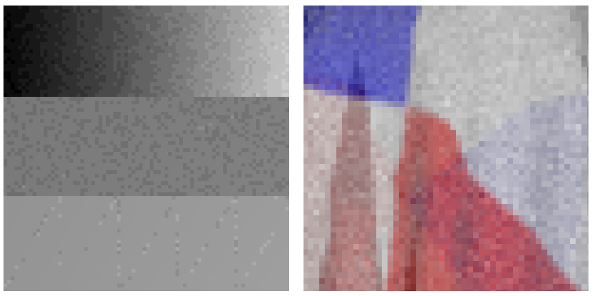
Eurographics Symposium on Rendering 2011
Ravi Ramamoorthi and Erik Reinhard
(Guest Editors)

Volume 30 (2011), Number 4

Stratified Sampling for Stochastic Transparency

Samuli Laine Tero Karras

NVIDIA Research


Previous work [ESSL10]

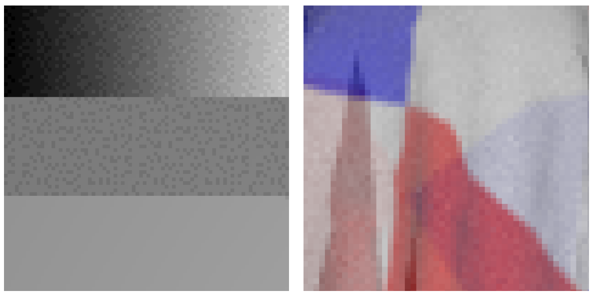

Our method

Figure 1: *Left of each pair: simple case with white primitives on top of black background, rendered using 16 samples per pixel. Top part is a quad with* α *ramp from 0 to 1, middle part is a quad with constant* $\alpha = 0.61$, *and bottom part is drawn as 5 quads with* $\alpha = 0.75$. *Right of each pair: cloth model with multiple transparent layers rendered using 64 samples per pixel. The original stochastic transparency algorithm has non-ideal spatial stratification (clumpy noise), problems at primitive edges, and its stratification properties degrade when there are overlapping layers. Our method dithers the noise properly, has no edge artifacts, and maintains better stratification for multiple layers.*

Abstract

The traditional method of rendering semi-transparent surfaces using alpha blending requires sorting the surfaces in depth order. There are several techniques for order-independent transparency, but most require either unbounded storage or can be fragile due to forced compaction of information during rendering. Stochastic transparency works in a fixed amount of storage and produces results with the correct expected value. However, carelessly chosen sampling strategies easily result in high variance of the final pixel colors, showing as noise in the image. In this paper, we describe a series of improvements to stochastic transparency that enable stratified sampling in both spatial and alpha domains. As a result, the amount of noise in the image is significantly reduced, while the result remains unbiased.

1. Introduction

Rendering semi-transparent surfaces is a persistent problem in real-time graphics. Traditional alpha blending requires that the surfaces are sorted in depth order, which is not always easy to achieve. Current GPUs can produce primitives procedurally using tessellation and geometry shaders, and there may be too much geometry to be sorted in e.g. hair and fur. Furthermore, when the depth ranges of primitives overlap, it often becomes impossible to sort them without conflicts—the primitives do not even need to intersect for this to happen. Because of these issues, there is still a need for methods that can render transparent surfaces without ordering constraints.

A-buffer [Car84], widely used in offline rendering, captures all surfaces rendered into a pixel, allowing the transparency to be resolved later when the rendering is complete. A GPU implementation was recently described by Yang et al. [YHGT10]. The main problems with A-buffer are its unbounded memory usage and the need to sort fragments in pixel in depth order before resolve. Another approach is to store a given maximum number of surfaces per pixel, which requires heuristically combining surfaces in overflow situations. Salvi et al. [SML11] present a recent GPU technique, but the idea has been explored and implemented in hardware already in the 90s (see e.g. [JC99] for references).

Unfortunately, incrementally combining surfaces can cause a snowball effect where slight changes in input may produce radically different outcomes, raising concerns of temporal coherence. Also, situation where a new surface appears between already combined surfaces cannot be handled correctly.

Stochastic transparency [ESSL10] operates in a fixed amount of storage and produces a result with the correct expected value. The downside of unbiasedness is that there is variance in the results, showing as noise in the rendered image. The amount of variance generally depends strongly on the sampling strategy used. Especially stratifying the samples in appropriate domains can bound the amount of sampling error in simple situations, and yield overall better sampling in complex situations.

