Stochastic Transparency

Eric Enderton, Erik Sintorn, Peter Shirley, and David Luebke, *Member*, *IEEE Computer Society*

Abstract—Stochastic transparency provides a unified approach to order-independent transparency, antialiasing, 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.

Index Terms—Rendering, transparency, shadow maps, deep shadow maps, stochastic sampling.

1 Introduction

Interactive rendering of complex natural phenomena such as hair, smoke, or foliage may require many thousands of semitransparent elements. Architectural 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 semitransparent objects present an additional challenge. Such shadows are in fact a version of the OIT problem, as they require some data structure 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 multisample antialiasing (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 multisample z-buffer hardware since the early 1990s [1], although modern GPUs have introduced more flexible support.) Furthermore, the technique easily extends to transparency shadow maps.

Stochastic transparency extends screen-door transparency with randomly chosen subpixel stipple patterns. For example, a full-pixel fragment with an opacity of 50 percent will randomly cover half of the subpixel samples; for those samples, it is fully opaque. Conventional z-buffering does the rest. If another 50 percent opaque fragment is closer to the camera, then the first fragment's color may end up in only 25 percent 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 [32] to formulate *depth-based stochastic transparency*, which stores only z values during the stochastic render pass and yields more accurate transparent sampling. We also discuss connections to Monte Carlo ray tracing and explore the statistical properties of our algorithm and extensions. Our interactive implementation shows visually pleasing results at encouraging frame rates.

No fragments were sorted in the making of these pictures.

2 Prior Work

The *A-buffer* [9] achieves order-independent transparency by storing a list of transparent surfaces per pixel. It has long been a staple of offline 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 [7], [14], [20] uses dual depth comparisons per sample to extract and composite each semitransparent layer in a separate rendering pass. It 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

- *E. Enderton, P. Shirley, and D. Luebke are with NVIDIA Corp., 2701 San Tomas Expwy, Santa Clara, CA 95050. E-mail: {eenderton, pshirley, dluebke}@nvidia.com.*
- *E. Sintorn is with Chalmers University of Technology, S-412 96 Gothenburg, Sweden. E-mail: erik.sintorn@chalmers.se.*

Manuscript received 18 May 2010; accepted 19 July 2010; published online 29 Sept. 2010.
Recommended for acceptance by M. Menezes de Oliveira Neto and D. Aliaga. For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCGSI-2010-05-0105.
Digital Object Identifier no. 10.1109/TVCG.2010.123.

depth [21]; 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 [31], 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 [6].

Screen-door transparency replaces transparent surfaces with a set of pixels that are fully on or off [16]. The choice of stipple patterns can be optimized so they are more visually pleasing to viewers [26]. The idea of random stipple patterns per polygon has also been long known, and was dubbed *cheesy translucency* in the original OpenGL "red book" [27]. The subpixel 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 "subpixel" ideas together with efficient hardware multisampling, as well as recent GPU features that allow the fragment shader to discard individual samples [5].

Random subpixel screen-door transparency is exactly analogous to using Russian roulette to decide ray absorption during batch ray tracing [25]. As an object-level analogy to screen-door transparency, triangles can be dropped from a mesh in proportion to transparency [30], 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 [8]. Finally, recent deferred shading schemes [19] can combine a transparent object with opaque objects by using a stipple pattern in the G-buffer and 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* [22]. These, like the A-buffer, require a variable length list per sample while the map is constructed. For interactive graphics, several recent techniques collect per-pixel statistics in a small fixed number of passes using fixed space, sacrificing perfect accuracy in exchange for predictable performance. Our method continues the work in this category. *Deep opacity maps* [33] 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 [13], [32]. Both these methods use a uniform subdivision of the depth range, creating accuracy issues for scenes with highly nonuniform distributions of transparency. Fourier opacity maps [17] are more robust to nonuniform distributions, but, since they represent only low-frequency changes in visibility, they are chiefly effective for clouds and smoke and can have ringing artifacts for discrete transparent surfaces.

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 that uses a randomized subpixel version of the "screen-door" stipple pattern. As with most stochastic methods, the main problem is noise. Section 3.1 describes the basic method for rendering transparent objects, and shows that Monte Carlo stratification can be used to reduce the noise. Section 3.2 introduces the use of stochastic transparency for shadows of transparent objects. In Section 3.3, we describe how to apply alpha correction for noise reduction, and in Section 3.4, we describe a three-pass but still order-independent method to reduce noise further. In Section 3.5, we show how our methods can be implemented efficiently using the multisample antialiasing features on modern hardware. Section 3.6 returns to the topic of stochastic shadow maps, using the refinements in the previous sections. 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 $i\text{th}$ triangle has color c_i, opacity α_i, and depth z_i. The desired final color for a pixel is given by the Porter-Duff "over" operator [29]:

$$C = \alpha_1 c_1 + (1 - \alpha_1)(\alpha_2 c_2 + (1 - \alpha_2)(\alpha_3 c_3 + \cdots)), \qquad (1)$$

which can be rearranged as

$$C = \sum_i \left(\prod_{z_j < z_i} (1 - \alpha_j) \right) \alpha_i c_i, \qquad (2)$$

showing that a triangle's 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:

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

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 percent gray background ($c_b = (0.25, 0.25, 0.25)$) with $N = 4$ white triangles ($c_f = (1, 1, 1)$). 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 α of R/S. We choose these subsets such that the expected (i.e., average) value of R/S is the fragment α, or in other words, the probability that a sample is covered is α. For example, for one of the fragments in Example 1 with $S = 4$ and $\alpha = 0.45$ R might take on any value of 0, 1, 2, 3, or 4, but on average, it should be $0.45S = 1.8$.

