Title

Fragment-Parallel Composite and Filter

Journal

Computer Graphics Forum, 29(4)

ISSN

0167-7055

Authors

Patney, Anjul

Tzeng, Stanley

Owens, John D

Publication Date

2010-06-01

DOI

10.1111/j.1467-8659.2010.01720.x

Eurographics Symposium on Rendering (2010)
Jason Lawrence and Marc Stamminger (Guest Editors)

Fragment-Parallel Composite and Filter

Anjul Patney, Stanley Tzeng and John D. Owens†

University of California, Davis

Abstract

We present a strategy for parallelizing the composite and filter operations suitable for an order-independent rendering pipeline implemented on a modern graphics processor. Conventionally, this task is parallelized across pixels/subpixels, but serialized along individual depth layers. However, our technique extends the domain of parallelization to individual fragments (samples), avoiding a serial dependence on the number of depth layers, which can be a constraint for scenes with high depth complexity. As a result, our technique scales with the number of fragments and can sustain a consistent and predictable throughput in scenes with both low and high depth complexity, including those with a high variability of depth complexity within a single frame. We demonstrate composite/filter performance of up to 60M fragments/sec for scenes with more than 1500 semi-transparent layers.

1. Introduction

Transparent and semi-transparent objects form an important class of primitives for high-quality rendering. They find applications in both interactive and off-line graphics, where they are used for rendering complex effects like particle systems, hair, foliage, and semi-transparent sprites and billboards. They are also significant for architectural and scientific visualization, where transparency can provide important cues on the structure and behavior of the visualized object.

The traditional pipeline for interactive graphics, however, is not well-suited towards such rendering tasks. It uses a depth buffer, typically implemented as fixed-function hardware, which is extremely efficient for rendering opaque primitives. However, rendering semi-transparent primitives using a depth buffer imposes a burden on the application programmer: objects must be dispatched in back-to-front order. Besides the extra complexity for the programmer, this method is not wholly robust: when objects overlap in depth, the resulting artifacts are tricky to avoid. As a result, effects like transparency are particularly difficult to render correctly and efficiently.

Thus researchers have recently concentrated on implementing methods for compositing in the graphics pipeline that can automatically, efficiently, and correctly composite fragments (samples) in an arbitrary order, leveraging parallelism over pixels in their implementations. While this approach delivers good performance for simple scenes with low depth complexity, complicated scenes with a larger depth complexity or more variance in the number of fragments per pixel lead to unacceptable performance. We propose to use the programmable features of modern graphics processors (GPUs) to instead parallelize over fragments and deliver good performance irrespective of the fragment distribution in the scene.

Our approach in using the programmable GPU for a traditionally fixed-function task is part of a larger trend of *programmable pipelines* [Pha06] as flexible and efficient interfaces to interactive rendering. This transition towards programmable rendering is further supported by a concurrent exploration in both industry and academia. Recent and upcoming GPUs provide native support for highly programmable rendering [SCS*08, LNOM08], which are exposed to graphics applications through compute shaders in the Direct3D 11 API [Mic08]. Researchers in academia have also demonstrated the ability to build rendering pipelines using the increased programmability of modern GPUs [ZHR*09, LHLW10]. Thus, interactive applications in the near future are likely to leverage these technologies for obtaining highly customizable rendering with real-time performance. As alternatives to hardware rendering, it is important to build programmable pipelines with maximal efficiency on GPU-like architectures, so that the cost of flex-

† {apatney, stzeng, jowens}@ucdavis.edu

ibility does not overshadow any loss in performance. This is especially important for pipeline stages that are typically implemented as fixed-function hardware.

In this paper, we study the parallelization of composite/filter on a modern GPU architecture, and propose a scheme that parallelizes composite/filter not just across pixel/subpixel locations, but over all available fragments (Figure 1). We demonstrate the applicability of this technique for several applications, and compare the obtained performance of the proposed fragment-parallel composite (FPC) against a conventional pixel-parallel composite (PPC) scheme, i.e. one that parallelizes only across pixel/subpixel locations. Using a synthetic benchmark, we also study the variation in the relative performance of FPC and PPC across the continuum from scenes with a few depth layers to those with a large number of depth layers. Our implementation maintains a generic setup, and allows an arbitrary number of semi-transparent layers for each subpixel location. The contributions of this paper are as follows:

- We identify a novel strategy to parallelize composite and filter across individual depth fragments, and describe an implementation on a modern GPU. We demonstrate that this technique is applicable to several areas with four examples: a particle system, a volume renderer, a polygon rasterizer and an interactive Reyes pipeline.
- We compare the performance of the proposed fragment-parallel strategy against a pixel-parallel version, and study the behavior of both with varying depth complexity. We show that for the same number of fragments, a pixel-parallel technique loses performance with high as well as variable depth complexity, while the performance of a fragment-parallel version is much more consistent.

