High Performance Graphics (2016)
Ulf Assarsson and Warren Hunt (Editors)

Exploring and Expanding the Continuum of OIT Algorithms

Chris Wyman†

NVIDIA Research, Redmond, WA, USA

Abstract

Order independent transparency (OIT) proves challenging for modern rasterization-based renderers. Rendering without transparency can limit the quality of visual effects, so researchers have proposed various algorithms enabling and approximating OIT. Unfortunately, this work generally has restrictions limiting its applicability.

To identify directions for improvement, we performed an in-depth categorization of existing transparency techniques and placed them on a multi-dimensional continuum. This categorization shows how prior published techniques relate to each other and highlights unexposed parts of the continuum where further research may prove beneficial. We also describe two new OIT algorithms that occupy previously unexplored regions of this continuum.

These novel algorithms include stochastic layered alpha blending *(SLAB), which provides a parameter to explicitly transition along the continuum between stochastic transparency and k-buffers, and* multi-layered coverage blending *(MLCB), which explicitly decorrelates visibility and opacity in multi-layered alpha blending.*

1. Introduction

Transparency involves partial occlusion of one surface by another, leading to a final pixel color combining contributions from two or more surfaces. The opacity of fragment i is often represented by an alpha value α_i [Gla15], and its transparency is $1-\alpha_i$. Porter and Duff [PD84] introduced a standard set of compositing operators that allow various combinations of transparent fragments. Commonly, Porter and Duff's *over* operator is used:

$$c_{result} = \alpha_0 c_0 + (1-\alpha_0)\alpha_1 c_1. \quad (1)$$

But most compositing operators require order-dependent processing in a consistent order, usually back-to-front, to achieve the desired result. Typically, renderers either sort geometry in advance (e.g., [GHLM05]) and render using the painter's algorithm or sort on a per-pixel basis using an A-buffer [Car84].

Both lack appeal for interactive applications. The painter's algorithm fails with complex models and requires an expensive per-frame geometry sort prior to rendering. A-buffers store lists of fragments affecting each pixel, consuming a significant, variable, and unbound amount of memory and requiring per-pixel sorts.

These issues have spurred research on order-independent transparency (OIT) techniques with deterministic memory consumption and computational costs. Order-independent algorithms consume arbitrarily-ordered primitives yet give identical results for all orderings. While designed for rasterizers, OIT algorithms also apply to ray tracing when acceleration structures, like bounding volume hierarchies, do not guarantee ordered ray intersections.

But current OIT techniques still have undesirable properties (see [And15]). Developers would prefer they better fit in existing pipelines and easily coupled with other algorithms, e.g., for antialiasing, depth of field, volumetric media, and deferred shading. Existing techniques often require separating opaque and transparent geometry into separate passes and can require (partial) sorting of transparent geometry. Essentially, transparency must be handled as a special case rather than just adding another triangle to render.

While exploring potential new approaches for order-independent transparency to address these issues, we revisited the literature and realized prior techniques have much more in common than usually recognized. During categorization, a number of holes in the algorithmic continuum became obvious. This paper describes our categorization, outlines some non-obvious ways prior techniques are related, and presents two new OIT algorithms with different tradeoffs than prior work. This paper's explicit contributions include:

- outlining a multi-dimensional space of order independent transparency techniques,
- exploring new regions of this space, introducing two novel order-independent transparency algorithms,
- introducing *stochastic layered alpha blending* (SLAB), which provides a parameter that continuously transitions from stochastic transparency [ESSL10] to hybrid transparency [MCTB13],

† E-mail: chris.wyman@acm.org

DOI: 10.2312/hpg.20161187

- and introducing *multi-layered coverage blending* (MLCB), which lies between multi-layered alpha blending [SV14] and stochastic transparency [ESSL10] on our continuum.

Note we *do not* claim to provide best practices for OIT; developers are generally unhappy with all existing algorithms, and we aim improve understanding of the space hoping to provide insights helpful for designing new approaches.

On applicability of our new algorithms, SLAB's continuous transition from hybid to stochastic transparency could empower faster predictive renderers that use biased k-buffering methods for speed then dial back to consistent and unbiased results for final renders. MLCB potentially provides additional insight into combining antialiasing and transparency in a single algorithm.

2. A Continuum for Order-Independent Transparency

Given that GPU-accelerated A-buffers (e.g, [YHGT10]) and A-buffer variants (e.g. [MP01, Wit01]) have unbound per-pixel memory consumption, the first question for a new OIT algorithm is how to *restrict memory usage* to a deterministic amount. There are three popular approaches: limiting the number of fragments stored per pixel (e.g., k-buffers [BCL*07]), fitting fragments to per-pixel transmission functions (e.g., deep shadow maps [LV00]), and trading storage for more rendering passes (e.g., depth peeling [Eve01]). Since unknown numbers of rendering passes and unbounded memory are equally unappealing, few renderers perform full depth peeling except as an incremental implementation of A-buffering.

Given limited storage, we need to somehow incorporate an incoming fragment into an existing pixel representation. This *insertion heuristic* varies between OIT techniques. Multi-layered depth peeling approaches [LHLW09, MB07] conditionally insert a fragment if it is one of the closest k fragments yet seen. Z^3 [JC99] and k-buffer variants (e.g. [BCL*07, SML11]) always account for each fragment, but merge less important fragments together to maintain a fixed memory footprint. Stochastic transparency [ESSL10] probabilistically inserts fragments into its stochastic pixel representation.

For algorithms that merge fragments, we need an appropriate *merging heuristic*. Possibilities include merging the furthest two fragments [JC99, SV14], identifying a combination with the smallest error [SML11], and fitting functional representations of the transmission [JB10, LV00, MB13]. To combine fragments, various operators can be used, including the standard *over* operator and approximate order-independent operators, like a weighted sum [Mes07] or weighted average [BM08].

Algorithms that discard fragments need to consider a similar question, whether to *normalize contributions* to account for discarded fragments. In stochastic transparency [ESSL10], adding a depth and alpha normalization reduces variance. Typically normalization terms use order-independent operators like weighted sum or weighted average to account for discarded fragments. Of the papers we reviewed, only stochastic transparency clearly discusses such a "normalization" or "correction factor." However, hybrid transparency [MCTB13] and McGuire et al.'s phenomenological models [MB13, MM16] perform mathematically equivalent operations.

A final consideration is if compositing uses a continuous alpha or discrete coverage mask to represent transparency. Usually, the decision between *alpha or coverage* defaults to alpha compositing. But screen-door transparency [FGH*85] via alpha-to-coverage [MGvW98] and stochastic transparency [ESSL10] use coverage masks, which better integrate with multisampling to represent transparency from partial occlusion (instead of translucency).

Symbol	Meaning
f_i	Fragment i in the current pixel
z_i	Fragment i's depth
c_i	Fragment i's color (not pre-multiplied with alpha)
α_i	Fragment i's alpha
m_i	Fragment i's coverage mask
j	Indices for sorted fragments, i.e., $z_{j-1} < z_j < z_{j+1}$
n	# of fragments in a pixel, i.e., $i, j \in [0 \dots n-1]$
b	# of bits in a coverage mask
$\|\|m_i\|\|_1$	ℓ_1-norm, i.e., # of bits set in a mask (equiv to $\mathbb{b}_i$)
$\mathbb{b}_i$	# of bits set in a mask (i.e., $bitCount(m_i)$ or $\|\|m_i\|\|_1$)
$\overline{\mathbb{b}_i}$	# of bits not set in a mask (i.e., $b - \mathbb{b}_i$)
$\mid$	Bitwise binary OR
&	Bitwise binary AND
$\sim$	Bitwise binary negation
$\bigvee_{s=0}^{t} m_i$	Bitwise OR of a sequence: $m_0 \mid m_1 \mid \dots \mid m_t$
$\lfloor x \rfloor$, $\lceil x \rceil$	Floor and ceiling of x

Table 2: *A summary of symbols and mathematical notation.*

To summarize, current OIT techniques vary along these axes:

- how they *restrict memory usage* per-pixel,
- the *insertion heuristic* selecting fragments that affect pixel color,
- the *merge heuristic* controlling which fragments to combine if limited resources are oversubscribed,
- how to *normalize contributions* if discarding fragments,
- and whether to use *alpha or coverage* to represent transparency.

