Computational Visual Media
https://doi.org/10.1007/s41095-018-0118-8 Vol. 4, No. 3, September 2018, 277–285
Research Article

GPU based techniques for deep image merging

Jesse Archer[1] (✉), **Geoff Leach**[1], **and Ron van Schyndel**[1]

Abstract Deep images store multiple fragments per-pixel, each of which includes colour and depth, unlike traditional 2D flat images which store only a single colour value and possibly a depth value. Recently, deep images have found use in an increasing number of applications, including ones using transparency and compositing. A step in compositing deep images requires merging per-pixel fragment lists in depth order; little work has so far been presented on fast approaches.

This paper explores GPU based merging of deep images using different memory layouts for fragment lists: linked lists, linearised arrays, and interleaved arrays. We also report performance improvements using techniques which leverage GPU memory hierarchy by processing blocks of fragment data using fast registers, following similar techniques used to improve performance of transparency rendering. We report results for compositing from two deep images or saving the resulting deep image before compositing, as well as for an iterated pairwise merge of multiple deep images. Our results show a 2 to 6 fold improvement by combining efficient memory layout with fast register based merging.

Keywords deep image; composite; GPU; performance

1 Introduction

This paper explores time and memory performance of storing and merging deep images on the GPU using OpenGL and GLSL. We assume the deep images are stored in graphics memory, leaving broader investigation of approaches which include reading deep images from persistent storage for future work.

A topic of increasing interest [1, 2], deep image compositing presents new opportunities and challenges compared to standard image compositing. Among these challenges is performance, as compositing many fragments per-pixel per-image requires more processing than just a single fragment per-pixel. GPUs are naturally suited for this task.

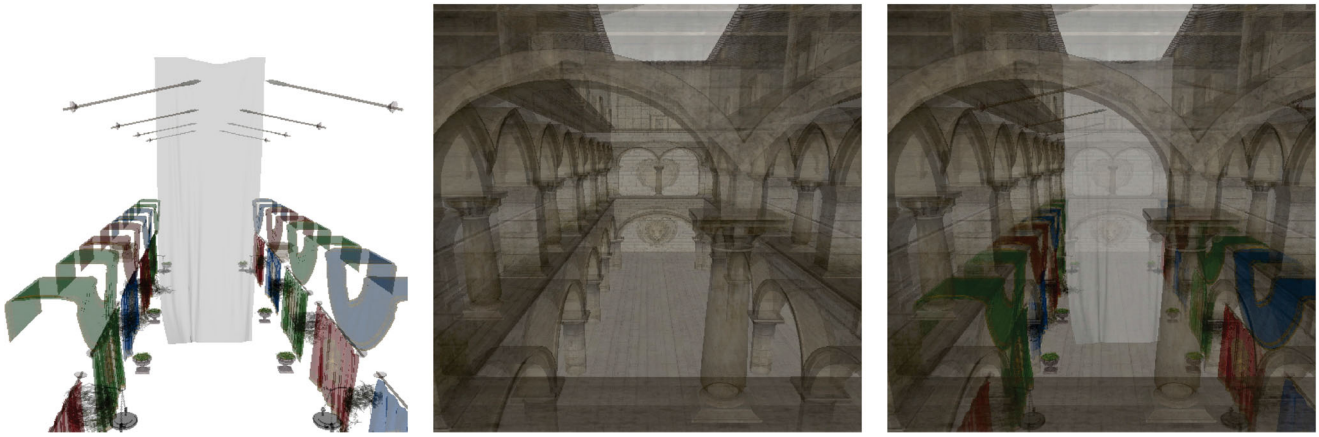
Merging two deep images (see Fig. 1) requires first loading them to GPU global memory, either entirely if possible, or in large blocks. Per-pixel threads then read data from both deep images, merging and compositing fragments to produce a final 2D (flat) image, or alternatively merging and saving the resulting deep image before compositing. The resulting deep image can then be used in other deep image operations, such as further merging in an iterated pairwise or k-way fashion. This paper specifically focuses on merging two deep images, either to give a merged result or for iterated pairwise merging, leaving the problem of k-way deep image merging for future work.

A simple merging approach is to step through fragment data in sorted order for both deep images, comparing fragments from each based on depth, and compositing before moving to the next fragment, using a basic linear time per-pixel stepwise merge of two sorted lists. Our approach improves on this by reading and processing blocks of data using registers.

Deep images are typically stored in graphics memory using one of two main formats for GPU processing: as per-pixel linked lists, or as linearised arrays of fragments. We explore differences between these approaches in terms of memory usage and processing time; linked lists require more memory while linearised arrays require more processing during construction. We also explore an interleaved array format which improves performance of deep image merging through better memory read coherence. We investigate performance of merging deep images in

1 School of Science, RMIT University, Melbourne, 3000, Australia. E-mail: J. Archer, jesse.archer@rmit.edu.au (✉); G. Leach, geoff.leach@rmit.edu.au; R. van Schyndel, ron.vanschyndel@rmit.edu.au.