The original stochastic transparency algorithm stratifies samples only locally within fragments, which has several weaknesses. In this paper, we describe a series of techniques for obtaining properly stratified sampling in both spatial and alpha domains, which reduces the amount of noise in the resulting image considerably. We restrict our scope to the basic single-pass method with no separate alpha gathering or post-processing. Multi-pass techniques described in [ESSL10] could be used to further improve image quality.

1.1. Basics of stochastic transparency

Let us consider the very basics of stochastic transparency. The naive algorithm is to perform the following computation for every covered sample.

1. Perform depth test, discard if failed.
2. Randomly choose an *opacity reference value* $x \in [0,1]$.
3. If $\alpha > x$, write sample to frame buffer and depth buffer.

This simple method yields the correct expected value for a pixel, but the result is extremely noisy. Already the original paper on stochastic transparency [ESSL10] abandons this approach in favor of local stratification of samples. If a fragment covers S samples and has opacity of α, the number of samples that are to be written is calculated as $R = \lfloor \alpha S + \xi \rfloor$, where ξ is a random number between 0 and 1. Then, R samples are chosen randomly among the covered samples. This greatly reduces the amount of noise, but as Figure 1 illustrates, the lack of spatial stratification makes the remaining noise somewhat clumpy. Also, because stratification is only achieved within each fragment, primitive edges have higher amount of noise than their interiors.

Our approach is based on the naive method outlined above, but we carefully assign each sample of the pixel an opacity reference value that is used in place of randomly chosen x. In contrast to the stratification of Enderton et al., we do not first decide how many samples we want to cover, but instead generate the opacity reference values so that the desired stratification is obtained. As shown in Figure 1, our method produces significantly less noise while remaining unbiased.

It is easy to see that we cannot statically assign opacity reference values to the samples. Consider e.g. two overlapping surfaces with the same opacity drawn on top of each other. With static assignment, both will be written into the exact same samples, and only the closer one will be visible, which is incorrect. This is what the hardware-supported alpha-to-coverage multisampling does today, preventing its use for stochastic transparency. On the other hand, if the surface is drawn in several parts into the same pixel, i.e., there is an edge inside the pixel, we want the assignment to remain consistent. This ensures stratification across the entire surface in the pixel instead of only within each individual fragment. Without consistent assignment, the interior edges of surfaces can become visible, as with the previous method.

Therefore, for a single surface the assignment has to remain static, whereas for different surfaces it has to be uncorrelated with previously drawn surfaces. This requires detecting when a new surface is started, which allows us to reassign the opacity reference values. In Section 2 we present a heuristic method for automatically detecting this situation.

There are several requirements for the opacity reference values. First, they need to be stratified within the pixel, and secondly, between nearby pixels. The latter ensures high-quality dithering instead of random noise. Thirdly, to support multiple surfaces, it must be possible to produce uncorrelated opacity reference value sequences without sacrificing the first two properties. In Section 3 we describe a suitable random number sequence that is also easy to generate.

Finally, even if each single surface is well stratified and separate surfaces are uncorrelated, there are no guarantees about the stratification of samples that are affected by multiple surfaces. We address this problem by sorting the existing samples, effectively grouping previously encountered surfaces together, which ensures that each surface that was previously drawn into the pixel obtains a stratified set of samples when a surface is drawn on top of them. This is described in Section 4.

2. Tracking of surfaces

In order to detect a new surface, we maintain a bitmask with one bit per sample within each pixel. Initially the mask is empty. In addition, we maintain a surface ID per pixel, starting at zero.

When a primitive is drawn into the pixel, we check if any of the samples it covers are set in the bitmask. If not, we

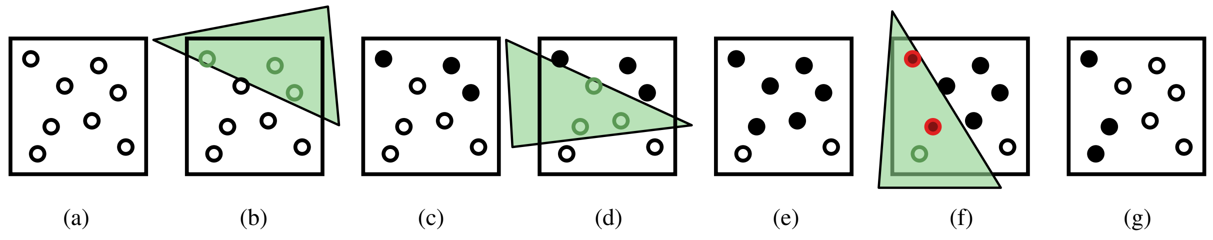


Figure 2: *An example of tracking of surface changes. (a) Bitmask is initially empty. (b) First triangle arrives, no conflicts with the bitmask. (c) Bits corresponding to coverage are set. (d) Second triangle of the same surface arrives, no conflicts with the bitmask. (e) Bits corresponding to coverage are set. (f) Third triangle conflicts with the bitmask and therefore requires opacity reference value sequence that is uncorrelated with the previous surface. (g) Surface ID is increased and bitmask is initialized to samples covered by the triangle.*