2.1. Derived Properties of OIT Techniques

While not explicitly chosen by developers, the parameters described above affect if an algorithm has *strict order independence*, is *unbiased*, or *consistently converges* as the number of layers and samples increases.

Algorithms that merge fragments reintroduce *weak order dependence* if their merge uses order-dependent operators. Some researchers define weak order dependence to mean that algorithms provide stable results if geometry order remains consistent between frames. This often suffices in games, rather than requiring strict order independence.

Todays only consistent and unbiased algorithms with strict order independence are simple variants of stochastic transparency [ESSL10]. Though, unlike in predictive rendering, many interactive applications need not produce unbiased or even consistent results. All modern algorithms, except empirical and phenomenological models, converge to A-buffering if given infinite resources.

3. Categorizing Prior OIT Work

To understand the space of real-time OIT approximations, we categorize existing techniques according to which choices they make on

Algorithm	Memory Limit	Insertion Heuristic	Merge Heuristic	Normalize?	Use Alpha or Coverage?
A-buffer [Car84]	none	always	no merging	no	either†
Alpha Testing	1 layer	if $\alpha >$ thresh	discard furthest	no	alpha
Alpha Compositing [PD84]	1 layer	always	*over* operator	no	alpha
Screen-Door Transparency [FGH*85]	k z-samples	always	z-test, discard occluded	no	coverage
Z^3 [JC99]	k layers	always	merge w/closest depths	no	alpha
Deep Shadow Maps [LV00]	k line segments	always	merge w/smallest error	no	alpha
Depth Peeling [Eve01]	1 layer	if closest	discard furthest	no	either†
Opacity Shadow Maps [KN01]	k bins	always	α-weighted sum	no	alpha
Density Clustering [MKBVR04]	k bins	always	k-means clustering	no	alpha
k-Buffers [BCL*07]	k layers	always	merge closest to camera	no	alpha
Sort-Independent Alpha Blending [Mes07]	1 layer	always	weighted sum	no	alpha
Deep Opacity Maps [YK08]	k bins	always	α-weighted sum	no	alpha
Multi-Layer Depth Peeling [LHLW09]	k layers	if in k closest	discard furthest	no	either†
Occupancy Maps [SA09]	k bins	always	discard if bin occupied	renormalize alpha	alpha
Stochastic Transparency [ESSL10]	k samples	stochastic	z-test, discard occluded	α-weighted average	coverage
Fourier Opacity Maps [JB10]	k Fourier coefs	always	sum in Fourier domain	no	alpha
Adaptive Volumetric Shadow Maps [SVLL10]	k layers	always	merge w/smallest error	no	alpha
Transmittance Function Maps [DGMF11]	k DCT coefs	always	sum in DCT basis	no	alpha
Adaptive Transparency [SML11]	k layers	always	merge w/smallest error	no	alpha
Hybrid Transparency [MCTB13]	k layers	always	discard furthest	α-weighted average	alpha
Weighted Blended OIT [MB13]	empirical func	never	discard all	α-weighted average	alpha
Multi-Layer Alpha Blending [SV14]	k layers	always	merge furthest	no	alpha
Phenomenological OIT [MM16]	empirical func	never	discard all	α-weighted average	alpha
(NEW) Stochastic Layered Alpha Blending	k layers	stochastic	discard furthest	α-weighted average	either‡
(NEW) Multi-Layer Coverage Blending	k layers	always	merge furthest	no	coverage

† *Data structure does not rely on storing opacity, so developers can use either.* ‡ *Algorithm has different variants that either use alpha or coverage.*

Table 1: *Categorizing transparency algorithms based on their choices along the axes of our continuum, as described in Section 2.*

the axes described in Section 2. We summarize this categorization in Table 1, with further details provided below. Table 2 highlights the notation we use throughout the rest of the paper.

3.1. Depth-Peeling Approaches

Depth-peeling [Mam89] is essentially multi-pass z-buffering. It restricts memory by storing a single fragment per pixel. In pass p, fragment j is conditionally inserted if and only if $p = j$. No merging of fragments occurs, and no normalization occurs to account for discarded fragments. Multi-layer peeling techniques [MB07] work identically, except they peel k layers simultaneously, so for pass p conditional insertion occurs if $k \cdot p \leq j < k \cdot (p+1)$. As $k \to \infty$ (or the number of passes increases), these converge to A-buffering. Since transparency has no influence while peeling, either alpha or discrete coverage masks can represent opacity.

3.2. k-Buffer Approaches

k-buffers [BCL*07] act like multi-layer peeling techniques, storing only k layers, but accumulate these layers in a single pass. All incoming fragments are inserted but once k fragments have been added, different algorithms vary in their merge criteria.

Bavoil et al. [BCL*07] merge the closest two fragments, assuming geometry is rasterized in approximately front-to-back order. Salvi et al. [SVLL10, SML11] merge the two fragments that, when combined, introduce the smallest error in the pixel's visibility function. Jouppi and Chang [JC99] attempt to merge fragments nearby in depth, assuming they belong to a continuous surface. Salvi and Vaidyanathan [SV14] merge the two furthest fragments using an *over* operator.

To our knowledge, all k-buffer techniques represent transparency via an alpha term, converge as $k \to \infty$, and are biased in various ways. Except the few algorithms that merge using order-independent operators, like the weighted average, k-buffers are not strictly order independent.

3.3. Methods Approximating the Transmittance Function

k-buffers approximate the transmission along a pixel using a discrete approximation to its *transmittance function* (sometimes called its visibility function). Salvi et al. [SML11] explicitly try to minimize the error of their approximation, but a separate class of work explores alternate transmission function representations, mostly in the context of casting shadows from transparent occluders.

Deep shadow maps [LV00] approximate the transmittance function for shadows using a piecewise linear sampling, with vertices generated at geometric boundaries and by regular piecewise sampling of volumetric media. Jansen and Bavoil [JB10] project the function into the frequency domain, Delalandre et al. [DGMF11] project onto a discrete cosine basis, and Gautron et al. [GDML13] add bounds to the projected bases to limit ringing in regions with no transparent media. With simplifications designed to accelerate rendering shadows in hair, Kim and Neumann [KN01] bin occluders into a small number of layers, Yuksel and Keyser [YK08] clus-

ter hair into bins with bounds extracted from geometric depth, and Mertens et al. [MKBVR04] use k-means clustering. Sintorn and Assarsson [SA09] use a bitmask to locate occlusion events along the transmission function.

Since they were designed for shadowing, rather than OIT, these techniques have slightly different characteristics. However, they all store a finite number of layers (or coefficients), merge layers with heuristics described above, and use alpha rather than coverage. Moreover, Sintorn and Assarsson [SA09] implicitly discard fragments that collide in their bitmask representation of the transmittance function, so they renormalize their final contributions.

