Pacific Graphics (2013), pp. 1–6
B. Levy, X. Tong, and K. Yin (Editors)

Backwards Memory Allocation and Improved OIT

P. Knowles, G. Leach, and F. Zambetta

RMIT University, Melbourne

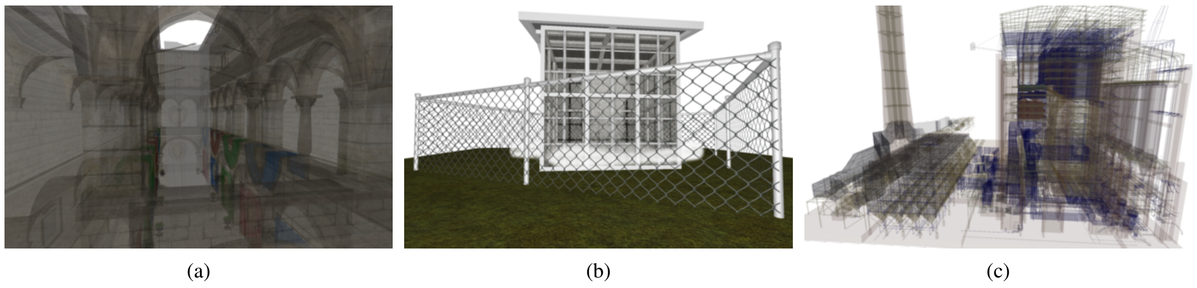


Figure 1: (a) The Sponza Atrium, (b) Smoke and (c) Power Plant scenes, rendered with exact transparency at 26, 12 and 5 FPS respectively using *backwards memory allocation* with 1680×1050 resolution and GeForce GTX 670.

Abstract
Order independent transparency *(OIT) is a graphics technique which sorts surfaces per-pixel for correct alpha blending. The sorting stage requires relatively large amounts of temporary memory in shaders that is usually conservatively allocated at a maximum, which impacts occupancy and performance. To address this issue we introduce* backwards memory allocation *(BMA), a strategy which creates a set of shaders with varying static allocation size in lieu of dynamic allocation. Batches of threads are then executed directly with the appropriate shader. This also allows optimizations for each generated shader such as choosing the sorting algorithm based on allocation size with no additional overhead. BMA gives both a more flexible OIT (BMA-OIT) for dynamic scenes of varying depth complexity and up to a 3× speedup.*

1. Introduction

Order-independent transparency (OIT) is a graphics technique which sorts surfaces per-pixel instead of per-polygon, thus independent of polygon rendering order. This is similar to the image based way the depth buffer solves hidden surface removal and has significant programming advantages. The underlying technique must capture and store all fragments during rasterization, which itself is non-trivial and is discussed in Section 2. Apart from transparency, the sorted fragment data is expected to be useful in other graphics applications.

Fragment sorting becomes a bottleneck for OIT in scenes with a depth complexity of approximately 32 or more fragments per pixel [KLZ12]. A significant sorting time factor in current implementations is the need to allocate a conservative maximum, or "worst case", amount of local memory in shaders in which to perform the sort. The performance impact arises due to local memory usage affecting GPU *occupancy*, discussed in Section 3. In this work, *shaders* refers to GLSL/HLSL programs.

We introduce *backwards memory allocation* (BMA) in Section 3, a strategy to allocate nearly — within a factor of two — the right amount of memory per thread and reduce the performance impact of large local memory requirements in shaders. We demonstrate BMA, applying it to the sorting stage of OIT (BMA-OIT) and show up to a 3× speedup. While BMA works well with OIT we believe it also has potential for other rendering applications.

2. Order-Independent Transparency

OIT with graphics hardware was first accomplished via *depth peeling* [Eve01], which required many rendering passes of the geometry. The introduction of random writes to global graphics memory and atomic operations in graphics hardware allows all rasterized fragments to be computed and stored in a single rendering pass. These features are available in OpenGL versions 3.2 (`ARB_shader_image_load_store`) and 4.1 (`ARB_shader_atomic_counters`). The fragments can be stored in various ways: (1) as a 3D array [LHLW09] where (x,y) per-pixel fragments are stacked along z, (2) in per-pixel linked lists [YHGT10] or (3) linearly packed in a 1D array [Pee08]. In this work we use per-pixel linked lists, usually requiring one rendering pass.