conclude that the primitive does not overlap with the current surface. We keep the same surface ID, and set the bits corresponding to the covered samples, effectively augmenting the current surface. Note that we examine all covered samples that survive the depth test. In particular, the analysis is performed before performing the opacity test that can cull some of those samples.

If one or more bits corresponding to the covered samples are already set, we infer that the primitive overlaps with the current surface, and therefore needs an uncorrelated set of opacity reference values. In this case we start a new surface, increment the surface ID, and initialize the bitmask according to the coverage of the primitive being drawn. The process is illustrated in Figure 2. It should be noted that perfect tracking of surfaces is not required—it is enough to guarantee that the same samples are not processed twice using the same surface ID, which the heuristic achieves.

However, the heuristic may fail to keep the same surface ID in certain cases. When a new surface is started and later a previous surface is resumed, it is obviously not possible to return to the sequence used in the previous surface. Another imperfection occurs when a previous surface covers only part of the pixel and a new surface is drawn using many primitives so that at first the primitives do not overlap the previous surface but later do. In this case, the change of surface is not detected until reaching the overlapping primitives of the new surface, yielding two uncorrelated stratification sequences for it.

We have not observed any visible problems caused by these failure modes, but it is certainly possible to construct test cases that intentionally mislead the heuristic.

3. Generation of opacity reference values

To be able to generate opacity reference values on the fly instead of storing them into memory, we need a function that maps an integer (constructed from sample index, pixel index, and surface ID) into a floating-point number (opacity reference value). Furthermore, a continuous span of indices in this sequence must produce a well-stratified sequence of opacity reference values, but disjoint spans should be mutually uncorrelated.

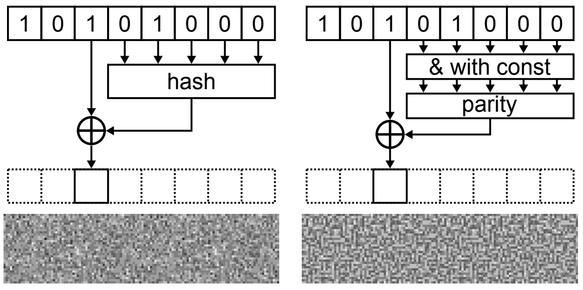
Let us begin by considering the standard base-2 radical inverse, obtained by reversing the bits of the index and placing a binary point in front:

Index	**Binary**	**Reversed**	**Decimal**
0	00000000	.00000000	0.000
1	00000001	.10000000	0.500
2	00000010	.01000000	0.250
3	00000011	.11000000	0.750
4	00000100	.00100000	0.125
5	00000101	.10100000	0.625
6	00000110	.01100000	0.375
7	00000111	.11100000	0.875

This sequence is extremely easy to generate and it fulfills the requirement that a continuous span of indices produces a well-stratified sequence. However, it has the problem that disjoint spans can be correlated. For example, spans $0 \ldots 3$ and $4 \ldots 7$ produce the exact same sequence of output values, shifted by 0.125.

If this correlation is not removed, the rendering results are biased when multiple surfaces are rendered into the same pixel. This is because each sample has almost constant opacity reference value regardless of surface ID. To remove the correlation, we perform a scrambling operation to the reversed bit sequence. Conceptually, for each bit in the reversed bit sequence, we want to take the bits below it, feed them into a hash function that returns 0 or 1, and XOR this with the bit we are examining. This produces the same result as performing an Owen scramble (see [KK02] for a description) to the original sequence, and therefore retains the stratification properties we are interested in.