2. Related Work

Efficient rendering of semi-transparent surfaces, both for interactive and offline rendering applications, has been a topic of active research in recent years. The gold standard for offline rendering is the A-buffer [Car84], a CPU-based serial compositing algorithm. Wittenbrink use a recursive formulation of the composite equation in their R-buffer [Wit01] approach, and Jouppi and Chang propose a fixed sample-storage in their Z^3 [JC99] approach, both showing that modifications to the original approach can achieve accurate and efficient transparency on highly constrained hardware.

More recently, there has been significant research in GPU-based techniques for interactive compositing. A common technique for obtaining transparency in interactive rendering is called depth peeling [Eve01]. This approach correctly renders multi-fragment effects, but requires a number of rendering passes equal to the maximum number of depth layers. For scenes with a high depth complexity, this can be extremely inefficient. More recent work [BCL*07, LHLW09] reduces the number of rendering passes by a constant factor,

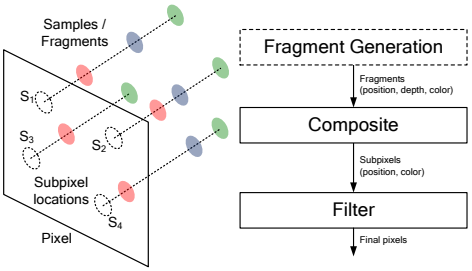


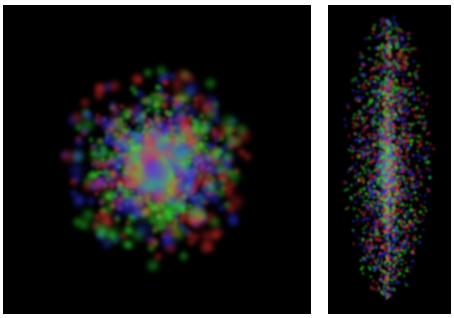
Figure 1: *The composite/filter stage of the graphics pipeline combines the contributions from all primitives that intersect a pixel into a color value for that pixel. The contribution of each primitive to its subpixel value is called a fragment. First, fragments are composited into a single subpixel value, then the filtering operation combines the subpixel colors into a final pixel color. Previous approaches parallelize over pixels or subpixels, but our technique parallelizes over fragments, achieving superior and more predictable performance especially on scenes with high depth complexity and with high variability of fragments per pixel.*

but the number of passes is still governed by the depth complexity of the scene. Wexler et al. [WGER05] use z-batches to accelerate Reyes-like hidden surface removal and depth peeling, but are still dependent on the depth. Their focus, however, was on non-interactive high quality rendering accelerated by the GPU rather than real-time rendering. Myers and Baviol's recent GPU-based A-buffer implementation [MB07] is also pixel-parallel.

Recently, Gruen et al. [GT10] demonstrated interactive order-independent transparency using GPU-based linked lists in Direct3D 11. Their approach parallelizes the task across screen pixels, but uses a serial linked-list traversal along the depth. Two notable recent implementations of programmable rendering pipelines [ZHR*09, LHLW10] each include composite stages written entirely on a programmable GPU. Both these techniques also parallelize the operation across pixels, but along depth, they remain serialized.

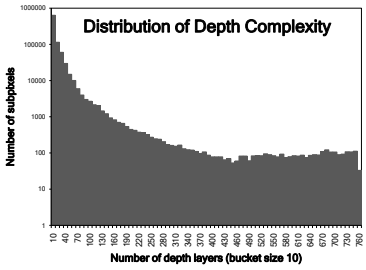
Stochastic transparency [ESSL10] uses screen-door-like methods to obtain approximate estimates of fragment contribution and generate a composite result. Our technique, in contrast, efficiently generates deterministic estimates of fragment contribution. We consider our problem of efficient compositing when all fragments are available orthogonal to their approach.

