Stochastic Transparency

Eric Enderton
NVIDIA

Erik Sintorn
Chalmers University of Technology

Peter Shirley
NVIDIA

David Luebke
NVIDIA

Abstract

Stochastic transparency provides a unified approach to order-independent transparency, anti-aliasing, and deep shadow maps. It augments screen-door transparency using a random sub-pixel stipple pattern, where each fragment of transparent geometry covers a random subset of pixel samples of size proportional to alpha. This results in correct alpha-blended colors on average, in a single render pass with fixed memory size and no sorting, but introduces noise. We reduce this noise by an alpha correction pass, and by an accumulation pass that uses a stochastic shadow map from the camera. At the pixel level, the algorithm does not branch and contains no read-modify-write loops other than traditional z-buffer blend operations. This makes it an excellent match for modern massively parallel GPU hardware. Stochastic transparency is very simple to implement and supports all types of transparent geometry, able without coding for special cases to mix hair, smoke, foliage, windows, and transparent cloth in a single scene.

Keywords: rendering, order-independent transparency, deep shadow maps, hair, smoke, foliage

1 Introduction

Interactive rendering of complex natural phenomena such as hair, smoke, or foliage may require many thousands of semi-transparent elements. Architectural renderings and CAD visualizations often depend on transparency as well. While the ubiquitous hardware z-buffer allows programmers to render opaque triangles efficiently and in any order, order-independent transparency (OIT) remains a difficult but highly desirable goal. Sorting transparent geometry at the object level results in artifacts when objects overlap in depth, while sorting at the fragment level is computationally expensive. Combining multiple approaches, each suitable to a different category of geometry, adds onerous complexity to interactive applications. Shadows cast by semi-transparent objects present an additional challenge. Such shadows are in fact a version of the OIT problem as they require some sort of transparency map that encodes opacity behavior.

We revisit the old-fashioned technique of screen-door transparency, describing algorithmic extensions that make it a practical interactive OIT method for both shadows and visible surfaces. Among its natural advantages, traditional screen-door transparency unifies all transparent geometry types in a single algorithm; works well with multi-sample anti-aliasing (MSAA); requires no sorting; requires no additional memory beyond the MSAA z-buffer; and is ideally suited to massively parallel hardware. (Indeed, it has been supported to some degree by multi-sample z-buffer hardware since the early 1990s [Akeley 1993], although modern GPUs have introduced more flexible support.) Furthermore the technique is easily extended to transparency shadow maps.

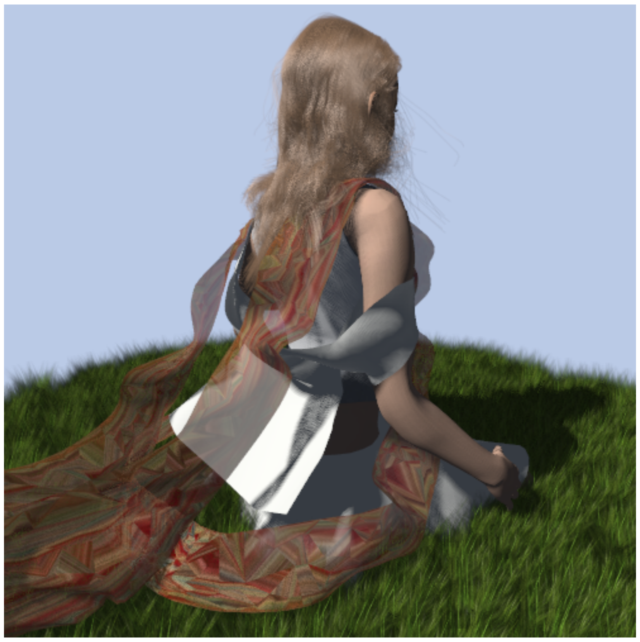

Figure 1: *15,000 transparent hairs, 6,000 transparency-mapped cards, transparent cloth, and opaque models all composite properly and shadow each other with only four order-independent render passes over the transparent geometry. Note for example the grass blades visible behind and in front of the scarf.*

Stochastic transparency extends screen-door transparency with randomly chosen sub-pixel stipple patterns. For example, a full-pixel fragment with an opacity of 50% will randomly cover half of the sub-pixel samples; for those samples, it is fully opaque. Conventional z-buffering does the rest. If another 50% opaque fragment is closer to the camera, then the first fragment's color may end up in only 25% of the total samples, regardless of which triangle is rasterized first. Handling all transparent geometry this way results in the correct alpha-blended color on average, but introduces noise.

We explore several methods for reducing that noise. In particular, we borrow the alpha correction and accumulation pass methods of Sintorn and Assarsson [2009] to formulate *depth-based stochastic transparency,* which stores only z values during the stochastic render pass, and yields more accurate transparent sampling. These techniques are combined with multi-sample anti-aliasing, yet remain simple and unified. Our interactive implementation shows visually pleasing results at encouraging frame rates. We also discuss connections to Monte Carlo ray tracing and to deep shadow maps.

No fragments were sorted in the making of these pictures.