Beyond discretely sampling via k-buffers, real time OIT techniques have largely ignored other representations of the per-pixel transmission function. The only exceptions are McGuire et al.'s [MB13, MM16] phenomenological models, which impose empirically determined transmittance functions for all pixels rather than deriving or approximating the function from scene geometry.

Approximate samplings of transmittance functions all converge as $k \to \infty$, are biased, and maintain strict order independence when using order-independent merges and normalization.

3.4. Stochastic Visibility Sampling

Rather than using deterministic sampling, like k-buffering, an alternative is to integrate transmittance via Monte Carlo sampling. Stochastic transparency [ESSL10] probabilistically renders fragments into the z-buffer based on surface opacity; a surface with opacity α emits an average of $\alpha \cdot s$ random samples into an s-sample framebuffer. This introduces noise, but is unbiased and converges with increasing sample count. Stratified sampling [LK11] improves the convergence rate, but remains noisy even with 64 samples.

Enderton et al. [ESSL11] observe that, just like the methods in Section 3.3, the stochastic samples can act as an oracle in a second geometry pass, reducing noise in exchange for bias. This acts as a normalization pass.

In the context of our axes from Section 2, stochastic transparency is layered, i.e., with s stochastic samples. Pixels store at most s fragments. Insertion occurs probabilistically based on α, and fragments are never merged, only discarded. Color can be renormalized to improve quality in exchange for bias, and transparency is stored as a (stochastic) coverage mask rather than a continuous alpha.

3.5. Bridging k-Buffers, Visibility Functions, and Stochasm

In Section 3, we largely kept the traditional rendering labels applied to these techniques. However, various algorithms fit into multiple categories, blurring the traditional labels.

We suggest depth-peeling and k-buffering techniques form a single category. Depth peeling is simply k-buffering that discards excess fragments, rather than merging them.

We also view stochastic transparency quite similarly to k-buffering. They both store a discrete number of samples per pixel (see Figure 1). Both techniques can be renormalized; in fact, we found the processing of hybrid transparency's "tail" (fragments after the k^{th}, see [MCTB13]) to be a renormalization equivalent to

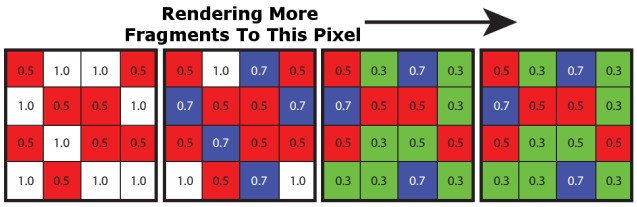

Figure 1: Stochastic transparency *renders to a multi-sample z-buffer, shown as a* 4×4 *sample grid for illustration. (Left) The pixel is first covered by a red fragment with* z *and* α *of* 0.5*; it randomly gets written to 8 of 16 sub-pixel samples. (Left center) A blue fragment with* $z = 0.7$ *and* $\alpha = 0.5$ *is rendered next; it gets written to 8 random subsamples, but the red fragment occludes a few. (Right center) Next is a green fragment with* $z = 0.3$ *and* $\alpha = 0.5$*, writing 8 random samples. (Right) A yellow fragment with* $z = 0.9$ *and* $\alpha = 0.5$ *has no effect, being fully occluded. In theory, a pixel stores up to 16 fragments per pixel, but in our example it can only store 3.*

Enderton et al.'s [ESSL11] depth and alpha correction. The key difference is s-sample stochastic transparency probabilistically inserts fragments into a k-buffer, with $k = s$; standard k-buffer techniques insert with probability 1.

Screen door transparency with alpha-to-coverage can also be seen bridging this gap. It can be viewed as stochastic transparency using fixed, instead of random, coverage masks for each α value. Alternatively, it can be viewed as a k-buffer with $k = 1$, using z-buffering to merge incoming fragments.

Our continuum from Section 2 has axes for merge heuristics and normalization. Enderton et al. [ESSL10] and Maule et al. [MCTB13] consider all discarded geometry as part of normalization; a k-buffer with normalization can be viewed as a k+1 buffer using an order-independent merge heuristic to combine "discarded" fragments into a new k+1$^{\text{st}}$ layer. In this context, McGuire et al.'s [MB13, MM16] phenomenological models are k-buffers (with $k = 0$) that merge into the k+1$^{\text{st}}$ layer via weighted sums.

All techniques we cite build some model of the transmittance function. Stochastic transparency builds this stochastically, adaptive transparency tries to minimize error, Fourier opacity mapping projects onto a suitable set of basis functions, phenomenological models derive a transmittance function empirically, and k-buffers capture or merge into a piecewise constant representation. This transmittance function can then be used as an oracle to generate color in a second geometry pass or directly visualized as a sequence of transparent surfaces.

3.6. Holes in the OIT Continuum

Looking at our continuum tabulated in Table 1, one poorly explored area jumps out—virtually all prior techniques uses alpha to represent opacity. The only exceptions are screen-door transparency, which has such poor quality that few use it for rendering complex transparent interactions, and stochastic transparency, which is high quality but noisy and somewhat expensive. This leaves a wide middle ground for alternative coverage-based transparency techniques.

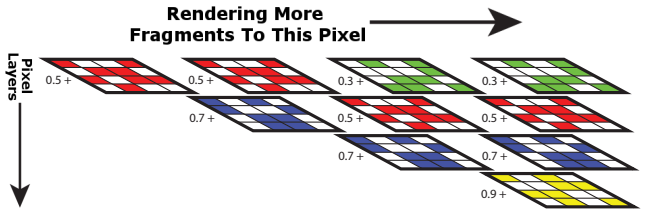

Figure 2: Stochastic layered alpha blending *(SLAB) decouples storage of depth and coverage samples. Here we show the same sequence of fragments and random coverage samples as in Figure 1, but explicitly store them in a layered representation. Since we no longer store one depth value per subpixel sample, this can reduce memory consumption. If we used $k = 2$, only the top two layers in each stack would be retained; the rest would be discarded.*

4. Stochastic Layered Alpha Blending

We describe a new algorithm using coverage masks, *stochastic layered alpha blending* (SLAB), that connects stochastic transparency and k-buffering. In fact, it has an explicit parameter that, by varying the value, gives results identical to both stochastic transparency and a k-buffer variant (hybrid transparency [MCTB13]). We present two different variants of SLAB that expose this parameter, the number of visibility bits b per layer, in different ways. The simple approach explicitly stores a b-bit visibility mask per layer, and our more complex approach stores a single alpha value per layer and computes a combinatoric insertion probability based on b virtual visibility bits.

Our stochastic approach has other benefits. It better stratifies samples along depth in stochastic transparency. This reduces noise, particularly in the presence of nearly opaque surfaces. Additionally, it decouples the size of the stochastic coverage mask b from the number of layers k, allowing an increase in b without significantly bloating memory consumption and bandwidth. However, this introduces bias in exchange.

4.1. Overview

Stochastic layered alpha blending explicitly extracts the layers in stochastic transparency. Rather than implicitly storing up to s fragments per pixel using s-sample stochastic transparency (as in Figure 1), SLAB explicitly allocates s layers (as in k-buffering) with each layer storing a fragment's depth and coverage (see Figure 2).

When encountering a new fragment, we stochastically insert it into our layer list based on a random coverage mask. If we have too many layers, we simply discard the furthest layer. Using $k = s$ we obtain the exact same results as stochastic transparency, modulo any subpixel geometric interpenetration that our representation (with a single z per fragment) cannot handle. A depth plus a random coverage mask per layer is the data structure for our simple, coverage mask implementation of SLAB.