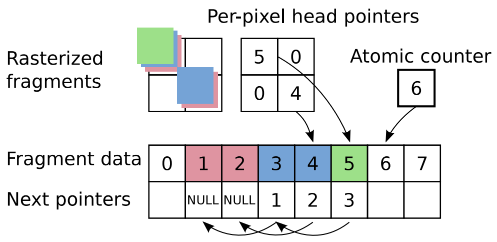


Figure 2: Per-pixel linked lists.

To construct per-pixel linked lists during rasterization, scattered writes are performed in the fragment shader instead of writing to the framebuffer. The process is visualized in Figure 2. Firstly, a global-scope atomic counter is incremented for a unique fragment storage index. An array of head-pointers, one for each pixel, is used to mark the start of each list. A next pointer is stored at the same index as the fragment, either with the fragment or in a parallel array. The fragment/next pointer node is inserted into the front of the list via an atomic exchange as follows:

```
head = imageAtomicExchange(headPtrs, pixel,
   index).r;
imageStore(nextPtrs, index, head);
imageStore(data, index, fragmentData);
```

The atomic counter contains the number of fragments generated after rendering. If there is not enough memory to store all fragments, data is lost and a resize and re-render is required. Hardware-supported atomic counters in recent GPUs have little overhead and the non-sequential storage has not been observed to affect OIT performance significantly [KLZ12]. The linked list data is used for both OIT and BMA-OIT, although BMA-OIT requires additional per-pixel fragment counts.

To render transparency, fragments are composited using alpha blending after sorting. Only one pass through the fragments is needed for alpha blending so storing the sorted result is unnecessary. Both sorting and compositing can be performed in the same shader, for example:

```
vec4 frags[MAX_FRAGS];
node = imageLoad(headPtrs, pixel).r;
while (node && count < MAX_FRAGS)
{
   frags[count++] = imageLoad(data, node);
   node = imageLoad(nextPtrs, node).r;
}
... sort frags
... composite frags
```

3. Backwards Memory Allocation

Fragments require sorting before computing transparency, and this quickly becomes the bottleneck for non-trivial scenes. Sorting is performed by a shader in local memory as global memory access has high latency. However, using local memory can be expensive as it can impact *occupancy*. BMA attempts to address this, but first we describe the problem in more detail.

Local memory in shaders exhibits the behaviour of residing in reserved memory equivalent to L1 cache and CUDA's *shared* memory, shared across each SIMD processor. SIMD processors swap execution of threads to keep busy and hide latency. To avoid copy operations for these context switches, resources for all active threads must remain resident during their lifetime. Therefore the number of possible active threads ("occupancy") is limited by the available resources and the portion individual threads require. Thus, as the allocated local memory size increases, the number of possible concurrently active threads in each SIMD processor decreases. Latency cannot be hidden without enough active threads in a SIMD processor and in extreme cases, there may not even be enough threads to fill the SIMD processor.

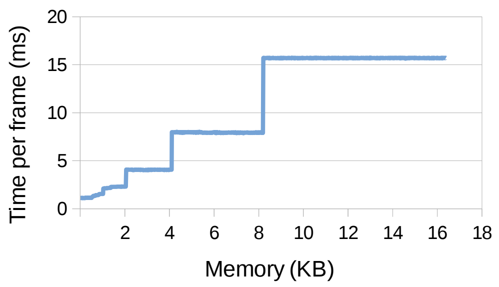


Figure 3: Effect on performance of increasing local memory usage in shaders.