Each sample does ordinary z-buffer comparisons, retaining the front-most fragment it sees, in a "winner take all" process. In this way, a fragment may lose some of its R samples to fragments in front of it. The fragment will end up occupying only the samples that belong to its subset and belong to the complement of the subset of each fragment in front of it. The accuracy of the method depends on those subsets being uncorrelated. If they are, then the expected size of that set intersection is simply the product of the expected set sizes (proportional to S). Restating this for a particular sample, the probability of fragment i occupying the sample in the end is the product of α_i and $(1 - \alpha_j)$ for all fragments j in front of this fragment. Since the pixel's displayed color is the average[1] of the sample color values, this probability is the expected contribution of c_i to the final pixel color. This expected contribution matches (2), proving that each sample is an unbiased estimator of the correct pixel color. This means that with increased samples per pixel, pixel colors approach the correct values.[2] It also means that, as in half-toning, a region of pixels will tend to average out to the correct color.

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 percent chance of using $R_0 = 1$ and an 80 percent chance of using $R_0 = 2$. (Or R could be dithered across a tile, for further stratification.) We then choose a random subset of size 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 Fig. 2).

1. In Section 3.5, a possibly weighted average.
2. Assuming the variance of R/S approaches zero. See Section 4.

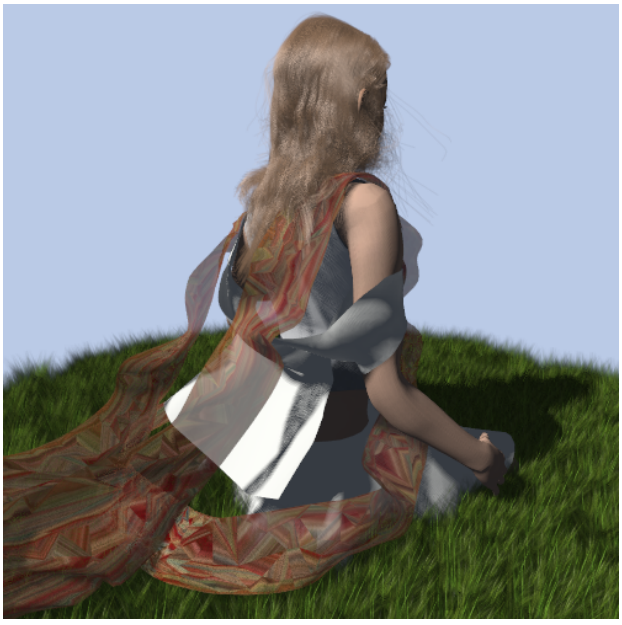


Fig. 1. 15,000 transparent hairs, 6,000 transparency-mapped cards, transparent cloth, and opaque models all composite properly and shadow each other, rendering at 26 frames per second with only four order-independent render passes over the transparent geometry. Note, for example, the grass blades visible behind and in front of the scarf.

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 one or two 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.

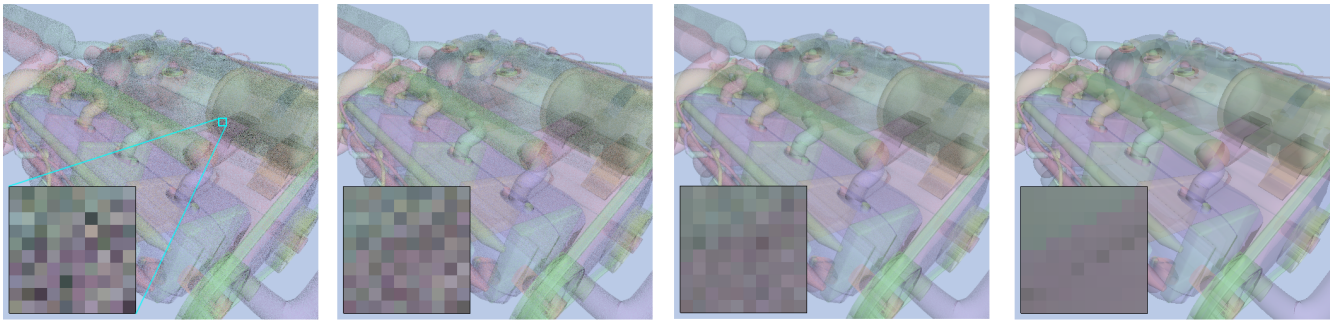


Fig. 2. Basic stochastic transparency with (a) 8, (b) 16, and (c) 64 samples per pixel, plus (d) a reference image from depth peeling. All images with no antialiasing, alpha 30 percent. Model contains 322K triangles.

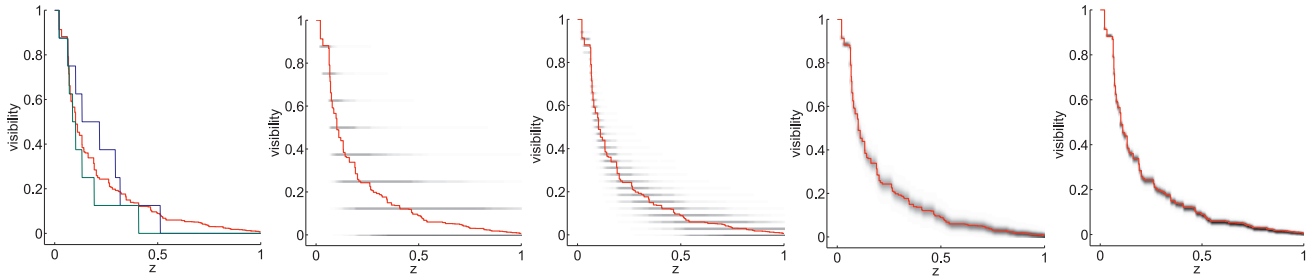


Fig. 3. The red line is the true visibility function $vis(z)$ for a random set of blockers. In the leftmost plot, two possible stochastic visibility functions $svis(z)$ are shown (in blue and green) when visibility is estimated with eight samples. In the remaining plots, hundreds of stochastic visibility functions have been superimposed to show the probability distribution for 8, 32, 256, and 1,024 samples.