Manuscript received: 2017-12-23; accepted: 2018-05-02



(a) Furnishings (b) Structure (c) Combined

Fig. 1 Merged interior and exterior *Atrium* deep images.

graphics memory using a stepwise approach, and an improved approach using blocks of registers. Finally we introduce a blocked interleaved array format which leverages blocked merging to give a combined 2 to 6 fold performance improvement.

2 Related work

Storing deep images in GPU memory as linked lists and linearised arrays has been explored in the context of transparency rendering in computer graphics [3, 4]. Linked lists have been found to generally provide better performance for processing data, while linearised arrays use less memory. Using fast GPU registers for sorting deep image data was presented in Ref. [5], a concept that this work extends.

General image compositing operations were first proposed in Ref. [6], and recently expanded to deep images [2, 7], using the OpenEXR format [8] for external storage. Such work focuses on how composite operations are performed. Performance of compositing deep images in memory on the GPU using different merging approaches and storage formats has to our knowledge not been presented, and is the focus of this work.

3 Background

3.1 Deep image formats

Arranging fragment data into appropriate buffers in memory is critical for fast processing on the GPU. As mentioned earlier, two main approaches exist: per-pixel linked lists and linearised arrays.

Building a deep image as per-pixel linked lists requires a global atomic counter and the allocation of three storage buffers, with one integer per-pixel for the head pointers and then buffers of arbitrary size for fragment data and next pointers. The head pointers are initialised to null (0) before rendering. If the size is too small to store all fragment data, then the atomic counter is used to allocate buffers of sufficient size before re-rendering.

As geometry is rasterized, fragments are added to the data array using a global atomic counter, and appended to the corresponding pixel's list using an atomic exchange which inserts the fragment at the front of the list. The fragment's next pointer is then set to the previous head node. In this fashion fragments are continuously added to the head of the corresponding pixel's linked list. Example GLSL code for adding fragment data to a linked list, and traversing it, is shown below:

```
// Building a linked list
uint fragIdx = atomicCounterIncrement(count);
if (fragIdx < size)
{
 uint headIdx = atomicExchange(headPtrs[pixel],
     fragIdx);
 nextPtrs[fragIdx] = headIdx;
 data[fragIdx] = frag;
}

// Traversing a linked list
uint node = headPtrs[pixel];
while (node != 0)
 node = nextPtrs[node];
```

Building a deep image can be done quickly, as fragments can be written to the next available place as they are rendered or captured. Traversing a pixel's fragment list starts at the index given by the pixel's

head pointer, and follows each fragment's next pointer respectively until a null terminator is reached.

Merging two deep images and saving the resulting merged deep image in this format reverses the order of the per-pixel fragment lists, as fragment data is, always added to the head of the list. This must be accounted for in the next merge or composite step.

Figure 2 shows per-pixel fragment colours stored as linked lists using three separate buffers with blue/red/green, blue/green, and blue/red fragment colours for the bottom left, bottom right, and top left pixels respectively. Fragments for a given pixel can be anywhere in the data buffer.

Linearised arrays require only two buffers: see Fig. 3, which shows the same per-pixel fragment colours as Fig. 2. Unlike the linked-list approach in which fragment data can be anywhere, in linearised arrays, fragment data for a given pixel is coherent: it is localised with all fragments for a given pixel stored contiguously.

Building a deep image in this format may be summarized as follows:

- Allocate buffer of per-pixel counts, initialised to zero.
- Render geometry depths and atomically increment counts in the fragment shader in an initial rendering pass.

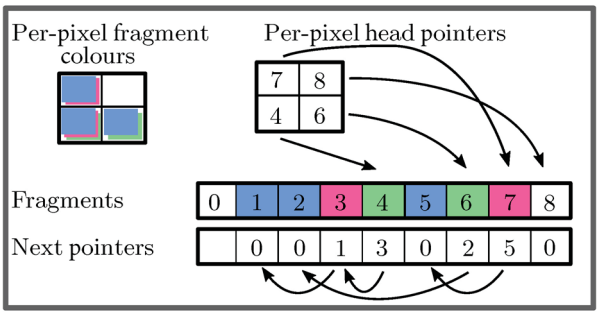


Fig. 2 Per-pixel blue/red/green, blue/green, and blue/red fragment colours as linked lists.

