Multi-Layer Alpha Blending

Marco Salvi* Karthik Vaidyanathan†

Intel Corporation

Figure 1: *These three complex scenes are rendered in real-time with multi-layer alpha blending. Each scene displays a variety of transparent objects that generate up to ten million fragments. The images are generated in a single rendering pass while using only 16 (left and middle images) or 32 (right image) bytes per pixel. The final result is virtually indistinguishable from the reference A-buffer based solution.*

Abstract

We introduce multi-layer alpha blending, a novel solution to real-time order-independent transparency that operates in a single rendering pass and in bounded memory. The main contribution of our method is a new scalable approximation for the compositing equation that makes possible to easily trade off better image quality for more memory and lower performance. We demonstrate improved image quality and performance over previously published methods, while also reducing memory requirements.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Bitmap and Framebuffer Operations

Keywords: order-independent transparency, alpha blending, compositing, compression, pixel synchronization

1 Introduction

Image space compositing of arbitrarily ordered geometric primitives is easily solved by buffering, sorting and blending of all fragments that contribute to a given pixel. This method, first introduced by Loren Carpenter [1984] is known as A-buffer and is extensively used by off-line renderers. It has not been adopted by the real-time rendering community due to its unbounded memory requirements and generally low performance.

Correctly compositing fragments is an important and long-standing open problem in real-time computer graphics. Significant progress was made in the last several years to develop new techniques that provide better rendering for transparent primitives. Nonetheless, even the most recent approaches suffer from a multitude of issues that make them still less than ideal solutions, such as reduced image quality, low performance or exorbitant memory consumption.

Multi-layer alpha blending (MLAB) enables high quality single-pass rasterization of transparent primitives while using a fixed amount of memory. Compared to other methods our algorithm is faster and due to its simplicity is easy to implement and deploy in real-world applications.

The most important result of our work is a novel streaming approximation for the compositing equation that often produces images indistinguishable from the reference solution and that is weakly dependent upon primitive submission order. Our method is scalable and allows to improve image quality by trading off memory requirements and performance.

2 Previous Work

The color contribution of a set of fragments $\mathbb{F}$ to the pixel color can be computed as follows:

$$\vec{A}_{transp} = \sum_{\mathbb{F}} \vec{p}_i t(z_i) \tag{1}$$

where $\vec{p}_i$ and z_i are respectively the fragment pre-multiplied alpha color $\vec{c}_i \alpha_i$ and distance from the viewer and $t(z_i)$ is the transmittance between the viewer, and the fragment:

$$t(z) = \prod_{\mathbb{F}} \begin{cases} 1 & z \leq z_i \\ 1 - \alpha_i & z > z_i \end{cases} \tag{2}$$

Sorting fragments in back-to-front order enables solving Equation 1 in a recursive fashion:

$$\begin{aligned} \vec{A}_0 &= \vec{p}_0 \\ \vec{A}_n &= \vec{p}_n + (1 - \alpha_n)\vec{A}_{n-1} \end{aligned} \tag{3}$$

*e-mail: marco.salvi@intel.com

†e-mail: karthik.vaidyanathan@intel.com

This method is commonly known as *alpha blending* and was first introduced by Porter and Duff [1984]. At each step a new fragment is composited onto a RGB tuple that can be directly stored in the frame buffer. Its simplicity, performance and widespread support in APIs and graphics hardware have made alpha blending the de facto standard compositing method for real-time applications. Nonetheless, sorting all fragments that contribute to a pixel is often impractical and failure to do so can generate significant artifacts (see alpha blending results in Figure 3).

The A-buffer algorithm [Carpenter 1984] applies Equation 3 to correctly composite previously buffered and spatially sorted fragments and therefore requires an amount of memory directly proportional to the total number of processed transparent fragments. A-buffer implementations running on the GPU [Yang et al. 2010] [Vasilakis and Fudos 2012] have not been widely adopted due to low, unstable performance and unbounded memory requirements. The Z^3 algorithm [Jouppi and Chang 1999] is a bounded memory and low-cost hardware A-buffer algorithm that similarly to our method keeps its storage requirements constant, at the expense of image quality. Depth peeling [Everitt 2001] also requires a fixed amount of memory by using n rendering passes to extract the first front-most n layers, making it not particularly suitable for real-time rendering of complex scenes. Other methods improve upon the original depth peeling formulation by capturing more than one layer per rendering pass [Bavoil and Callahan 2008] [Liu et al. 2009].