Note that these real-time techniques for order-independent transparency exploit parallelism in the screen dimensions, but do not do so along the depth dimension. While this is acceptable and indeed an efficient solution for scenes with depth complexity up to a few tens of layers, performance will be affected if the number of depth layers approaches hundreds or even thousands, or if the variance



(a) Top view of the scene (800 depth layers)

(b) Side view (170 depth layers)



(c) Distribution of Depth Complexity

Figure 2: *This figures shows two views of a 3000 particle system that involves a large variation in depth. When the collection of particles is viewed from the side, we see a maximum of 170 depth layers. However, viewing the particles head-on generates up to 800 depth layers for pixels. A robust rendering system should be able to handle both these cases. Depth variation within a single view is also important—the histogram showing the number of subpixels having a given depth complexity (note that the y-axis is in log-scale) shows that most subpixels have low depth complexity, but very few of them have an extremely large number of layers. If handled serially, these can become bottlenecks in the composite process, hampering the performance of the entire system.*

between depths is large. For such applications, an approach that extends parallelization to all fragments is likely to be a better choice.

Sintorn et al. [SA09] present a technique that parallelizes blending across available fragments for rendering hair with constant opacity. Our method is generic and can handle fragments with arbitrary geometry and opacity values.

3. Data-Parallel Composite and Filter

In this section, we describe the two strategies for programmable composite and filter: pixel-parallel composite (PPC) and fragment-parallel composite (FPC). While implementations of PPC are common in past implementations (see Section 2), FPC is a new contribution of this paper.

For this discussion, we assume a pipeline structure as shown in Figure 1. Note that fragment generation is independent of the composite/filter stages, and could be performed using any technique, for instance triangle rasterization or Reyes-style sampling. The composite and filter stages are only provided with a set of fragments that contain their position, color, transparency, and depth, and their task is to compute pixel colors using the provided fragments.

3.1. Pixel-Parallel Composite (PPC)

In the pixel-parallel composite approach, the obtained fragments are first sorted such that fragments belonging to the same subpixel occur together. Then, the composite operation is performed for each subpixel location in parallel. To perform composite and obtain the final subpixel color C_s, each subpixel computes the following blending expression:

$$\begin{aligned} C_s = &\{\alpha_1 \cdot C_1 + (1-\alpha_1) \cdot \{\alpha_2 \cdot C_2 + (1-\alpha_2) \\ &\times \{\alpha_3 \cdot C_3 + (1-\alpha_3) \cdot \{\alpha_4 \cdot C_4 + (1-\alpha_4) \\ &\times \ldots \\ &\times (\alpha_N \cdot C_N + (1-\alpha_N) \cdot C_B)\}\ldots\} \end{aligned} \tag{1}$$

Here, α_i represents the alpha value (opacity) for the ith fragment, sorted in a front-to-back order. C_i represents the color of the ith fragment; C_B is the background color.

The above computation is performed in parallel for all subpixels—each thread, which we refer to as a unit of parallel operation and its interpretation may be implementation-dependent, sorts its list of fragments according to their depth, and then visits all fragments in back-to-front order, accumulating their contribution to the final subpixel color. Note that it is possible to avoid sorting if original fragments are assumed to be depth-sorted. Of course, this assumption depends on the specific pipeline and is not true in general.

The above procedure computes final colors for each subpixel. Assuming multiple subpixel locations per pixel, we now need to compute the final pixel colors. This is performed using a similar approach, parallelized over individual pixels: for each pixel in parallel, visit each associate subpixel and accumulate their contribution to the final pixel color. Algorithm 1 describes the combined approach for pixel-parallel composite and filter.

3.2. Fragment-Parallel Composite (FPC)

Now we describe the fragment-parallel strategy for composite and filter. We identify a unique parallelization for the composite operation, and use it to implement high-performance composite and filter. We start with Equation 1

Algorithm 1 Pixel-Parallel Composite and Filter

procedure PPC (samples)

1: Sort samples according to their subpixID
2: Use scan/compact to get subpixel and pixel starting positions
3: **for all** subpixels s **do** {in parallel}
4: Sort all fragments corresponding to s
5: $C_s = C_B$
6: **for all** fragments f **do** {back-to-front}
7: $C_s = (1-\alpha_f)\cdot C_s + \alpha_f \cdot C_f$
8: **end for**
9: **end for**
10: **for all** pixels p **do** {in parallel}
11: $C_p = 0$
12: **for all** subpixels s **do**
13: $C_p = C_p + \kappa_s \cdot C_s$ // κ is the filter coefficient
14: **end for**
15: **end for**

end procedure

and rewrite it by expanding parentheses:

$$\begin{aligned} C_s = {} & \alpha_1 \cdot C_1 + (1-\alpha_1)\cdot\alpha_2\cdot C_2 \\ & + (1-\alpha_1)\cdot(1-\alpha_2)\cdot\alpha_3\cdot C_3 \\ & + (1-\alpha_1)\cdot(1-\alpha_2)\cdot(1-\alpha_3)\cdot\alpha_4\cdot C_4 \\ & + \ldots \\ & + \left(\prod_{i=1}^{N}(1-\alpha_i)\right)\cdot 1\cdot C_B \\ C_s = {} & F_1 + F_2 + F_3 + \ldots + F_N + F_{N+1}, \end{aligned}$$

where $F_k = \left(\prod_{i=1}^{k-1}(1-\alpha_i)\right)\cdot\alpha_k\cdot C_k$ for the kth sorted fragment represents its color contribution to the final sample color. Here, the $(N+1)$th fragment is opaque and represents the background color.