Figure 3a illustrates this concept for a single bit. It does not matter whether the operations are performed in parallel or sequentially, as the potential flips of less significant



(a) Our method (b) Sobol sequence

Figure 3: *To produce a bit sequence where disjoint spans are uncorrelated, we scramble each bit of the sequence based on some hash of bits below it (⊕ denotes a* XOR *operation). The insets at bottom show the results when 8 white surfaces with* $\alpha = 0.1$ *are drawn on top of black background, using 8 samples per pixel. (a) We employ a generic hash function to provide randomness to the sequence. (b) Sobol sequence is equivalent to using a particularly simple hash function consisting of an* AND *operation with a bit-specific constant followed by parity count. When multiple surfaces are drawn, significant correlations remain.*

bits can be thought of being incorporated in the hash function that is in any case random. Contrasting our method with Sobol sequence (see [BF88] for a practical algorithm), we introduce more randomness into the sequence at the expense of potentially worsening its discrepancy characteristics. However, it is essential to remove any unwanted correlation among adjacent pixels and surfaces, or distracting artifacts may appear as shown in Figure 3b.

Below is an example of a scrambled sequence:

Index	**Reversed**	**Scrambled**	**Decimal**
0	00000000	11000000	0.750
1	10000000	01000000	0.250
2	01000000	00000000	0.000
3	11000000	10000000	0.500
4	00100000	01100000	0.375
5	10100000	11100000	0.875
6	01100000	00100000	0.125
7	11100000	10100000	0.625

We can see that for every aligned sequence of length 2, the highest bit of the scrambled index has both values of 0 and 1, and therefore the values for such spans are well stratified (difference is always 0.5). Similarly, for aligned sequences of length 4 the two highest bits get all possible combinations of 00, 01, 10, and 11. In fact, every aligned, continuous sequence with power-of-two length is perfectly stratified as in the previously considered radical inverse sequence. In contrast to that, disjoint spans are not correlated anymore, and therefore we can easily obtain uncorrelated but internally stratified subsequences by jumping into a different place in the sequence.

```
function CalcAlphaRef(sampleId, pixelId, surfaceId)
1: $x \leftarrow sampleId + pixelId \cdot samplesPerPixel$
2: $x \leftarrow \text{reverseBits}(x) + surfaceId$
3: $x \leftarrow x \oplus (x \cdot \texttt{0x6C50B47Cu})$
4: $x \leftarrow x \oplus (x \cdot \texttt{0xB82F1E52u})$
5: $x \leftarrow x \oplus (x \cdot \texttt{0xC7AFE638u})$
6: $x \leftarrow x \oplus (x \cdot \texttt{0x8D22F6E6u})$
7: return $x$ scaled to range $[0,1]$
```

Figure 4: *Pseudocode of the opacity reference value calculation algorithm based on multiply-*XOR *hash. Variable x is a 32-bit unsigned integer.*

3.1. Construction in practice

Evaluating a hash function per bit can be too expensive for a practical solution. Fortunately, computing hashes and XORing the bits can be approximated cheaply by multiplications and word-long XOR operations. Multiplying the index by an even random number produces a bit string where each input bit may affect all of the bits above it. XORing that with the index itself flips each bit based on a hash-like function of the bits below it. Using multiplication as the hash function leaves a lot of structure in the results, and therefore one such operation is not enough to provide enough scrambling. Based on our tests, doing four rounds with different constants are sufficient to remove any detectable structure.

Figure 4 gives the pseudocode of the opacity reference value calculation using multiplication-based hashing. The multiplication constants are the ones used in generating the images in this paper, but they carry no particular significance beyond that. To obtain perfect stratification among the samples in a pixel, the sample index is placed into the lowest bits of the index, and for spatial stratification (dithering), the pixel index is placed right above it. The pixel index has to be determined so that nearby pixels have nearby indices. Z curve (Morton curve) is not adequate, because it gives every 2×2, 4×4, etc. pixel block exactly the same structure, causing, e.g., pixels with even y coordinate to have a different expected value than pixels with odd y coordinate. Hilbert curve produces much better results that have no apparent structures, and we have used it when generating the images in this paper. The surface index is placed in the lowest bits of the reversed bit sequence, i.e., the highest bits of the index. This ensures that whenever there is a change of surface, we jump to a completely new part of the sequence and obtain values that are uncorrelated with previously generated values.

