Adaptive Transparency

Marco Salvi[*] Jefferson Montgomery[†] Aaron Lefohn[‡]
Intel Corporation

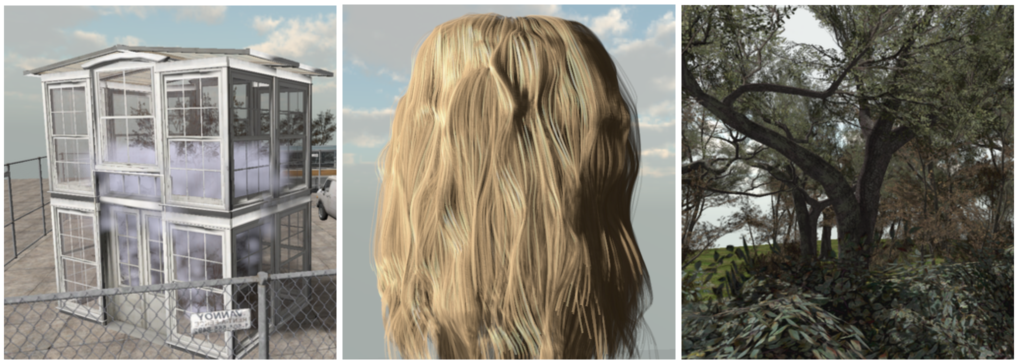

Figure 1: *Each image shows millions of transparent fragments correctly composited in real-time with adaptive transparency. The maximum per-pixel depth complexity in these scenes is 320 fragments (left), 671 fragments (middle), and 45 fragments (right); yet our technique accurately represents the visibility information using only 24, 16, and 8 samples (left to right). The scenes in the left and right image are derived from assets provided by Valve Software and Cem Yuksel provided the hair model.*

Abstract

Adaptive transparency is a new solution to order-independent transparency that closely approximates the ground-truth results obtained with A-buffer compositing but, like a Z-buffer, operates in bounded memory and exhibits consistent performance. The key contribution of our method is an adaptively compressed visibility representation that can be efficiently constructed and queried while rendering. The algorithm supports a wide range and combination of transparent geometry (e.g., foliage, windows, hair, and smoke). We demonstrate that adaptive transparency is five to forty times faster than real-time A-buffer implementations, closely matches the image quality, and is both higher quality and faster than other approximate order-independent transparency techniques: stochastic transparency, uniform opacity shadow maps, and Fourier opacity mapping.

Keywords: order independent transparency, visibility, compression

1 Introduction

Efficient and accurate support for order-independent transparency in feed-forward, rasterization rendering pipelines is a long-standing open problem. Properly rendering scenes containing transparent objects requires renderers to determine the amount of light absorbed by the objects between a given fragment and the viewer. Ray tracers naturally compute this quantity while tracing rays from the viewing position into the scene because ray intersections are ordered. However, rasterization pipelines may generate fragments out of order and must therefore take extra measures to correctly composite transparent fragments. Offline feed-forward renderers such as Reyes capture all fragments in an A-buffer, then sort and composite the fragments to generate the final image. Graphics hardware rasterization pipelines, limited by the memory and performance constraints of real-time rendering, support order-independent rendering only for opaque geometry. They support transparency using alpha compositing operations that are only correct when fragments are generated in front-to-back or back-to-front order. This requirement is often impossible to comply with for complex scenes with multiple types of overlapping transparent objects (see Figure 1). Recent research has recently introduced a number of real-time order-independent transparency techniques such as stochastic transparency, GPU A-buffer implementations, and others; however, they suffer from low image quality, highly variable performance and memory usage, or restrictions on the type of supported transparent objects.

Adaptive transparency (AT) supports order-independent rasterization of transparent primitives, but operates in bounded memory. We constrain memory usage by building a compressed representation of per-pixel visibility; however, compared to previous fixed-memory methods our adaptive algorithm works with a wide range of transparent primitives (glass, hair, foliage, smoke, etc.) and accurately represents per-pixel visibility functions without noise and with minimal compression artifacts.

The main contribution of our work is an adaptive visibility representation that minimizes the error introduced by compressing data and can be efficiently constructed and sampled while rendering. Our method requires changes to graphics architectures to operate in fixed memory, but we demonstrate a variable-memory prototype running on current GPUs. AT closely approximates the A-buffer-based true visibility solution, performs consistently regardless of fragment ordering, is up to forty times faster than GPU A-buffer solutions, and is performance-competitive with other approximate real-time transparency methods while delivering higher image quality.

[*]e-mail:marco.salvi@intel.com
[†]e-mail:jefferson.d.montgomery@intel.com
[‡]e-mail:aaron.lefohn@intel.com

2 Background and Related Work

There are a number of exact and approximate methods for handling transparency in a rasterizer, and this section gives a brief overview of the range of techniques available.

We begin by defining a transparent fragment f_i characterized by its distance from the viewer z_i and its color and transmittance $(c_i, 1 - \alpha_i)$. Given a visibility function $vis(z_i)$ that represents the total transmittance between f_i and the viewer, we can compute the contribution of f_i to the final pixel color as:

$$c_i \alpha_i vis(z_i). \tag{1}$$

with the contribution of multiple overlapping fragments given by:

$$\sum_{i=0}^{n} c_i \alpha_i vis(z_i) \tag{2}$$

While the transmittance and color of each fragment are known during fragment shading, computing $vis(z_i)$ is the focus of rasterization transparency algorithms, and this paper introduces a new technique for computing this function.