The benefit of 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 one opaque sample, becomes a version of the "balls and bins" problem (dropping M balls at random into S bins), where the number of nonempty bins determines total alpha. The asymptotic variance in the number of empty bins increases linearly with M [15]. 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 benefit 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 [23].

3.2 Stochastic Shadow Maps

A transparent shadow map can be rendered in exactly the same way, by stochastically discarding fragments and storing only depth. The goal of any transparent shadow map technique is to estimate $\mathrm{vis}(z)$, the visibility along the ray (or beam) through a shadow map pixel. Put another way, $\mathrm{vis}(z)$ represents the proportion of the light that reaches depth z. Since each surface diminishes the light by one minus the alpha for that surface, the correct visibility is a product of surface intersections:

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

From the discussion in the previous section, it follows that the approximation of this function for some fragment at depth z is obtained by simply taking S samples from the stochastic shadow map within some small filter region and counting how many samples are closer than z:

$$\mathrm{svis}(z) = \mathrm{count}(z \leq z_i)/S \approx \mathrm{vis}(z). \tag{4}$$

In other words, the visibility at any depth is obtained from a stochastic shadow map by simple percentage closer filtering (PCF).

Fig. 3 illustrates that the approximation of svis converges to the true function as S goes to infinity. Contrasted with deep shadow maps [22], the approximation is crude—it is quantized to $1/S$, and must consider many samples for tight convergence—but it is compact, regular, and parallel-friendly. The z values need never be sorted. We discuss ways to improve the approximation in Section 3.6.

3.3 Alpha Correction

The correct (nonstochastic) total alpha of the transparent fragments covering a pixel is

$$\alpha_{total} = 1 - \prod (1 - \alpha_i). \tag{5}$$

(This is $1 - \mathrm{vis}(\infty)$, or one minus the contribution of the background color.) Since the product is independent of the order of the fragments, this equation 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 [32], we can use total alpha as a correction factor. We multiply our average color of stochastic samples by $\alpha_{total}/(R/S)$. For pixels with a

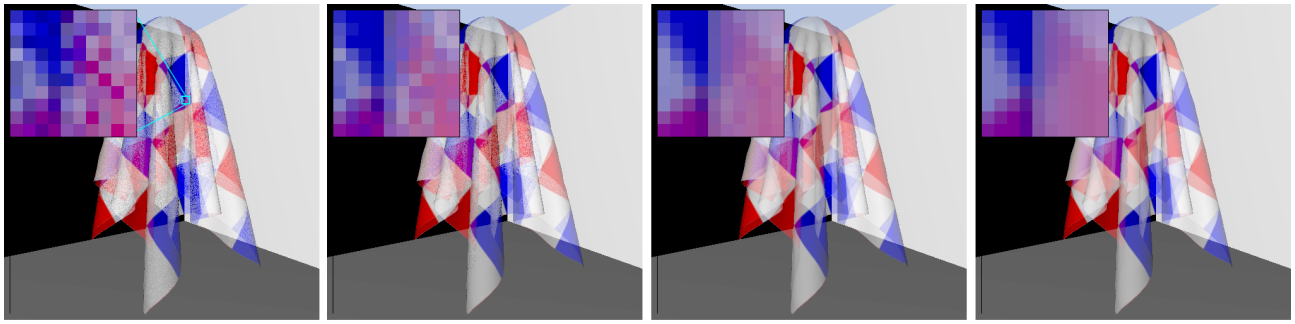


Fig. 4. Comparison of noise reduction methods. (a) Basic stochastic transparency has noise in all transparent regions. (b) Alpha correction eliminates noise in areas where all layers have similar colors. (c) Depth-sampled stochastic transparency is more accurate in complex regions. (d) Reference image from depth peeling. All images use eight samples per pixel, no antialiasing, and an alpha of 40 percent.

single transparent layer, the result is now exact. Empirically, the error in multilayer pixels is reduced overall, though for some pixels it is increased (see Fig. 4). This technique does add bias (see Section 4).

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.

3.4 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.

Looking at (2) and (3), we see that $\mathrm{vis}(z)$ is not only the light reaching depth z from a light, it is also the light reaching the camera from a fragment at depth z:

$$C = \sum \mathrm{vis}(z_i)\alpha_i c_i. \tag{6}$$

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

Given an oracle that estimates $\mathrm{vis}(z)$, we can estimate (6) 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 $\mathrm{vis}(z)$. 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. [32] 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 correct total alpha to the accumulated alpha, obtained by replacing c_i with 1 in (6).

In our case, the oracle is a stochastic depth 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 (6), with visibility estimated by (4). 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 Fig. 4). 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:

$$\sigma = (0.10, 0.17, 0.18) \rightarrow (0.0, 0.05, 0.18).$$

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:

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

3.5 Spatial Antialiasing

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

We use the multisample antialiasing (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) is per sample. For display, the pixel's samples are averaged (in hardware), or a better filter, such as a 2×2 Gaussian filter, can be applied (in software). The algorithms below can work with any linear filter.