4. Improving stratification of overlapping surfaces

We are now able to ensure that the samples drawn by individual surfaces are not mutually correlated, providing correct expected value. However, only the nearest surface is guaranteed to end up covering the desired stratified number of sam-

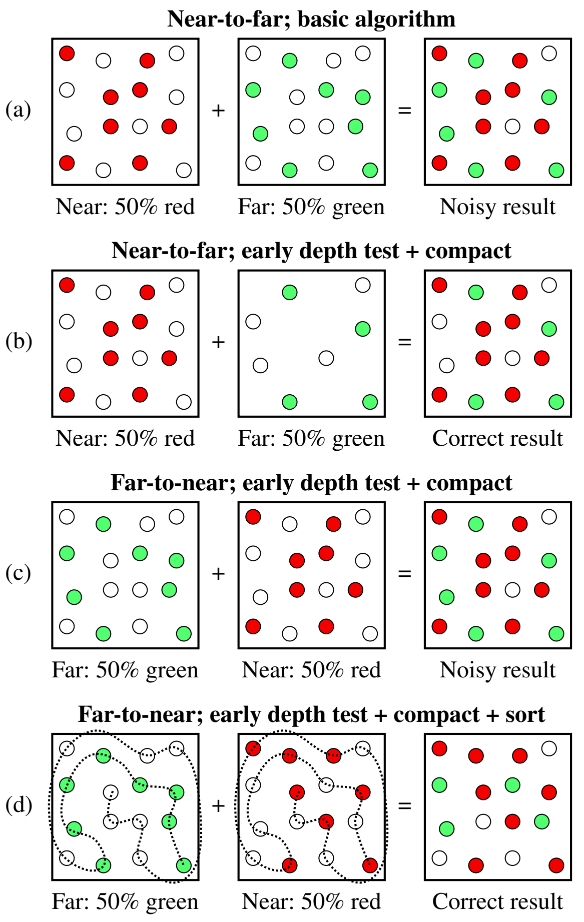


Figure 5: *Ensuring correct stratification with multiple surfaces. (a) With no correlation between surfaces, the result is correct on average, but there are no guarantees that any but the nearest surface covers exactly the right number of samples. (b) Considering only the samples that pass the depth test solves the near-to-far case. (c) When rendering far-to-near, compaction does not help as all samples of the red surface pass the depth test, and the number of green samples it covers is again random. (d) Sorting the samples according to depth of previously drawn surfaces groups them into continuous spans, restoring the stratification.*

ples. To be precise, given N samples per pixel, the nearest surface will cover either $\lfloor \alpha N \rfloor$ or $\lceil \alpha N \rceil$ samples with correct probabilities, but there are no similar bounds for surfaces behind it. As an example, consider a situation where a surface is drawn with $\alpha = 0.5$ and $N = 16$. Due to stratification of opacity reference values this will cover exactly 8 samples in every pixel. Now, if another such surface is drawn behind the first one, it will cover 4 samples on the average after depth test. However, it can cover any number between 0 and 8 samples, depending on the opacity reference values, and therefore it will be noisy. Similarly, if the second surface is drawn in front of the first one, it can occlude between 0 and 8 samples of the first surface, again causing noise on the far surface. In both cases the correct result is to have the far surface cover exactly 4 samples in the end, but so far we are unable to make this happen. Figure 5a illustrates the front-to-back case.

The first step to remedy this situation is to apply per-sample depth test before any other computation, and assign sample IDs continuously to the surviving samples. This is illustrated in Figure 5b. By not considering samples that would be occluded by existing surfaces in the pixel, we obtain correct stratification among the covered, unoccluded fragments.

However, when surfaces are drawn in the opposite order, the early depth test does not help, as illustrated in Figure 5c. In order to improve the result in this particular case, we must make the later drawn, occluding surface cover exactly half of each previously drawn surface. We accomplish this by sorting the samples before assigning them indices. We first look at the previous depth values stored in the depth buffer at each of the covered samples. We then sort the samples in increasing order based on these depths. This groups the surfaces in the pixel together and ensures that each previously drawn surface gets a continuous span of indices from the opacity reference value sequence, ensuring good stratification, as illustrated in Figure 5d. There is no need to detect individual surfaces in any way, and it is in fact desirable to be able to stratify groups of surfaces (e.g. distant foliage) as a whole. Therefore, sorting the samples is preferable to trying to group the samples into discrete surfaces.

One more detail needs to be taken into account when applying the early depth test. If we always start the numbering of samples that pass the depth test from zero, we cannot any more guarantee stratification within the same surface if it is drawn in multiple fragments. This again leads to edge artifacts. In order to fix this, we need to know how many samples we have already drawn from the current surface and continue the numbering of samples from there. Fortunately, this information is easily obtained by counting the number of bits set in the current surface coverage bitmask used for detecting surface changes. In case of starting a new surface after a conflict, we start the counting from zero.