Many renderers solve Equation 2 recursively by rendering fragments in back-to-front order using the following compositing equations:

$$\begin{aligned} C_0 &= \alpha_0 c_0 \\ C_n &= \alpha_n c_n + (1 - \alpha_n) C_{n-1}. \end{aligned} \tag{3}$$

where C_n is the final pixel color. This solution, known as *alpha blending* in current graphics APIs, was first introduced by Porter and Duff [1984] and modern graphics processors include fixed-function hardware to accelerate these particular read-modify-write operations on C_i. A key benefit of this approach is that each fragment's visibility is computed recursively and $vis(z)$ is never actually stored. However, the requirement of fragment ordering is left to the user to ensure, and can be costly, complicated, or impossible (see Figure 2). Depth peeling [Everitt 2001] allows geometry to be submitted in any order but requires k complete rendering passes to capture k layers, and so is quite costly and has highly variable performance, although more recent implementations [Liu et al. 2009] can capture multiple layers in one pass. The A-buffer algorithm [Carpenter 1984] also solves transparency using Equation 3, but removes the restriction of generating the fragments in-order by storing all fragments in variable-length lists and sorting them after all geometry is rendered. The technique requires variable, unbounded memory and the sorting performance is sensitive to the number and order of fragments stored; but it can nonetheless give good real-time results on current graphics hardware [Yang et al. 2010]. The Z^3 algorithm [Jouppi and Chang 1999] is a bounded A-buffer implementation that can dynamically merge fragments in order to keep its per-pixel memory footprint constant, at the expense of image quality. Lastly, the F-Buffer [Mark and Proudfoot 2001] captures all rasterized fragments in a FIFO for multi-pass shading of all transparent fragments. The F-buffer and A-buffer have similar storage requirements yet to support order-independent transparency, the F-buffer requires a global sort to perform the final compositing instead of the A-buffer's per-pixel sorting.

A second class of algorithms explicitly computes the visibility function $vis(z)$ in a separate rendering pass. Specifically, the transmittance from z to the viewer is defined as the product of the transmittance of every transparent object that intersects the ray from the viewer to z:

$$vis(z) = \prod_{0<z_i<z} (1 - \alpha_i). \tag{4}$$

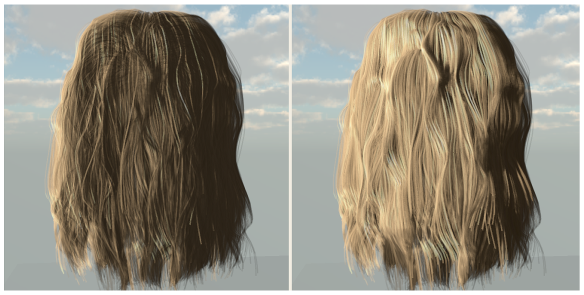

Figure 2: *Left: a complex hair object rendered with alpha-blending presents large artifacts. Right: the same object is correctly composited in real-time with our method (53ms, 18 million transparent fragments).*

For example, Patney et al. [2010] introduced a data-parallel solver for Equation 4 that uses segmented sort and scan operations to load-balance across the irregular number of fragments per pixel.

The visibility function, in conjunction with Equation 2, carries all of the fragments' depth ordering and transmittance information and thus suggests a general method for transparency:

1. Render fragments and build a representation of visibility.
2. Render fragments and composite them using Equation 2.

This refactoring of the transparency problem was first introduced by Sintorn et al. [2009], who represent $vis(z)$ in a bitfield where each bit represents a step function of constant height spaced at regular depths. The technique, occupancy maps, works well in scenes where all transparent geometry has the same transmittance, such as hair. Enderton et al. [2010] point out that any solution for shadows from transparent occluders is also a general transparency solution, and proposes stochastic transparency: a stochastic sampling extension to the alpha-to-coverage functionality in current graphics hardware that builds a stochastic representation of visibility. The solution works for any transparent geometry, but requires a large number of visibility samples to avoid noise artifacts (the amount of noise increases with depth complexity). Opacity shadow maps (OSM) [Kim and Neumann 2001] accumulate opacity on a uniformly discretized grid, thereby exhibiting banding artifacts in the visibility function. Deep opacity maps [Yuksel and Keyser 2008] extends OSMs by warping the planes by a per-pixel offset from the closest layer. Fourier opacity maps (FOM) [Jansen and Bavoil 2010] store $vis(z)$ in the Fourier basis, which works well for soft occluders such as smoke but is less accurate for sharp occluders such as hair or glass. Adaptive volumetric shadow maps (AVSM) [Salvi et al. 2010] store an adaptive, piecewise linear representation of $vis(z)$ optimized for casting shadows from volumetric media.

Adaptive transparency is an order-independent visibility algorithm designed for determining direct visibility of transparent objects, i.e., querying the visibility function from the same direction it was generated. We represent visibility with a step function basis to efficiently support a wide range of transparent object types and present analysis showing that the lossy compression scheme, which is inspired by the scheme used in adaptive volumetric shadow maps [Salvi et al. 2010], is especially well suited for order-independent transparency. We compare AT to other real-time order-independent transparency techniques: A-buffer, stochastic transparency, opacity shadow map transparency, and Fourier opacity map transparency.