These spatial antialiasing samples can be used as our transparency samples as well. (Philosophically, each transparent surface can be considered one more dimension for distributed ray tracing [11].) Current hardware already supports an alpha to coverage (a2c) mode that 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 antialiased edges of transparent surfaces, total alpha must be rendered with multisampling. 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 multisampled z-buffer.
2. Render total alpha into a separate multisampled buffer. (One pass.)
3. Render the transparent primitives into a separate multisampled 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 multisampled 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 (4).

The final antialiased depth-based algorithm is this:

1. Render the opaque background into a multisampled z-buffer.
2. Render total alpha into a separate multisampled 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 multisampled 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 eight samples per pixel (on current hardware), the algorithm can be iterated with different random seeds and the results averaged.[3] Since true alpha is independent of the random variables, only two extra passes are needed for each additional eight 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 rerendered with each new seed as well.

3.6 Stochastic Shadow Maps Revisited

We can build on the ideas in the preceding sections to improve our stochastic shadow mapping. Recall that the light's visibility for the fragment to be shaded is obtained by using percentage closer filtering on a stochastic shadow map. This map can be created by simply rendering a standard shadow map (one sample per pixel), but discarding fragments with probability α in the fragment shader. However, the benefits of stratified sampling apply here as well, and we can obtain a higher quality shadow map by rendering an MSAA z-buffer and discarding a stochastic set of samples, as described in Section 3.5. The shader will usually require at least one texture lookup to obtain the fragment's alpha and another lookup (or some significant per fragment computation) to obtain a random number. Since

3. 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.

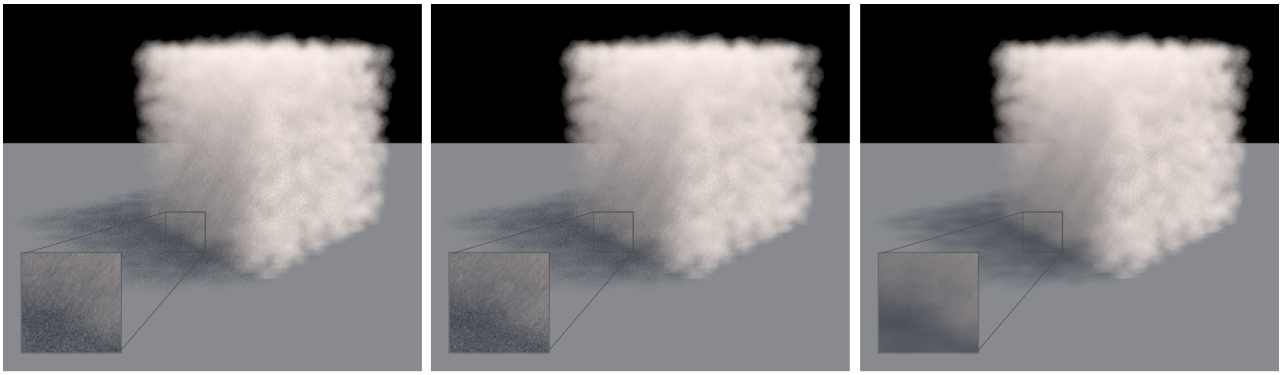


Fig. 5. Comparing noise reduction in stochastic shadow maps. All images are eight-sample MSAA. Shown are (a) naive sampling, (b) stratified sampling, and (c) stratified alpha corrected shadows.

MSAA shades only once per S samples, rendering an MSAA stochastic shadow map is faster (although shadow lookup can be slower). Since the MSAA render uses S-choose-R stratification, while a one-sample shadow map uses a separate coin toss per sample, the MSAA shadows have somewhat less noise (see Fig. 5).

Alpha correction (Section 3.3) can be applied to transparent shadows as well. By accumulating the total drop in visibility for each shadow map texel, dividing that by the estimated drop and using this as a correction factor for our estimated visibility at depth z, we have rescaled our visibility function to better fit the true visibility function, on average. However, the estimated visibility at a particular depth may actually be less accurate after correction. The total-alpha texture can also be used to obtain the correct transparent shadow for any opaque surface. Note that alpha correcting shadow maps require us to split the shadow maps into opaque and transparent, losing some generality. Fig. 5 shows a comparison of standard and alpha corrected shadows.

When soft or blurry shadows are acceptable, simply rendering a reasonably sized nonstratified stochastic shadow map and using a large PCF filter produces pleasing results. Unfortunately, unless a very large number of taps are taken (or shadows are alpha corrected) the resulting shadows will come out grainy and unconvincing. A number of papers have addressed the problem of prefiltering shadow maps to improve lookup performance, but of the ones we have considered (i.e., [12], [3], [2]) none trivially extend to stochastic shadow maps. Randomly choosing samples per fragment from a large filter (as suggested in [24]) improves visual appearance by trading blur for aliasing artifacts, as the effects of any one shadow sample are randomly distributed over a wide area (see for example Fig. 1).

4 Bias

We now discuss whether each of our rendering techniques is an *unbiased estimator*, meaning that the expected or average color of a pixel is the correct blended color, or if not, whether it is at least a *consistent* Monte Carlo method, meaning that the pixel color approaches the correct color in the limit as the number of samples goes to infinity [4].

For this section, we assume the input triangles are numbered in order of increasing depth $z_1 < z_2 < \cdots < z_N$, although they may still be drawn in any order. We recall (2) for the correct combined color of all transparent fragments

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

and note that their correct combined alpha is the same but with colors replaced by unity:

$$\alpha_C = \alpha_{total} = \sum_i \left(\prod_{z_j < z_i} (1 - \alpha_j) \right) \alpha_i.$$

Further, for any sample, let w be the index of the fragment that "wins," that is, the fragment whose depth and color are stored in that sample at the end of a stochastic rendering pass. We use the Kronecker delta δ_{wi} to be 1 when $w = i$ and 0 otherwise.