We can now deduce a fragment-parallel formulation. To do this, we break down the computation into two parts:

1. In parallel for each fragment, compute F_k, where k is the sorted fragment index. Note that this is equivalent to performing a parallel-scan (with the multiplication operator) over $(1-\alpha_i)$s, and post-multiplying the result by $\alpha_k \cdot C_k$.
2. Perform parallel reduction to add all F_ks corresponding to a subpixel. The result of this reduction is C_s, the final color for this subpixel.

As seen above, dividing the composite task into the two parts offers a unique opportunity for parallelization. Scan and reduction are well-known parallel primitives and efficient GPU implementations are readily available [HOS*09]. Furthermore, it is also possible to perform these operations over all pixels in parallel. In this case, the primitives to be used are parallel-segmented-scan (mul) and parallel-segmented-reduction (sum). Efficient GPU implementations are available for these as well [HOS*09].

Let us consider the filter operation now. We need to average final colors of subpixels corresponding to a pixel. This can again be formulated as a parallel-segmented-reduction (sum), followed by normalization. A simple average assumes a box-filter. For other filters, one can pre-multiply subpixel colors with weights corresponding to the filter and subpixel location.

We thus have a three-step parallelization for composite and filter. Interestingly, this can be further reduced to two steps by recognizing that the final two steps, both parallel-segmented-reductions (sum), can be merged into a single parallel-segmented-reduction (sum). The first step will now use parallel-segmented-scan (mul) to compute $F_k = \left(\prod_{i=1}^{k}(1-\alpha_i)\right)\cdot\alpha_k\cdot C_k$ and will also pre-multiply F_k by the filter-weight. The second step will perform a parallel-segmented-reduction (sum) over all fragments belonging to a pixel (instead of a subpixel), which will give the final pixel colors.

Algorithm 2 Fragment-Parallel Composite and Filter

procedure FPC (samples)

1: Sort samples using (subpixID,depth) as a double key
2: Get pixel and subpixel starting flags, $flag_p$ and $flag_s$
3: Obtain an array A containing $(1-\alpha)$ for all fragments
4: Create array F = SegmentedScan(mul, $flag_s$, A)
5: **for all** fragments f **do** {in parallel}
6: $F_f = F_f \cdot \alpha_f \cdot C_f \cdot \kappa_f$ // κ is the filter coefficient for f
7: **end for**
8: Final pixels P = SegmentedReduce(add, $flag_p$, F)

end procedure

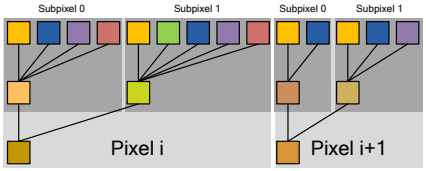
These two steps assume fragments for each subpixel occur together, and arranged in a front-to-back fashion. Subpixels corresponding to a pixel must also occur together. We can achieve this by pre-sorting the initial list of generated fragments using a two-key sorting scheme, with subpixel ID as the first key, and depth as the second one. This method of sorting fragments is similar to the one used in RenderAnts [ZHR*09].

This parallelization strategy is the primary contribution of this paper, and is described in detail in Algorithm 2.

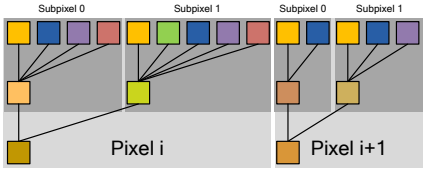
4. Results

We implemented the PPC and FPC schemes on a NVIDIA GTX280 graphics processor. We used the CUDA programming model, which is similar to other alternatives for programming GPUs, like OpenCL and Direct3D 11 Compute Shader. Hence, we expect our technique to be applicable to applications written using these models as well.