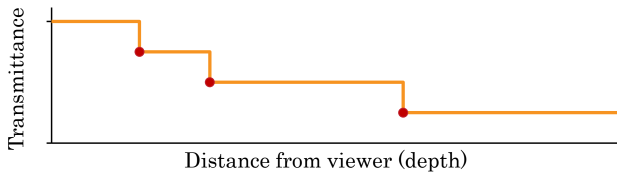

Figure 3: *Piece-wise constant representation of visibility. The red dots denote nodes containing the depth and transmittance values that parameterize the basis functions $H_{z,\alpha}$.*

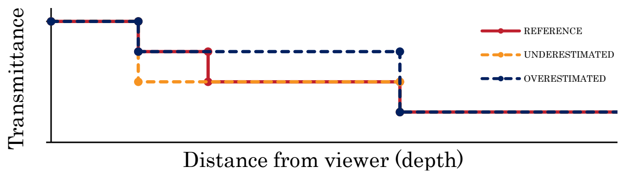

Figure 4: *When a node is removed, the result is an over-estimate (blue) of the true function (red). Alternatively, the previous basis can be modified to yield an under-estimate (orange).*

3 Algorithm Overview

Adaptive transparency is an order-independent transparency technique based on first building a representation of visibility as described in Section 2. The algorithm was designed by stressing the importance of accurately representing visibility while retaining some of the characteristics that have made the Z-buffer the *de facto* standard algorithm in real-time primary visibility determination: namely bounded memory requirements and consistent performance.

3.1 Basis Functions

When representing a complex function, it is common practice to expand it as a sum or product of simpler basis functions. While the choice of basis function can be arbitrary, it is important to adopt functions that match the features of the original signal so that as few terms as possible are required to represent it. In our case, the original visibility function for transparent surfaces is comprised of many instantaneous reductions in transmittance (see Figure 3), so we start by rewriting Equation 4 using the Heaviside step function:

$$vis(z) = \prod_{i=0}^{n} \Big(1 - \alpha_i H(z - z_i)\Big) \tag{5}$$

$$H(x) = \left\{ \begin{array}{ll} 0 & x \le 0 \\ 1 & x > 0 \end{array} \right.$$

Equation 5 can be trivially expanded out into an even cleaner expression for Equation 4 involving a new step function $H_{z,\alpha}(x)$:

$$vis(z) = \prod_{i=0}^{n} H_{z_i,\alpha_i}(z) \tag{6}$$

where $H_{z,\alpha}(x)$ is defined as:

$$H_{z,\alpha}(x) = \left\{ \begin{array}{ll} 1 & x \le z \\ 1-\alpha & x > z \end{array} \right.$$

Given the simplicity of Equation 6 it is straightforward to adopt $H_{z,\alpha}(x)$ as basis function and represent visibility as a sequence of depth and transmittance pairs $(z_i, 1 - \alpha_i)$ that we call *nodes* (see Figure 3).

3.2 Visibility Compression

In order to fit this representation into bounded memory, we assign a fixed memory budget of N nodes to each pixel. If more than N fragments contribute to a given pixel we compress our data by eliminating existing nodes. While methods like alpha-to-coverage or stochastic transparency [Enderton et al. 2010] use simpler schemes to eliminate existing nodes, adaptive transparency selects an optimal candidate based on minimizing change to the current visibility function.

When a node is inserted that does not fit in budget, we remove the node that generates the smallest error to the integration over the visibility function which, due to the step-nature of our basis function, is a simple comparison of rectangle areas. If nodes are ordered in depth, such that node $i-1$ is the next closest node from node i, then we remove node j given by Equation 7. Note that the zeroth node (which represents the transmittance before the first fragment, i.e., 1) is implicit and consumes no memory nor can be removed.

$$\operatorname*{argmin}_j \{(vis(z_{j-1}) - vis(z_j))(z_j - z_{j-1})\} \tag{7}$$

There are two alternative ways of eliminating a node. As shown in Figure 4, if the node is simply removed from the representation, the resulting compressed visibility function overestimates the uncompressed transmittance. Alternatively, when the node is removed the previous node's transmittance can also be modified such that the compressed representation is an underestimate.

Underestimation generally produces less objectionable artifacts, as shown in Figure 5. This is mainly due to the fact that, should a subsequent fragment fall within the compressed depth range, overestimated transmittance makes it more visible while underestimated transmittance de-emphasizes the fragment's contribution which is less jarring to observers. Figure 5 also shows an implementation where the node removal technique is selected on a node-by-node basis, based on the compression metric shown in Equation 7. While this further minimizes error in the represented visibility function, it can introduce objectionable high-frequency artifacts.

We've found that the best image quality is obtained by always removing the node using the underestimating technique. Another important side-effect of this, is that the represented visibility at infinity (the total transmittance of all fragments) is always correct, even if the furthest node is removed. As will be shown in Section 4 this property is very important for performance and image quality.

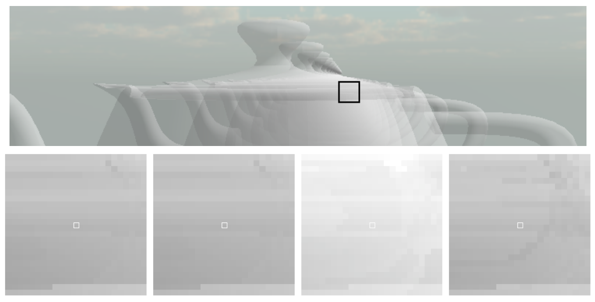

Figure 5: *Bottom row, left to right: results from an uncompressed visibility function and those from a compressed function using the underestimate technique, the overestimate technique, and selecting the technique on a node-by-node basis.*

3.3 Efficient Visibility Representation

Computing the compression metric requires the nodes to be sorted in depth, but sorting the representation as part of the compression operation is prohibitively expensive. We therefore maintain the list of nodes discussed in Section 3.1 in sorted (front-to-back) order by inserting new nodes in the appropriate location.