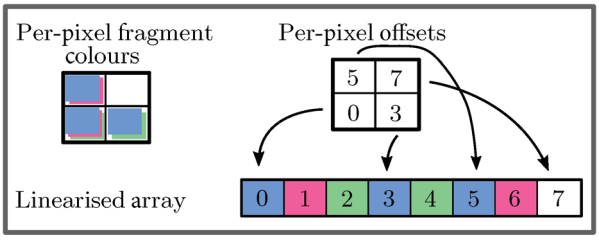


Fig. 3 Per-pixel blue/red/green, blue/green, and blue/red fragment colours as linearised arrays.

- Perform parallel prefix sums scan on counts to produce an array of offsets. These determine the location of each pixel's memory in the global data array.
- Allocate data buffer of size given by final offset.
- Render full geometry data in a second rendering pass; offsets are atomically incremented in the fragment shader to give the location at which each fragment is written in the data buffer.

Traversing a pixel's fragment data requires reading the index offset and number of fragments, given by subtraction from the next pixel's offset, then reading the fragment data sequentially. Example GLSL code is given below for adding fragment data to a linearised array, and traversing it; note that the same buffer is used for both counts and offsets:

```
// Counting the number of fragments per-pixel
atomicAdd(offsets[pixel], 1);

// Building the linearised array.
uint idx = atomicAdd(offsets[pixel], 1);
data[idx] = frag;

// Traversing the array
uint start = pixel > 0 ? offsets[pixel-1] : 0;
uint end = offsets[pixel];
for (uint node = start; node < end; node++)
    ...
```

Building a deep image in this format is typically slower, as it requires computing offsets from per-pixel fragment counts in a separate initial counting pass before writing fragment data in a second capturing pass. However, it requires less memory as there are no next pointers.

Deep image compositing requires fragment lists in depth sorted order. As mentioned in Section 2, the currently fastest technique for deep image sorting is register-based block sort [5] which uses a sorting network of fast registers. In cases where lists are longer than the number of available per-thread registers, backwards memory allocation [9] partitions the sort into blocks. This combined approach is used for sorting deep images in this paper.

3.2 Memory hierarchy

Deep images can be large; on the GPU, pixels are processed per-thread in parallel. GPUs have a hierarchy of memory as shown in Fig. 4, with a large amount of relatively slow global memory, and a smaller amount of fast memory such as local memory, and then an even smaller number of very fast

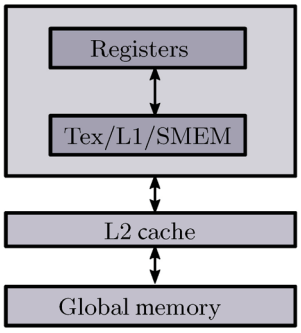


Fig. 4 Example memory hierarchy of an nVidia GPU.

registers. On the nVidia Pascal architecture, local memory (CUDA shared L1 memory) and registers are available per-streaming multiprocessor (SM) while global memory (CUDA local memory) and L2 cache are available to all threads.

As stated previously, compositing deep images requires first loading or capturing them to slow global memory. Global memory has high latency, particularly as fragment reads are not necessarily coherent. A stepping approach that reads then composites before reading the next fragment in turn is highly vulnerable to this latency.

Processing data by reading blocks from slow to fast memory is an established concept, and applies to merging. One approach is to merge blocks of data by reading fragments from global memory to local memory before compositing, reducing the impact of latency. Using blocks of local memory requires copying data from global to local memory, then reading from local memory to perform the comparison and composition operations in registers.

Registers are much faster than global and local memory. GPUs typically have on the order of thousands of registers, typically 255 per-thread or core, so fragments can be read to per-thread blocks of registers directly rather than to local memory first. This has the benefit of both reducing the impact of latency, and avoiding writing to and then reading from local memory.

4 Deep image merging

4.1 Register block merging

The merging operation is performed by reading blocks of data directly from global memory to fast registers, bypassing local memory. We term this approach *register block merging* (RBM). It is summarised in the following steps, which performs a stepwise merge, reading to blocks of registers:

- Begin with two per-pixel sorted fragment lists and two register blocks, one for each deep image.
- If either register block is empty, read values from the corresponding deep image.
- Merge values in both blocks in depth order until one block is exhausted.
- Merged data is either written to an output deep image, or composited to a flat (2D) image.
- After exhausting one fragment list, merge the remaining block and fragment values from the other list.