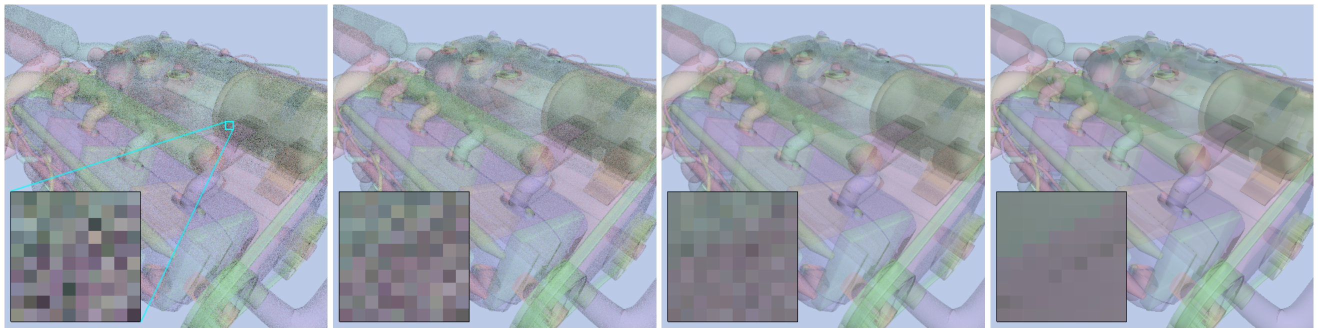

Figure 2: *Basic stochastic transparency with (a) 8, (b) 16, and (c) 64 samples per pixel, plus (d) a reference image from depth peeling. All images with no anti-aliasing, alpha 30%. Model contains 322K triangles.*

2 Prior Work

The *A-buffer* [Carpenter 1984] achieves order-independent transparency by storing a list of transparent surfaces per pixel. It has long been a staple of off-line rendering and can now be implemented on recent graphics hardware. However, the A-buffer must store all transparent fragments at once, resulting in unpredictable and virtually unbounded storage requirements. This requires either dynamic memory allocation, or dynamic tile sizing to keep the memory requirement below a fixed limit. Both are challenging to make efficient in a massively parallel computational environment such as modern GPUs.

Depth peeling [Everitt 2001; Liu et al. 2006; Bavoil and Meyers 2008] uses dual depth comparisons per sample to extract and composite each semi-transparent layer in a separate rendering pass. This approach makes efficient use of rasterization hardware, but requires an unbounded number of rendering passes (enough to reach the maximum depth complexity of the transparent image regions). Complex geometry, tessellation, skinning, scene traversal, and CPU speed limitations can all combine to make each rendering pass quite expensive, feasible only a few times per frame. Depth peeling can also be applied to fragments bucketed by depth [Liu et al. 2009]; this works well, but the number of passes still varies with the maximum number of collisions.

Billboard sorting renders the objects in sorted order. This is efficient and can be done entirely on the GPU [Sintorn and Assarsson 2008], but is only appropriate for geometry types that don't overlap in depth. When approximate results are acceptable, a partially sorted list of objects can be rendered with the help of a *k-buffer* for good results [Bavoil et al. 2007].

Screen-door transparency replaces transparent surfaces with a set of pixels that are fully on or off [Fuchs et al. 1985]. The choice of stipple patterns can be optimized so they are more visually pleasing to viewers [Mulder et al. 1998]. The idea of random stipple patterns per polygon has also been long known, and was dubbed *cheesy translucency* in the original OpenGL "red book" [Neider and Davis 1993]. The sub-pixel variation of screen-door transparency that has been implemented in hardware is *alpha-to-coverage*, which turns samples on or off to implicitly encode alpha. Because alpha-to-coverage uses a fixed dithering pattern, multiple layers occlude each other instead of blending correctly, leading to visual artifacts.

This paper develops the combination of the "random" and "sub-pixel" ideas together with efficient hardware multi-sampling, as well as recent GPU features that allow the fragment shader to discard individual samples [Balci et al. 2009].

Random sub-pixel screen-door transparency is exactly analogous to using Russian roulette to decide ray absorption during batch ray tracing [Morley et al. 2006]. As an object-level analogy to screen-door transparency, triangles can be dropped from a mesh in proportion to transparency [Sen et al. 2003], but this can require a dynamic tessellation as an object changes size on screen. In order to keep rendering times in proportion to screen size, Callahan et al. drop some polygons and increase the opacity of the remaining ones [Callahan et al. 2005]. Finally, recent deferred shading schemes [Kircher and Lawrance 2009] can combine a transparent object with opaque objects using a stipple pattern in the G-buffer, by searching for appropriate samples during the deferred shading pass.

A variety of techniques has been used to account for shadows cast on and by transparent objects. In the film industry the most common method is deep shadow maps [Lokovic and Veach 2000]. These, like the A-buffer, require a variable length list per sample while the map is constructed. *Deep opacity maps* [Yuksel and Keyser 2008] have clear visual artifacts unless a very large number of layers are used. *Occupancy maps* are promising but are so far applicable only to objects of uniform alpha [Sintorn and Assarsson 2009; Eisemann and Décoret 2006]. All such methods that use a uniform subdivision of the depth range suffer accuracy issues for scenes with highly non-uniform distributions of transparency.

Our technique combines several features not addressed together in prior approaches. Stochastic transparency uses fixed memory, renders a fixed number of passes, adapts to the precise location of layers, and scales in effectiveness with MSAA hardware improvements. The main tradeoff for these features is the amount of time and memory required to compute and store many samples. In hardware terms this translates to very aggressive MSAA buffer use. Despite this drawback we argue that the robustness and simplicity of stochastic transparency make it an attractive option when designing interactive graphics software systems that must handle transparency.

3 Stochastic Transparency

Stochastic transparency is simply screen-door transparency using a randomized sub-pixel version of the "screen-door" stipple pattern. As with most stochastic methods, the main problem is noise. Section 3.1 describes the basic method both for rendering and shadowing transparent objects, and shows that Monte Carlo stratification can be used to reduce the noise. In Section 3.2 we describe how to apply alpha correction for noise reduction, and in Section 3.3 we describe a three-pass but still order-independent method to reduce noise further. In Section 3.4 we show how our methods can be implemented efficiently using the multi-sample anti-aliasing features on modern hardware. Throughout the discussion we refer to "triangles", but the ideas extend naturally to other primitives. We also often use "fragment" which refers to the part of the triangle inside a given pixel.

3.1 Sampling Transparency

We assume a collection of triangles where the ith triangle has color c_i and opacity α_i, so that its correct contribution to a pixel is $\alpha_i c_i$ modulated by the transparency of the triangles in front of it. We introduce two running examples.