In the basic stochastic transparency method, described in Section 3.1, each sample's color is the color of the winning fragment,

$$B = \sum \delta_{wi} c_i.$$

We argue in that section that this is unbiased, that is, the expected value is correct:

$$E(B) = C.$$

Recall that we take S samples per pixel, and select a subset of R samples to be opaque for a given fragment. The assumptions we require are that the subsets (or *masks*) are chosen such that:

1. they are proportional, so on average $R = \alpha S$,
2. they are uncorrelated, so on average the intersection of subsets i and j has size $\alpha_i \alpha_j S$, and
3. they are improving, so that the variance of the previous two quantities goes to zero in the limit as S goes to infinity.

The first condition guarantees correctness for a single transparent layer, while the second condition guarantees it for two or more layers. In practice of course, randomness is never perfect, and if too weak a random number generator (or too small a table) is used, bias or increased noise can result.

4.1 Bias of Alpha Correction

Let B_1 be the occupancy of a sample, equal to one if the sample contains a fragment, and zero otherwise. Then, B_1 is analogous to B with the colors replaced by unity:

$$B_1 = \sum \delta_{wi}.$$

Let $\mathcal{F}$ be the downsampling filter operator that averages (possibly with a weighted average) its argument expression over all the samples in a pixel. In particular, if each sample contains the color value B, then the pixel will be displayed with color $\mathcal{F}(B)$. Then, the alpha-corrected pixel value is given by

$$A = \begin{cases} \alpha_C \mathcal{F}(B)/\mathcal{F}(B_1), & \text{if } \mathcal{F}(B_1) > 0, \\ 0, & \text{otherwise.} \end{cases}$$

The "otherwise" hints at a problem for convergence of this method. Consider a pixel that contains transparent fragments (so that $\alpha_C > 0$) and yet whose samples are all empty, due to bad luck with stochastic sampling. Such an empty pixel demonstrates a failure case for alpha correction, as there is no color value to scale up to the desired total alpha. Indeed, we find that for a single transparent layer with $\alpha = \frac{1}{2S}$, half the pixels have a single sample occupied while half the pixels are empty, and the average pixel value is

$$E(A) = \alpha c/2,$$

half the value it should be. This counterexample shows that alpha correction is, in general, neither unbiased nor consistent.

How does A behave in images without empty pixels of this kind? For instance, if every pixel has a fragment with $\alpha \geq 1/S$, then stratified sampling guarantees the pixels are nonempty. For such images, A is consistent, by the same limit-of-quotient argument we give next for the depth-based method. However, A is still biased. This is easily demonstrated using a two-fragment example. For example, suppose $N = 2$ and all $\alpha_i = 1/S$. Then each fragment occupies exactly one sample. With probability $1/S$, these will be the same sample, in which case

$$\begin{aligned} \mathcal{F}(B) &= c_1/S, \\ \mathcal{F}(B_1) &= 1/S, \\ A &= \alpha_C c_1. \end{aligned}$$

The rest of the time, two samples are occupied, and

$$\begin{aligned} \mathcal{F}(B) &= (c_1 + c_2)/S, \\ \mathcal{F}(B_1) &= 2/S, \\ A &= \alpha_C (c_1 + c_2)/2. \end{aligned}$$

Multiplying out the expected value of A, we find it does not match C. For $S = 4$, $c_1 =$ red, and $c_2 =$ green, we calculate

$$\begin{aligned} E(A) &\approx (0.273, 0.164, 0) \text{ while} \\ C &= (0.25, 0.1875, 0), \end{aligned}$$

so the front fragment is biased about 10 percent too high, while the back fragment is biased about 10 percent too low.

4.2 Bias of Depth-Based Methods

Uncorrected depth-based stochastic transparency sums the colors of all fragments, each weighted by its alpha and its stochastically estimated visibility:

$$U = \sum \text{svis}(z_i)\alpha_i c_i.$$

For the corrected depth-based method, the numerator of the correction factor is again α_C (total alpha), while the denominator is again the uncorrected value with all colors replaced by unity:

$$U_1 = \sum \text{svis}(z_i)\alpha_i.$$

The division is done after downsampling,[4] yielding final pixels as

$$D = \begin{cases} \alpha_C \mathcal{F}(U)/\mathcal{F}(U_1), & \text{if } \mathcal{F}(U_1) > 0, \\ 0, & \text{otherwise.} \end{cases}$$

The key to analyzing this is to consider the visibility estimate of a single sample. If fragment w wins the sample, then fragments in front of z_w are fully visible, while fragments behind z_w have no visibility, and (4) reduces to

$$\text{svis}(z_i) = \begin{cases} 1, & \text{for } i \leq w, \\ 0, & \text{for } i > w. \end{cases}$$

For an empty sample, we consider w to be $N + 1$ and z_{N+1} to be large. In particular, $w \geq 1$ always, and so fragment 1 is always visible. This means that $\mathcal{F}(U_1)$ cannot be zero unless the pixel has no fragments, and so the "otherwise" case for D does not cause the problems that it does for A.

For example, consider one sample with six fragments where fragment 3 wins ($S = 1, N = 6, w = 3$). Then the uncorrected color, correction denominator, and corrected color are simply

$$\begin{aligned} U &= \alpha_1 c_1 + \alpha_2 c_2 + \alpha_3 c_3, \\ U_1 &= \alpha_1 + \alpha_2 + \alpha_3, \\ D &= \alpha_C (\alpha_1 c_1 + \alpha_2 c_2 + \alpha_3 c_3)/(\alpha_1 + \alpha_2 + \alpha_3). \end{aligned}$$

In other words, depth-based sample color is a weighted average of fragment colors, scaled by total alpha. The weights ignore the depth order—everyone contributes based on their own alpha—regardless of the alpha of fragments in front of them. For instance, c_2's relative weight is α_2, when it should correctly be $(1 - \alpha_1)\alpha_2$, which is lower. However, the sums include only fragments 1 through w. Fragments further from the camera are weighted too highly, but are less likely to contribute at all.