Local variables or arrays with fixed indices known at compile time must be used in order to ensure that the GLSL compiler will store fragments in fast GPU registers. Code examples for reading fragment data from a deep image are shown below, along with the intermediate shader assembly output produced by the nVidia compiler, based on a similar example given in Ref. [5]:

```
// Local memory
#define SIZE 4
Fragment data[SIZE];
uniform int count;

// Loop limit not known at compile time
for (int i = 0; i < count; i++)
    data[i] = readNext(...);

produces:
...
TEMP lmem0[4];
TEMP RC, HC;
...
MOV.S R0.x, {0, 0, 0, 0};
REP.S ;
SGE.S.CC HC.x, R0, c[0].x;
BRK (NE.x);
MOV.U R0.y, R0.x;
MOV.S lmem0[R0.y].x, {0, 0, 0, 0};
ADD.S R0.x, R0, {1, 0, 0, 0};
ENDREP;
...

// Registers
#define SIZE 4
Fragment data[SIZE];
uniform int count;

// Loop limit known at compile time
for (int i = 0; i < count && i < SIZE; i++)
    data[i] = readNext(...);

produces:
...
TEMP R0, R1, R2, R3;
TEMP RC, HC;
...
SLT.S R0.y, {0, 0, 0, 0}.x, c[0].x;
MOV.U.CC RC.x, -R0.y;
IF NE.x;
```

```
MUL.S R0.y, 0, c[0].x;
MUL.S R0.y, R0, {4, 0, 0, 0}.x;
MOV.U R0.y, R0;
SLT.S R0.z, {1, 0, 0, 0}.x, c[0].x;
MOV.U.CC RC.x, -R0.z;
LDB.S32 R3.x, sbo_buf0[R0.y];
...
ENDIF;
...
```

The first program iterates a number of times determined at runtime and therefore the loop cannot be unrolled at compile time. As register usage is decided at compile time, registers cannot be used in this case and local memory is used instead, seen by the `lmem0[8]` local memory allocation. The use of registers requires either manual loop unrolling or use of a bounding compile-time constant, as shown by `R0`, `R1`, `R2`, `R3` in the second example. The same unrolling technique is used when reading and merging. A block of registers can also be used when writing the merged deep image, although we found this to be faster only when using linearised arrays.

As GPUs keep all active threads resident, the number of threads that can be scheduled and executed simultaneously is limited by available per-thread resources such as local memory and registers. However, instead of threads causing waiting when reading from global memory, other threads are executed to reduce the impact of memory latency and increase throughput. This means GPUs typically have many more active threads than available cores. Storing fragments in per-thread registers reduces the number of possible simultaneous threads. To achieve greater throughput this needs to be balanced by keeping block sizes relatively small, typically using 4 to 16 fragments.

4.2 Interleaved arrays

Instead of using linked lists and linearised arrays to store deep images, a faster technique is to use interleaved arrays. GPUs execute threads in groups, where instructions across the group are executed in lock-step. This means the first fragment for each pixel in a thread group is processed before the second fragment. Improved memory performance requires coherent memory reads for fragment data in a thread group, rather than for each individual pixel. Arranging fragment data in order of per-group reads instead of per-pixel reads improves memory coherence.

One approach is to interleave fragment data across groups for all pixels, based on the group's maximum fragment count. This requires padding each group so that all lists have the same length, consequently increasing memory requirements; this was by a factor of 2–3 for our test scenes. We instead interleave up to the shortest fragment list for any pixel in a group, with remaining fragments stored at the end of each group with no padding as shown in Fig. 5.

Building a deep image in this format is done in a similar way to the approach used for a linearised array, and requires an extra buffer for per-group minimum counts, and a buffer of per-pixel counts in addition to the offsets:

- Minimum counts are allocated as number of pixels divided by 32, initialised to zero.
- Per-pixel counts are determined in the same manner.
- Compute threads are executed for each group of 32 pixels, each thread determining and writing the minimum count of its respective group to the minimum counts buffer.
- The prefix sums scan then computes per-pixel offsets from per-pixel counts as for the linearised arrays case, and allocates a data buffer of sufficient size.
- Complete geometry is rendered in a second pass and saved as for linearised arrays, but with modified indexing.

Example GLSL code for adding fragment data to and traversing an interleaved array is given below:

```
// Adding data to an interleaved array
#define GROUP_SIZE 32
uint run = pixel % GROUP_SIZE;
uint group = pixel / GROUP_SIZE;
uint groupOffset = offsets[group * GROUP_SIZE];
uint minCount = minCounts[group];
uint currCount = atomicAdd(counts[pixel], 1);
if (currCount < minCount)
    data[groupOffset + run + GROUP_SIZE *
    currCount] = frag;
else
{
    uint adjst = offsets[pixel] - minCount *
    run;
    data[minCount * GROUP_SIZE + adjst + (
    currCount - minCount)] = frag;
}

// Traversing an interleaved array.
uint interOffset = (pixel % GROUP_SIZE) +
    offsets[(pixel / GROUP_SIZE) * GROUP_SIZE];
uint extraOffset = offsets[pixel] - minCounts[
    pixel / GROUP_SIZE] * (pixel % GROUP_SIZE);
uint minCount = minCounts[pixel / GROUP_SIZE];
uint end = counts[pixel];
for (uint node = 0; node < end; node++)
    if (node < minCount)
        uint idx = interOffset + GROUP_SIZE *
    node;
    else
        uint idx = minCount * GROUP_SIZE +
    extraOffset + node - minCount;
```