Other algorithms render the transparent elements of the scene a first time to compute an exact or approximate per-pixel transmittance function (see Equation 2), followed by a second rendering pass where the final pixel color is evaluated via Equation 1. Occupancy maps [Sintorn and Assarson 2009] encode the transmittance function at regular intervals with a series of equal-height steps. This approach works well with geometry like hair that tend to have the same transmittance while occupying a well defined region of space. Opacity shadow maps [Kim and Neumann 2001] allow the transmittance function to take arbitrary values but only at regular intervals which typically causes under-sampling errors, while Fourier opacity maps [Jansen and Bavoil 2010] encodes $t(z)$ in Fourier space which is optimal for soft occluders like smoke but requires many expansion terms to encode thin objects such as glass.

Stochastic transparency [Enderton et al. 2010] extends alpha-to-coverage techniques in order to build a stochastic representation of transmittance. The solution leverages hardware support for multi-sample anti-aliasing but requires a large number of visibility samples to avoid noise artifacts. Adaptive transparency [Salvi et al. 2011] uses a fixed number of piece-wise step functions to represent transmittance. This choice makes it possible to faithfully account for a large variety and number of transparent objects, but due to graphics hardware limitations the algorithm cannot work with a fixed amount of memory. Lastly Hybrid Transparency [Maule et al. 2013] encodes the head of the transmittance function using the n front-most layers while the tail is represented by a single average term. This hybrid representation is advantageous because it is specifically tailored to be built using a bounded amount of memory, without incurring data race issues due to multiple fragments simultaneously contributing to the same pixel.

Our approach, called multi-layer alpha blending (MLAB), aims for the simplicity and performance of alpha blending by adopting a single rendering pass method that directly evaluates Equation 1 without explicitly building a separate transmittance function. We do so by generalizing Equation 3 to use an arbitrary, but bounded, number of terms for each per-pixel. This makes it possible to reduce, and in some cases completely eliminate, alpha blending compositing errors due to processing fragments in the wrong order. We compare MLAB to an A-buffer based reference solution and to most recent methods like adaptive transparency (AT) and hybrid transparency (HT).

3 Algorithm

We begin by observing that the strict ordering requirements of alpha blending can be relaxed by storing per-pixel transmittance T and depth Z, alongside the accumulated color $\vec{A}$.

The depth value can be used to determine whether a first fragment $\vec{f_a} = (\vec{p}_a, t_a, z_a)$ is behind or in front of a second fragment $\vec{f_b} = (\vec{p}_b, t_b, z_b)$, and to composite them accordingly by selecting the over or the under blending operator [Porter and Duff 1984]:

$$\begin{aligned} \vec{A}_0 &= \vec{p}_a, \quad T_0 = t_a, \qquad Z_0 = z_a \\ \vec{A}_1 &= \begin{cases} \vec{p}_b + \vec{A}_0 t_b & z_b < Z_0 \\ \vec{A}_0 + \vec{p}_b T_0 & z_b \geq Z_0 \end{cases} \end{aligned}$$

While one can try to generalize this method to many fragments:

$$\begin{aligned} &\vec{A}_0 = \vec{p}_0, \qquad T_0 = t_0, \qquad Z_0 = z_0 \\ &\vec{A}_n = \begin{cases} \vec{p}_n + \vec{A}_{n-1} t_n & z_n < Z_{n-1} \\ \vec{A}_{n-1} + \vec{p}_n T_{n-1} & z_n \geq Z_{n-1} \end{cases} \\ &Z_n = \begin{cases} z_n & z_n < Z_{n-1} \\ Z_{n-1} & z_n \geq Z_{n-1} \end{cases} \\ &T_n = T_{n-1} t_n \end{aligned} \tag{4}$$

this improved algorithm cannot still correctly handle an arbitrary number of fragments. To prove this we note that a third fragment $\vec{f_c}$ will be correctly composited with $\vec{f_a}$ and $\vec{f_b}$ only if it is not located between them. Therefore, as more fragments are blended together, the distance between the closest and furthest fragments can grow, making it even more likely to incorrectly composite future fragments.

3.1 Blending Array

To address this problem we further generalize alpha blending to recursively solve Equation 1 by using up to m terms per pixel. We store these terms in a *blending array* $\mathbf{B}$, with each row j including accumulated pre-multiplied color $\vec{A}_j$, transmittance T_j and distance from the viewer Z_j. If the blending array rows are sorted in front-to-back order Equation 1 becomes:

$$\vec{A}_{transp} = \sum_{j=0}^{m-1} \vec{A}_j \prod_{i=0}^{j-1} T_i \tag{5}$$

where $T_{-1} = 1$ (see Equation 2). Note that the last row transmittance T_{m-1} is not used to calculate $\vec{A}_{transp}$ but it is required to composite the transparent fragments over an opaque background to obtain the final pixel color:

$$\vec{A}_{pixel} = \vec{A}_{opaque} \prod_{i=0}^{m-1} T_i + \vec{A}_{transp} \tag{6}$$