The effect of increasing local memory usage in shaders is shown in Figure 3, where the following fragment shader is executed, rendering a full screen polygon to an 800 × 600 window 64 times with varying `SIZE`. The uniform variable `zero` is required to keep the compiler from optimizing out the array. Note the stepped shape of the graph produced by memory-limited occupancy. This demonstrates how important it can be to maintain low local memory usage in shaders.

```
#define SIZE set_by_application
vec4 myArray[SIZE];
uniform int zero;
out vec4 fragColour;
void main()
{
    fragColour = myArray[zero];
}
```

Dynamic local memory allocation is unavailable in shaders so a common approach, when running out of memory would be unacceptable, is to allocate a maximum. For example, significant artifacts can occur in OIT when there is not enough memory (see Figure 6, Section 4) and allocating a maximum introduces an unnecessary performance overhead as much of it is unused. OIT is a good example as scenes commonly have a low overall depth complexity relative to the maximum and just a few pixels require a lot more memory for sorting, as shown in Figures 5 and 7 (Section 4). This performance characteristic is expected to occur in general for applications with both varying and occupancy-limiting local memory requirements

We introduce BMA to address the performance penalty of large local memory allocation in the cases where it is underused. BMA groups threads by their memory requirements at run-time and execute each using a different shader program with appropriate local memory defined. We term this strategy backwards memory allocation primarily as permutations of shaders must be pre-compiled with fixed memory, reversing the memory demand/request order of common dynamic allocation. Secondly, the concept is also somewhat unusual compared to typical CPU programming practice in the context of local memory. The term *binned* or *batched memory allocation* is an accurate alternative.

BMA can increase occupancy for threads with varying local memory usage and has potential for improving SIMD coherency since similar threads are executed together, although this has not been explicitly observed in our transparency experiments.

Our implementation of BMA-OIT begins by defining a set of allocation intervals to group per-pixel threads, each with varying numbers of fragments to process and hence varying local memory requirements. Shader programs to sort and composite fragments are then generated for each interval. Given the steps in Figure 3, we choose intervals in powers of two beginning at eight, for example: 1-8, 9-16, 17-32, 33-64, 65-128. The last interval processes pixels of 65 fragments and up but only sorts the first 128 and discards the rest. Note that zero is omitted from the first interval to ignore pixels without fragments. We have found exhaustively optimizing these intervals for specific views produces similarly exponentially spaced values and gives performance increases of the order of 5%, but in general the powers of two approach works well.

The BMA-OIT shader structure is as follows, where `MAX_FRAGS` is set to the upper bound of each interval:

```
#define MAX_FRAGS set_by_application
vec4 frags[MAX_FRAGS];
int pixel = PIXEL_ADDR(gl_FragCoord.xy);
int count = loadFragments(pixel);
sortFragments(count);
fragColour = compositeFragments(count);
```

Per-pixel fragment counts are required for BMA-OIT, which are recorded while rendering the scene in addition to the linked lists of fragment data. We use `imageAtomicAdd` to compute counts, although additive blending and incrementing the stencil buffer are alternatives.

The stencil buffer is then used to process only pixels with fragment counts in each interval by the appropriate shader program. The process is summarized as follows:

1. Render the scene to linked lists, storing all fragments and computing per-pixel counts.
2. For each fragment count interval,
    a. Clear stencil buffer.
    b. Render a full-screen polygon to the stencil buffer, discarding for pixels with fragment counts outside the interval.
    c. Bind the OIT sort-and-composite shader for the current interval and render another full-screen polygon, using the stencil buffer to mask out discarded pixels in the previous step.

Using the stencil buffer instead of calling discard in the fragment shader is fundamental to BMA as only shaders for the correct intervals are executed. An alternative to using the stencil buffer is to compute lists of pixel IDs within each range and execute threads directly, although we have found the stencil buffer performs better.

Incrementing the stencil buffer while rendering the scene can be faster than `imageAtomicAdd` and avoids clearing and re-rendering the mask for each interval. Unfortunately this method, unlike the one outlined above, imposes a limit of 255 fragments per-pixel. If stencil buffer incrementing is used, BMA-OIT can be computed by rendering full-screen polygons for intervals in descending order with the following stencil attributes:

```
glStencilFunc(GL_LEQUAL, intervalMin, 0xFF);
glStencilOp(GL_KEEP, GL_ZERO, GL_ZERO);
```

This processes all pixels within the current interval and removes them from following passes by zeroing the stencil value.

With fragment lists already grouped into depth complexity intervals and separate shader programs, only small changes are required to optimize each program for its interval, such as using sorting algorithms appropriate to the fragment count range. While this is also possible dynamically in standard OIT, we have observed better results with BMA. This is discussed further in Section 4.