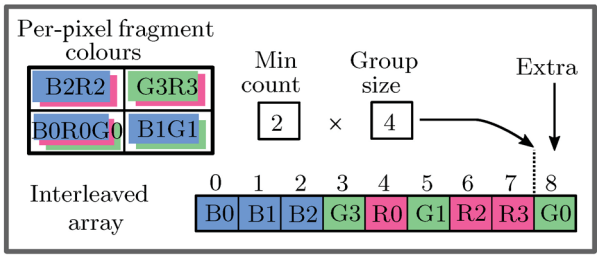


Fig. 5 Per-pixel blue/red/green, blue/green, blue/red, and green/red fragment colours as an interleaved array.

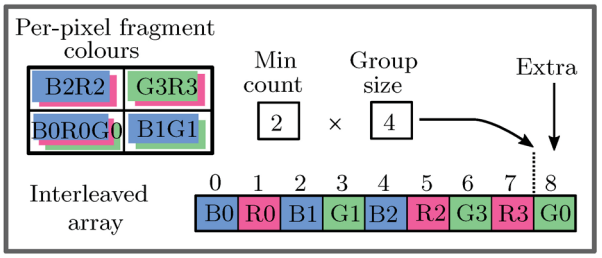


Fig. 6 Per-pixel blue/red/green, blue/green, blue/red, and green/red fragment colours as a blocked interleaved array with block size 2.

4.3 Blocked interleaving

When using register block merging, coherence is further improved by interleaving fragments in blocks rather than individually when generating deep image data: see Fig. 6. This means the first block of fragments for the first pixel is written to the deep image, then the first block for the second pixel in turn. The same block size is used when building and merging the deep images. With blocked interleaving, the per-group minimum counts must be a multiple of the block size, which can result in more non-interleaved fragments stored at the end.

Building a deep image in this format follows the same approach as for an interleaved array, with modified indexing in the fragment shader, as shown below:

```
// Adding data to a blocked interleaved array.
...
#define BLOCK_SIZE 4
if (currCount < minCount)
    data[groupOffset + (currCount / BLOCK_SIZE)
     * GROUP_SIZE * BLOCK_SIZE + run *
    BLOCK_SIZE + currCount % BLOCK_SIZE] = frag
    ;
else
{
    uint adjst = offsets[pixel] - minCount *
    run;
    data[minCount * GROUP_SIZE + adjst + (
    currCount - minCount)] = frag;
}

// Traversing a blocked interleaved array.
...
for (uint node = 0; node < end; node++)
    if (node < minCount)
        idx = groupOffset + (node / BLOCK_SIZE)
     * GROUP_SIZE * BLOCK_SIZE + run *
    BLOCK_SIZE + node % BLOCK_SIZE;
    else
        idx = minCount * GROUP_SIZE +
    extraOffset + node - minCount;
```

If the final result is written back to global memory, compute threads can be executed in order of memory layout. However, if the resulting image is rasterized, then threads are instead executed in pixel rasterization order. nVidia GPUs typically rasterize pixels in a tile-based fashion, where a 2×8 tile is rasterized in a zig-zag pattern. Figure 7 shows the rasterization order for a 4×8 block of pixels; numbers represent execution order of per-pixel fragment shader threads as determined by atomic counters. Indexing pixels to more closely match the repeating 2 × 8 tiled raster pattern when building the deep image improves merging performance by approximately 1.5 to 2 fold for all approaches.