Example 1. Consider a pure red ($c_0 = (1, 0, 0)$) triangle in front of a pure green ($c_1 = (0, 1, 0)$) triangle, each with an opacity of 0.45 ($\alpha_0 = \alpha_1 = 0.45$). If the background is pure blue ($c_b = (0, 0, 1)$) then the resulting pixel color is:

$$c = \alpha c_0 + (1 - \alpha)\alpha c_1 + (1 - \alpha)^2 c_b = (0.45, 0.2475, 0.3025),$$

which corresponds to the "over" operator as defined by Porter and Duff [1984]. This simple artificial example isolates contributions from each triangle and the background into separate color channels.

Example 2. Consider N triangles each with $\alpha = 1/N$ and color c_f. The resulting pixel color is:

$$c = (1 - \alpha)^N c_b + \sum_{i=0}^{N-1} (1 - \alpha)^i \alpha c_f.$$

Here we consider a 25% grey background ($c_b = (0.25, 0.25, 0.25)$) with $N = 4$ triangles. This example resembles the situation when multiple thin, softly lit, partially transparent "smoke blobbies" overlap in screen space, and helps illustrate the inherent complexity of transparency: the color contributed by a triangle depends on α of all closer triangles at that pixel.

Consider an ordinary z-buffer where each pixel in the frame buffer consists of a number of samples S. During rasterization, a fragment *covers* some or all of those samples. Each sample retains the depth and color of the front-most fragment that has covered it. For our purposes we want to encode transparency for a fragment using the percentage of samples we enable during rasterization, so we begin by considering all S samples to be located at the center of the pixel. In this case, an opaque fragment will cover all the samples, or none.

A transparent fragment will cover a stochastic subset of R samples, resulting in an effective $\alpha = R/S$. Each sample does ordinary z-buffer comparisons, retaining the front-most fragment it sees, so this is a "winner take all" process. The expected value of the alpha and color for the fragment is correct provided the probability a sample is covered is equal to α. For example, if we make these opacity choices independently for $S = 4$, then for one of the fragments in Example 1 with $\alpha = 0.45$, R might take on any value of $0, 1, 2, 3, 4$ but on average it should be $0.45S = 1.8$.

For a fragment that is not in front, its effective α is the product of its own α and $(1 - \alpha_i)$ for all triangles i in front of this fragment. The expected product of two random variables is the product of the expected values, if the variables are uncorrelated. Thus using stochastic subsets for each triangle works correctly, provided that our method for selecting those subsets ensures no correlation. There will of course be some variance, i.e., random noise, around the correct value. Reducing that noise can be achieved by increasing the number of samples S, but for a given triangle this will have the classic Monte Carlo problem of *diminishing returns* where halving the average error requires quadrupling S.

The standard way to attack diminishing returns is to use *stratified sampling*. That can be accomplished here by setting the R samples as a group rather than independently. One approach is to first set R:

$$R_i = \lfloor \alpha_i S + \xi \rfloor$$

where ξ is a "canonical" random number (uniform in $[0, 1)$). For example, with $S = 4$ and $\alpha_0 = 0.45$, we have a 20% chance of using $R_0 = 1$ and an 80% chance of using $R_0 = 2$. (Or R could be dithered across a tile, for further stratification.) We then choose a random subset of R samples. This allows the error for one fragment to decrease linearly, thus avoiding diminishing returns. All the images in this paper use stratified sampling. See Figure 2.

Put differently, naive sampling flips an α-weighted coin for each sample, while stratified sampling randomly selects one of the S-choose-R possible subsets of samples. We illustrate the practical impact of this difference with the two examples at the beginning of the section with $S = 4$ samples per pixel. For Example 1, the RMS error (standard deviation) rounded to hundredths per RGB channel is:

$$\sigma_{\text{naive}} = (0.25, 0.21, 0.23).$$

When stratified sampling is used, each of the triangles will have 1 or 2 samples representing them. For the front triangle all samples will count, while for the back triangle they will count only if not also covered by the front triangle. This introduces more stability:

$$\sigma_{\text{stratified}} = (0.10, 0.17, 0.18).$$

For Example 2 with $S = 4$, we have for each RGB channel:

$$\sigma_{\text{naive}} = 0.35,$$

$$\sigma_{\text{stratified}} = 0.12.$$

This example shows that although we do not have stratification between the fragments, we do still benefit significantly from the stratification within a pixel. The actual number is also interesting because the example resembles cloudy sprites on a dark background. An RMS error of 0.12 in this situation, while somewhat higher than practical, is approaching a usable level.

The benefits of the stratification in the "one opaque sample per fragment" example raises the question of whether in all cases stochastic transparency has error that decreases with $1/S$ and thus avoids the law of diminishing returns. Unfortunately this is not the case. Our example with S fragments each with 1 opaque sample becomes a version of the "balls and bins" problem (dropping M balls at random into S bins), where the number of empty bins determines alpha. The asymptotic variance in the number of empty bins increases linearly with M [Flajolet and Sedgewick 2009]. When $M = S$ (the "thin media" case) and we normalize by S to get our opacity, this implies variance decreases as $1/S$ and thus the RMS error decreases as $1/\sqrt{S}$. So the benefits of stratification for the thin media case comes in the constant factor. This behavior is similar in spirit to jittered sampling for opaque surfaces where the error varies between $O(1/\sqrt{S})$ and $O(1/S)$ depending on the details of the fragments in the pixel [Mitchell 1996].

A transparent shadow map can also be rendered by stochastically discarding fragments and storing only depth. The simplest case is one sample per pixel, at a high resolution. Percentage closer filtering then approximates the transparent shadow map value. Alternatively, several depths can be stored at each pixel. This can be viewed as a Deep Shadow Map [Lokovic and Veach 2000] that encodes each ray's visibility function using S depth values $z_1, z_2, ...z_S$. Visibility is approximated by

$$\text{vis}(z) \approx \text{count}(z \leq z_i)/S. \tag{1}$$

This is crude but compact and regular, and the z's need never be sorted.