Further, we note that both the compression metric and the final composite (see Equation 2) require the evaluation of the visibility function at each node. Instead of re-computing these values, we take advantage of node ordering and pre-compute all partial products in Equation 6, storing instead $(z_i, \prod_{m=0}^{i}(1-\alpha_m))$ in each node rather than $(z_i, 1-\alpha_i)$. This operation can be efficiently carried out every time we insert a new fragment by simply multiplying the current visibility curve with the step function representing the new fragment.

3.4 Compression Bias

To understand how the probability of a node to be removed is spatially distributed we must analyze the rate of variation of the compression metric with respect to the viewing distance. Since our metric corresponds to variations of the area under the visibility function, our question translates to computing the second derivative of the area formed by a set of (z_i, α_i) with $0 \leq \alpha_i < 1$:

$$\begin{aligned}\frac{\partial^2}{\partial x^2}\int_0^x vis(t)dt &= \frac{\partial}{\partial x}\left\{\prod_{i=0}^{n} H_{z_i,\alpha_i}(x)\right\}\\ &= \frac{\partial}{\partial x}\left\{\exp\left[\sum_{i=0}^{n} ln\Big(H_{z_i,\alpha_i}(x)\Big)\right]\right\}\\ &= vis(x)\left[\sum_{i=0}^{n}\left(-\frac{\alpha_i\delta(x-z_i)}{H_{z_i,\alpha_i}(x)}\right)\right] \qquad (8)\end{aligned}$$

with $\delta(x)$ defined as Dirac delta function:

$$\delta(x) = \left\{\begin{array}{ll} 0 & x \neq 0 \\ \infty & x = 0 \end{array}\right. \qquad \int_{-\infty}^{\infty}\delta(x)dx = 1$$

The term in square brackets makes Expression 8 zero for all $x \neq z_i$ and negative in all remaining cases. Since $vis(z)$ is a monotonically decreasing function as we move further away from the viewer the non-zero terms approach zero and consequentially the rate of variation of our compression metric reduces.

Put differently, $vis(z)$ is given by the product of functions bounded in the interval $(0, 1]$ (see Equation 6), as we add more fragments to it nodes located in the tail of the curve will generate smaller area variations, which makes them more likely candidates for removal. Therefore our compression metric is naturally biased towards preserving details that are closer to the viewer. This is a good property of our metric, because as shown by Equation 2, fragments located in the tail of $vis(z)$ are more likely to give little contribution to the final pixel color, hence it is advantageous to have a more accurate reconstruction of the head of the curve.

4 Implementation

The adaptive transparency algorithm fits into a rasterization rendering pipeline as a new framebuffer operation, e.g., an alternative to or an extension of the depth test. This can be implemented as either a fixed-function capability in the framebuffer operation stage or in the pixel shading stage if pixel shaders support read-modify-write operations on the framebuffer protected by critical sections. Regardless of whether the renderer implements the algorithm in the pixel shader or framebuffer operation stage, users render unsorted transparent geometry using the two-pass technique described in Section 2 which first captures the visibility representation and then uses it to composite final color.

Other approximate transparency techniques, such as occupancy maps [Sintorn and Assarson 2009] or stochastic transparency [Enderton et al. 2010] have noted the importance to image quality in correctly representing the total transmittance and this value is also needed to composite opaque fragments that may have been rendered separately. However those algorithms require an extra pass over transparent geometry to compute the correct value, which is then used as a correction factor when compositing the color buffer. However, as discussed in Section 3.2, the compression used by adaptive transparency never modifies this value, maintaining the exact total transmittance, so no extra passes are required nor corrections to the compositing equation.

While it is possible to render both opaque and transparent geometry using AT as the visibility function, it is more efficient to first render the opaque geometry using a traditional Z-buffer. The transparency rendering passes can then reject fragments occluded by opaque geometry by reading from this Z-buffer. The resulting algorithm for opaque and transparent geometry proceeds as follows:

1. Render opaque geometry into color buffer A
2. Render transparent geometry to build per-pixel visibility function, $vis(z)$
3. Render transparent geometry, determine visibility by sampling $vis(z)$ and composite into color buffer B
4. Generate final image by combining color buffers A and B

We use the above approach to implement the fixed-memory implementations of opacity shadow map transparency (OSMTF) and Fourier opacity map transparency (OSMTF), by building $vis(z)$ on a uniform grid (OSMTF) or with Fourier basis functions (FOMTF). We also add a rendering pass over transparent geometry to compute the minimum and maximum per-pixel depth bounds in order to give the best possible results for these techniques.

We implement depth-based stochastic transparency with alpha correction as described in [Enderton et al. 2010], which requires the four computation passes described above plus an additional pass to compute the correct per-sample total transmittance. $vis(z)$ is represented in a multi-sample depth buffer where each sample indicates a constant change in transmittance. Each fragment writes its depth into a subset of samples specified by a stochastically-generated stipple pattern. All possible stipple patterns are pre-generated and stored in a look-up table in a random order, then one is selected based on both the fragment's transmittance as well as the following pseudo-random indexing, which gives qualitatively similar results to [Enderton et al. 2010]: $PrimitiveId + Position.x^2 + Position.y$. The one place our implementation differs from [Enderton et al. 2010] is when using more stochastic samples than the optimally-supported MSAA rate: we sample into multiple buffers during a single pass, whereas [Enderton et al. 2010] perform multiple passes with different random seeds and average the results into the final render.