To obtain optimized implementations of parallel-segmented-scan and parallel-segmented-reduction, we use



(a) Pixel-Parallel Composite and Filter



(b) Fragment-Parallel Composite and Filter

Figure 3: *This figure shows the graphical formulation of the pixel-parallel and fragment-parallel composite and filter techniques. Ample parallelism is available in both these approaches, however, a pixel-parallel approach suffers from serialization across depth samples which could be a performance bottleneck. Fragment-parallel composite avoids this by compositing in parallel across depth samples, and uses Segmented Scan and Reduction primitives for this.*

the CUDPP library [HOS*09]. Unfortunately, the library does not provide a parallel-segmented-reduction, so we use the parallel-segmented-scan to obtain the same result.

4.1. Application Examples

Our contributions are best applied to domains of rendering where high depth complexities are common. In this Subsection, we will cover four common application domains. They are implemented using the same infrastructure as above, and form the front-end of the programmable pipeline in Figure 1.

Note that wherever required, we implemented our own software CUDA-based rasterization/sampling routines. Despite our best efforts, they may not be the most efficient. Thus, for performance results, we do not include the time taken for fragment generation. However, both PPC and FPC will work equally well if used with a built-in GPU rasterizer. Since CUDA does not have direct access to rasterized fragments, a corresponding implementation would require using Direct3D Compute Shader. Such an implementation is not covered in this paper because a software rasterizer fits better in our effort towards fully software rendering pipelines. Also, the impact of using a built-in rasterizer would be generally orthogonal to the composite/filter results presented here. Finally, we are unaware of any publicly-available efficient implementations of the required parallel primitives.

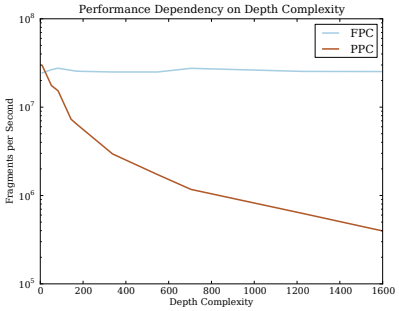


Figure 4: *Comparison between PPC and FPC with varying depth complexity. In the synthetic fragment generator we generate scenes with varying depth complexity and measure the fragments processed per second with each approach. Notice that at low depth complexities, PPC will perform slightly better than FPC, but for scenes with very high depth complexities, FPC always stays at roughly a constant while the performance of PPC decreases dramatically.*

Particle and Sprite Rendering Particle systems are useful rendering primitives to depict complex effects like fire, smoke, and clouds. They consist of a large number of particles, each of which is usually rendered as a semi-transparent polygon. We implemented a simple particle system intended to depict a smoke-like effect. We render each particle as a quad aligned with the camera-plane and filled with a semi-transparent texture. We rasterize these quads using a CUDA kernel where each thread generates fragments for one quad. The particles move in 3D space around the origin, approximating the desired effect as shown in Plate I.

Volume Rendering Volumes as rendering primitives are important for several applications, most notably scientific visualization. To test FPC over such a workload, we implemented a major-axis slicing technique to generate fragments for a 3D texture. Since quads used for this purpose are larger than those used for particles, we implemented a CUDA-based rasterizer for large quads, which parallelizes fragment-generation over several threads. Plate I shows an example rendering of a $256 \times 256 \times 256$ CT scan skull dataset.

Even though a major-axis slicing volume renderer generates fragments in back-to-front order, both PPC and FPC treat them as generic and unsorted. Thus the obtained results represent generalized composite/filter performance.

Real-time Reyes We also implemented a Reyes-style micropolygon renderer as a source of fragments for PPC and FPC techniques. We map all stages of the pipeline—smooth-surface subdivision, shading and sampling—to a GPU, and composite the obtained samples using the proposed technique. Results for this application are shown in Plate I.

Application	Number of fragments	Max Depth	FPC time	PPC time
Particles	2.9M	400	58 ms	440 ms
Volume	2.8M	256	60 ms	143 ms
Reyes (grass)	0.88M	~40	18 ms	15 ms
Polygon	1.1M	34	22 ms	21 ms

Table 1: *This table shows the obtained performance of FPC and PPC approaches for the four test applications. For each pipeline implemented, we give the time taken by each algorithm for the composite and filter tasks. Note that while the variation in performance for FPC is primarily due to the number of samples composited, for PPC the depth complexity has a direct impact. This is further studied in Figure 4.*

