting to know whether there was overflow or not. Performance degradation due to such stalls can be seen in all results (Figure 5, 6, 7 and 8), especially in the peaks of PLL and PPLL plots in Figure 8.

As can be seen in Figure 6, the DFB performance degrades smoothly as the resolution increases, and its memory consumption is lower than the other approaches (Figure 7). This makes the DFB the most suitable technique for applications that require rendering large image resolutions.

C. Performance Impact due to Memory Allocation

The DFB was named after its ability of adjusting itself at each frame to use the minimum memory possible, while

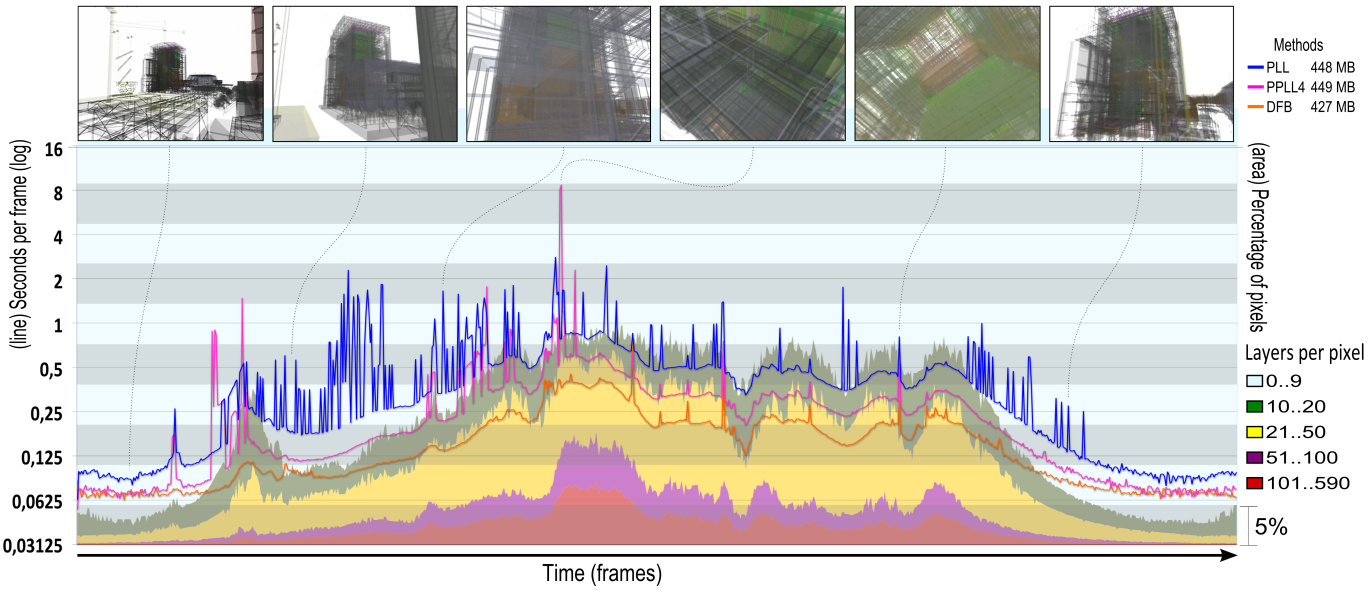

Fig. 8. A tour around the Power Plant model: only DFB, PPLL4 and PLL were able to visualize the full Powerplant with **12,748,510** triangles and up to **590** transparent layers, at 800x600. Screenshots show frames captured during the tour. The graph lines show the seconds to generate each frame of the tour. Color-coded areas in the background show the percentage of pixels with different numbers of transparent layers; each strip indicates ranges of 5%, up to a total of 50%. The other half is filled by fragments with [0..9] layers. Peaks in PLL and PPLL4 are caused by memory reallocation.

keeping the necessary fragments to correctly evaluate OIT using less memory than the others. Since it requires contiguous storage, an amount of memory must be pre-allocated to assure contiguity.

PPLL and B3D techniques also need contiguous memory, but they cannot estimate the correct buffer size because, having a single geometry pass, they are unable to count the number of fragments that must be stored in each frame. Because of that, both techniques may waste memory, or lose fragments when the allocated number of layers is not enough (overflow).

Overflow cases may produce incorrect renderings. PPLL can detect these cases, resize the buffer and rebuild the frame. Since this technique does not know how many fragments were lost, it can not precisely increase the buffer size. Thus, a heuristical increase of the buffer size is applied and the frame is re-generated, overflow is checked again, and the process repeated until the buffer size is large enough, or until there is no more memory available. The consequences of this iterative memory allocation are the elevated costs presented in Figure 6, and the peaks of performance loss in Figure 8. Because the DFB monitors closely the number of fragments per pixel, and the total number per frame, an overflow only occurs when the GPU memory is insufficient to handle the frame.

D. Performance Impact due to Memory Addressing

Both the DFB and B3D have sequential per-pixel arrays of fragments, which helps to improve memory locality. However, B3D uses an array of fixed size, thus forcing a conservative choice of slots per pixel, leading to memory wasting. When arrays are processed in B3D, the look-ahead cache may copy trash from unused entries. The array organization of DFB avoids both wasting and copying trash to the cache.

Differently from DFB and B3D, the main PLL drawbacks with respect to memory addressing are the bottlenecks generated due to the use of a single counter for all pixels, leading to serialization, and the random memory access. Both problems are also shared by PPLL, which reduces them by creating pages of consecutive fragments. But, it is still hard to define a good page size. If the page is too small, many random accesses are required to assemble the list; if the page is too big, memory is wasted and trash is copied to the cache. The same does not occur with the DFB, which has fragments from the same pixel sequentially organized in the shared buffer.