We observed that stochastic transparency *only* uses a coverage mask to probabilistically insert fragments into the z-buffer; it converts this mask back to continuous alpha when accumulating color. By decoupling depth and coverage, our stochastic layered approach need not maintain an explicit coverage mask if we instead provide a metric controlling stochastic k-buffer insertion.

4.2. Insertion Probability

When encountering fragment f_i, we attempt to insert it into our layered structure and find j fragments in front of f_i. We find the total occlusion from these surfaces, m_{occl}, by bitwise ORing their coverage masks:

$$m_{occl} = \bigvee_{t=0}^{j-1} m_t. \tag{2}$$

Insertion to our subpixel structure occurs if $(m_{occl} \mid m_i) \neq m_{occl}$ when using explicit coverage masks.

Since the location of coverage bits in m_i are randomly selected, we instead derive a combinatoric probability function $P_b(\mathbb{b}_{occl}, \mathbb{b}_i)$ that depends only on the number of random coverage bits in m_{occl} and m_i, not their location. This allows use of a single random number to stochastically insert our fragment without computing a new random coverage mask per fragment.

4.2.1. Random Occlusion Probability Function

Given f_i with a random b-bit mask m_i containing $\mathbb{b}_i$ set bits that is potentially occluded by geometry with aggregate mask m_{occl} with $\mathbb{b}_{occl}$ set bits, we want to analytically compute the probability of insertion $P_b(\mathbb{b}_{occl}, \mathbb{b}_i)$.

For a naive, unstratified random sampling our mask m_i depends on the surface transparency α_i. Each bit in m_i is turned on with probability α_i. Fragment f_i is occluded if none of the $\overline{\mathbb{b}_{occl}}$ unset bits in m_{occl} are set in m_i. This occurs with probability $(1-\alpha_i)^{\overline{\mathbb{b}_{occl}}}$. In a world of discretized visibility, $\alpha_i = \mathbb{b}_i/b$, so our probability of insertion (i.e., of our new fragment's visibility) is:

$$P_b(\mathbb{b}_{occl}, \mathbb{b}_i) = 1 - \left(1 - \frac{\mathbb{b}_i}{b}\right)^{\overline{\mathbb{b}_{occl}}} = 1 - \left(\frac{\overline{\mathbb{b}_i}}{b}\right)^{\overline{\mathbb{b}_{occl}}} \tag{3}$$

4.2.2. Stratified Occlusion Probability Function

With a stratified random sampling, the probabilities of two bits being set is not independent. Instead, a stratified b-bit coverage mask of a surface with α_i with have either $\lfloor \alpha_i b \rfloor$ or $\lceil \alpha_i b \rceil$ bits set.

A mask m_i will be occluded by a mask m_{occl} if all $\mathbb{b}_i$ bits are occluded by one of the $\mathbb{b}_{occl}$ bits. The first bit will be occluded with probability $\mathbb{b}_{occl}/b$. Given that bit is occluded, another will be occluded with probability $(\mathbb{b}_{occl}-1)/(b-1)$. Given those occlusions, a third bit will be occluded with probability $(\mathbb{b}_{occl}-2)/(b-2)$, etc. This gives us a stratified probability of fragment f being visible:

$$P_b(\mathbb{b}_{occl}, \mathbb{b}_i) = \begin{cases} 1 - \frac{\mathbb{b}_{occl}!(b-\mathbb{b}_i)!}{b!(\mathbb{b}_{occl}-\mathbb{b}_i)!} & : \mathbb{b}_i \leq \mathbb{b}_{occl} \\ 1 & : \mathbb{b}_i > \mathbb{b}_{occl} \end{cases} \tag{4}$$

Where the factorial term is a rearrangement of $(s)_t/(b)_t$, where $(s)_t$ is the falling factorial, $s(s-1)...(s-t+1)$ and $s = \mathbb{b}_{occl}$ and $t = \mathbb{b}_i$. Equation 4 has factorials that are costly to compute on the fly; we precompute a $(b+1) \times (b+1)$ lookup table for $\mathbb{b}_i, \mathbb{b}_{occl} \in [0 \ldots b]$.

4.3. Algorithm: Creating Stochastic Layers

Given these probabilities, creating a stochastic layered buffer is straightforward. Each incoming fragment traverses the per-pixel layer list, identifies occluders, and accumulates their occlusion.

Given a number of set bits $\mathbb{b}_i$, determined by fragment i's alpha, and the number of occluder bits $\mathbb{b}_{occl}$, we take random $\xi \in [0\ldots1]$ and compare to our probability $P_b(\mathbb{b}_{occl}, \mathbb{b}_i)$. If $\xi < P_b(\mathbb{b}_{occl}, \mathbb{b}_i)$, we insert our fragment to our per-pixel layer list. If we have more than the maximum number of layers k, we discard the furthest.

When storing explicit coverage masks per layer, we count set bits to find $\mathbb{b}_i$ and $\mathbb{b}_{occl}$ (finding occlusion via Equation 2). If storing an α per layer, we compute $\mathbb{b}_i$ by discretizing $\alpha_i b$ and $\mathbb{b}_{occl}$ by discretizing $(1-\prod_{t\in[0\ldots j-1]}(1-\alpha_t))b$.

4.4. Algorithm: Rendering Final Colors

We can accumulate final color in exactly the same way as stochastic transparency [ESSL11], in a second geometry pass using the per-pixel layers to provide the inputs for the Porter and Duff *over* operator [PD84]:

$$C = \alpha_1 c_1 + (1-\alpha_1)(\alpha_2 c_2 + (1-\alpha_2)(\alpha_3 c_3 + ...)), \tag{5}$$

which can be rearranged as:

$$C = \sum_i \left(\prod_{z_j<z_i} (1-\alpha_j) \right) \alpha_i c_i. \tag{6}$$

α_i and c_i come directly from the fragment, and $\prod(1-\alpha_j)$ is the probability that fragment i is visible.

We can also perform Enderton et al.'s [ESSL10] depth and alpha correction by accumulating opacity of all transparent fragments, α_{total}, and the visibility-weighted sum of fragment alphas, α_{sum}:

$$\alpha_{total} = 1 - \prod_{\text{all } i} (1-\alpha_i), \tag{7}$$

$$\alpha_{sum} = \sum_{\text{all } i} \left(\prod_{z_j<z_i} (1-\alpha_j) \right) \alpha_i. \tag{8}$$

The final normalized color can then be computed as:

$$C_{norm} = \frac{a_{total}}{a_{sum}} C \tag{9}$$

4.5. Further Considerations for Using Continuous α

Decoupling stochastic transparency's coverage and depth samples allows us increase the number of coverage samples per layer faster than the number of layers. In fact, we can increase the number of bits to "infinity" and use a continuous alpha value.

But the insertion probabilities $P_b(\mathbb{b}_{occl}, \mathbb{b}_i)$ from Equations 3 and 4 still require discrete values for $\mathbb{b}_i$ and $\mathbb{b}_{occl}$. For both random and stratified sampling schemes, we bilinearly interpolate the probability functions based on the probabilities with the discrete values $\lfloor \alpha_i b \rfloor$, $\lceil \alpha_i b \rceil$, $\lfloor \alpha_{occl} b \rfloor$, and $\lceil \alpha_{occl} b \rceil$ based on a user-selectable number of "virtual bits" b.