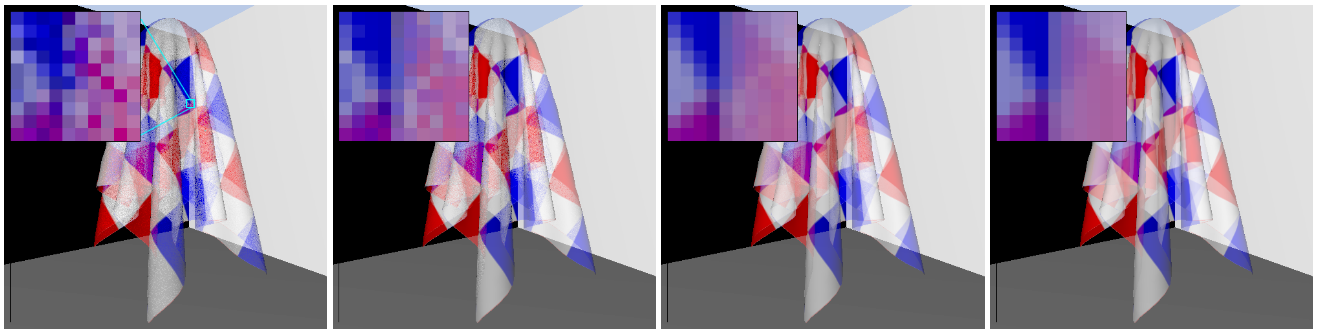

Figure 3: *Comparison of noise reduction methods. (a) Basic stochastic transparency has noise in all transparent regions. (b) Alpha correction eliminates noise in areas where all layers have similar colors. (c) Depth-sampled stochastic transparency is more accurate in complex regions. (d) Reference image from depth peeling. All images use 8 samples per pixel, no anti-aliasing, and an alpha of 40%.*

3.2 Alpha Correction

The exact (non-stochastic) total alpha of the transparent fragments covering a pixel is

$$\alpha_{total} = 1 - \prod (1 - \alpha_i) \tag{2}$$

Since this equation is independent of the order of the fragments, it can be evaluated in a single render pass, without sorting. Ordinary alpha blending leaves this result in the alpha channel. It is only the color channels that depend on proper depth ordering.

Following Sintorn and Assarsson [2009], we can use total alpha as a correction factor. We multiply our (stochastic) average color of samples by $\alpha_{total}/(R/S)$. For pixels with a single transparent layer, the result is now exact. Empirically, the error in multi-layer pixels is reduced overall, though for some pixels it is increased; see Figure 3. This technique does add bias, but in a way that vanishes as S increases, so it is technically a *consistent Monte Carlo* method [Arvo and Kirk 1990].

For our two examples from the last section, the effects of alpha correction are instructive. The RMS error for Example 1 changes as:

$$\sigma = (0.10, 0.17, 0.18) \rightarrow (0.17, 0.14, 0.0)$$

Note that the error associated with the first triangle goes up. This is to be expected because the contribution from the first visible triangle is optimally stratified and changes due to other triangles can easily make it worse. The background (blue) component error goes to zero as expected, and the back triangle's contribution is improved. In less pathological examples, the error in each component is a mixture of errors from all contributors, so as long as error is usually reduced per fragment it should improve per pixel. For the more naturalistic Example 2, the error goes to zero because the fragment colors are all the same:

$$\sigma = 0.12 \rightarrow 0.0$$

While this extreme case is unusual, whenever fragments have similar colors we will get some of this benefit.

Stochastic shadow maps may be alpha corrected by storing total alpha per pixel, and filtering it with the same kernel as the percentage closer filter. This eliminates quantization effects, particularly for shadows thrown onto opaque objects.

3.3 Depth-based Stochastic Transparency

One source of error in stochastic transparency is that each fragment is weighted only by how many of its samples are finally visible. With one additional order-independent render pass, we can improve these weights significantly. We first describe the formula for the weights in exact (non-stochastic) alpha blending, followed by our approximation.

Let $vis(z)$ be the visibility along a sample ray, that is, the proportion of the light that reaches depth z. Since each surface diminishes the light by one minus its alpha, visibility is a product of surface intersections:

$$\text{vis}(z) = \prod_{z_i < z} (1 - \alpha_i). \tag{3}$$

(This is α_{total} at the first opaque surface, and zero beyond that.) The standard alpha blending formula can then be rearranged to identify the contribution of each fragment to the final composited pixel:

$$\text{Final color} = \sum \text{vis}(z_i)\alpha_i c_i. \tag{4}$$

In words, each fragment contributes in proportion to its visibility and its opacity.

Given an oracle that estimates vis(z), we can estimate Equation 4 using a single additive render pass over all fragments, in any order. A transparency shadow map from the camera is such an oracle, a way of estimating vis(z). This is a pleasing duality: any transparency *shadow* method is also an order-independent transparency *rendering* method, using one extra accumulation pass. For example, this is how Sintorn et al. [2009] use Occupancy Maps for both shadows and rendering. They point out that it is important to alpha-correct the result by the ratio of the exact total alpha to the accumulated alpha, obtained by replacing c_i with 1 in Equation 4.

In our case, the oracle is a stochastic transparency buffer. Here is the resulting method for *depth-based stochastic transparency*, for samples that are all in the center of the pixel:

1. Render any opaque geometry and the background. All further steps treat only the transparent geometry, culling away fragments that are behind opaque geometry.

2. Render total alpha for each pixel into a separate buffer. (One pass.)

3. Render stochastic transparency with S samples per camera pixel, storing only z. (One pass, multi-sampled.)

4. Accumulate fragment colors (including alpha to be used for alpha correction) by Equation 4, with visibility estimated by Equation 1. The shader reads all S of the z values for the pixel from the output texture of the previous step, and compares them with the current fragment's z. (One pass.)