4.1 Prototype on Current GPUs

It is not possible to implement the adaptive transparency algorithm as described above on current DirectX11-class GPUs because multiple fragments that map to the same pixel can be concurrently shaded causing data races. In fact pixel shaders do not support critical sections nor can AT be implemented with the existing fixed-function framebuffer operations, unlike OSMTF and FOMTF that only require additive blending.

To guarantee no data races will occur we can constrain the GPU to shade at most one fragment per pixel by operating via a fullscreen pass. The prototype DirectX11 adaptive transparency implementation involves the following rendering passes:

1. Render transparent geometry and capture depth, color, transmittance and coverage mask of all fragments in per-pixel linked lists (variable length data structure)

2. Perform a per-pixel (fullscreen) computation pass that traverses the per-pixel linked lists twice to:

    (a) Generate adaptive transparency compressed visibility curve, $vis(z)$

    (b) Determine visibility of each fragment by sampling $vis(z)$ and composite into color buffer

This prototype does not operate in bounded memory, but it does result in a real-time transparency algorithm that we use to compare image quality with other techniques: A-buffer, stochastic transparency, uniform opacity map transparency, and Fourier opacity map transparency.

For completeness, we implement variable-memory implementations of opacity shadow map transparency (OSMT) and Fourier opacity map transparency (FOMT) using a similar approach, adding an additional pass over the linked lists to compute the per-pixel depth bounds before we build $vis(z)$.

4.2 Spatial Anti-Aliasing

To support MSAA for the fixed-memory transparency algorithms, we render to multi-sampled render targets and shade/sample depth bounds and visibility data textures at sample frequency. For stochastic transparency, we implement MSAA by accumulating total transmittance and applying the alpha-correction at sample frequency which gives good results despite the visibility function still being represented at pixel frequency.

For the variable-memory transparency implementations, we support multi-sampled anti-aliasing (MSAA) with up to 4 samples per pixel on all the implemented methods. For the reference solution, AT and variable memory OSMT/FOMT we adopt the A-buffer like method introduced by Yang et al. [2010] by storing a coverage mask in the least significant four bits of fragment depth. In the full-screen resolve pass we shade at sample frequency and only process those fragments that cover the current sample (as indicated by the fragment's stored coverage mask).

5 Results

We evaluate image quality, memory and bandwidth requirements, and performance of the transparency methods described in the previous section using the GPU-based A-Buffer implementation recently introduced by Yang et al. [2010] as a reference solution. We select scenes representative of modern real-time graphics applications but where sorting draw calls on the CPU(s) is not practical, and so exhibit objectionable artifacts when rendered with alpha blending. All results are gathered at a resolution of 1280×720 on an ATI Radeon HD 5870 GPU with an Intel Core 2 quad-core CPU running at 2.4 GHz running Windows 7 (64-bit).

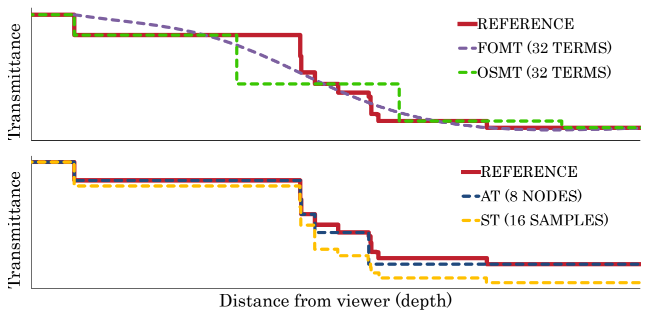

Figure 8: *Comparison of visibility representations used by different transparency algorithms. The AT visibility representation adheres most closely to the correct function, despite using fewer samples.*

5.1 Visual Quality Analysis

We qualitatively compare the baseline image rendered against five transparency methods: adaptive transparency (AT), stochastic transparency (ST), Fourier opacity map transparency (FOMT), opacity shadow map transparency (OSMT), and traditional alpha blending in multiple scenes in Figure 6. For AT, ST, OSMT, and FOMT we also show difference images from the baseline, which for clarity reasons have been enhanced by a factor of eight.

We further confirm the accuracy of our method by inspecting a number of visibility functions. We find that AT produces curves that are much closer to the ground truth result than any other approximate technique (see Figure 8), with negligible sensitivity to primitives ordering when using more than ˜8 nodes per pixel.

5.1.1 SMOKE Scene

The SMOKE scene contains over 10 million transparent fragments generated by many overlapping and intersecting objects such as smoke, foliage, windows, wire fences, and vehicles. As shown in the top rows of Figure 6, this scene is particularly problematic for alpha-blending which composites scene elements in the wrong order (see smoke rendered in front of glass windows and wire fence). While the particles could be sorted before rendering, this would only fix compositing errors within the smoke effect, leaving the remaining rendering artifacts unaddressed. The scene also generates very complex visibility functions (involving, in some cases, more than 300 fragments per pixel) which both the opacity shadow map (OSM) and Fourier opacity map (FOM) methods fail to sufficiently encode over a large depth range. This behavior is not unexpected as both algorithms use non-adaptive approximations, moreover FOM trigonometric basis functions are a poor choice for thin light blockers and cause major artifacts due to ringing issues (see heavily darkened smoke in Figure 6 - FOM case).