Naive random samples could instead be linearly interpolated between $\lfloor \alpha_{occl} b \rfloor$, and $\lceil \alpha_{occl} b \rceil$ using the following probability:

$$P_b(\mathbb{b}_{occl}, \alpha_i) = 1 - (1-\alpha_i)^{\overline{\mathbb{b}_{occl}}} \tag{10}$$

but bilinearly interpolating the stratified probability from Equation 4 gives better results.

4.5.1. Connecting Stochastic Transparency to k-Buffers

Once storing continuous α, the precision of discretization is simply a parameter to the algorithm, specifying a number of bits b in our virtual coverage mask.

Interestingly, the probability functions P_b approach 1 as $b \to \infty$ (except if $b = \mathbb{b}_{occl}$, i.e., $\alpha_{occl} = 1$). For a k layer stochastic representation, $b \in [k\ldots\infty]$ provides a smooth continuum between stochastic transparency (if $b = k$) to variants of k-buffering (if b is large). The k-buffering variant to which SLAB converges depends on the normalization used. Without stochastic transparency's depth and alpha correction, it converges to k-layer depth peeling; with correction, it converges to k layer hybrid transparency [MCTB13]. In retrospect, this convergence is not particularly surprising. As stochastic transparency's per-pixel sample count increases it converges to ground truth (or A-buffering), and we simply limit the number of layers stored per pixel.

Note that $b < k$ is valid, but uninteresting, as no more than b layers can be stored per pixel. Each fragment requires at least one unique bit in its virtual coverage mask to be visible.

5. Multi-Layered Coverage Blending

Integrating multi-layer algorithms with multisampling in a performant way is challenging. We could store k layers for each subpixel coverage sample, but this dramatically increases bandwidth and memory usage. In theory, an alpha value already contains information about coverage as well as transparency [Gla15], so storing k alpha values per subpixel potentially duplicates coverage information already baked into the alpha channel. Instead, as noted in Section 3.6, we observe that most OIT techniques focus on α-based representations and propose using explicit coverage to represent both coverage and transparency.

We introduce *multi-layered coverage blending* (MLCB), which is a variant of multi-layered alpha blending (MLAB) [SV14] that stores per-layer coverage rather than alpha.

5.1. Coverage Mask Computation

At a high level MLAB and MLCB work identically, except MLCB substitutes a coverage mask for MLAB's per-layer alpha value. We compute this coverage mask in two parts. A fragment's transparency is converted to a coverage mask, $m_{\alpha 2c}$, using alpha-to-coverage. Each fragment also computes geometric coverage m_{geom} based on subpixel coverage from the rasterizer (e.g., from `gl_SampleMask` or `SV_Coverage`). Fragment i's coverage mask is computed with a bitwise AND:

$$m_i = m_{\alpha 2c} \,\&\, m_{geom}$$

Techniques using alpha-to-coverage tend to introduce correlation artifacts (e.g., as in screen-door transparency [FGH*85]). To avoid that, we randomize the coverage mask for each fragment. This can introduce noise and flickering, similar to stochastic transparency. If this is objectionable, using the same per-pixel pseudo-random seed every frame keeps the noise fixed between frames.

5.2. Algorithm: Inserting and Merging Fragments

Multi-layered coverage blending is a straightforward derivative of MLAB. Each incoming fragment f_i gets inserted into a sorted per-pixel list of fragments. Lists store k fragments, and each fragment contains a depth z_i, color c_i, and coverage mask m_i. To avoid baking alpha's conflated coverage and opacity into the color term, we explicitly do not use pre-multiplied alpha in c_i, below, unlike Salvi et al. [SV14].

Since we have an explicit coverage mask, we can account for occlusion from closer fragments as part of insertion, i.e., instead of storing a fragment's mask m_i, we store a pre-occluded mask m_i':

$$m_i' = m_i \;\&\; \sim\left(\bigvee_{z_j<z_i} m_j\right). \tag{11}$$

If this insertion increases our per-pixel list size beyond the maximum size k, we merge the two most distant fragments, f_k and f_{k+1}, together into a new fragment f_k'':

$$c_k'' = w_k c_k + w_{k+1} c_{k+1} \tag{12}$$
$$m_k'' = m_k' \mid m_{k+1}' \tag{13}$$
$$z_k'' = z_k' \tag{14}$$

where w_k and w_{k+1} can be thought of a discrete alpha values:

$$w_k = \frac{\mathbb{b}_k'}{\mathbb{b}_k''} = \frac{\left\|m_k'\right\|_1}{\left\|m_k' \mid m_{k+1}'\right\|_1}, \quad w_{k+1} = \frac{\mathbb{b}_{k+1}'}{\mathbb{b}_k''} = \frac{\left\|m_{k+1}'\right\|_1}{\left\|m_k' \mid m_{k+1}'\right\|_1}.$$

Essentially, when merging two fragments we linearly interpolate the colors with weights based on the number of coverage bits visible in each fragment. To avoid double counting contributions from a coverage sample, it is important to use occluded masks m_i' rather than a fragment's original mask m_i.

5.3. Algorithm: Rendering Final Colors

Once we have an array of k layers with color, depth, and pre-occluded coverage masks, our final pixel color can easily be computed as follows:

$$C = \frac{1}{b}\left(\left\|\sim\bigvee_{i=1}^{k} m_i'\right\|_1 c_{bg} + \sum_{i=1}^{k}\left\|m_i'\right\|_1 c_i\right). \tag{15}$$

This weighs each layer by the number of bits in its coverage mask and the background based on the number of bits unset in any mask. This basically rearranges Equation 6 for our discrete sampling.

5.4. Avoiding Banding Artifacts

The major problem moving from α-based algorithms to coverage-based algorithms is that the number of representable levels of transparency decreases from 2^b to $b+1$. Since the number of bits developers allocate likely will not change, this may cause banding on transparent surfaces.

One way to partially mitigate this problem would store both alpha and coverage, explicitly representing geometric information as coverage bits and opacity as alpha. For instance, most engines use at most $4\times$ MSAA. In such a situation, each of our k per-pixel layers could store 4 bits of geometric coverage and a 4-bit alpha value, representing $(4+1)\times 2^4 = 80$ transparent values with an 8-bit field. Randomization can also replace banding with noise.

5.5. Context of MLCB in the OIT Continuum

By design, multi-layered coverage blending is one step away from MLAB on our continuum (see Table 1), simply switching out a per-layer α value for a per-layer coverage value. Once we randomize the coverage mask to reduce correlation, MLCB starts to resemble stochastic layered alpha blending, with per-layer random coverage masks. The key difference between our new algorithms is that MLCB uses an order dependent merge, rather than a normalization term, and always adds a fragment to its per-pixel list (rather than stochastically inserting them).

One could also view multi-layer coverage blending as an extension of screen-door transparency, decoupling coverage bits and depth samples by storing multiple layers per-pixel and randomizing the generated alpha-to-coverage pattern.

6. Implementation

We implemented our new algorithms, plus various prior techniques, using OpenGL 4.5 with a few extensions. In particular, we used *NV_shader_atomic_int64* and *NV_fragment_shader_interlock* for synchronization.

Our goal with stochastic layered alpha blending and multi-layered coverage blending is not necessarily to suggest their use over prior OIT algorithms, but rather to show that exploring the OIT continuum provides developers tools that give more control over algorithmic tradeoffs.

One example: while implementing hybrid transparency we made an interesting observation. Both Maule et al. [MCTB13] and Salvi et al. [SV14] discuss one-pass versions of hybrid transparency that only touch the geometry once. As SLAB connects hybrid and stochastic transparency, this optimization may carry through, raising the possibility of implementing stochastic transparency in one pass. We implemented a variant of SLAB derived from one-pass hybrid transparency. Due to higher bandwidth requirements this approach does not have competitive performance today. But future hardware improvements that provide appropriate granularity synchronization could enable more competitive performance, potentially providing a big win over multi-pass stochastic techniques.