5. Composite that sum of fragments, times the alpha correction, over the opaque image.

Note that Step 1 in this algorithm is an optimization (of both speed and quality) and that the algorithm holds for $\alpha = 1$ as well.

This is almost always significantly less noisy and more accurate than the basic stochastic transparency algorithm; see Figure 3. One reason is that basic stochastic transparency effectively quantizes alpha to multiples of $1/S$, while depth-based stochastic transparency quantizes visibility but not alpha. Another reason is that the basic method collects only fragments that "win" and remain in the z-buffer for one or more samples, while in the depth-based method, all fragments contribute.

Using the depth-only pass without alpha correction reduces the RMS error for the red and green triangles of Example 1:

$$\sigma = (0.1, 0.17, 0.18) \rightarrow (0.0, 0.05, 0.18)$$

Note that the background's contribution (blue) does not change. The errors for the other channels are so low that in this case alpha correction will likely do more harm than good, which is indeed the case:

$$\sigma = (0.0, 0.05, 0.18) \rightarrow (0.14, 0.12, 0.0)$$

While the example suggests that for some cases we may not want to apply alpha correction, we have found it essential for avoiding artifacts in all scenes we have tried.

For Example 2, the RMS error drops using only the depth-only test, and adding alpha correction again drives the error to zero because the fragments are all the same color:

$$\sigma = 0.12 \rightarrow 0.08 \rightarrow 0$$

3.4 Spatial Anti-aliasing

We now consider samples that are distributed spatially over the pixel. The resulting algorithms are surprisingly simple.

We use the multi-sample anti-aliasing (MSAA) modes supported by current hardware graphics pipelines, where each pixel contains several (currently up to 8) samples at different positions within the area of the pixel, and each rasterized fragment carries a coverage mask indicating which of these samples lie inside it. This is efficient because shading is per fragment while visibility (z-buffer comparisons) are per sample. For display, the pixel's samples are averaged (in hardware), or a better filter such as a 2x2 Gaussian filter can be applied (in software). The algorithms below can work with any filter.

These spatial anti-aliasing samples can be used as our transparency samples as well. (Philosophically, each transparent surface can be considered one more dimension for distributed ray tracing [Cook et al. 1984].) Current hardware already supports an alpha to coverage (a2c) mode which does this. It *and*s the coverage mask with a screen door mask computed from alpha. Unfortunately the screen door mask is always the same; samples are added to it in a specified order as alpha is increased. The key difference between current a2c and our stochastic transparency approach is that we ensure masks are uncorrelated between samples in the same pixel.

There is one subtlety. For alpha correction to produce anti-aliased edges of transparent surfaces, total alpha must be rendered with multi-sampling. For alpha correction of the transparent geometry, only the filtered (per pixel) total alpha is needed. But for compositing the background pass, the background must be multiplied by $(1 - \alpha_{\text{total}})$ sample by sample. This is chiefly in case there is an edge in the background pixel that corresponds to the edge of the transparent material, which commonly happens at the edges of a window or the silhouette of skin with hairs behind it.

The basic algorithm with alpha correction is this:

1. Render the opaque background into a multi-sampled z-buffer.
2. Render total alpha into a separate multi-sampled buffer. (One pass.)
3. Render the transparent primitives into a separate multi-sampled buffer, culling by the opaque z-buffer, and discarding samples by stochastic alpha-to-coverage. (One pass.)
4. Compositing passes: First dim (multiply) the opaque background by one minus total alpha at each sample. Then read the filtered transparent color, correct to the filtered total alpha, and blend over the filtered, dimmed background.

In the depth-based algorithm, the shader no longer needs to compare z to all S of the z_i values in the transparency map, because we can use the multi-sampled z-buffer hardware to do this. We use the z-buffer to accumulate each fragment into only those samples where $z \leq z_i$. When the accumulated samples are averaged for display, the effect is that the fragment is multiplied by Equation 1.

The final anti-aliased depth-based algorithm is this:

1. Render the opaque background into a multi-sampled z-buffer.
2. Render total alpha into a separate multi-sampled buffer. (One pass.)
3. Render the transparent primitives into the opaque z-buffer, discarding samples by stochastic alpha-to-coverage, and storing only z. (One pass.)
4. Accumulation pass: Render the transparent primitives in additive blending mode, into a separate multi-sampled color buffer, comparing against the combined z-buffer from the previous step. Starting with black, add $\alpha_i c_i$ to all samples where $z_{\text{fragment}} \leq z_{\text{buffer}}$. (One pass.)
5. Compositing passes: Dim the opaque background by one minus total alpha at each sample. Then read the filtered accumulated sum, correct to the filtered total alpha, and blend over the filtered, dimmed background.

In all, we use three passes over the transparent geometry. To get more than 8 samples per pixel (on current hardware), the algorithm can be iterated with different random seeds and the results averaged.[1] Since true alpha is independent of the random variables, only two extra passes are needed for each additional 8 samples, for a total of $1 + 2(S/8)$ passes. An additional computational cost is that shadow maps may be larger, and ideally they are re-rendered with each new seed as well.

4 Results

Our final anti-aliased depth-based algorithm can be implemented on recent GPUs that support programmable fragment coverage output, supported since DirectX 10.1 and OpenGL shader model 4.1 (we use the `ARB_sample_shading` extension [Balci et al. 2009]). All timings were measured on a pre-release version of an NVIDIA DirectX 11-capable GPU, code named "Fermi." Timings were largely independent of CPU speed.

[1]The average of two 8-sample images is noisier than one 16-sample image, unless extra care is taken to split one stratified 16-bit sample mask per pixel across both images.