5.1.2 HAIR Scene

Correctly rendering and compositing hair and fur is a notoriously difficult problem to solve in real-time. Applications such as games often implement fragile approximations, or avoid the problem by using hats or hairless characters. Therefore we test our algorithms with a complex hair mesh that generates 15 million transparent fragments per frame with a maximum depth complexity of over 600 layers per pixel (shown in the middle rows of Figure 6). Adaptive transparency produces stable results without temporal artifacts and it is virtually indistinguishable from the baseline. Stochastic transparency exhibits noticable noise artifacts, even with 24 samples per

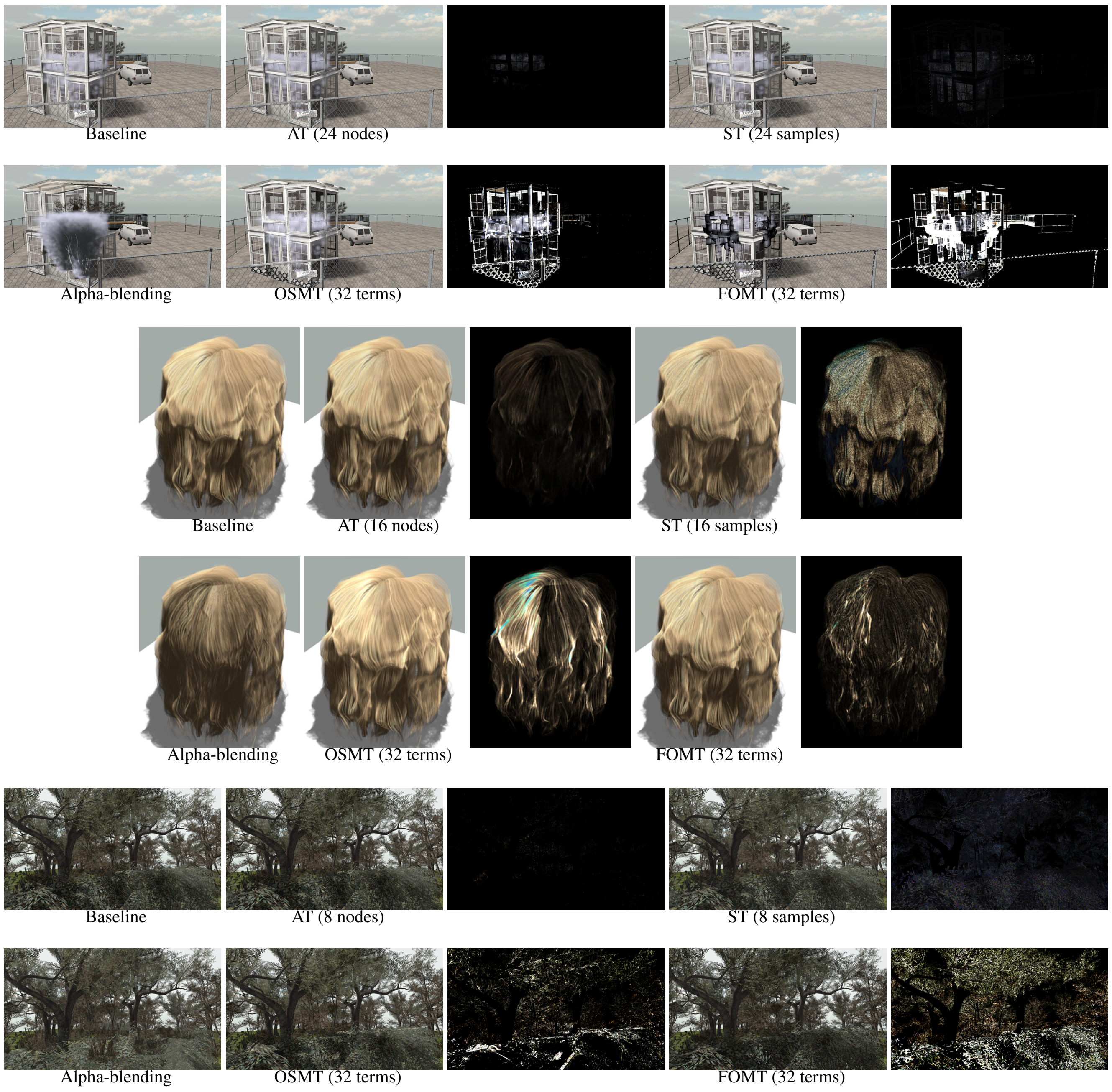

Figure 6: *Several scenes comparing adaptive transparency (AT) to A-buffer (baseline), stochastic transparency (ST), opacity shadow map transparency (OSMT), Fourier opacity map transparency (FOMT), and traditional alpha blending. The difference images are enhanced by a factor of eight. The SMOKE (top) and FOREST (bottom) scenes are derived from assets provided by Valve Software; the HAIR scene (middle) is courtesy of Cem Yuksel.*

Scene	Max Depth Complexity	Fragment Count	AT/OSMTV/FOMTV	OSMTF/FOMTF	ST
SMOKE	312	10.6 M	125.4 MBytes	63.3 MBytes	93.8 MBytes
HAIR	663	15.0 M	174.6 MBytes	63.3 MBytes	65.0 MBytes
FOREST	45	6.0 M	72.6 MBytes	63.3 MBytes	36.9 MBytes
POWER PLANT	35 (60)	2.69 (3.47) M	30.8 (39.7) MBytes	63.3 (253.1) MBytes	65.0 (91.4) MBytes

Table 1: *Scene statistics and memory requirements for each method. Please note that all our variable storage methods allocate the same amount of graphics memory, including AT and the iterative and recursive reference solutions. The number of samples used by each method is specified in Figure 6. Results between parenthesis obtained with 4X multi-sampling.*