We first initialize $\mathbf{B}$ with black and fully transparent fragments located at infinite distance from the viewer:

$$\begin{pmatrix} \vec{A}_0 = \vec{0} & T_0 = 1 & Z_0 = \infty \\ \vec{A}_1 = \vec{0} & T_1 = 1 & Z_1 = \infty \\ \vdots & \vdots & \vdots \\ \vec{A}_{m-1} = \vec{0} & T_{m-1} = 1 & Z_{m-1} = \infty \end{pmatrix}$$

To add a new fragment $\vec{f}_{new} = (\vec{p}_{new}, t_{new}, z_{new})$ we perform an ordered insertion of a new j_{th} row vector such that:

$$Z_{0\ldots j-1} < z_{new} \leq Z_{j\ldots m-1}$$

This operation generates a new blending array $\mathbf{B}'$ with an additional row:

$$\begin{pmatrix} \vec{A}'_0 = \vec{A}_0 & T'_0 = T_0 & Z'_0 = Z_0 \\ \vdots & \vdots & \vdots \\ \vec{A}'_{j-1} = \vec{A}_{j-1} & T'_{j-1} = T_{j-1} & Z'_{j-1} = Z_{j-1} \\ \vec{A}'_j = \vec{p}_{new} & T'_j = t_{new} & Z'_j = z_{new} \\ \vec{A}'_{j+1} = \vec{A}_j & T'_{j+1} = T_j & Z'_{j+1} = Z_j \\ \vdots & \vdots & \vdots \\ \vec{A}'_m = \vec{A}_{m-1} & T'_m = T_{m-1} & Z'_m = Z_{m-1} \end{pmatrix}$$

which is too large to be stored in memory since we previously decided to work within a fixed memory budget of m rows per pixel.

3.2 Data Compression

To address our storage problem we compress $\mathbf{B}'$ to generate a new blending array $\mathbf{B}$, and we do so while minimizing compression artifacts. Removing one row would throw away important information, instead we opt for merging two neighboring rows i and $i+1$ into a new i_{th} row. This allows to retain part of the original data and it also reduces the likelihood of incorrectly compositing future fragments that are spatially located in between compressed rows.

We ask that the color contribution of to-be-merged rows must be equivalent to the color contribution of the row that will replace them. This constraint translates to (see Equation 5):

$$\vec{A}_{merge} = \vec{A}'_i + A'_{i+1}T'_i \tag{7}$$

$$T_{merge} = T'_i T'_{i+1} \tag{8}$$

where $\vec{A}_{merge}$ and T_{merge} are respectively the pre-multiplied alpha color and transmittance associated to the merged rows. Lastly we assign Z_i, the depth of the closest row, to the newly merged row. This guarantees convergence with the image rendered by a z-buffer when a fragment's transmittance is equal to zero (i.e. fully opaque fragments).

The new blending array $\mathbf{B}$ obtained by compressing $\mathbf{B}'$ becomes:

$$\begin{pmatrix} \vec{A}_0 = \vec{A}'_0 & T_0 = T'_0 & Z'_0 = Z_0 \\ \vdots & \vdots & \vdots \\ \vec{A}_{i-1} = \vec{A}'_{i-1} & T_{i-1} = T'_{i-1} & Z_{i-1} = Z'_{i-1} \\ \vec{A}_i = \vec{A}'_i + A'_{i+1}T'_i & T_i = T'_i T'_{i+1} & Z_i = Z'_i \\ \vec{A}_{i+1} = \vec{A}'_{i+2} & T_{i+1} = T'_{i+2} & Z'_{i+1} = Z'_{i+2} \\ \vdots & \vdots & \vdots \\ \vec{A}_{m-1} = \vec{A}'_m & T_{m-1} = T'_m & Z_{m-1} = Z'_m \end{pmatrix}$$

3.3 Merge Error Metrics

Designing a rule to merge rows that preserves pixel color is not sufficient to fully address the image quality problem. A new incoming fragment spatially located between previously merged rows will erroneously affect both rows, not just the one behind it. Therefore, at each data compression step, it is also important to select for merging the most appropriate row pair out of the m possible.

Since we do not know in advance the location of the next incoming fragments, it is not generally possible to always merge a row pair that guarantees, or at least minimizes, the error introduced by future fragment insertions. To reduce artifacts generated by compressing the transmittance function, algorithms such as adaptive transparency [Salvi et al. 2011] discard the data that generates the smallest variation of the integral of $t(z)$. Not only a similar idea can be applied to our blending array, but it can be pushed one step further by including other data like pre-multiplied alpha color, which is typically not available to 2-pass methods like adaptive, stochastic or hybrid transparency.