Current hardware supports a maximum of only 8 samples per pixel (8x MSAA). To simulate more samples, we average together multiple passes with different random seeds. Finding an S-choose-R mask is accelerated by a precomputed look-up table, indexed by alpha in one dimension and a pseudo-random seed in the other. This quantizes alpha to approximately 10 bits, acceptable in practice. For modest S, the table can include all possible masks; for 8 samples, S-choose-R is at most 70, and even for 16 samples, it is at most 12,870. We use instead a fixed table width of 2,047 masks.

We have tested our implementation on a wide variety of semi-transparent geometry including hair, smoke, alpha-mapped foliage, sheer cloth, and a CAD model, as well as on a scene combining several of these. Table 1 shows the execution time of each of the render passes of the anti-aliased depth-based stochastic transparency algorithm, for three of the scenes, each rendered at 500x500 with 8 samples per pixel and 16 samples from a stochastic shadow map of total size either 2048x2048 or 4096x4096. (The latter is rendered with supersampling rather than MSAA.) We also include for each scene a baseline time equal to the time required for one MSAA pass over all visible transparent geometry with full shading (including shadow look-ups) and alpha blending. This is a plausible lower bound on the render time for anti-aliased transparency with any algorithm – the time to draw the transparent geometry if it were already sorted. Each render pass is shown in milliseconds and as a multiple of this baseline.

Our algorithm's run time is between 3 and 4 times the baseline, whether the depth complexity is modest (the motor, Figure 2) or very high (the windy hill, Figure 1). Indeed, each of the four passes over the transparent geometry is of a similar order as the base time. Of those four passes, only one executes the full surface shader to compute fragment color; the other three only need to run enough of the shader to compute fragment alpha. Computing alpha may require a texture look-up, but it will not typically involve shadow look-ups and filtering, for instance. Two of the passes are actually stochastic, meaning they use randomly selected sample masks, while the other two render each transparent fragment over the whole pixel. (In the case of the accumulation pass, many of the samples are culled by the stochastic z-buffer.) Note that our scenes are dominated by transparent geometry; in practice the increased cost of stochastic transparency will be less for scenes containing significant opaque geometry.

Comparison to depth peeling. Depth peeling can produce perfect results for visibility but requires rendering the scene $O(D)$ times for depth complexity D. For the motor scene, 10 peels are enough to convey the content, as the last few layers are mostly obscured. Our straight-forward depth-peeling implementation took 1.5 times longer to draw 10 layers – without MSAA but also without noise – as the stochastic transparency renderer took to draw all layers – with MSAA and with noise. As scene complexity increases, depth peeling becomes quadratic in the number of primitives P ($O(P)$ passes with $O(P)$ primitives, assuming depth complexity scales linearly with P) while stochastic transparency remains linear with P. For complex scenes the advantage skyrockets; see Figure 4. Furthermore, as the view or scene moves, D can vary, with strong influence on depth peeling render times. Stability of run time is a major strength of stochastic transparency.

Comparison to specialized approaches. For smoke, depth peeling is inappropriate, but sorting the particles on the GPU can be very fast [Sintorn and Assarsson 2008; Cohen et al. 2009]. Furthermore, geometric anti-aliasing can be replaced by shader anti-aliasing. While a specialized smoke renderer will be faster than stochastic transparency, the flexibility of our approach means developers can avoid multiple transparency implementations for different scenarios. This flexibility is another strength of stochastic transparency.

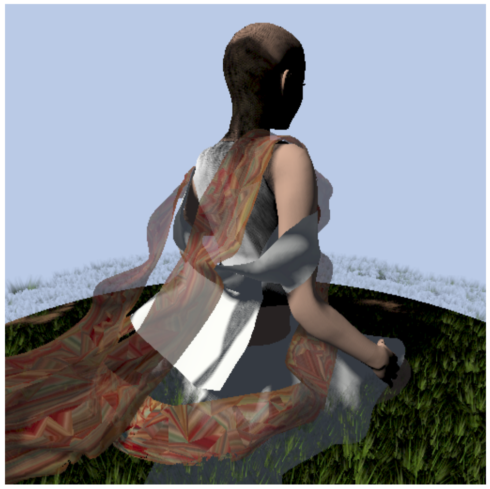
Figure 4: *In the time that Figure 1 was rendered with stochastic transparency, depth peeling could only draw 5 layers, with surreal results.*

Algorithms that sample opacity in regular slices [Kim and Neumann 2001; Sintorn and Assarsson 2009; Liu et al. 2009] have had impressive results, particularly for hair. Our randomized method resulted from an effort to overcome limitations of these algorithms with very uneven distributions of opacity. If for example two puffs of smoke are separated by a large empty space, it is difficult for a regular sampling approach to capture the variation of light within each puff. Unlike uniform slicing methods, stochastic transparency depends only on the depth order of fragments, and is insensitive to the actual depths. And unlike most bit-mask methods, it does not depend on all fragments having similar opacities; random sampling approaches any distribution in the limit.

Qualitative assessment. To our eyes, depth-based stochastic transparency with 8 samples is pleasant and undistracting in the detailed naturalistic scenes, while 16 samples is adequate, although not completely satisfying, for the less detailed cloth scene and the broad smooth surfaces of the CAD model. Recall that 16 samples requires only 5 passes. For basic stochastic transparency without alpha correction, 64 samples seems like a minimum. This may seem prohibitive today, but if graphics hardware exploits Moore's Law for ever-increasing hardware MSAA sample rates, the basic algorithm may win out by (in Kurt Akeley's phrase) "the elegance of brute force."