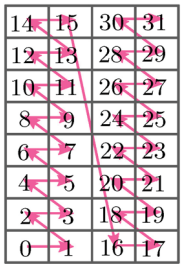
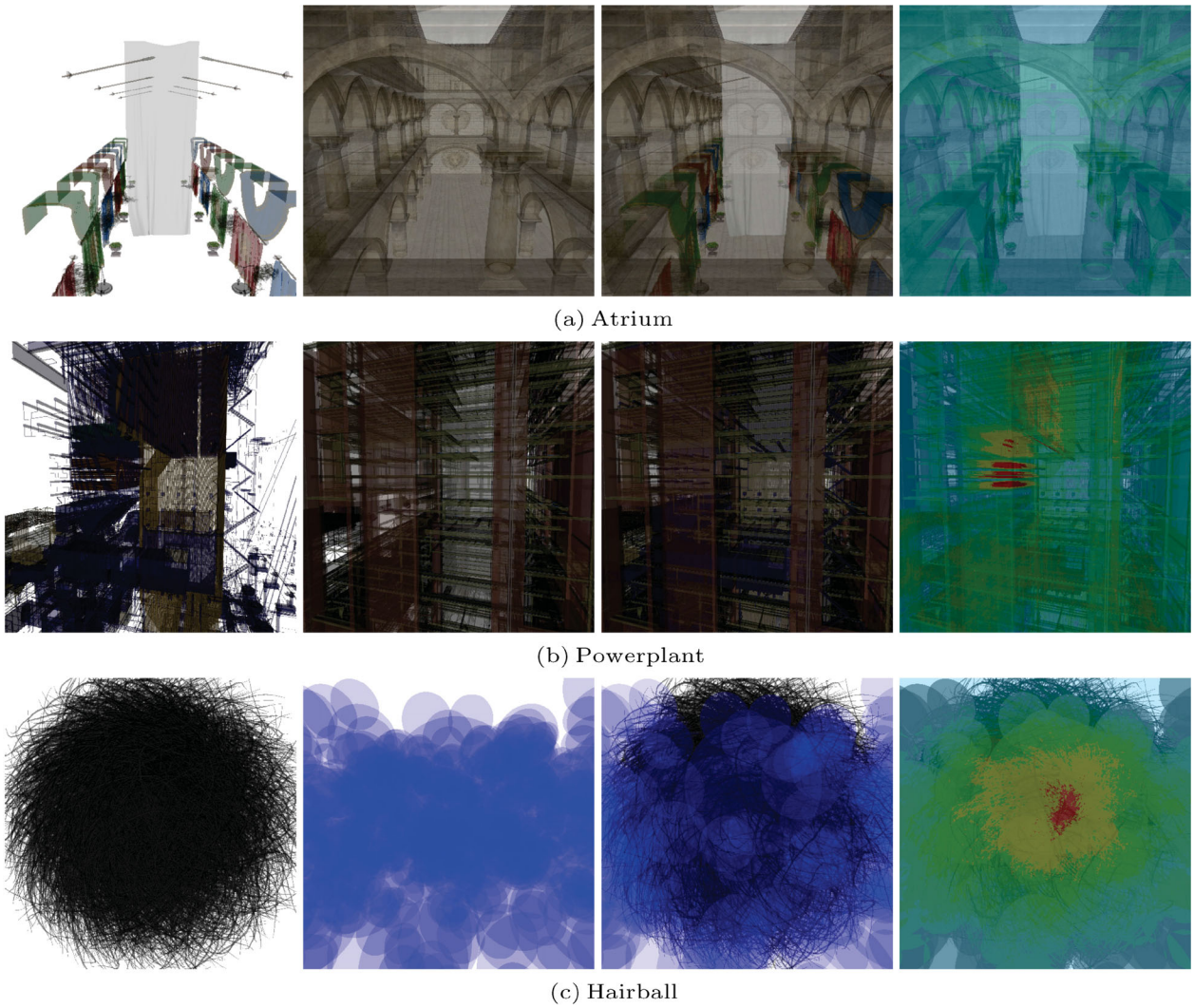


Fig. 7 Typical thread execution order (raster pattern) for a 4×8 block of pixels on an nVidia GPU.

5 Results

We compare performance of merging two deep images using three different scenes: see Fig. 8. Additionally we compare an iterated pairwise merge, where four deep images are first merged to give the two deep images shown, before merging the two resulting deep images.

The first and second scenes are the *Sponza Atrium* and the *Powerplant*, each separated into interior and exterior deep images. The third referred to as the *Hairball*, is a synthetic scene of a hairball merged with a set of randomly generated spheres. These scenes are available from Ref. [10]. Not shown is another synthetic scene referred to as the *Planes*, which has 256 screen-aligned quads with linked list data in



(a) Atrium

(b) Powerplant

(c) Hairball

Fig. 8 Test scenes with separated and merged deep images. Heatmap gives depth complexity with blue up to 16 fragments, green 16–64, yellow 64–128, and red 128–512.

approximately coherent order merged with a set of randomly generated spheres. For all measurements, deep image data is arranged in raster pattern, which is 1.5–2 times faster for all test cases.

The *Atrium* scene typically has fewer than 20 per-pixel fragments in each deep image, while the other scenes have up to hundreds. The *Atrium* and *Powerplant* are divided mainly into interior and exterior geometry. Thus, as merging progresses, data is mainly read from one deep image then the other in turn. The *Hairball* and *Planes* have spheres randomly distributed, with memory being read more evenly across both deep images as a consequence.