4. Results

All performance results vary scene depth complexity in scale and distribution with a fixed 1680 × 1050 resolution. We compare performance of standard OIT and BMA-OIT using a benchmark approach. We use the three scenes in Figure 1, also used in previous work, including a fly-through of the Atrium. Depth complexity is a major factor in the performance of OIT and is visualized for the scenes in Figure 4. The distribution within each scene is further shown in Figure 5. All results were obtained using an NVIDIA GeForce GTX 670.

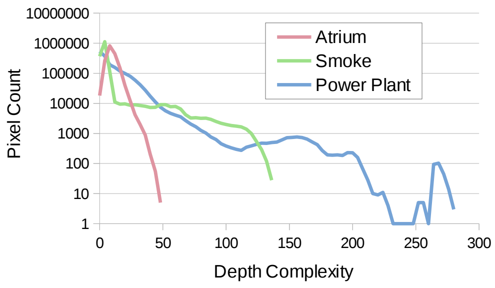
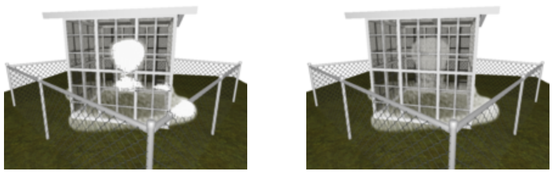


Figure 5: Depth complexity histogram of the three scenes in Figure 1. Note the log scale — most pixels have a relatively low depth complexity.

Exceeding the maximum sorting memory and discarding fragments can produce wildly incorrect results, as Figure 6 shows. This is why it is important to conservatively allocate a large amount of memory for standard OIT and handle the worst case complexity, even though this maximum is not used in many views.



(a) (b)

Figure 6: Potential artifacts due to an insufficient sorting array size. Image (a) sorts up to 32 fragments, missing window and fence fragment and (b) correctly sorts up to 64.

BMA increases occupancy during the execution of threads with low local memory usage relative to the maximum. BMA-OIT adapts to the current depth complexity on a per-pixel basis. Table 1 shows the speedup BMA-OIT provides is a result of processing the lower depth complexity intervals faster. BMA-OIT gives the speed of smaller allocation (for both various areas of a static view and various views of a scene) and still supports correct results where higher allocation is required.

Interval	Atrium	Smoke	Power Plant
8	1.14	2.11	2.14
16	0.97	1.18	2.74
32	0.99	1.67	3.78
64	1.00	2.32	3.03
128		1.68	1.99
256		1.00	1.48
512			1.00

Table 1: Estimated speedup BMA-OIT gives over OIT at each interval. As OIT does not have intervals, minor error may exist due to measuring per-interval thread execution.

Table 2 shows rendering times for three scenes using OIT and BMA-OIT with varying local memory allocation. Note that for the Power Plant, 256 fragments is insufficient for some pixels. Sorting 1024 fragments was not possible using the GTX 670. Results include using different sorting algorithms based on fragment count, which we discuss later.

Comparing the highest **Max Alloc** interval in which pixels exist shows a fair comparison between OIT and BMA-OIT within the same view. In this case BMA-OIT performs the same as OIT for the Atrium scene and gives a 2.08× and 2.93× speedup for the Smoke and Power Plant scenes.

The overhead for BMA-OIT is computing per-fragment counts and executing OIT for pixels in batches. From Table 2, supporting an additional **Max Alloc** interval (in which no pixels exist so no computation is performed) adds an overhead of 0.5% and 0.1% for the Atrium and Smoke scenes respectively. In contrast, standard OIT rendering increases by 151% and 176% for the additional and unnecessary increase.

	Atrium		Smoke		Power Plant	
Total Polys	279,178		13,468		12,701,147	
Total Frags	14,449,340		10,072,458		18,479,488	
Peak Depth	46		136		278	
Max Alloc	64	128	256	512	256	512
OIT (ms)	37.6	53.8	167.7	273.6	372.0	604.0
BMA-OIT (ms)	37.6	37.8	80.7	80.9	203.0	206.0
Speedup (×)	1.00	1.42	2.08	3.38	1.83	2.93

Table 2: Comparing standard OIT and BMA-OIT rendering times. **Max Alloc** is the local memory allocated for OIT and upper limit for BMA-OIT.