Looking at the uncorrected color U and comparing it to the correct color C,

$$\begin{aligned} C = \alpha_1 c_1 &+ (1 - \alpha_1)\alpha_2 c_2 + \cdots \\ &+ (1 - \alpha_1)(1 - \alpha_2) \cdots (1 - \alpha_5)\alpha_6 c_6, \end{aligned}$$

we see that c_1's contribution is correct, while c_2's contribution is too high. But c_1 is included in every sample, regardless of which fragment wins, whereas c_2 is included

4. When implementing $S > 8$ by combining multiple hardware render passes, there are two things to be aware of. First, when averaging across passes, it is best to average U and U_1 separately and divide at the end. Second, results are slightly improved if the stochastic masks are stratified across all passes, rather than across each 8 samples separately.

TABLE 1
Run Times of the Phases of the Final Algorithm for Figs. 2, 7, and 1

	stochastic	shading	Motor		Dog		WindyHill	
			ms	rel	ms	rel	ms	rel
Base	no	full	3.6	1.00	14.7	1.00	9.5	1.00
shadow	yes	alpha	-	-	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	

only $(1-\alpha_1)$ of the time, so its expected contribution is correct. In this way, we prove that the expected value of U is in fact C, that is, U is consistent and unbiased.

Alas, U can easily be greater than one (white), if the alphas add up to more than one. Since the displayed U is clamped to $\max(U,1)$, which is in general less than U, the image is then too dark. In practice, the displayed U is *not* consistent.

On the other hand, the corrected color D is equal to α_C times a convex combination of colors. That guarantees each color channel is no more than α_C, so it will not be clamped by the display. D is also likely to be less noisy, because it has less dynamic range, and for the following reason. Consider the correction factor $D/U = \alpha_C/U_1$ as a function of w. For any multilayered pixel, if $w = 1$, then

$$U_1 = \alpha_1 < \alpha_C$$

and so $D/U > 1$. If $w = N$, then

$$U_1 = \sum \alpha_i > \alpha_C$$

and $D/U < 1$. Furthermore, it is monotonic in w. So, for pixels where w is low for most samples, the samples will include fewer fragments, but they will be scaled up, whereas for pixels where w is mostly high, the samples will include many fragments, but they will be scaled down.

Does D converge to the correct image, as samples increase? We have just proved that $E(U) = C$. Therefore,

$$\lim_{S\to\infty} \mathcal{F}(U) = C.$$

The same proof with $c_i = 1$ shows that

$$E(U_1) = \alpha_C,$$

and therefore,

$$\lim_{S\to\infty} \mathcal{F}(U_1) = \alpha_C.$$

Since $\alpha_C > 0$ (in pixels of interest), the limit of the quotient is the quotient of the limits, and

$$\begin{aligned}\lim_{S\to\infty} D &= \alpha_C \lim_{S\to\infty} \mathcal{F}(U) / \lim_{S\to\infty} \mathcal{F}(U_1) \\ &= \alpha_C C/\alpha_C \\ &= C,\end{aligned}$$

that is, D is consistent.

However, D is biased. Since U is unbiased, and D weights the fragment colors differently than U does, D cannot also be unbiased. (For a small case, such as $N = 2$, it is straightforward to expand the weights of c_1 and c_2 in $E(D)$, and see that they differ from C.) An interesting topic for future research would be to analyze and bound the bias as a function of S.

5 Results

Our final antialiased 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 [5]). All timings were measured on a prerelease version of an NVIDIA DirectX 11-capable GPU, code named "Fermi." Timings were largely independent of CPU speed.

Current hardware supports a maximum of only eight 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 lookup table, indexed by alpha in one dimension and a pseudorandom 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 eight 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 semitransparent 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 antialiased depth-based stochastic transparency algorithm, for three of the scenes, each rendered at 500×500 with eight samples per pixel and 16 samples from a stochastic shadow map of total size either $2{,}048 \times 2{,}048$ or $4{,}096 \times 4{,}096$. (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 lookups) and alpha blending. This is a plausible lower bound on the render time for antialiased 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 three and four times the baseline, whether the depth complexity is modest (the motor, Fig. 2) or very high (the windy hill, Fig. 1). Indeed, each of the four passes over the transparent geometry is of a

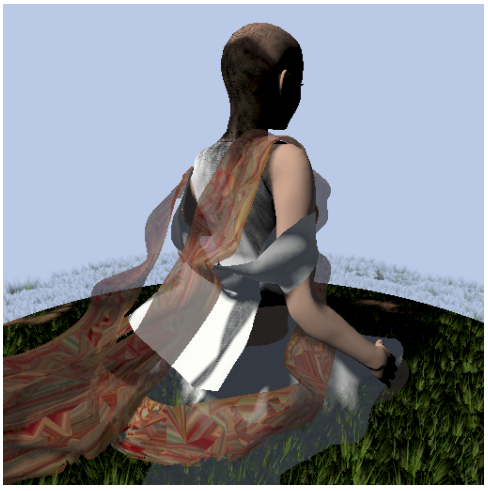


Fig. 6. In the time that Fig. 1 was rendered with stochastic transparency, depth peeling could only draw five layers, with surreal results.

Fig. 7. A dog with 300K hairs. 26 FPS.