5 Limitations and Extensions

Temporal coherence. When we seed the pseudorandom number generator that selects the mask, we include pixel (x,y) in the seed to get variation across the screen, but we also need independent masks for each fragment within a pixel. Seeding by fragment z is effective for still images, but causes the noise pattern of moving objects to change completely each frame. Seeding by primitive ID (a different number for each triangle, line, or point drawn) is much more stable. Where noise is visible, primitives appear to move through a noise field fixed in screen space. The drawback is that the noise field changes at triangle boundaries, and these boundaries are occasionally visible during motion (see video). An alternative is to seed by an application-defined object ID. However if a curved object presents several fragments to a pixel, these fragments' masks

			Motor		Dog		WindyHill	
	stochastic	shading	ms	rel	ms	rel	ms	rel
Base	no	full	3.6	1.00	14.7	1.00	9.5	1.00
shadow	yes	full	-	-	8.9	0.61	13.1	1.39
opaque			2.6	0.71	1.2	0.08	2.8	0.30
total alpha	no	alpha	3.3	0.91	8.6	0.58	6.9	0.73
stoch depth	yes	alpha	2.7	0.75	8.7	0.59	7.0	0.74
accumulation	no	full	3.4	0.93	9.5	0.64	6.9	0.73
Other			1.6	0.44	1.5	0.1	1.7	0.18
Total			13.6	3.74	38.4	2.61	38.4	4.05
			73 FPS		26 FPS		26 FPS	

Table 1: *Run times of the phases of the final algorithm for Figures 2, 5, and 1.*

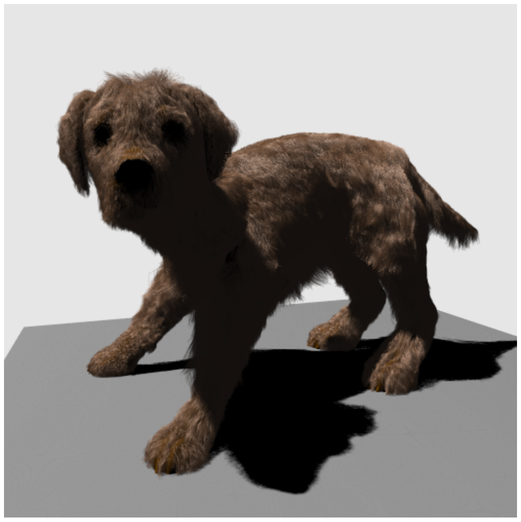

Figure 5: *A dog with 300K hairs. 26 FPS.*

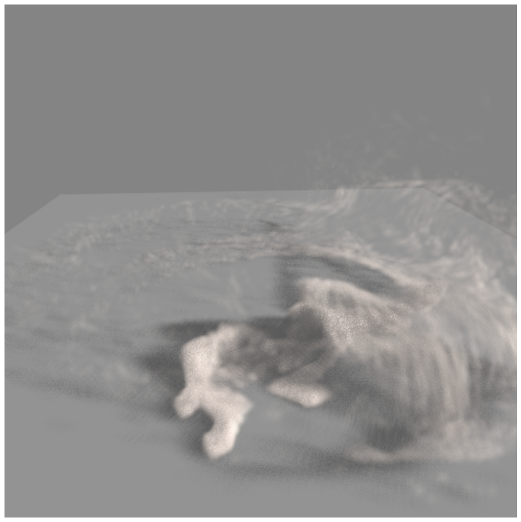

Figure 6: *A smoke plume with 100K particles. 38 FPS.*

will be identical, and they will occlude each other rather than blend. An extra seed bit for front-facing versus back-facing triangles will avoid this in the simple cylindrical case, but not in general. Typically however this method yields satisfactory results with few animation artifacts, at the expense of some effort by the programmer.

Uneven noise between pixels. Our noise reduction methods makes some regions noisier than others, which may be objectionable. Stratified masks reduce noise in most pixels, but pixels containing edges see a mix of two masks, one from each contributing fragment, and so have increased variance. Under specific conditions, with matching and somewhat flat shading on both sides of the boundary, an object boundary can cause a visible stripe of noisier pixels even in a static image. Similarly, alpha correction eliminates noise from regions with just one transparent layer, giving them a different texture than adjacent regions. One option is to forgo stratified masks or alpha correction, making all pixels as noisy as the worst ones, and instead combat noise with many more samples.

Adaptive sampling. Another possibility is to adaptively render more passes for pixels showing larger variance. This would have the great advantage of a noise level that is uniform and under artist control. It would have the disadvantage of, like depth peeling, requiring more passes for more complex images. But the number of passes is likely to still be many fewer than for depth peeling.

Post processing. One approach to reducing stochastic noise is to post-process the image. The bilateral filter [Paris et al. 2007] is a popular way to reduce noise while preserving edges. To preserve edges from all visible layers of transparency, we experimented with cross-filtering the transparent image with the (non-stochastic) total alpha channel. This is inexpensive, and blurs out noise somewhat, but also blurs surface detail in the transparent layers. However it is possible that there are weighting functions that produce good results.

Mask selection. Ideally, we would also reduce the variance of the overlap of the masks chosen for different fragments in the same pixel, leaving just the right amount of correlation. This can be done with two known fragments [Mulder et al. 1998], but it is not clear how to do this in general, given that we choose each mask without knowledge of the other fragments in the pixel. Furthermore, a pair of random S-choose-R masks does reasonably well. For instance, with $S = 8$ and two fragments with $\alpha = 0.5$, total samples covered is within one sample of correct 97% of the time. It would seem difficult to improve on that significantly.

Leveraging Monte Carlo techniques. The stochastic transparency algorithm is equivalent to backwards Monte Carlo ray tracing, with no changes in ray direction. At each ray-surface intersection, Russian roulette decides whether the ray (the sample) is absorbed or transmitted. In this view, Equation 4 is merely the sum over the probability distribution of paths: $vis(z_i)$ is the probability of a ray reaching the fragment at z_i, and α_i is the probability of the ray stopping there. This explains why alpha correction is necessary in the depth-based algorithm: since $vis(z_i)$ is only approximate, the probabilities do not sum to one.