Figure 6 illustrates the effect of sorting when drawing a cloth model that has multiple transparent layers and triangles ordered randomly. In Figure 6a we can see regions of high and low amount of noise. In places where the near surface is drawn first, the compaction of samples is enough to guarantee that the far surface gets the correct number of samples, and hence low amount of noise. However, when the far surface is drawn first, there is no guarantee of it having the correct number of samples after the near surface is drawn over it. These regions exhibit high amount of noise. In Fig-

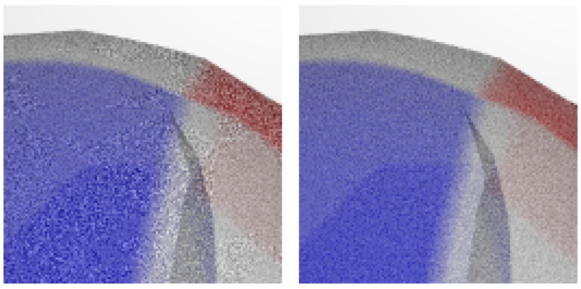


Figure 6: *Effect of sample sorting. (a) Without sorting, the result is more noisy in places where the surfaces are drawn in far-to-near order. (b) With sort the level of noise is the same everywhere.*

```
procedure DrawFragment(pixel $p$, samples $S$)
 1: for each sample $s \in S$ do
 2:     if $s.z > p.depth[s.sampleId]$ then remove $s$ from $S$
 3: end for
 4: if for any $s \in S$ bit $p.covered[s.sampleId]$ is set then
 5:     $p.surfaceId \leftarrow p.surfaceId + 1$
 6:     clear $p.covered$
 7: end if
 8: $n \leftarrow$ number of set bits in $p.covered$
 9: sort $S$ in order of increasing $p.depth[s.sampleId]$
10: for each sample $s \in S$ do
11:     $ref \leftarrow \text{CalcAlphaRef}(n, p.pixelId, p.surfaceId)$
12:     if $s.opacity > ref$ then
13:         $p.depth[s.sampleId] \leftarrow s.z$
14:         $p.color[s.sampleId] \leftarrow s.color$
15:     end if
16:     $n \leftarrow n + 1$
17: end for
18: for each $s \in S$ do set $p.covered[s.sampleId]$
```

Figure 7: *Pseudocode of the final stratified stochastic transparency algorithm. See text for walkthrough.*

ure 6b, we see that when the samples are sorted, all parts of the image have the same low amount of noise.

Figure 7 gives pseudocode for the final stratified stochastic transparency algorithm. Set S contains the samples of the fragment being drawn. Lines 1–3 perform depth test and lines 4–7 and 18 track the current surface. Lines 8–11 and 16 ensure correct ordering of opacity reference values, and lines 12–15 perform the opacity test and conditional write. It should be particularly noted that on line 9 the samples of the fragment are sorted according to previously drawn depths at the covered sample positions, not the depths of the samples in the fragment being drawn.

spp	MOTOR our	MOTOR comp.	MOTOR ratio	CLOTH our	CLOTH comp.	CLOTH ratio
4	29.5	36.3	0.81	17.0	19.3	0.88
8	17.9	25.3	0.71	9.7	12.3	0.79
16	10.3	17.2	0.60	5.6	8.4	0.67
32	6.3	12.1	0.52	3.3	5.8	0.57
64	4.0	8.7	0.46	2.0	4.1	0.49

Table 1: *RMSE in 8-bit color units. As the number of samples per pixel grows, our RMSE improves faster than with the comparison method, indicating better stratification. At 64 samples per pixel, our method about halves the magnitude of noise compared to the previous method.*

5. Results and Discussion

Our algorithm is primarily targeted at hardware implementation, and it is most probably not feasible to implement it in shaders like the original stochastic transparency algorithm [ESSL10]. Therefore, we shall not provide execution time statistics but focus only on image quality. We compare our method against the basic method of Enderton et al., assuming that they do depth tests before determining the number of samples that a fragment covers—the original paper leaves room for interpretation, so we chose the alternative that favors the comparison method.

Figure 9 shows the results using our method and the basic method of Enderton et al. in case of a multi-layered, complex model. The model has constant $\alpha = 0.3$, and all surfaces are rendered using diffuse shading. The effectiveness of our method can be seen in both image quality and the RMSE values computed against a reference image rendered using thousands of samples per pixel. Figure 10 shows a similar improvement for a colorful cloth model with $\alpha = 0.4$. Table 1 summarizes the RMSE results. We can see that our RMSE falls faster when the number of samples is increased, indicating better stratification. Judging from the images and the RMSE results, our method with 16 samples per pixel is quite close to the comparison method with 64 samples per pixel, allowing significant memory, computation and bandwidth savings.