7. Results

Figure 3 shows geometry from Unity's Blacksmith demo, with varied amounts of transparency added. This figure compares results from various algorithms, demonstrating some important qualities.

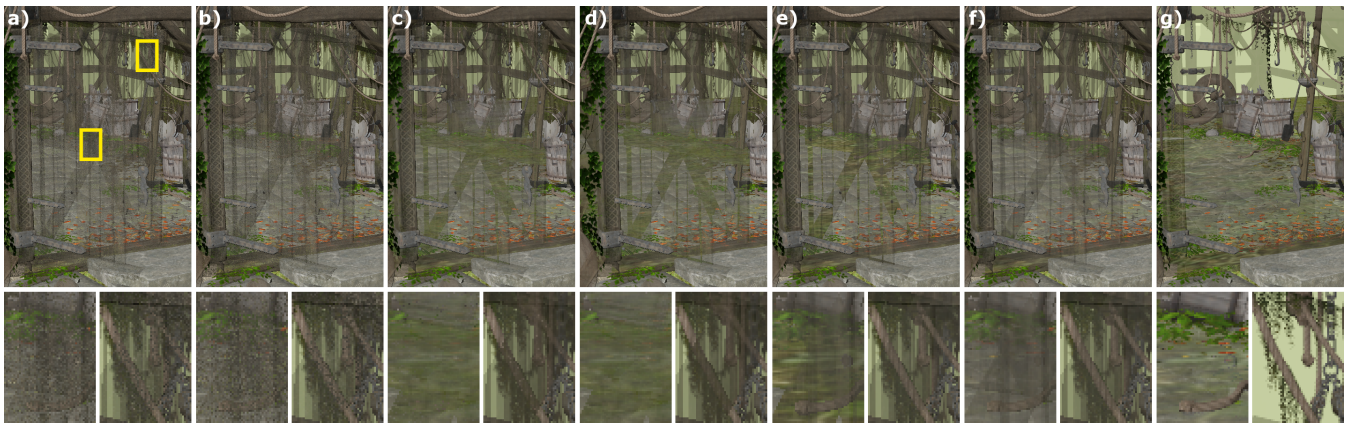

Figure 3: *Geometry from Unity's "The Blacksmith" demo under varying OIT algorithms: (a) stochastic transparency with 4 samples, (b) stochastic layered alpha blending with $k = 4$ and 4 coverage bits, (c) SLAB with $k = 4$, $b = 16$, and continuous α, (d) hybrid transparency with $k = 4$, (e) MLAB with $k = 4$, (f) ground truth, (g) and $8\times$ MSAA with alpha-to-coverage.*

Figure 4: *The blacksmith model comparing (top) MLCB with 32 coverage samples with (bottom) MLAB. In both cases, we use $k = 4$.*

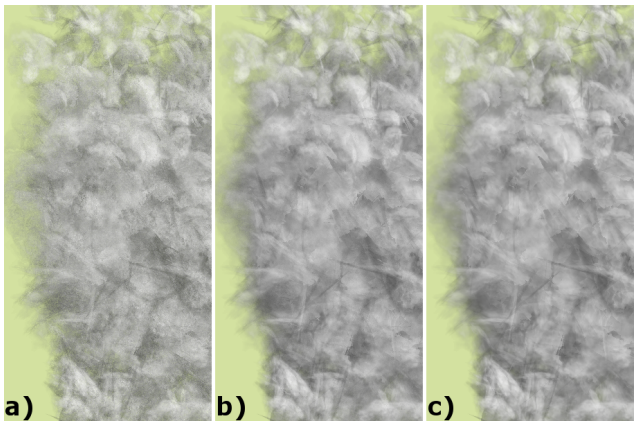

Figure 6: *A complex particle system rendered with (a) MLCB using $b = 32$, (b) MLCB using $b = 128$, and (c) MLAB. All use $k = 4$.*

It compares stochastic transparency and stochastic layered alpha blending using identical sampling rates, showing equivalent results modulo varied random noise. It also demonstrates the bias arising from using only k layers, namely overweighting of background geometry when overflows occur (see inset where beams on the door add additional layers). It highlights the catastrophic problems with fixed alpha-to-coverage dither patterns. Finally, it compares with a weakly order-independent algorithm; in some circumstances algorithms like MLAB capture details from more than k layers, but this benefit is nonuniformly distributed and hard to predict. Larger versions of these images are provided as supplementary material.

Figure 4 demonstrates how multi-layered coverage blending compares with multi-layered alpha blending. Explicilt coverage masks enable antialiasing and accumulating transparency simultaneously, though even with a 32-bit mask we see noise in the results. Figure 5 compares the antialiasing abilities in MLAB, MLCB and MSAA with alpha-to-coverage and demonstrates that MLCB converges to MLAB with sufficient samples and has correlation artifacts reminiscent of MSAA without sufficient samples.

Figure 6 shows a complex billboard cloud particle system with MLAB and MLCB, demonstrating they converge with a large enough mask. Our supplementary material contains images of this particle system with other rendering algorithms.

Figure 7 shows the Foliage Map from the Unreal Engine SDK, with α of 0.75 applied to all surfaces, and demonstrates a relatively continuous transition from stochastic transparency to hybrid transparency. For SLAB, we use explicit coverage masks to demonstrate the noise reduction with increasing mask size b. We also compare to SLAB using alpha, rather than a coverage, with $b = 64$ virtual bits. This is virtually indistinguishable from our k-buffer. Figure 8 shows how SLAB behaves with increasing layer count.

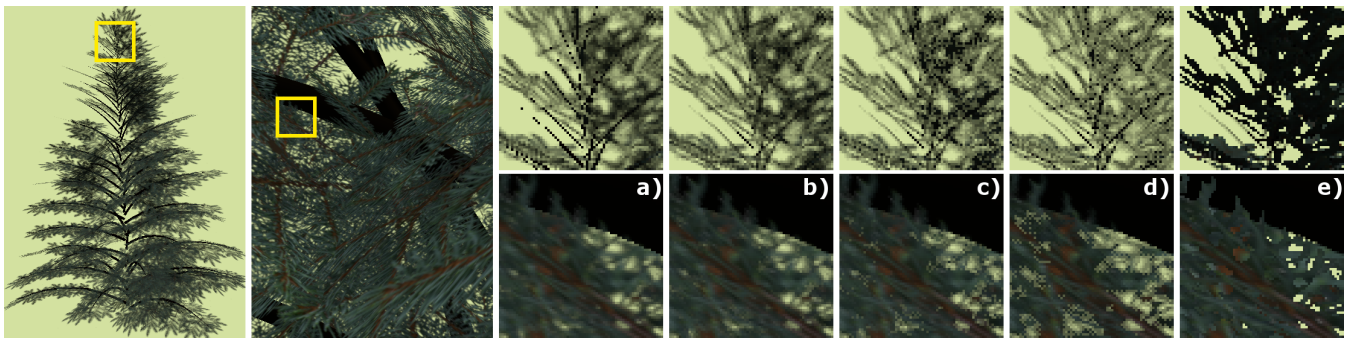

Figure 5: *Two views of a pine tree with many overlapping alpha textures. Insets rendered using a variety of algorithms: (a) MLAB, (b) MLCB with 8 geometric samples and a 128 bit coverage mask, (c) MLCB with 8 geometric samples and a 16 bit mask, (d) 8× MSAA with alpha-to-coverage, and (e) 8× MSAA with alpha testing. Full images for all algorithms are included as supplementary material.*