The stratification from Section 3.1 could be achieved by using stratified seeds between the rays. This raises the question of whether other Monte Carlo optimization techniques can apply, such as importance sampling. The relative variance is highest when we have light transparent surfaces over a dark background. Making S higher in pixels where the background is dark would be a promising start towards importance sampling.

6 Conclusion

Stochastic transparency using sub-pixel masks provides a natural implementation of order-independent transparency for both primary visibility and shadowing. It is inexact, but with the refinements presented in the paper, it produces pleasing results for low enough sampling rates that it is practical for interactive systems.

The resulting algorithm has a unique combination of desirable qualities. It uses a low, fixed number of render passes – on current hardware, three passes for 8 samples per pixel, or five passes for 16 samples. It uses a fairly high but fixed and predictable amount of memory, consisting of a couple of extra MSAA z-buffers. Its run time is fairly stable and linear with the number of fragments. It is unaffected by uneven spatial distribution of fragments, and responsive to uneven opacities among fragments. It is very simple to implement. And it provides a unified technique for all types of transparent geometry, able without coding for special cases to mix hair, smoke, foliage, windows, and transparent cloth in a single scene.

Stochastic transparency provides a unified approach to order independent transparency, anti-aliasing, and deep shadow maps. It is closely related to Monte Carlo ray tracing. Yet the algorithm does not branch and contains no read-modify-write loops other than traditional z-buffer blend operations. This makes it an excellent match for modern, massively parallel GPU hardware.

References

AKELEY, K. 1993. Reality engine graphics. In *Proceedings of SIGGRAPH*, 116.

ARVO, J., AND KIRK, D. 1990. Particle transport and image synthesis. In *Proceedings of SIGGRAPH*, 63–66.

BALCI, M., BOUDIER, P., BROWN, P., ROTH, G., SELLERS, G., AND WERNESS, E., 2009. Arb_sample_shading. OpenGL extension http://www.opengl.org/registry/specs/ARB/sample_shading.txt.

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

BAVOIL, L., CALLAHAN, S., LEFOHN, A., COMBA, J., AND SILVA, C. 2007. Multi-fragment effects on the GPU using the k-buffer. In *Symposium on Interactive 3D graphics and games*, 104.

CALLAHAN, S., COMBA, J., SHIRLEY, P., AND SILVA, C. 2005. Interactive rendering of large unstructured grids using dynamic level-of-detail. In *Proceedings of Visualization*, 199–206.

CARPENTER, L. 1984. The A-buffer, an antialiased hidden surface method. In *Proceedings of SIGGRAPH*, 103–108.

COHEN, J., TARIQ, S., AND GREEN, S., 2009. Real time 3d fluid and particle simulation and rendering. NVIDIA Demo, http://www.nvidia.com/object/cuda_home.html.

COOK, R., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *Proceedings of SIGGRAPH*, 137–145.

EISEMANN, E., AND DÉCORET, X. 2006. Fast scene voxelization and applications. In *SIGGRAPH Sketches*, 8.

EVERITT, C., 2001. Interactive order-independent transparency. NVIDIA white paper, http://developer.nvidia.com.

FLAJOLET, P., AND SEDGEWICK, R. 2009. *Analytic Combinatorics*. Cambridge University Press, Cambridge, UK.

FUCHS, H., GOLDFEATHER, J., HULTQUIST, J., SPACH, S., AUSTIN, J., BROOKS JR, F., EYLES, J., AND POULTON, J. 1985. Fast spheres, shadows, textures, transparencies, and image enhancements in Pixel-Planes. In *Proceedings of SIGGRAPH*, 111–120.

KIM, T., AND NEUMANN, U. 2001. Opacity shadow maps. In *Rendering Techniques 2001*, 177–182.

KIRCHER, S., AND LAWRANCE, A. 2009. Inferred lighting: fast dynamic lighting and shadows for opaque and translucent objects. In *Proceedings of the SIGGRAPH Symposium on Video Games*, 39–45.

LIU, B., WEI, L.-Y., AND XU, Y.-Q. 2006. Multi-layer depth peeling via fragment sort. Tech. rep., Microsoft Research.

LIU, F., HUANG, M., LIU, X., AND WU, E. 2009. Bucket depth peeling. In *Proceedings of High Performance Graphics*, 50.

LOKOVIC, T., AND VEACH, E. 2000. Deep shadow maps. In *Proceedings of SIGGRAPH*, 385–392.

MITCHELL, D. 1996. Consequences of stratified sampling in graphics. In *Proceedings of SIGGRAPH*, 277–280.

MORLEY, R. K., BOULOS, S., JOHNSON, J., EDWARDS, D., SHIRLEY, P., ASHIKHMIN, M., AND PREMOŽE, S. 2006. Image synthesis using adjoint photons. In *GI '06: Proceedings of Graphics Interface 2006*, 179–186.

MULDER, J., GROEN, F., AND VAN WIJK, J. 1998. Pixel masks for screen-door transparency. In *Proceedings of Visualization*, 351–358.

NEIDER, J., AND DAVIS, T. 1993. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

PARIS, S., KORNPROBST, P., TUMBLIN, J., AND DURAND, F. 2007. A gentle introduction to bilateral filtering and its applications. In *ACM SIGGRAPH 2007 courses*, 1.

PORTER, T., AND DUFF, T. 1984. Compositing digital images. In *Proceedings of SIGGRAPH*, 253–259.

SEN, O., CHEMUDUGUNTA, C., AND GOPI, M. 2003. Silhouette-opaque transparency rendering. In *Sixth IASTED International Conference on Computer Graphics and Imaging*, 153–158.

SINTORN, E., AND ASSARSSON, U. 2008. Real-time approximate sorting for self shadowing and transparency in hair rendering. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, 157–162.

SINTORN, E., AND ASSARSSON, 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.

YUKSEL, C., AND KEYSER, J. 2008. Deep opacity maps. In *Proceedings of EUROGRAPHICS*, 675–680.