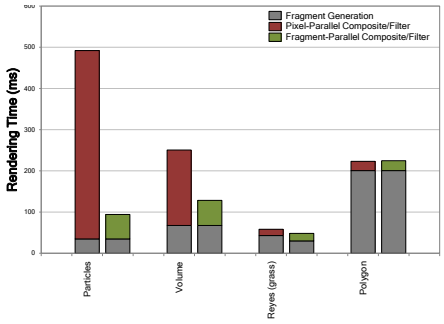


Figure 5: *Breakdown of rendering time for applications shown in Table 1. Note that FPC performs similar to PPC for simpler rendering environments, but when the number of depth layers increases the difference in performance can become significant. The difference in fragment generation time for Reyes is an implementation-level detail affecting sample storage, which is out of scope for this paper.*

Polygon Rendering We implemented a polygon rasterizer in CUDA to show the effects of FPC on traditional polygon models. Our rasterizer parallelizes over triangles and quads similar to a fixed-function GPU. Plate I shows the results of our polygon rasterizer on the AMD Mecha model. Analyzing the time shows that the majority of the time is spent on rasterization, with only ten percent being spent on FPC itself. This model generates roughly 1.2M fragments per frame.

4.2. Performance Results

Comparison between PPC and FPC We now look at the performance behavior of the PPC and FPC techniques. For the applications described above, Table 1 shows the obtained performance for composite and filter and Figure 5 shows the breakdown of total rendering time. We compare the two for varying depth complexities and applications. Table 1 shows that the performance of FPC is dependent mainly on the number of fragments generated per scene. PPC is not only dependent on this, but also on the depth complexity of the scene. Figure 5 shows that the impact of using FPC may not be immediately apparent for some applications (e.g. polygon rendering), but for others (e.g. particles) the impact is notable. To fully examine this behavior of depth dependencies, we turn to a synthetic fragment generator.

Performance Variation For a more concrete comparison between the performance behavior of pixel-parallel and fragment-parallel composite strategies, we designed a synthetic experiment. We generate a fixed number of fragments, randomly distributed across the screen and in depth. Given a constant number of fragments, varying the screen extent of fragment positions also varied the scene depth complexity. Thus, we obtain a dynamic balance between pixel- and depth-parallelism. Figure 4 shows the resulting variation.

We measure performance using the metric of fragments processed per second. All of our test cases are based on 1M fragments. FPC shows a constant performance even at high depth complexity. While PPC outperforms FPC at scenes with very low depth complexity, PPC shows a drastic decrease in performance at even moderately high depth complexities. This resonates with our original insight that FPC can extend the parallelism of composite beyond per-pixel composite and utilize the GPU more efficiently.

5. Discussion

From the results above, it is clear that the proposed FPC is an efficient technique for rendering scenes with semi-transparent objects. However, we note that performance advantages of FPC are obvious only for scenes with high depth complexities. Thus, while it is an acceptable alternative for composite and filter in programmable rendering, fragment-parallel composite is much more beneficial for applications like particle systems and high-quality pipelines where the number of depth layers is usually large.

We see two possible shortcomings of FPC. The first is a possible increase in accesses to the GPU memory due to multiple kernel passes, and the second is the high memory footprint due to temporary queues for fragment storage. In the future we would like to reduce that overhead.

Looking forward, we believe compositing scenarios where FPC outperforms PPC will become more common as scenes become more complex. As a result, parallelizing over fragments will be increasingly more rewarding than parallelizing over pixels. But does pixel parallelism offer enough parallelism moving forward? We believe the answer is no: Given the modest memory allocations of modern GPUs and the rapid increase in scene complexity, it is unlikely that all fragments in a complex scene will fit into GPU memory. Offline CPU renderers combat this complexity by dividing the screen into tiles rendered one at a time. Considering smaller and smaller times will tend to diminish the parallelization domain for PPC, while as GPUs become wider, it will become increasingly difficult to keep the their execution units

occupied. Because storage is proportional to the number of fragments and not the number of pixels, we feel that producing enough fragments to fill both the GPU's memory and its execution units and then parallelizing across fragments is a more sensible strategy going forward.

6. Conclusion and Future Work

The combination of programmable hardware and programmable pipelines gives us the opportunity to rethink the basic assumptions of the graphics pipeline. In graphics hardware, composite/filter has historically exploited parallelism across pixels. While this is sound for scenes with low depth complexity and variance, we argue in this paper that fragment parallelism is the right approach, particularly as scenes get more complex. Our work also provides a viable solution for order-independent transparency.

The programmable pipeline abstraction offers a potentially enormous benefit to programmers, allowing them to apply the creativity we've seen only in shaders to the entire pipeline. We can expect that future graphics pipelines will not only encompass traditional real-time and offline pipelines but also allow customized hybrid pipelines built from a variety of traditional stages. We are excited about the recent work in parallel subdivision and rasterization algorithms, and hope that our composite/filter work will serve as another core building block for next-generation programmable pipelines.