To test which method works best, we evaluated through visual inspection metrics based on the smallest pixel color deltas, transmittance and distance from the viewer. We also tested all unique combinations of products of these three metrics, for a total of six different metrics. While more complex metrics seemed to occasionally lower the error in a significant way, they failed to produce spatially and temporally consistent results. Overall we found that merging the row pair with the smallest transmittance, produced the most visually pleasing images. Since the transmittance function is monotonically decreasing [Salvi et al. 2011], the row pair with the smallest transmittance always comprises the last two rows of the blending array $\mathbf{B}'$. This makes our implementation both simpler and faster.

4 Implementation

As shown in Listing 1 our algorithm can be expressed in a few lines of code. We note that concurrently shaded fragments that cover the same pixel will update the same memory locations, causing data races. Since it is not possible to map our algorithm to a sequence of atomic operations, the entire per-pixel data structure must be updated atomically. Moreover, our data compression scheme is weakly order-dependent (i.e. it becomes order-independent only when the number of rows in the blending array is as large as the number of contributing fragments), therefore updates must also happen in a well-defined order to guarantee temporally stable results. Even software or hardware support for per-pixel spin-locks or critical sections wouldn't be enough to correctly implement our algorithm on the GPU, since 3D APIs do not dictate any particular order for fragment shading.

Listing 1: *Multi-layer alpha blending fragment insertion code.*

```
MLAB(in Fragment f, inout Fragment B[m+1])
{
    // 1-pass bubble sort to insert fragment
    Fragment temp, merge;
    for (int i = 0;i < m; i++) {
        if (f.z <= B[i].z) {
            temp = B[i];
            B[i] = f;
            f    = temp;
        }
    }

    // Compression (merge last two rows)
    merge.a = B[m-1].a + B[m].a * B[m-1].t;
    merge.t = B[m-1].t * B[m].t;
    merge.z = B[m-1].z;
    B[m-1] = merge;
}
```

Similarly to adaptive transparency [Salvi et al. 2011] a possible implementation strategy for our algorithm includes rendering per-pixel lists of all transparent fragments, which are then processed

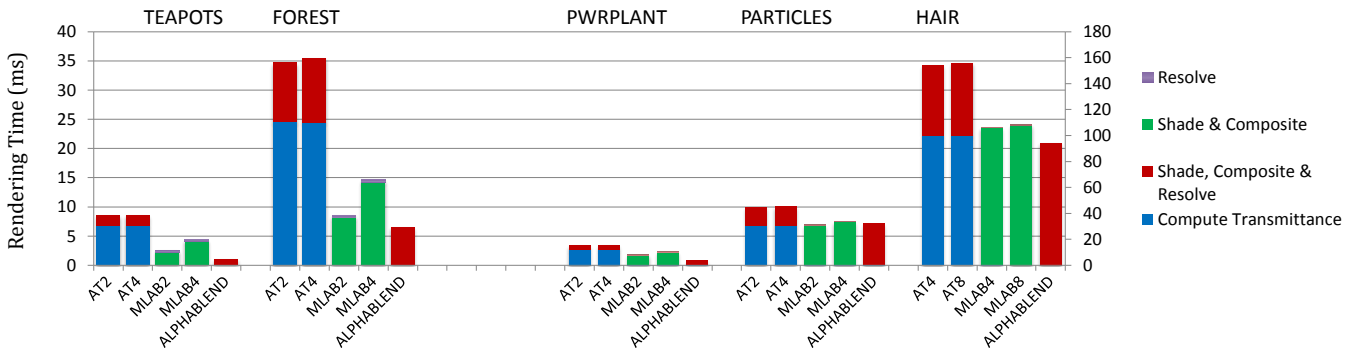

Figure 2: *Rendering times for all tested algorithms. Since MLAB and HT performance is very similar and for convenience we only report numbers for our method. Note that due to large performance gap between different scenes we use two time scales: one for TEAPOTS and FOREST (left) and one for PWRPLANT, PARTICLES and HAIR (right).*

in a single threaded fashion (i.e. one GPU thread assigned to each pixel) to compute the final pixel color. We do not use this method since the construction of per-pixel lists requires a potentially unbounded amount of memory and it does not guarantee a deterministic processing order.