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 lookup, but it will not typically involve shadow lookups 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 test 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 Fig. 6). 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 [10], [31]. Furthermore, geometric antialiasing can be replaced by shader antialiasing. 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 (see Fig. 8).

Algorithms that sample opacity in regular slices [18], [21], [32] 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 bitmask 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 eight samples is pleasant and undistracting in the detailed naturalistic scenes (Figs. 1 and 7), 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 require only five passes. For basic stochastic transparency without alpha correction, 64 samples seem 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."

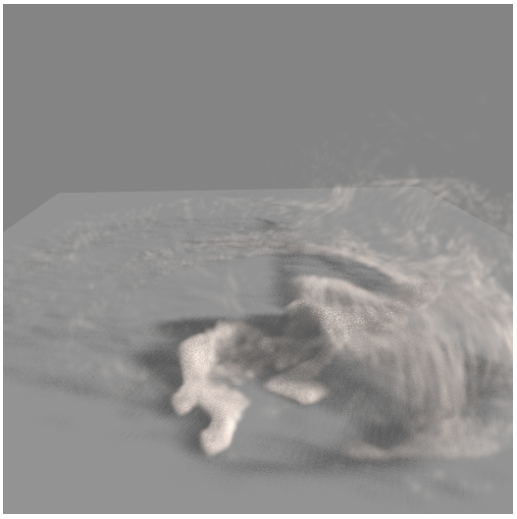


Fig. 8. A smoke plume with 100K particles. 38 FPS.

6 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. 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 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 make 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's 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 fewer than for depth peeling.

Post processing. One approach to reducing stochastic noise is to postprocess the image. The bilateral filter [28] is a popular way to reduce noise while preserving edges. To preserve edges from all visible layers of transparency, we experimented with crossfiltering the transparent image with the (nonstochastic) 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 [26], 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 percent of the time. It would seem difficult to improve on that significantly.

Gamma. Most graphics pipelines use *gamma correction*, where images in both the frame buffer and image files have a discretized nonlinear representation for image colors. In the simplest case, the displayed intensity is $I = p^\gamma$, where $p \in [0, 1]$ is the stored pixel color and $\gamma \approx 2.2$. This nonlinearity gives rise to a concern that the signed error in our method might not average to zero in this nonlinear space, such as when a viewer's eye blurs together a small region of adjacent pixels. Fortunately, this is not the case. Like any renderer, we use linear colors in our calculations (e.g., c_i), including when averaging samples together to form the desired (linear) pixel value (call it d). And like any renderer, we should gamma-compress the desired pixel value before sending it to the frame buffer:

$$p = d^{1/\gamma}, \quad \text{so that} \quad I = (d^{1/\gamma})^\gamma = d.$$

Since averaging of samples occurs before the gamma compression, and averaging of pixels (by the viewer) occurs after the display's gamma expansion, the intermediate nonlinear representation does not disturb the linear nature of the error, and its expected value remains at zero.

Leveraging Monte Carlo techniques. The stochastic transparency algorithm is equivalent to backward 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, (6) 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.

7 Conclusion

Stochastic transparency using subpixel 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 eight 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. 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, antialiasing, 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

[1] K. Akeley, "Reality Engine Graphics," *Proc. ACM SIGGRAPH,* p. 116, 1993.

[2] T. Annen, T. Mertens, P. Bekaert, H.-P. Seidel, and J. Kautz, "Convolution Shadow Maps," *Proc. Eurographics Symp. Rendering Techniques,* pp. 51-60, 2007.

[3] T. Annen, T. Mertens, H.-P. Seidel, E. Flerackers, and J. Kautz, "Exponential Shadow Maps," *Proc. ACM Int'l. Conf. Graphics Interface (GI '08),* pp. 155-161, 2008.

[4] J. Arvo and D. Kirk, "Particle Transport and Image Synthesis," *Proc. ACM SIGGRAPH,* pp. 63-66, 1990.

[5] M. Balci, P. Boudier, P. Brown, G. Roth, G. Sellers, and E. Werness, "Arb_Sample_Shading," http://www.opengl.org/registry/specs/ARB/sample_shading.txt, 2009.

[6] L. Bavoil, S.P. Callahan, A. Lefohn, J.L.D. Comba, and C.T. Silva, "Multifragment Effects on the GPU Using the k-Buffer," *Proc. Symp. Interactive 3D Graphics and Games,* p. 104, 2007.

[7] L. Bavoil and K. Meyers, "Order Independent Transparency with Dual Depth Peeling," technical report, NVIDIA, Feb. 2008.

[8] S.P. Callahan, J.L.D. Comba, P. Shirley, and C.T. Silva, "Interactive Rendering of Large Unstructured Grids Using Dynamic Level-of-Detail," *Proc. Conf. Visualization,* pp. 199-206, 2005.

[9] L. Carpenter, "The A-Buffer, an Antialiased Hidden Surface Method," *ACM SIGGRAPH Computer Graphics,* vol. 18, no. 3, pp. 103-108, 1984.

[10] J. Cohen, S. Tariq, and S. Green, "Real Time 3d Fluid and Particle Simulation and Rendering," http://www.nvidia.com/object/cuda_home.html, 2009.

[11] R.L. Cook, T. Porter, and L. Carpenter, "Distributed Ray Tracing," *Proc. ACM SIGGRAPH,* pp. 137-145, 1984.

[12] W. Donnelly and A. Lauritzen, "Variance Shadow Maps," *Proc. 2006 Symp. Interactive 3D Graphics and Games,* pp. 161-165, 2006.

[13] E. Eisemann and X. Décoret, "Fast Scene Voxelization and Applications," *Proc. 2006 Symp. Interactive 3D Graphics and Games,* pp. 71-78, 2006.