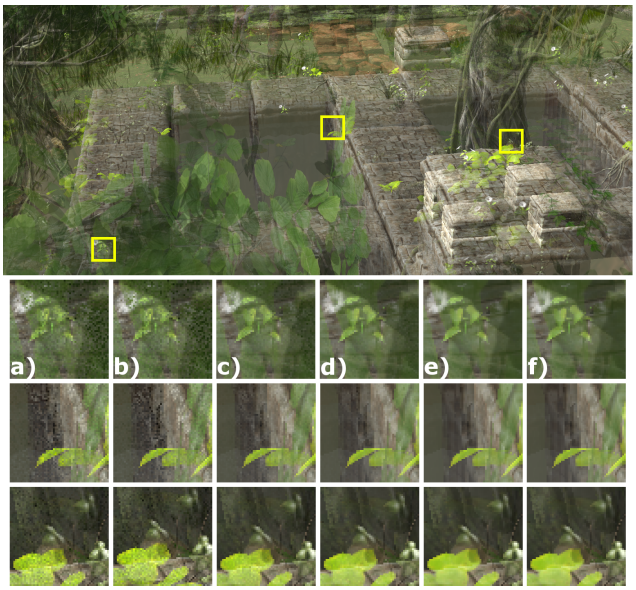

Figure 7: *The Unreal Foliage Map, all surfaces* $\alpha = 0.75$*, with insets rendered with various methods: (a) stochastic transparency, 8 samples, (b) SLAB with* $b = 8$ *and* $k = 8$*, (c) SLAB with* $b = 32$ *and* $k = 8$*, (d) SLAB with* $b = 128$ *and* $k = 8$*, (e) SLAB storing continuous* α *with* $k = 8$ *and* $b = 32$ *virtual bits, and (f) k-buffer with* $k = 8$*.*

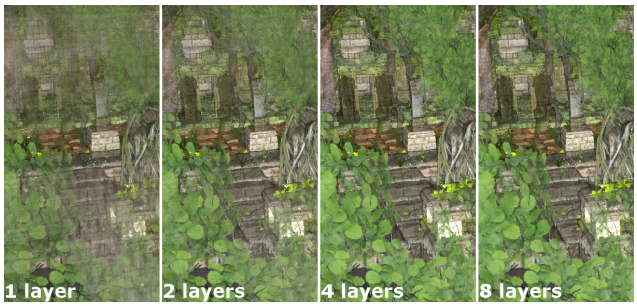

Figure 8: *The Unreal Foliage Map, all surfaces* $\alpha = 0.75$*, rendered with SLAB using* $b = 32$ *and varying numbers of layers.*

Algorithm Cost in msec / frame	Pine Tree	Car	Black-smith	Foliage Map
SLAB	8.1	15.6	48.1	89.2
Stochastic transp. [ESSL10]	8.2	21.3	86.6	162.0
Hybrid transp. [MCTB13]	8.7	24.6	91.8	175.9
MLCB	18.4	40.3	89.9	168.1
MLAB [SV14]	29.0	65.7	191.2	346.1

Table 3: *Rough, prototype performance numbers on an NVIDIA GeForce Titan X at* 2560×1440 *where all polygons in the scene are partially transparent. Stochastic transparency uses 8 coverage samples. Layered techniques use* $k = 4$ *layers. SLAB and MLCB use* $b = 32$ *coverage bits. Performance is unoptimized; please see text for caveats.*

7.1. Performance

Our goal was not identifying a "best" OIT algorithm for any particular application, nor to suggest either of our algorithms over prior work. For that reason, we do not provide much comparative performance data. With a few exceptions, all the algorithms in Table 1 are fast enough to ship in modern games when limited to scenes with a reasonable number of transparent objects. A few algorithms requiring more complex GPU syncronization, e.g., Salvi et al. [SV14], may not yet be performant on the majority of GPUs.

For development, we used a NVIDIA GeForce Titan X. Rough performance numbers from our prototype are shown in Table 3. Please note: performance was not optimized for speed, as our code was designed to allow tuning knobs to smoothly transition from one algorithm to another. Not all algorithms have been optimized an equal amount. In theory, hybrid transparency and stochastic layered alpha buffering have roughly equivalent performance, though our SLAB implementation does not keep its k layers sorted, which significantly reduces work inside the critical section.

For our implementations, SLAB with 4 layers with 32-bit coverage requires the same memory footprint as stochastic transparency

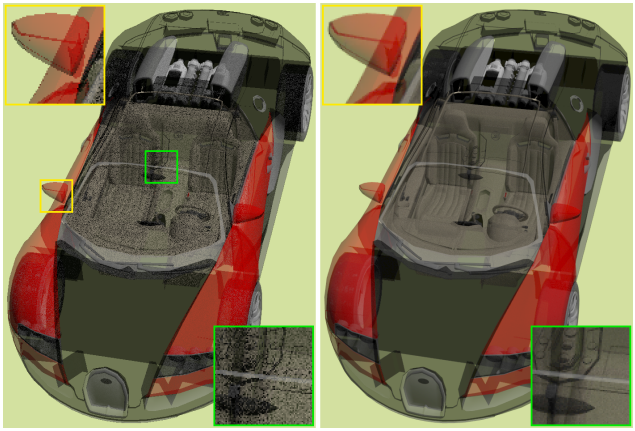

Figure 9: *A car rendered using stochastic layered alpha blending without (left) and with (right) temporal antialiasing.*

using 8 samples per pixel. Increasing SLAB's coverage mask to 96-bits per layer doubles the memory footprint (to 128 bits per layer).

MLCB and MLAB have more complex critical sections but in well optimized implementations they should achieve roughly equal performance. They have larger per-layer storage requirements since they work in a single pass. This slows performance in comparison to SLAB or stochastic transparency, especially in simpler scenes.

7.2. Coupling and Decoupling Antialiasing and Transparency

Our initial motivation in developing MLCB was enabling efficient antialiasing as part of our OIT algorithm. However, pitfalls when merging fragments that only partially cover pixels complicate this process, as highlighted by Jouppi and Chang [JC99]. We borrowed a high quality temporal antialiasing algorithm (e.g., [Kar14]) from another project and discovered this seamlessly handled sub-pixel geometric interpenetration that proves difficult in Z^3 and MLCB.

Temporal antialiasing's ability to handle geometric aliasing reassured us when decoupling antialiasing from stochastic transparency. By splitting coverage and depth samples in SLAB, we lost stochastic transparency's inherent geometric antialising. However, applying temporal filtering as a post process not only removes this aliasing but filters much of the noise from the stochastic process (see Figure 9).

8. Conclusions

We presented a new categorization of interactive order-independent transparency techniques. Fundamentally, we view all current OIT algorithms as using simplified representations of the transmittance function, typically using k layers or samples. Their key differences lie in how new fragments are inserted into per-pixel structures, how these fragments are merged together, if normalization occurs, and whether the transmittance approximation relies on coverage masks or continuous alpha values.

Categorizing this work enabled us to identify various unexplored areas. We presented SLAB and MLCB, which both leverage coverage masks.

Of our new algorithms, we find the stochastic layered approach more interesting. It provides a parameter to transition between stochastic transparency and k-buffering, two algorithms with previously non-obvious similarities. This might allow professional applications, like CAD software, that aim for predictive rendering to add a knob that could dynamically increase bias in exchange for performance. SLAB also suggests stochastic insertion is a viable insertion metric for k-buffering, which may provide inspiration for new Monte Carlo-based interactive rendering approximations.