To avoid atomicity and ordering issues we make use of *pixel synchronization*, a new feature recently introduced by INTEL. Pixel synchronization enables fragments that map to the same pixel to perform any type and number of read-modify-write operations in primitive submission order and without incurring data races. The performance impact of pixel synchronization is lower, compared to atomic operations; in all tested scenes it never exceeded more than 10% of the total rendering time. We note that a similar feature is available through OpenGL extensions that enable reading the content of the frame buffer from the pixel shader, therefore guaranteeing ordered atomic updates of the blending array (e.g. GL_EXT_shader_framebuffer_fetch) .

We use pixel synchronization also for our hybrid transparency (HT) implementation. Although HT is fully order-independent, this new feature enables a single pass implementation, while the original implementation based on 32 bit atomic operations requires two distinct rendering passes [Maule et al. 2013]. This change significantly improves performance over the original method.

Listing 2: *Hybrid transparency fragment insertion code.*

```
HT(in Fragment f, inout Fragment B[m],
   inout Tail T)
{
    // 1-pass bubble sort to insert fragment
    Fragment temp;
    for (int i = 0;i < m - 1; i++) {
        if (f.z <= B[i].z) {
            temp = B[i];
            B[i] = f;
            f    = temp;
        }
    }

    // Update tail if necessary
    if (B[m-1].t != 1) {
        T.accColor += B[m-1].a;
        T.accAlpha += 1 - B[m-1].t;
        T.fragCount++;
    }
}
```

4.1 MLAB and HT

By comparing Listing 1 and 2 we see that the main difference between MLAB and HT is related to how the last to-be-compressed row is processed. MLAB simply merges the last two rows of the blending array, while HT removes the last row and accumulates it to build a specialized representation for the tail of the transmittance curve. This seemingly small difference is what enable MLAB to provide improved image quality and performance (see Section 5).

Every time we exceed per-pixel storage, HT will encode the overflow fragment in its approximate tail representation, potentially introducing an error. On the other hand, MLAB compression step does not introduce any error in the final pixel color; only future fragments can generate compositing errors and only if they fall in between previously merged rows. MLAB naturally exploits the spatial coherence of the stream of incoming fragments, while its image quality tends to deteriorate when such coherence is lost. For instance MLAB can correctly composite any number of fragments that happen to be sorted in front-to-back or back-to-front order using a blending array with just one entry, while in the same case HT will deviate from the correct result as more and more fragments overflow the front-most layers and are composited in the tail representation.

4.2 Storage

As mentioned in Section 3.1 we allocate memory to store a blending array for each pixel. In particular each array row encodes premultiplied alpha, color (3 bytes), transmittance (1 byte) and distance from the viewer (4 bytes), for a total of 8 bytes. Performance, image quality and storage requirements can be easily scaled by increasing or decreasing m, the number of rows in the blending array.

For a fair comparison between MLAB and HT we allocate the same per-pixel memory for both. MLAB uses $8 \times m$ bytes for the blending array and our HT implementation uses $8 \times (m - 1)$ bytes to encode the head of the transmittance curve. The tail of the curve uses the remaining 8 bytes to store accumulated color (30 bit), accumulated alpha (16 bit) and fragment count (16 bit).

Lastly, we have implemented a reference A-buffer solution and an adaptive transparency (AT) solution. It is not possible to perform an iso-memory comparison between MLAB and the A-buffer solution since it requires an amount of memory proportional to the number of transparent fragments in the image. Although a fixed-memory 2-pass implementation of AT is feasible with pixel synchronization, we compare our method against the original unbounded memory

implementation [Salvi et al. 2011].

5 Results

We present performance and image quality comparisons of multi-layer alpha blending (MLAB), adaptive transparency (AT) and hybrid transparency (HT) against our reference A-buffer implementation. All results have been generated at a resolution of 1280×720 pixels on a laptop with an Intel i7-4950HQ processor and Iris Pro 5200 integrated graphics (47W TDP shared between CPU and GPU).

5.1 Visual Analysis

In order to better understand image quality with the different methods, we performed a visual inspection of five complex scenes exhibiting different types and distributions of transparent geometry (see Figure 3). To perform a realistic evaluation of all techniques each scene was evaluated from arbitrary view points and without submitting the geometric primitives in any particular order.

Since all algorithms are scalable, we tested two configurations for each of them. For MLAB the smallest size of the blending array was selected on a per scene basis by inspecting different configurations. We then chose a blending array size that resulted in images that were not perceivably different from the reference solution. As mentioned in Section 4.2 we used the same memory configurations for HT and MLAB. Finally we also tested AT with a number of nodes identical to the number of rows used for MLAB. In Table 1 we report the memory requirements for each scene and algorithm.