When a large number of layers are rendered on top of each other, our stratification attempts are mostly in vain as can be expected. Figure 8 shows a case where dozens of transparent layers are rendered on top of each other, as is often done for particle effects. In these kind of situations algorithms that merge layers adaptively (e.g., [SML11]) can perform much better unless temporal coherence becomes an issue.

Considering GPU implementation, the execution of our algorithm would be best suited in ROP (raster operation) stage that follows shading and traditionally performs blending. It should be noted that the *for* loops in the pseudocode (Figure 7) can be trivially parallelized, because the samples are independent.

Our method
RMSE = 5.5
Comp. method
RMSE = 5.7
Depth complexity
White = 200

Figure 8: *With an extreme number of layers, our method loses most of its effectiveness but never performs worse than the comparison method. This cloud of smoke consists of textured billboards with* $\alpha = 0.05$ *and is rendered with 16 samples per pixel.*

6. Future work

Combining the multi-pass methods of Enderton et al. [ESSL10] with our stratification algorithm should improve the results as in the original paper. Quantifying the improvement remains an interesting task. As an alternative to gathering alpha values in a separate pass, it may be possible to filter out the remaining noise using some kind of post-process algorithm in the spirit of Shirley et al. [SAC*11]. Examining whether this approach is as well suited for transparency as for other stochastic effects and how exactly the filter widths should be decided provide further avenues for future work.

References

[BF88] BRATLEY P., FOX B.: Algorithm 659. Implementing Sobol's quasirandom sequence generator. *ACM Transactions on Mathematical Software 14*, 1 (1988), 88–100. 4

[Car84] CARPENTER L.: The A-buffer, an antialiased hidden surface method. *SIGGRAPH Comput. Graph. 18* (1984), 103–108. 1

[ESSL10] ENDERTON E., SINTORN E., SHIRLEY P., LUEBKE D.: Stochastic transparency. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games* (2010), pp. 157–164. 1, 2, 6, 7

[JC99] JOUPPI N. P., CHANG C.-F.: Z3: An economical hardware technique for high-quality antialiasing and transparency. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (1999), pp. 85–93. 2

[KK02] KOLLIG T., KELLER A.: Efficient multidimensional sampling. *Computer Graphics Forum 21*, 3 (2002), 557–564. 3

[SAC*11] SHIRLEY P., AILA T., COHEN J., ENDERTON E., LAINE S., LUEBKE D., MCGUIRE M.: A local image reconstruction algorithm for stochastic rendering. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games* (2011), pp. 9–13. 7

[SML11] SALVI M., MONTGOMERY J., LEFOHN A.: Adaptive order independent transparency: A fast and practical approach to rendering transparent geometry. Game Developers Conference, March 2011. 2, 6

[YHGT10] YANG J., HENSLEY J., GRÜN H., THIBIEROZ N.: Real-time concurrent linked list construction on the GPU. *Computer Graphics Forum 29*, 4 (2010), 1297–1304. 1