The Atrium fly-through in Figure 7 shows the performance of OIT and BMA-OIT as the scene and its depth complexity distribution changes. The 33–64 and 65–128 fragment count intervals, shown separately in Figure 8, have relatively few pixels and there are no pixels in the 129–256

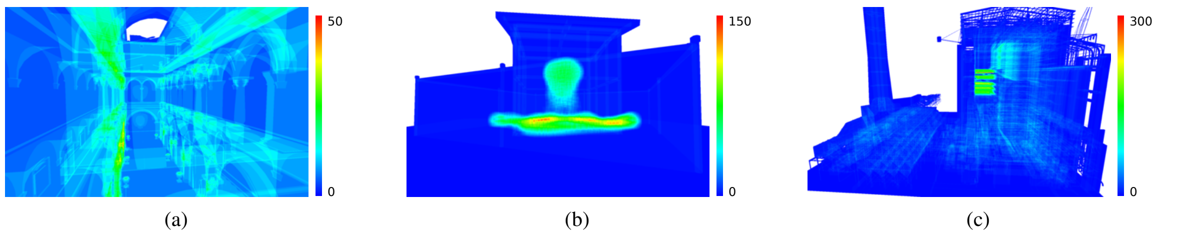


Figure 4: Depth complexity of the three scenes in Figure 1.

range. It can be seen that increasing OIT local memory to correctly support up to 128 fragments for the peak depth complexity significantly reduces performance in all views — by a factor of around 1.4×.

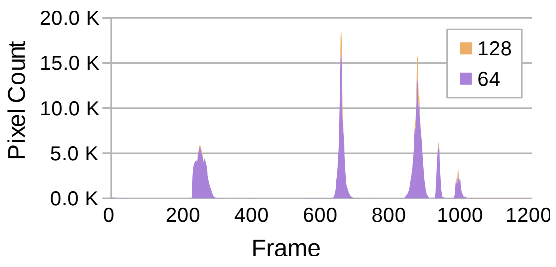


Figure 8: Separated depth complexity during the flythrough in Figure 7, showing only ranges 33–64 and 65–128.

The sorting algorithm chosen has a significant affect on OIT performance. Figure 9 shows the performance of different sorting algorithms relative to insertion sort for sorting and compositing different BMA intervals in the Power Plant scene. Based on these and similar results we use insertion sort for depth complexities of 16 and less, and merge sort for higher depth complexities. In some cases shell sort may give a small performance benefit, for example the 33-64 interval. BMA-OIT can compile the program for each interval that only uses the best sorting algorithm without overhead. In contrast standard OIT must branch at runtime, potentially affecting thread coherency, however this still provides a significant benefit. For standard OIT we found using insertion sort for intervals up to 64 and merge sort for higher intervals to be most effective. Note that the threshold at which different sorting algorithms become beneficial is different when selecting dynamically. The above optimizations increase standard OIT performance by 1.5× and 1.3× for the Smoke and Power Plant scenes respectively while BMA-OIT performs 2.3× and 2.1× better. Previous results include these sorting improvements.

5. Conclusion

Large and unused local memory allocation can significantly impact shader performance. We have introduced BMA to reduce this effect, demonstrated its use by applying it to OIT and achieved up to a 3× speedup. Moreover, BMA-OIT is more flexible allowing fast rendering times for low depth complexity scenes while supporting correct rendering of high depth complexity scenes. In contrast, without BMA the OIT result is either rendered incorrectly for high depth complexity scenes or performance suffers with low depth complexity scenes.

We have shown choosing the sorting algorithm for OIT based on the fragment count at a per-pixel level improves performance significantly and works especially well with BMA-OIT, as the optimization is applied for each BMA interval shader and not dynamically.

This work is part of many recent compound improvements bringing exact transparency rendering of complex scenes, such as the power plant, to within interactive speeds.

While we have demonstrated BMA with OIT, it may be of benefit to other applications where occupancy is also limited by varying and unavoidably large local memory usage.