5.1.1 HAIR and PARTICLES Scenes

The HAIR and PARTICLES scenes represent some of the most complex scenarios we have tested. They include thousands of hair strands and particles that generate six to ten million transparent fragments. In both scenes, we use adaptive volumetric shadow maps [Salvi et al. 2010] for improved realism and more importantly to increase color variability and contrast across transparent fragments, which can significantly enhance image artifacts.

As seen in Figure 3, MLAB generates images very similar to the reference HAIR image with just four layers (i.e. a 4-row blending array), while both HT and AT cannot produce the same image quality even with twice the number of layers. The PARTICLES scene exhibits a similar behavior with MLAB generating a very high quality image with only two layers and HT producing significantly larger errors. We believe this is due to the fact that HT uses a single average term to represent the tail of the transmittance curve and is therefore unable to handle scenarios where several thousand particles contribute to the same region of the image.

AT produces noticeable darkening artifacts especially in the PARTICLES scene. This observation is not unexpected, since the AT algorithm adopts a biased compression scheme and tends to underestimate the transmittance function [Salvi et al. 2011]. This confirms similar findings reported by Maule et al [2013].

5.1.2 FOREST scene

The FOREST scene is an example of how order-independent transparency algorithms can be used to replace techniques based on alpha-testing to render (virtually) alias-free foliage. In this case all tested methods can generate high quality images very similar to the reference A-buffer solution. Nonetheless, when compared to HT, our algorithm converges to the correct solution using half the memory.

Scene	Fragments	A-buffer/AT	MLAB/HT
HAIR	10.6 M	124.8	28.2(56.4)
PARTICLES	6.1 M	73.3	14.1(28.2)
FOREST	6.0 M	72.2	14.1(28.2)
PWRPLANT	2.3 M	29.8	14.1(28.2)
TEAPOTS	6.0 M	72.2	14.1(28.2)

Table 1: *Memory requirements in megabytes for all tested methods. The memory used by A-buffer and AT partially depends on the number of transparent fragments in the image (e.g. 12 bytes per fragment). MLAB and HT use a fixed amount of memory for each pixel. Numbers between parenthesis indicate the memory required by the more costly configuration of the two tested for each scene.*

5.1.3 PWRPLANT and TEAPOTS Scenes

These scenes are designed to test configurations with batches of colorful geometry, that are submitted to the GPU without respecting any particular order. Such scenarios are typical of urban and architectural models. In the PWRPLANT scene all tested algorithms can render good quality images. While MLAB can approach the reference solution with only 2 layers, HT requires twice the data to produce results of similar, if not better quality. AT produces darkening artifacts similar to the other test cases.

5.2 Performance

In Figure 2 we report rendering times for MLAB, HT, AT and alpha blending. We omitted performance numbers for the reference A-buffer based solution due to the fact that is generally more than an order of magnitude slower than the other methods. We also do not explicitly quote rendering times for HT since they are virtually identical to MLAB rendering times. This is due to the fact the both implementations use, by design, the same amount of storage and memory bandwidth and perform very similar computations. Despite this, as shown in the previous section, MLAB often generates higher quality images than HT when using the same amount of memory.

We first note that with very complex scenes like HAIR and PARTICLES, performance is not particularly sensitive to the number of layers used by the algorithms. We attribute this to the fact that the execution of complex fragment shaders for volumetric lighting and shadowing consumes the vast majority of the computations and memory bandwidth, dwarfing the cost of the transparency solutions. In fact scenes employing simple shaders, such as TEAPOTS and FOREST, show how rendering times for MLAB and HT scale almost linearly with the number of layers. The performance sensitivity of AT is reduced by the first pass of the algorithm that runs expensive atomic operations to create per-pixel lists, that do not depend on the number of layers.

In all tested scenes MLAB provides real-time performance, from tens to hundreds of frames per second, and it is often significantly faster than AT. We also note that rendering times with alpha blending are not too dissimilar from MLAB and HT, while generating images of significantly worse quality in most cases. This observation suggests that the continued progress towards more powerful and efficient algorithms for real-time order-independent transparency might lead to large scale adoption of these techniques.

5.3 Failure Cases

As noted in Section 4 our algorithm, unlike HT, is not fully order-independent. Although in most cases this does not constitute a problem, complex transparent objects overlapping in image space