The storage approaches discussed in Sections 3 and 4 were compared: linked lists (LLs), linearised arrays (LAs), interleaved linearised arrays (IAs), and blocked interleaved linearised arrays (BIAs). Merging techniques were stepwise (S) and register block merging (RBM). The test platform was an nVidia GeForce GTX 1060, driver version 390.25. The deep images were HD (1920×1080) resolution. For each technique we report memory usage for the deep images and total merging time in milliseconds. We do not report the memory cost of RBM or stepwise merging, as these techniques do not require extra global memory.

Results for compositing when merging two deep images are shown in Table 1, while those for merging and saving the resulting merged deep image are shown in Table 2. Iterated pairwise merging results are shown in Table 3 for the *Atrium* and *Powerplant* scenes, with geometry divided into two interior and two exterior deep images.

Results are average time from rendering and

capturing scenes as separate deep images on the GPU and then merging; merge time reported includes either compositing a flat (2D) image or saving the resulting deep image.

The results in Tables 1–3 show RBM offers up to 4-fold performance improvement in the best case and no performance penalty in the worst case, regardless of whether compositing during merging, saving the merged image or using pairwise merging. This is due to memory latency for incoherent reads being reduced by reading memory in blocks. The largest performance improvement by RBM is for linearised arrays and blocked interleaved arrays, as block memory reads are typically more coherent in these formats. RBM offers a smaller performance improvement for mostly coherent data, or when there is little data to merge, as in the *Atrium* scene.

Blocked interleaved arrays are faster than interleaved arrays when compositing, regardless of whether RBM or stepwise merging is used, being up to 1.3 times faster in the case of the *Planes*. When saving the merged deep image or using pairwise merging, blocked interleaved arrays are only faster when combined with RBM.

RBM is more effective with blocked interleaved arrays, as memory is specifically arranged to improve this approach. Compared to the worst case approach of each scene, BIA-RBM gives a 2 to 6 fold performance improvement. This improvement is less significant in the *Atrium* scene where less geometry is present and thus fewer merging operations performed.

When saving the merged deep image or using an iterated pairwise approach, linked lists are typically faster for the *Atrium* scene. Saving a deep image using linearised arrays, interleaved arrays, or blocked interleaved arrays requires first building an array of offsets before writing any fragment data, unlike linked lists for which next pointers and fragments are written simultaneously. The cost of first building the offsets is outweighed by any potential merging improvements when less geometry is present.

Linearised arrays use less space than other formats as shown in Table 4, while interleaved arrays and blocked interleaved arrays require a little more due to the per-group minimum fragment counts, which depend on image resolution. Linked lists use the most memory in all cases, as expected, due to the next pointers. In all cases RBM and stepwise merging require no extra global memory.

Table 1 Merging time for two input deep images, compositing during merging

Approach	Atrium	Powerplant	Hairball	Planes
LL-S	1.8 ms	16.8 ms	36.2 ms	23 ms
LL-RBM	1.8 ms	14.7 ms	30.5 ms	16.5 ms
LA-S	1.9 ms	6.8 ms	12.5 ms	46.3 ms
LA-RBM	1.7 ms	5.2 ms	7.3 ms	24.6 ms
IA-S	**1.4 ms**	6.4 ms	9.1 ms	14.4 ms
IA-RBM	1.5 ms	6.1 ms	8.4 ms	14.5 ms
BIA-S	1.5 ms	5.4 ms	8.0 ms	14.5 ms
BIA-RBM	1.5 ms	**4.6 ms**	**5.9 ms**	**10.7 ms**

Table 2 Merging time for two input deep images, saving result as a deep image

Approach	Atrium	Powerplant	Hairball	Planes
LL-S	**4.7 ms**	25.3 ms	47.9 ms	36.2 ms
LL-RBM	**4.7 ms**	19.8 ms	34.7 ms	**30.8 ms**
LA-S	14.1 ms	43.2 ms	60.0 ms	198.6 ms
LA-RBM	6.2 ms	14.6 ms	19.0 ms	54.0 ms
IA-S	5.5 ms	14.9 ms	19.2 ms	42.6 ms
IA-RBM	5.6 ms	13.5 ms	17.6 ms	38.6 ms
BIA-S	7.3 ms	21.6 ms	28.8 ms	70.5 ms
BIA-RBM	6.0 ms	**12.6 ms**	**15.8 ms**	33.2 ms

Table 3 Merging time for four input deep images using iterated pairwise merging

Approach	Atrium	Powerplant
LL-S	**9.5 ms**	39.2 ms
LL-RBM	9.6 ms	36.3 ms
LA-S	28.3 ms	76.2 ms
LA-RBM	14.0 ms	31.1 ms
IA-S	13.4 ms	29.8 ms
IA-RBM	13.6 ms	29.0 ms
BIA-S	16.9 ms	40.7 ms
BIA-RBM	14.3 ms	**28.5 ms**