B3D has better performance than DFB since it has only one texture read to get the fragment list, while DFB needs two. PLL and PPLL need to follow several links of the list, using random memory accesses, to assemble the entire fragment list.

E. Image Quality

All the algorithms evaluated were able to compute correct images of transparent scenes if the system provides enough memory. Since the physical components are limited, management of the existing memory is necessary to avoid waste and allow generation of complex scenes with several layers.

DFB is able to dynamically require more memory while producing the frame. Thus, an overflow only occurs when there is no more memory available for the GPU. The DxDFB implementation could do the same, but the implementation fixed the buffer to an average of eight fragments per pixel. For simple scenes this is enough, since some of the pixels might not have transparency layers, while others may have more than the average number.

Differently, PPLL detects and treats overflow cases, as described above. Once the algorithm detects an overflow, it

increases the buffer by an estimated size until the overflow no longer happens, or the memory required is not available. For all techniques, artifacts can happen if the memory is not enough. By using less memory, DFB is less susceptible to such artifacts. B3D could use a similar heuristic to increase the buffer size, but the reallocation of the entire buffer due to one missed fragment could result in severe performance loss.

VI. Conclusions and Future Work

In this paper we introduced DFB, a memory-efficient algorithm to render, at interactive rates, high-quality images from scenes composed of several transparent layers. When compared to fixed-size approaches such as B3D, DFB presents memory savings proportional to the asymmetry of layers among pixels in the frame. With increasing number of transparent layers and screen resolution, B3D becomes impractical due to its high memory requirements. Compared with PPLL, the DFB performance advantage increases with the number of transparent layers and screen resolution, due to the interactive memory allocation used by PPLL.

Results shown that DFB has the best memory usage, with the same image quality at competitive frame rates. It also presented significant improvements over the DxDFB technique, being able to achieve interactive frame rates, thus making our solution competitive with the state of the art OIT techniques. Even for scenes with a large amount of geometry, like the Power Plant, DFB proved to be the best choice.

There are several avenues of future work. We would like to extend our algorithm to support larger image resolutions for massive models, possibly using some space partitioning in the scene. In addition, we would like to revisit recurrent problems of OIT methods, such as z-fighting and aliasing.

Acknowledgment

This work was sponsored by a scholarship by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES). We also like to thank Vitor Pamplona for additional support.

References

[1] Microsoft, "Oit11 sample," http://msdn.microsoft.com/en-us/library/ee416572%28v=VS.85%29.aspx, 2010.

[2] A. Mammen, "Transparency and antialiasing algorithms implemented with the virtual pixel maps technique," *IEEE Comput. Graph. Appl.*, vol. 9, no. 4, pp. 43–55, 1989.

[3] L. Carpenter, "The a-buffer, an antialiased hidden surface method," in *SIGGRAPH '84: Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, 1984, pp. 103–108.

[4] M. Maule, J. ao L.D. Comba, R. P. Torchelsen, and R. Bastos, "A survey of raster-based transparency techniques," *Computers & Graphics*, vol. 35, no. 6, pp. 1023 – 1034, 2011.

[5] J. C. Yang, J. Hensley, H. Grűn, and N. Thibieroz, "Real-time concurrent linked list construction on the GPU," *Computer Graphics Forum*, vol. 29, no. 4, pp. 1297–1304, 2010.

[6] C. Crassin, "Paged per pixel linked lists," http://blog.icare3d.org/2010/07/opengl-40-abuffer-v20-linked-lists-of.html, 2010.

[7] A. A. Vasilakis and I. Fudos, "S-buffer: Sparsity-aware multi-fragment rendering," *Eurographics Short Paper*, 2012.

[8] W. R. Mark and K. Proudfoot, "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*, 2001, pp. 57–64.

[9] C. M. Wittenbrink, "R-buffer: a pointerless a-buffer hardware architecture," in *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, 2001, pp. 73–80.

[10] N. P. Jouppi and C.-F. Chang, "Z3: an economical hardware technique for high-quality antialiasing and transparency," in *HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, 1999, pp. 85–93.

[11] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu, "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*, 2010, pp. 75–82.

[12] M. Salvi, J. Montgomery, and A. Lefohn, "Adaptive transparency," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ser. HPG '11. New York, NY, USA: ACM, 2011, pp. 119–126. [Online]. Available: http://doi.acm.org/10.1145/2018323.2018342

[13] L. Bavoil, S. P. Callahan, A. Lefohn, a. L. D. Comba, Jo and C. T. Silva, "Multi-fragment effects on the GPU using the k-buffer," in *I3D '07: Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, 2007, pp. 97–104.

[14] C. Crassin, "Fast and accurate single-pass a-buffer using OpenGL 4.0+," http://blog.icare3d.org/2010/06/fast-and-accurate-single-pass-buffer.html, 2010.

[15] C. Everitt, "Interactive order-independent transparency," Technical report, NVIDIA Corporation, 2001.

[16] L. Bavoil and K. Myers, "Order independent transparency with dual depth peeling," Technical report, NVIDIA Corporation, 2008.

[17] E. Enderton, E. Sintorn, P. Shirley, and D. Luebke, "Stochastic transparency," in *I3D '10: Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2010, pp. 157–164.

[18] nVidia, "Thrust," http://code.google.com/p/thrust/.