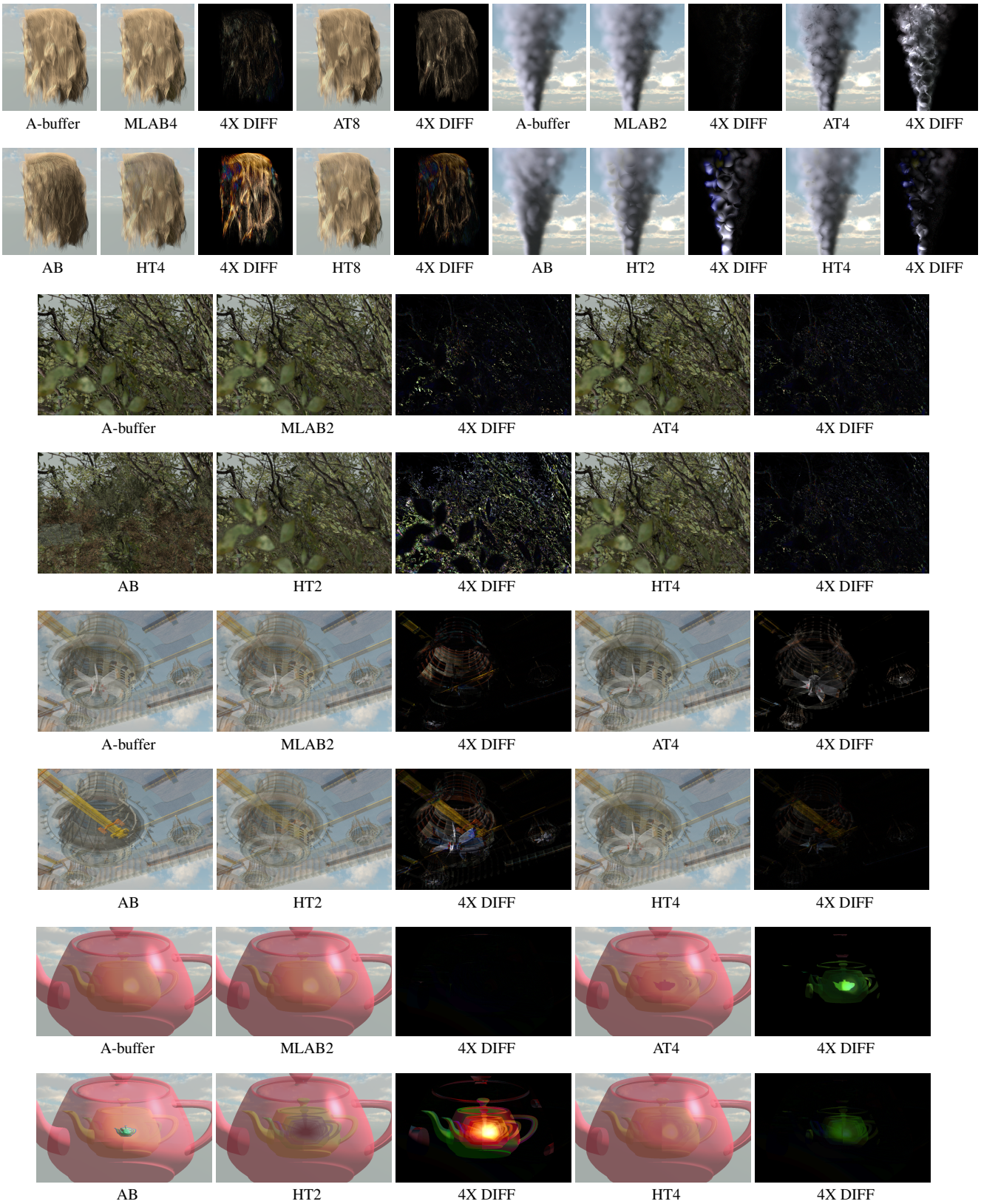

Figure 3: *We compare MLAB, HT, AT to our reference A-buffer implementation over five scenes exhibiting different types of transparent geometry, including hair, foliage and volumetric particles. From left to right, top to bottom: HAIR, PARTICLES, FOREST, PWRPLANT and TEAPOTS. The single digit associated to each method respectively indicates the number of rows of the blending array (MLAB), the number of terms used to encode the head and the tail of the transmittance curve (HT) and the number of nodes used to encode the transmittance function (AT). Please note that the difference images are enhanced by a factor of four. For comparison purposes we also provide images rendered with alpha blending (AB).*

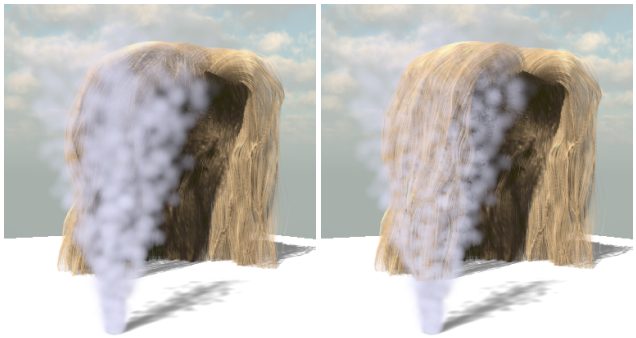

Figure 4: *Rendering order still matters for complex transparent objects. Left: we render particles, followed by hair. Right: swapping the rendering order of particles and hair causes a visible degradation in image quality. Both images are generated with MLAB8.*

might sometimes reveal order-dependent artifacts. As illustrated in Figure 4 simply swapping the order of rendering of a hair mesh and a group of particles can generate visible artifacts. In these cases a substantial increase in the number of layers does not usually produce a significantly improved image.