[14] C. Everitt, "Interactive Order-Independent Transparency," white paper, NVIDIA, http://developer.nvidia.com, 2001.

[15] P. Flajolet and R. Sedgewick, *Analytic Combinatorics,* Cambridge Univ. Press, 2009.

[16] H. Fuchs, J. Goldfeather, J.P. Hultquist, S. Spach, J.D. Austin, F.P. Brooks Jr, J.G. Eyles, and J. Poulton, "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes," *Proc. ACM SIGGRAPH,* pp. 111-120, 1985.

[17] J. Jansen and L. Bavoil, "Fourier Opacity Mapping," *Proc. 2010 Symp. Interactive 3D Graphics and Games,* pp. 165-172, 2010.

[18] T.-Y. Kim and U. Neumann, "Opacity Shadow Maps," *Proc. Eurographics Symp. Rendering Techniques,* pp. 177-182, 2001.

[19] S. Kircher and A. Lawrance, "Inferred Lighting: Fast Dynamic Lighting and Shadows for Opaque and Translucent Objects," *Proc. ACM SIGGRAPH Symp. Video Games,* pp. 39-45, 2009.

[20] B. Liu, L.-Y. Wei, and Y.-Q. Xu, "Multilayer Depth Peeling via Fragment Sort," technical report, Microsoft Research, 2006.

[21] F. Liu, M.C. Huang, X.H. Liu, and E.H. Wu, "Bucket Depth Peeling," *Proc. Eurographics Conf. High Performance Graphics,* pp. 51-57, 2009.

[22] T. Lokovic and E. Veach, "Deep Shadow Maps," *Proc. ACM SIGGRAPH,* pp. 385-392, 2000.

[23] D.P. Mitchell, "Consequences of Stratified Sampling in Graphics," *Proc. ACM SIGGRAPH,* pp. 277-280, 1996.

[24] M. Mittring, "Finding Next Gen: Cryengine 2," *Proc. ACM SIGGRAPH Course,* pp. 97-121, 2007.

[25] R. Keith Morley, S. Boulos, J. Johnson, D. Edwards, P. Shirley, M. Ashikhmin, and S. Premože, "Image Synthesis Using Adjoint Photons," *Proc. Graphics Interface (GI '06),* pp. 179-186, 2006.

[26] J.D. Mulder, F.C.A. Groen, and J.J. Van Wijk, "Pixel Masks for Screen-Door Transparency," *Proc. Conf. Visualization,* pp. 351-358, 1998.

[27] J. Neider and T. Davis, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1,* Addison-Wesley Longman Publishing Co., Inc., 1993.

[28] S. Paris, P. Kornprobst, J. Tumblin, and F. Durand, "A Gentle Introduction to Bilateral Filtering and its Applications," *Proc. ACM SIGGRAPH Course,* p. 1, 2007.

[29] T. Porter and T. Duff, "Compositing Digital Images," *Proc. ACM SIGGRAPH,* pp. 253-259, 1984.

[30] O. Sen, C. Chemudugunta, and M. Gopi, "Silhouette-Opaque Transparency Rendering," *Proc. Sixth IASTED Int'l Conf. Computer Graphics and Imaging,* pp. 153-158, 2003.

[31] E. Sintorn and U. Assarsson, "Real-Time Approximate Sorting for Self Shadowing and Transparency in Hair Rendering," *Proc. 2008 Symp. Interactive 3D Graphics and Games,* pp. 157-162, 2008.

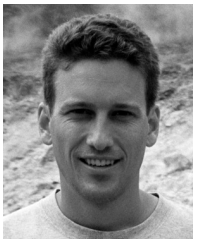
[32] E. Sintorn and Ulf Assarsson, "Hair Self Shadowing and Transparency Depth Ordering Using Occupancy Maps," *Proc. 2009 Symp. Interactive 3D Graphics and Games,* pp. 67-74, 2009.

[33] C. Yuksel and J. Keyser, "Deep Opacity Maps," *Computer Graphics Forum,* vol. 27, no. 2, pp. 675-680, 2008.

Eric Enderton received the master's degree in computer science from the University of California at Berkeley. He is a research scientist at NVIDIA whose interests include film effects for real-time rendering and GPU acceleration for offline rendering. Prior to joining NVIDIA in 2003, he developed rendering and animation software at Lucasfilm's Industrial Light & Magic and other studios.

Erik Sintorn received the MSc degree, in 2007, and is currently pursuing the PhD degree at the Department of Computer Engineering, Chalmers University of Technology in Gothenburg, Sweden. His main research interests include real-time shadows, transparency, and GPGPU computing. He has published several papers on these topics. He received a 2009 NVIDIA Graduate Fellowship award.

Peter Shirley received the BA degree in physics from Reed College, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign. He is a principal research scientist at NVIDIA, and adjunct professor in the School of Computing at the University of Utah. He is the coauthor of three books and dozens of technical articles. He spent four years as an assistant professor at Indiana University, and two years as a visiting assistant professor at the Cornell Program of Computer Graphics, before moving to Utah, where he was a professor of computer science for twelve years.



David Luebke received the PhD degree under Fred Brooks at the University of North Carolina, in 1998. David Luebke helped found NVIDIA Research, in 2006, after eight years on the faculty of the University of Virginia. His principal research interests include GPU computing and real-time computer graphics. His honors include the NVIDIA Distinguished Inventor award, the NSF CAREER and DOE Early Career PI awards, and the ACM Symposium on Interactive 3D Graphics "Test of Time Award." He has coauthored a book, a SIGGRAPH Electronic Theater piece, a major museum exhibit visited by more than 110,000 people, and dozens of papers, articles, chapters, and patents. He is a member of the IEEE Computer Society.