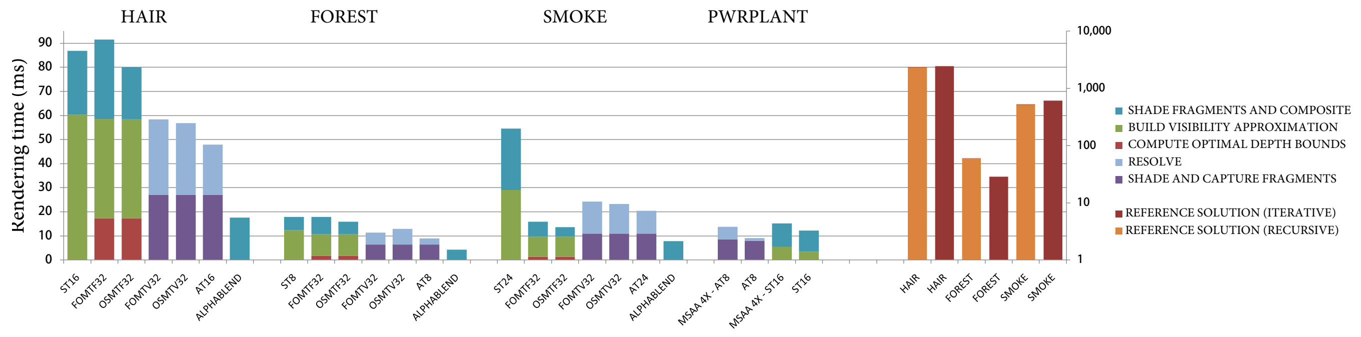

Figure 7: *This histogram shows each transparency method performance in the HAIR, FOREST and SMOKE scenes. Performance for 4X MSAA is tested only for the PWRPLANT scene for AT and ST. Given the large performance gap between the different methods we plot the reference solutions again the right vertical axis that uses a log scale. All other methods rendering times are plotted against the left axis.*

pixel, due to the random nature of the visibility function approximation. In contrast, AT computes and retains the most significant nodes of the visibilty curve. Opacity shadow map transparency and Fourier opacity map transparency both render images that deviate substantially from the baseline and have some temporal artifacts. Their artifacts, however, are not as severe as in the SMOKE scene because the HAIR scene exhibits a more compact depth range.

5.1.3 FOREST Scene

Real-time applications often render finely-detailed foliage using opaque geometry as a proxy with textures that encode the anti-aliased silhouettes of branches and leaves. In particular, techniques such as alpha testing and alpha-to-coverage convert texture alpha into varying coverage and can be used to work around compositing issues. However, high multi-sampling rates may be necessary to avoid spatial and temporal artifacts. These same assets can be applied to AT, which enables the rendering of high-quality and alias-free foliage or similar objects like wire fences or metallic grids. Instead of employing alpha test or alpha-to-coverage, we simply render and composite transparent fragments from pre-anti-aliased proxy geometry based on the opacity values stored in textures. AT and ST can be used in this way to render perfectly composited and anti-aliased foliage at high speed, as shown in the bottom rows of Figure 6 and 7. While the FOREST scene exhibits much lower depth complexity than the HAIR and SMOKE scenes (as detailed Table 1) OSMT and FOMT generate only moderate improvements over an alpha-blending approach.

5.2 Memory Storage Requirements and Spatial Anti-Aliasing

The storage requirements for fixed memory transparency implementations are driven by frame buffer resolution and multi-sampling rates, and Table 1 illustrates graphics memory requirements for each scene. These techniques can support different flavors of multi-sampling anti-aliasing. As mentioned in Section 4.2 OSMTF and FOMTF use a brute force approach, while our ST implementation computes the visibility function at pixel frequency, with both methods giving excellent visual results. On the other hand, the per-pixel visibility used by ST uses less memory and is more flexible as it decouples visibility function and color buffer rate which can then be scaled independently.

Variable memory implementations require an amount of memory proportional to the number of visible transparent fragments. This unbounded memory characteristic is certainly an undesirable feature, but the techniques provide other benefits and can be useful when known to be applied to a limited amount of transparent geometry. Moreover, memory requirements of variable memory algorithms scale well with MSAA because, as discussed in Section 4.2, storing coverage masks does not affect the per-pixel list node size, but it does increase the number of pixel shader invocations and therefore transparent fragments. This increase is typically in the range of 10% to 50%, and often translates to a more than 50% impact on performance in the AT resolve pass (as also shown in Figure 7). Please note that MSAA gives excellent results with the variable memory implementation of adaptive transparency (Figure 9).

5.3 Performance of Variable-Memory GPU Prototype

While the fixed memory implementation of adaptive transparency requires changes to GPU hardware (as discussed in Section 4), we measured the performance of our real-time prototype of AT and other transparency methods on a current DirectX11-class GPU. These performance results are not directly predictive of the fixed-memory algorithm's performance, but they do demonstrate that AT can deliver consistent performance that is up to forty times faster than sort-based methods. In particular $vis(z)$ is generated and consumed within in a single fullscreen pass and therefore it can be stored close to the arithmetic units in on-chip memory like registers, caches, or scratchpad memory (see Section 4.1). We also note that because $vis(z)$ is not explicitly stored in graphics memory, changing the number of nodes, Fourier terms, etc. does not require changing GPU buffer allocations—only using more registers.

For comparisons we use two GPU A-buffer implementations as reference solutions based on Equations 2 and 3 (iterative and recursive compositing). For the SMOKE scene, the A-buffer solution takes over 600 milliseconds. AT and ST closely approximate the image while respectively being thirty and ten times faster (see Figure 7). Fixed and variable memory implementations of OSMT and FOMT are also much faster than the reference solution, but their rendered images contain numerous objectionable artifacts. The HAIR scene takes 2300 milliseconds to render using an A-buffer, ST is twenty times faster but generates substantial noise artifacts, and AT is over forty times faster and generates an image nearly indistinguishable from the A-buffer image. For the FOREST scene, AT is seven times faster than the A-buffer solution and only half the speed of traditional alpha blending (less than 10 milliseconds).