Acknowledgments

We would like to thank Matt Pharr, Aaron Lefohn, and Mike Houston for their suggestions in the early stages of the work; the anonymous reviewers and Mike Roberts for their excellent reviews; Justin Hensley (AMD) for the Mecha model; Jeff Stuart for the volume model and parsing code; and Shubho Sengupta for reading early drafts of the paper. Thanks to Headus Inc. (http://www.headus.com.au) for the Killeroo model. Thanks also to our funding agencies, the National Science Foundation (Award 0541448) and the SciDAC Institute for Ultrascale Visualization, and to NVIDIA for equipment donations and an NVIDIA Graduate Fellowship.

References

[BCL*07] BAVOIL L., CALLAHAN S. P., LEFOHN A., COMBA J. L. D., SILVA C. T.: Multi-fragment effects on the GPU using the k-buffer. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2007), ACM, pp. 97–104. 2

[Car84] CARPENTER L.: The A-buffer, an antialiased hidden surface method. In *Computer Graphics (Proceedings of SIGGRAPH 84)* (July 1984), pp. 103–108. 2

[ESSL10] ENDERTON E., SINTORN E., SHIRLEY P., LUEBKE D.: Stochastic transparency. In *Proceedings of the 2010 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), pp. 157–164. 2

[Eve01] EVERITT C.: *Interactive Order-Independent Transparency*. Tech. rep., NVIDIA Corporation, May 2001. http://developer.nvidia.com/object/Interactive_Order_Transparency.html. 2

[GT10] GRUEN H., THIBIEROZ N.: OIT and indirect illumination using DX11 linked lists. In *Proceedings of Game Developers Conference 2010* (Mar. 2010). http://developer.amd.com/gpu_assets/OITandIndirectIlluminationusingDX11LinkedLists_forweb.ppsx. 2

[HOS*09] HARRIS M., OWENS J. D., SENGUPTA S., ZHANG Y., DAVIDSON A.: CUDPP: CUDA data parallel primitives library. http://gpgpu.org/developer/cudpp/, 2009. 4, 5

[JC99] JOUPPI N. P., CHANG C.-F.: Z3: An economical hardware technique for high-quality antialiasing and transparency. In *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (Aug. 1999), pp. 85–93. 2

[LHLW09] LIU F., HUANG M.-C., LIU X.-H., WU E.-H.: Efficient depth peeling via bucket sort. In *Proceedings of High Performance Graphics 2009* (Aug. 2009), pp. 51–57. 2

[LHLW10] LIU F., HUANG M.-C., LIU X.-H., WU E.-H.: 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* (Feb. 2010), pp. 75–82. 1, 2

[LNOM08] LINDHOLM E., NICKOLLS J., OBERMAN S., MONTRYM J.: NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro 28*, 2 (Mar./Apr. 2008), 39–55. 1

[MB07] MYERS K., BAVOIL L.: Stencil routed A-buffer. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches* (New York, NY, USA, 2007), ACM, p. 21. 2

[Mic08] MICROSOFT CORPORATION: Introduction to the Direct3D 11 graphics pipeline. http://www.microsoft.com/downloads/details.aspx?familyid=E410716F-12BF-4E8F-AC41-97B4440C3B90, 2008. 1

[Pha06] PHARR M.: Interactive rendering in the post-GPU era. Keynote, Graphics Hardware 2006. http://www.graphicshardware.org/previous/www_2006/presentations/pharr-keynote-gh06.pdf, Sept. 2006. 1

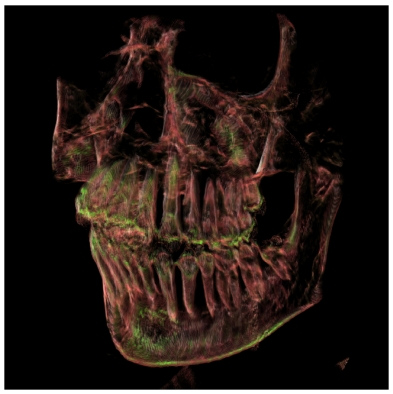
[SA09] SINTORN E., ASSARSSON U.: Hair self shadowing and transparency depth ordering using occupancy maps. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2009), ACM, pp. 67–74. 3

[SCS*08] SEILER L., CARMEAN D., SPRANGLE E., FORSYTH T., ABRASH M., DUBEY P., JUNKINS S., LAKE A., SUGERMAN J., CAVIN R., ESPASA R., GROCHOWSKI E., JUAN T., HANRAHAN P.: Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3 (Aug. 2008), 18:1–18:15. 1