Table 4 Data usage for different deep image formats

Format	Atrium	Powerplant	Hairball	Planes
LL	282 MB	5578 MB	682 MB	1834 MB
LA	**157 MB**	**309 MB**	**378 MB**	**1018 MB**
IA	173 MB	325 MB	394 MB	1034 MB
BIA	173 MB	325 MB	394 MB	1034 MB

6 Conclusions

This paper has presented RBM and shown it to be a better merging approach, and has shown blocked interleaved arrays to be a better deep image format. It has also explored and compared stepwise merging and other existing deep image formats. Interleaved deep images have little memory overhead and fast merging time due to improved memory coherence, while register block merging improves performance of

merging fragment data. Combined, these approaches give up to 2 to 6 fold performance improvement compared to non-interleaved stepwise merging.

The interleaved arrays and blocked interleaved arrays approaches interleave fragment data based on per-group minimum fragment counts, with all remaining fragments stored in a non-interleaved linear fashion. Interleaving remaining fragment data past the minimum fragment list length without padding may offer further performance improvement. As iterated pairwise merging requires multiple writes to global GPU memory, an alternative is to use k-way merging which we suspect may offer improved performance as it only writes to global memory once per-fragment.

Acknowledgements

The authors would like to thank Pyar Knowles for his original deep image software on which this work is based. It is available at `https://github.com/pknowles/lfb`.

References

[1] Heckenberg, D.; Saam, J.; Doncaster, C.; Cooper, C. Deep compositing. 2010. Available at `http://www.johannessaam.com/deepImage.pdf`.

[2] Duff, T. Deep compositing using lie algebras. *ACM Transactions on Graphics* Vol. 36, No. 3, Article No. 26, 2017.

[3] Maule, M.; Comba, J. L. D.; Torchelsen, R.; Bastos, R. Memory-efficient order-independent transparency with dynamic fragment buffer. In: Proceedings of the 25th SIBGRAPI Conference on Graphics, Patterns and Images, 134–141, 2012.

[4] Knowles, P.; Leach, G.; Zambetta, F. Efficient layered fragment buffer techniques. In: *OpenGL Insights*. Cozzi, P.; Riccio, C. Eds. CRC Press, 279–292, 2012.

[5] Knowles, P.; Leach, G.; Zambetta, F. Fast sorting for exact OIT of complex scenes. *The Visual Computer* Vol. 30, Nos. 6–8, 603–613, 2014.

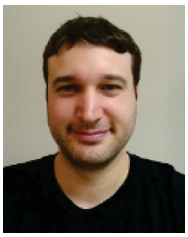
[6] Porter, T.; Duff, T. Compositing digital images. In: Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, 253–259, 1984.

[7] Egstad, J.; Davis, M.; Lacewell, D. Improved deep image compositing using subpixel masks. In: Proceedings of the 2015 Symposium on Digital Production, 21–27, 2015.

[8] Hillman, P. The theory of OpenEXR deep samples. Technical Report. Weta Digital Ltd., 2013.

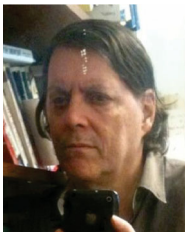
[9] Knowles, P.; Leach, G.; Zambetta, F. Backwards memory allocation and improved OIT. In: Proceedings of the Pacific Graphics, 59–64, 2013.

[10] McGuire, M. Computer graphics archive. 2017. Available at `https://casual-effects.com/data`.



Jesse Archer is a Ph.D. student at RMIT University, Melbourne. His research interests are in realtime computer graphics and GPU computing. He completed his bachelor of computer science in 2008, bachelor of IT (games and graphics programming) in 2010, and honours in computer science in 2015 at RMIT University.

Geoff Leach is a lecturer in the School of Science at RMIT University. His major research interests include computer graphics, computational science, and GPU computing. He mostly teaches computer graphics, and has been using OpenGL since version 1.1. He holds a M.App.Sci. degree from RMIT University.



Ron van Schyndel is a senior lecturer from School of Science (formerly School of Computer Science and IT) at RMIT University. He is and has been an active researcher in the domain of digital watermarking for more than 2 decades, and is co-author to some of the most cited papers in the field. He obtained his Ph.D. degree from Monash University on the then nascent topic of digital watermarking, and has obtained many industry grants on watermarking and other applications. His other research interests beyond digital watermarking include signal, image, and vision processing, as well as software infrastructure specifically as applied to mobile navigation for the blind and visually impaired.