Combined with the ability to leverage optimizations from one OIT technique into another, we hope our categorization inspires future work exploring and optimizing other areas of the OIT continuum, and perhaps even suggesting new axes for algorithms to explore.

9. Acknowledgments

Many people at NVIDIA Research, including Marco Salvi, Eric Enderton, Aaron Lefohn, Anton Kaplanyan, and Anjul Patney, provided insightful discussions on order-independent and stochastic transparency as well as feedback on paper drafts. Nir Benty's *Falcor* prototyping framework was used as a base for this research.

Thanks to Epic Games for making models available as part of the Unreal Development Kit and to Unity for releasing their Blacksmith Demo models.

References

[And15] ANDERSSON J.: The rendering pipeline: Challenges & next steps. In "Open problems in real-time rendering," *ACM SIGGRAPH Courses*, 2015. 1

[BCL*07] BAVOIL L., CALLAHAN S., LEFOHN A., COMBA J., SILVA C.: Multi-fragment effects on the gpu using the k-buffer. In *Symposium on Interactive 3D Graphics and Games* (2007), pp. 97–104. 2, 3

[BM08] BAVOIL L., MYERS K.: *Order independent transparency with dual depth peeling*. Tech. rep., NVIDIA, 2008. 2

[Car84] CARPENTER L.: The a-buffer, an antialiased hidden surface method. In *Proceedings of SIGGRAPH* (1984), pp. 103–108. 1, 3

[DGMF11] DELALANDRE C., GAUTRON P., MARVIE J.-E., FRANÇOIS G.: Transmittance function mapping. In *Symposium on Interactive 3D Graphics and Games* (2011), pp. 31–38. 3

[ESSL10] ENDERTON E., SINTORN E., SHIRLEY P., LUEBKE D.: Stochastic transparency. In *Symposium on Interactive 3D Graphics and Games* (2010), pp. 157–164. 1, 2, 3, 4, 6, 9

[ESSL11] ENDERTON E., SINTORN E., SHIRLEY P., LUEBKE D.: Stochastic transparency. *IEEE Transactions on Visualization and Computer Graphics 17*, 8 (2011), 1036–1047. 4, 6

[Eve01] EVERITT C.: *Interactive Order-Independent Transparency*. Tech. rep., NVIDIA, 2001. 2, 3

[FGH*85] FUCHS H., GOLDFEATHER J., HULTQUIST J. P., SPACH S., AUSTIN J. D., BROOKS JR. F. P., EYLES J. G., POULTON J.: Fast spheres, shadows, textures, transparencies, and imgage enhancements in pixel-planes. In *Proceedings of SIGGRAPH* (1985), pp. 111–120. 2, 3, 7

[GDML13] GAUTRON P., DELALANDRE C., MARVIE J.-E., LECOCQ P.: Boundary-aware extinction mapping. *Computer Graphics Forum 32*, 7 (2013), 305–314. 3

[GHLM05] GOVINDARAJU N. K., HENSON M., LIN M. C., MANOCHA D.: Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In *Symposium on Interactive 3D Graphics and Games* (2005), pp. 49–56. 1

[Gla15] GLASSNER A.: Interpreting alpha. *Journal of Computer Graphics Techniques (JCGT) 4*, 2 (2015), 30–44. 1, 6

[JB10] JANSEN J., BAVOIL L.: Fourier opacity mapping. In *Interactive 3D Graphics and Games* (2010), pp. 165–172. 2, 3

[JC99] JOUPPI N. P., CHANG C.-F.: Z3: An economical hardware technique for high-quality antialiasing and transparency. In *Graphics Hardware* (1999), pp. 85–93. 2, 3, 10

[Kar14] KARIS B.: High-quality temporal supersampling. In "Advances in real time rendering," *ACM SIGGRAPH Courses*, 2014. 10

[KN01] KIM T.-Y., NEUMANN U.: Opacity shadow maps. In *Eurographics Rendering Workshop* (2001), pp. 177–182. 3

[LHLW09] LIU F., HUANG M.-C., LIU X.-H., WU E.-H.: Efficient depth peeling via bucket sort. In *High Performance Graphics* (2009), pp. 51–57. 2, 3

[LK11] LAINE S., KARRAS T.: Stratified sampling for stochastic transparency. *Computer Graphics Forum 30*, 4 (2011), 1197–1204. 4

[LV00] LOKOVIC T., VEACH E.: Deep shadow maps. In *Proceedings of SIGGRAPH* (2000), pp. 385–392. 2, 3

[Mam89] MAMMEN A.: Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications 9*, 4 (1989), 43–55. 3

[MB07] MYERS K., BAVOIL L.: Stencil routed a-buffer. In *ACM SIGGRAPH Sketches* (2007). 2, 3

[MB13] MCGUIRE M., BAVOIL L.: Weighted blended order-independent transparency. *Journal of Computer Graphics Techniques (JCGT) 2*, 2 (December 2013), 122–141. 2, 3, 4

[MCTB13] MAULE M., COMBA J. A., TORCHELSEN R., BASTOS R.: Hybrid transparency. In *Symposium on Interactive 3D Graphics and Games* (2013), pp. 103–118. 1, 2, 3, 4, 5, 6, 7, 9

[Mes07] MESHKIN H.: Sort-independent alpha blending. GDC Session, 2007. 2, 3

[MGvW98] MULDER J. D., GROEN F. C. A., VAN WIJK J. J.: Pixel masks for screen-door transparency. In *IEEE Visualization* (1998), pp. 351–358. 2

[MKBVR04] MERTENS T., KAUTZ J., BEKAERT P., VAN REETH F.: A self-shadow algorithm for dynamic hair using density clustering. In *ACM SIGGRAPH Sketches* (2004). 3, 4

[MM16] MCGUIRE M., MARA M.: A phenomenological scattering model for order-independent transparency. In *Symposium on Interactive 3D Graphics and Games* (February 2016), p. 10. 2, 3, 4

[MP01] MARK W. R., PROUDFOOT K.: The f-buffer: A rasterization-order fifo buffer for multi-pass rendering. In *Graphics Hardware* (2001), pp. 57–64. 2

[PD84] PORTER T., DUFF T.: Compositing digital images. In *Proceedings of SIGGRAPH* (1984), pp. 253–259. 1, 3, 6

[SA09] SINTORN E., ASSARSSON U.: Hair self shadowing and transparency depth ordering using occupancy maps. In *Symposium on Interactive 3D Graphics and Games* (2009), pp. 67–74. 3, 4

[SML11] SALVI M., MONTGOMERY J., LEFOHN A.: Adaptive transparency. In *High Performance Graphics* (2011), pp. 119–126. 2, 3

[SV14] SALVI M., VAIDYANATHAN K.: Multi-layer alpha blending. In *Symposium on Interactive 3D Graphics and Games* (2014), pp. 151–158. 2, 3, 6, 7, 9

[SVLL10] SALVI M., VIDIMCE K., LAURITZEN A., LEFOHN A.: Adaptive volumetric shadow maps. *Computer Graphics Forum 29*, 4 (2010), 1289–1296. 3

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

[YHGT10] YANG J. C., HENSLEY J., GRÜN H., THIBIEROZ N.: Real-time concurrent linked list construction on the gpu. *Computer Graphics Forum 29*, 4 (2010), 1297–1304. 2

[YK08] YUKSEL C., KEYSER J.: Deep opacity maps. *Computer Graphics Forum 27*, 2 (2008), 675–680. 3