[WGER05] WEXLER D., GRITZ L., ENDERTON E., RICE J.: GPU-accelerated high-quality hidden surface removal. In *2005 SIGGRAPH / Eurographics Conference on Graphics Hardware* (July 2005), pp. 7–14. 2

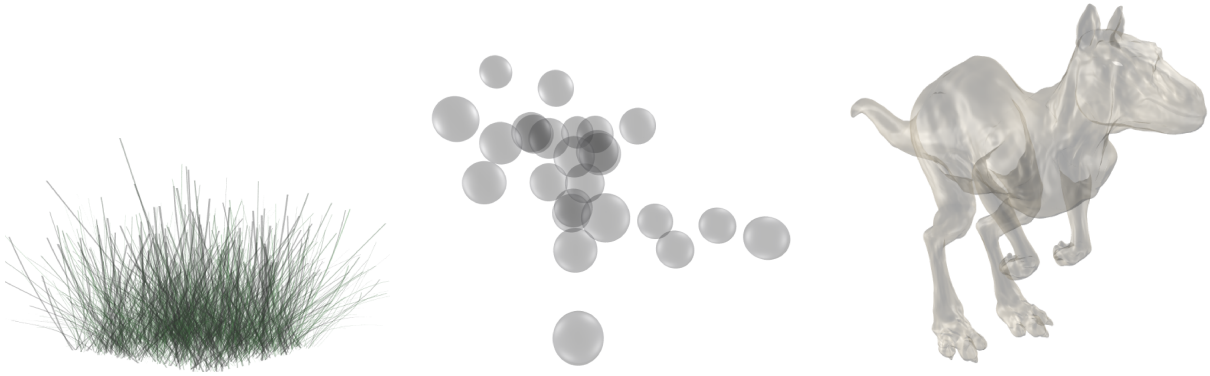
[Wit01] WITTENBRINK C.: R-buffer: A pointerless A-buffer hardware architecture. In *Graphics Hardware 2001* (Aug. 2001), pp. 73–80. 2

[ZHR*09] ZHOU K., HOU Q., REN Z., GONG M., SUN X., GUO B.: RenderAnts: Interactive Reyes rendering on GPUs. *ACM Transactions on Graphics 28*, 5 (Dec. 2009), 155:1–155:11. 1, 2, 4

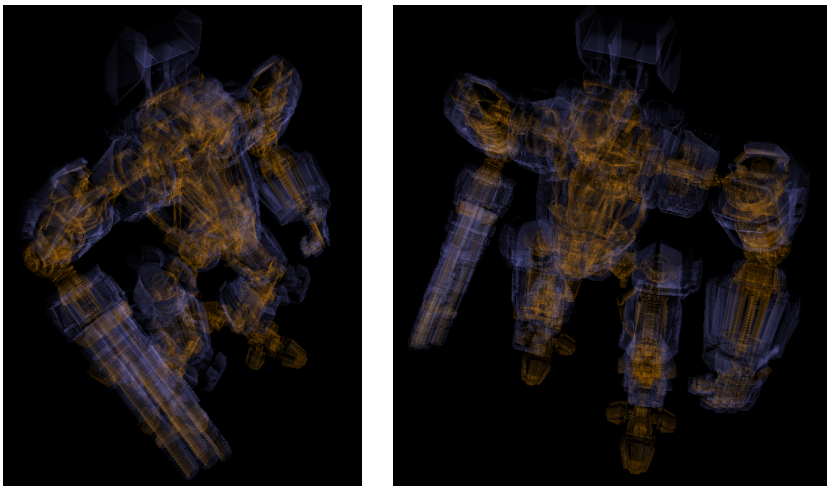


Volume Renderer

Particle Renderer



Reyes



Mecha Model

Plate I: *Examples of Fragment-Parallel Composite in action for a variety of applications. Top Row Left: This volume consists of a set of 256×256×256 voxels, and for the current viewpoint generates about 2.5M fragments. The composite time for this scene is around 60 ms. Top Row Right: This scene consists of 3000 particles, each rendered as a camera-aligned billboard with an alpha-texture. The maximum depth complexity for this scene is about 650 layers, and rasterization generates about 15M fragments. FPC takes about 250 ms to composite them. Center Row: obtained images from our GPU-based Reyes renderer, which uses the FPC approach to composite and filter the generated fragments. Bottom Row: A polygonal rendering of the AMD Mecha model. The polygon model rasterizes into roughly 1–1.2M fragments, which are composited using FPC in about 30 ms.*