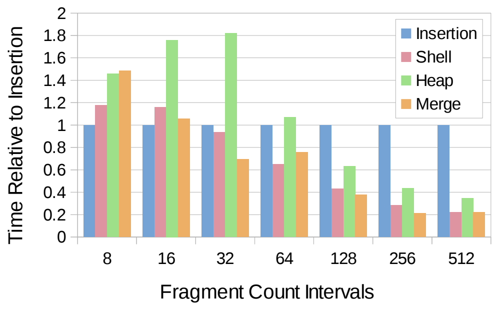


Figure 9: Sorting algorithm time relative to insertion sort at different fragment count intervals for BMA-OIT with the Power Plant scene.

References

[Eve01] Everitt C.: *Interactive Order-Independent Transparency*. Tech. rep., NVIDIA Corporation, 2001. 2

[KLZ12] Knowles P., Leach G., Zambetta F.: Efficient layered fragment buffer techniques. In *OpenGL Insights*, Cozzi P.,

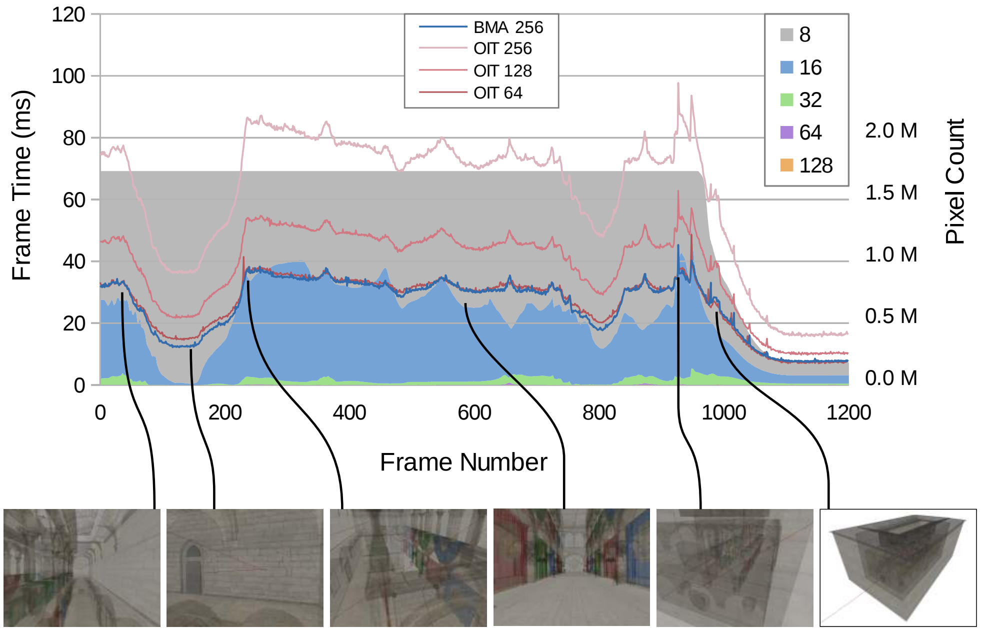


Figure 7: A fly-through of the Sponza Atrium, showing each frames rendering time for standard OIT and BMA-OIT. The background area shows the number of pixels with depth complexity in each range. Note range 8 does not include zero depth complexity.

Riccio C., (Eds.). CRC Press, July 2012, pp. 279–292. http://www.openglinsights.com/. 1, 2

[LHLW09] Liu F., Huang M.-C., Liu X.-H., Wu E.-H.: Single pass depth peeling via cuda rasterizer. In *SIGGRAPH 2009: Talks* (New York, NY, USA, 2009), SIGGRAPH '09, ACM, pp. 79:1–79:1. URL: http://doi.acm.org/10.1145/1597990.1598069, doi:http://doi.acm.org/10.1145/1597990.1598069. 2

[Pee08] Peeper C.: Prefix sum pass to linearize a-buffer storage, patent application, microsoft corp., December 2008. 2

[YHGT10] Yang J. C., Hensley J., Grün H., Thibieroz N.: Real-time concurrent linked list construction on the gpu. *Comput. Graph. Forum 29*, 4 (2010), 1297–1304. URL: http://dblp.uni-trier.de/db/journals/cgf/cgf29.html#YangHGT10. 2