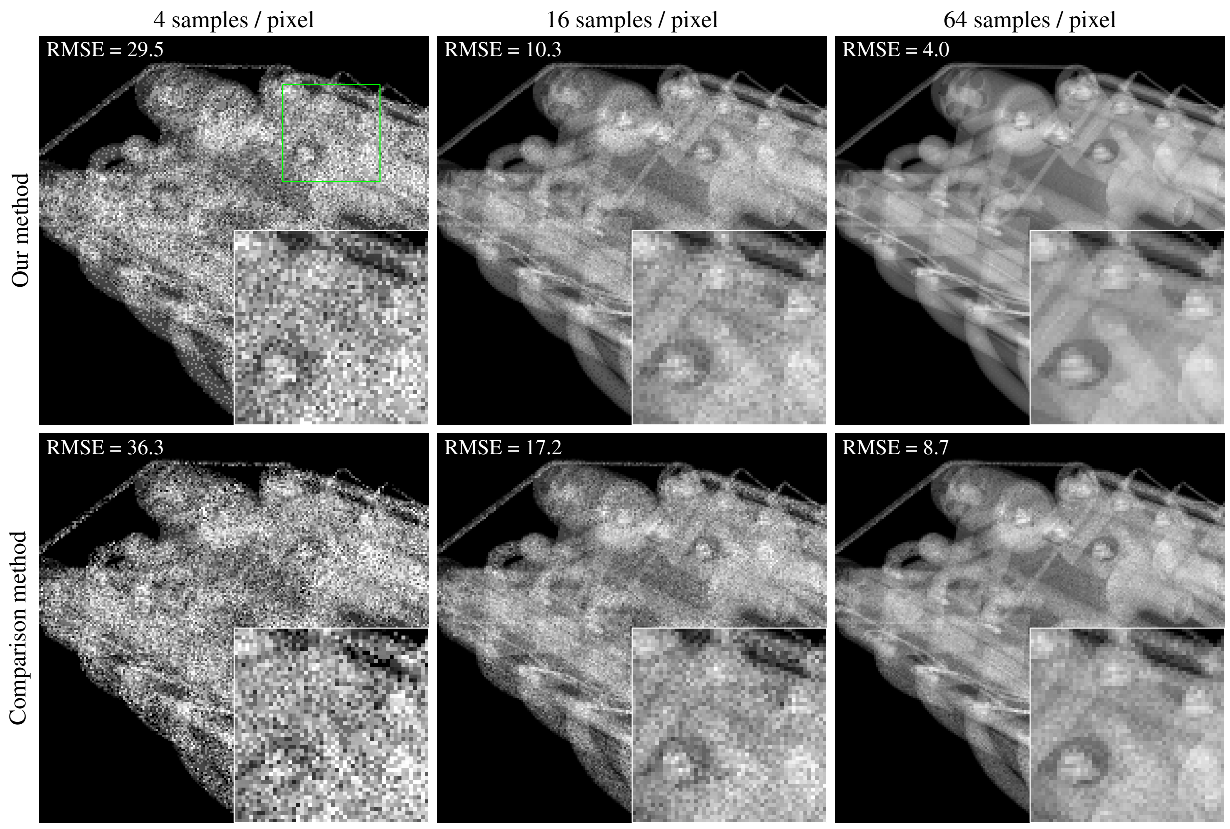


Figure 9: *A complex motor model rendered using white untextured surfaces, diffuse shading, and* $\alpha = 0.3$*. The RMSE values are in 8-bit RGB units, i.e., RMSE of 1 would indicate that the magnitude of noise is 1/256 of the full brightness range.*

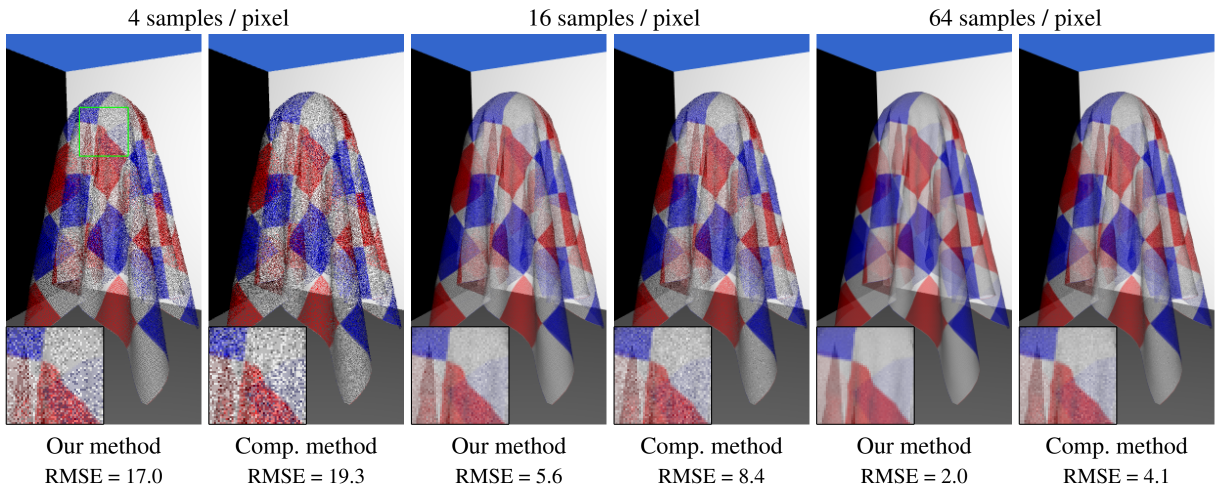


Figure 10: *A textured cloth model rendered using diffuse shading and* $\alpha = 0.4$*.*