To better understand this problem we first note that with both rendering orders MLAB compression step will merge particle and hair fragments located at the interface between the two objects. However, if the terms being merged do not contribute much to the pixel color, subsequent fragments falling in-between compressed rows are less likely to generate visible artifacts. This is the case when rendering hair last: MLAB will be merging rows mostly located behind a thick layer of particles. Instead, if we swap the rendering order of hair and particles, merged rows will be only partially occluded (i.e. not all particles will have been rendered yet) and may produce visible artifacts.

6 Conclusion and Future Work

We have demonstrated a new single-pass and bounded-memory order-independent transparency algorithm that can render images faster and with higher quality than previously published techniques. The key idea behind our method is a scalable extension to alpha blending that uses multiple terms to solve the compositing equation in a streaming fashion. Our data representation is compact and often provides better image quality in iso-memory comparisons.

In the future, we would like to investigate ways of making MLAB more robust and resilient to changes in the primitives submission order, particularly in scenarios with complex distributions of transparent fragments. We would also like to extend our algorithm to support multi-sample anti-aliasing (MSAA) using a per-pixel data structure, instead of storing and updating a per-sample blending array. This can possibly be accomplished by leveraging techniques like Z^3 [Jouppi and Chang 1999] to store and update a coverage mask and depth gradients for each row in the blending array.

7 Acknowledgments

We thank the Advanced Rendering Technology team at Intel, Aaron Lefohn and the reviewers for providing valuable feedback. We also thank Chuck Lingle and Tom Piazza for supporting this research. Finally we thank Jason Mitchell and Wade Schinn at Valve Software for the data sets used in the FOREST scene and Cem Yuksel for the hair model.

References

BAVOIL, L., AND CALLAHAN, MEYERS, K. 2008. Order independent transparency with dual depth peeling. Tech. rep., NVIDIA Corporation.

CARPENTER, L. 1984. The A-buffer, an Antialiased Hidden Surface Method. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, ACM, vol. 18, 103–108.

ENDERTON, E., SINTORN, E., SHIRLEY, P., AND LUEBKE, D. 2010. Stochastic transparency. In *I3D '10: Proceedings of the 2010 Symposium on Interactive 3D Graphics and Games*, 157–164.

EVERITT, C. 2001. Interactive order-independent transparency. Tech. rep., NVIDIA Corporation, May.

JANSEN, J., AND BAVOIL, L. 2010. Fourier opacity mapping. In *I3D '10: Proceedings of the 2010 Symposium on Interactive 3D Graphics and Games*, 165–172.

JOUPPI, N. P., AND CHANG, C.-F. 1999. Z3: an economical hardware technique for high-quality antialiasing and transparency. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, ACM, New York, NY, USA, HWWS '99, 85–93.

KIM, T.-Y., AND NEUMANN, U. 2001. Opacity shadow maps. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, 177–182.

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

MAULE, M., COMBA, J. A., TORCHELSEN, R., AND BASTOS, R. 2013. Hybrid transparency. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '13, 103–118.

PORTER, T., AND DUFF, T. 1984. Compositing Digital Images. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, ACM, vol. 18, 253–259.

SALVI, M., VIDIMČE, K., LAURITZEN, A., AND LEFOHN, A. 2010. Adaptive volumetric shadow maps. In *Eurographics Symposium on Rendering*, 1289–1296.

SALVI, M., MONTGOMERY, J., AND LEFOHN, A. 2011. Adaptive transparency. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ACM, New York, NY, USA, HPG '11, 119–126.

SINTORN, E., AND ASSARSON, U. 2009. Hair self shadowing and transparency depth ordering using occupancy maps. In *I3D '09: Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, 67–74.

VASILAKIS, A., AND FUDOS, I. 2012. S-buffer: Sparsity-aware multi-fragment rendering. In *Eurographics (Short Papers)*, 101–104.

YANG, J. C., HENSLEY, J., GRÜN, H., AND THIBIEROZ, N. 2010. Real-time concurrent linked list construction on the gpu. In *Proceedings of the 21st Eurographics Conference on Rendering*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGSR'10, 1297–1304.