5.4 Bandwidth Requirements

To qualitatively understand the memory bandwidth requirements of the implemented methods we develop a very simple memory model for each algorithm. This simplified model is a worst-case analysis

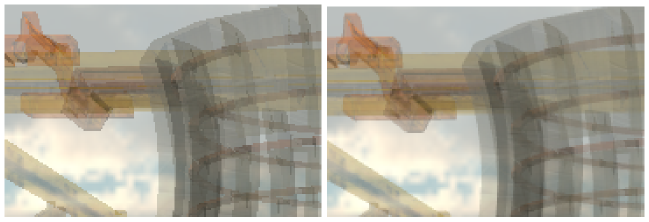

Figure 9: *Detail of the PWRPLANT scene rendered with 8-node AT without (left) and with (right) 4× multi-sampled anti-aliasing.*

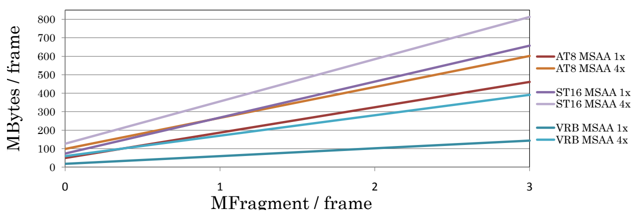

Figure 10: *Per-frame memory bandwidth as function of the number of composited transparent fragments and image resolution (720p).*

that does not take into account key bandwidth saving mechanisms employed by GPUs such as caches, color and depth compression.

The plot in Figure 10 compares three techniques with and without multi-sampling. VRB identifies three variable memory algorithms (AT, FOMTV and OSMTV) which share the same memory bandwidth profile. ST16 models stochastic transparency with 16 samples per pixel. AT8 identifies a fixed memory implementation of adaptive transparency (8 nodes) via read-modify-write operations from pixel shaders, that might be available on future graphics hardware architectures. To model multi-sampling bandwidth requirements we assume a model similar to our stochastic transparency implementation with a single visibility function per pixel, irrespective of the multi-sampling rate. Please note that we don't use the same number of samples for adaptive transparency and stochastic transparency because the former often exhibits better image quality than the latter even with 50% less samples.

First we note that variable memory implementations consume less memory bandwidth than any other method. This is not surprising, because as described in Section 5.3 variable memory implementations keep the visibility functions close to the arithmetic units at all times, which saves a considerable amount of bandwidth. This indicates that variable memory methods could still be a good choice for simple applications that don't generate many transparent fragments running on low bandwidth graphics hardware.

Secondly we note that within the limits of our simple model a fixed memory implementation of adaptive transparency clearly has a bandwidth per-image quality advantage over stochastic transparency. On the other hand we are not taking in consideration multi-sampling compression that might give ST an edge over AT. This result also emphasizes the importance of new hardware compression schemes if adaptive transparency is implemented as a fixed-function framebuffer operation.

6 Conclusions and Future Work

We have demonstrated that adaptive transparency renders images at higher quality and performance than previous fixed-memory order-independent solutions. AT image quality is nearly indistinguishable from exact transparency methods, while often providing more than an order of magnitude better performance, all independent of the ordering of the transparent fragments.

The key to AT's speed and robustness is a new compact and fixed length visibility representation for visibility functions that are rendered and sampled from the eye. Moreover our representation is simple to update and query.

Adaptive transparency requires changes to future graphics hardware in order to run in fixed memory and points toward investigating architectural changes that enable rendering AT's visibility functions and other user-defined data structures in bounded memory, such as locks in pixel shaders or new framebuffer operations and formats.

Acknowledgements

We thank Craig Kolb, Matt Pharr, Charles Lingle, the rest of the Advanced Rendering Technologies (ART) team, and Elliot Garbus at Intel Corporation for supporting this research and for their valuable feedback. We also thank Kiril Vidimče at Intel for writing the hair rendering code, Jason Mitchell and Wade Schinn at Valve Software for the assets from Left-for-Dead-2, and Cem Yuksel from Texas A&M University for the hair model.

References

CARPENTER, L. 1984. The a-buffer, an antialiased hidden surface method. *SIGGRAPH Comput. Graph. 18*, 3, 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 *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, 51–57.

MARK, W. R., AND PROUDFOOT, K. 2001. The F-buffer: A rasterization-order FIFO buffer for multi-pass rendering. In *Graphics Hardware 2001, pages =*.

PATNEY, A., TZENG, S., AND OWENS, J. D. 2010. Fragment-parallel composite and filter. *Computer Graphics Forum (Proceedings of EGSR 2010) 29*, 4 (June), 1251–1258.

PORTER, T., AND DUFF, T. 1984. Compositing digital images. *SIGGRAPH Comput. Graph. 18*, 3, 253–259.

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

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.

YANG, J., HENSLEY, J., GRÜN, H., AND THIBIEROZ, N. 2010. Real-time concurrent linked list construction on the gpu. In *Rendering Techniques 2010: Eurographics Symposium on Rendering*, Eurographics, vol. 29.

YUKSEL, C., AND KEYSER, J. 2008. Deep opacity maps. *Computer Graphics Forum 27*, 2 (Apr.), 675–680.