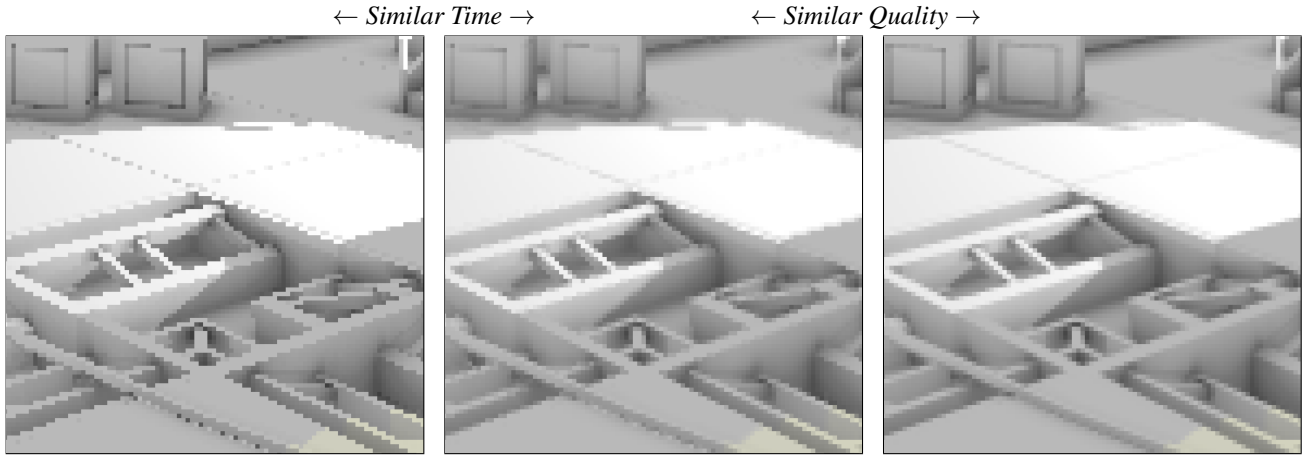
Subpixel Reconstruction Antialiasing for Deferred Shading

Matthäus G. Chajdas*
Technische Universität München and NVIDIA

Morgan McGuire
NVIDIA and Williams College

David Luebke
NVIDIA



(a) 1× Shading + Box (poor, fast) (b) **NEW: 1× Shading + SRAA (good, fast)** (c) 16× Shading + Box (good, slow)

Figure 1: *Subpixel Reconstruction Antialiasing produces an image approaching* $16\times$ *supersampling quality using the shading samples from a regular* $1\times$ *grid. It applies a joint bilateral filter inside each pixel based on subpixel geometric samples in a Latin square. Scene from Marvel Ultimate Alliance 2 (see Figure 8), courtesy of Vicarious Visions. Shown: 4 geometric samples/pixel, planes+normals depth metric.*

Abstract

Subpixel Reconstruction Antialiasing (SRAA) combines single-pixel ($1\times$) shading with subpixel visibility to create antialiased images without increasing the shading cost. SRAA targets deferred-shading renderers, which cannot use multisample antialiasing.

SRAA operates as a post-process on a rendered image with superresolution depth and normal buffers, so it can be incorporated into an existing renderer without modifying the shaders. In this way SRAA resembles Morphological Antialiasing (MLAA), but the new algorithm can better respect geometric boundaries and has fixed runtime independent of scene and image complexity.

SRAA benefits shading-bound applications. For example, our implementation evaluates SRAA in 1.8 ms (1280×720) to yield antialiasing quality comparable to 4-16$\times$ shading. Thus SRAA would produce a net speedup over supersampling for applications that spend 1 ms or more on shading; for comparison, most modern games spend 5-10ms shading. We also describe simplifications that increase performance by reducing quality.

CR Categories: I3.3 [Picture/Image Generation]: Antialiasing—; I3.7 [Three-Dimensional Graphics and Realism]: Color, shading, shadowing, and texture—;

Keywords: antialiasing, deferred shading

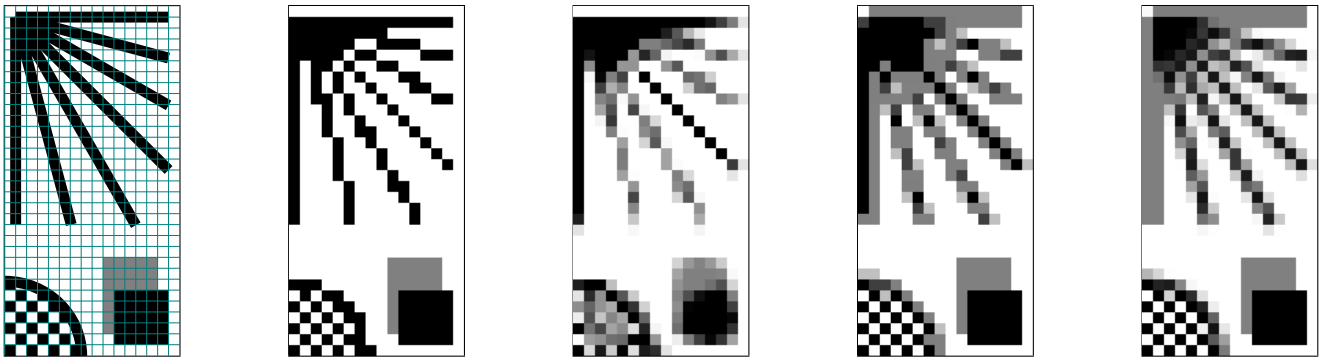
*chajdas@tum.de, {momcguire, dluebke}@nvidia.com

1 Introduction

Deferred lighting and **multisample antialiasing** (MSAA) are powerful techniques for real-time rendering that both work by separating the computation of the shading of triangles from the computation of how many samples they cover. Deferred lighting uses deferred shading [Saito and Takahashi 1990] to scale complex illumination algorithms up to large scenes. MSAA resolves edges by shading multiple samples per pixel; unlike SSAA each primitive is shaded at most once.

Unfortunately, these two techniques are incompatible (see Section 2), so developers must currently choose between high quality lighting and high quality antialiasing. To achieve antialiasing under deferred lighting, programs tend to either super sample – at linear cost in the resolution – or perform Morphological Antialiasing (MLAA) [Reshetov 2009], a sort of heuristic "smart blur" of the final image.

We introduce a new technique for **subpixel reconstruction antialiasing** (SRAA). The core idea is to extend the success of MLAA-style postprocessing with enough input to accurately reconstruct subpixel geometric edges. SRAA operates as a postprocess that combines a G-buffer sampling strategy with an image reconstruction strategy. The key part is to sample the shading at close to screen resolution while sampling geometry at subpixel precision, and then estimate a superresolution image using a reconstruction filter. That superresolution image is then filtered into an antialiased screen-resolution image. In practice, the reconstruction and downsampling occur simultaneously in a single reconstruction pass. Our results demonstrate that SRAA can approximate the quality of supersampling using many fewer shading operations, yielding a net 4-16$\times$ speedup at minor quality degradation compared to shading at each subpixel sample.



(a) Vector Input (b) 1× Shading (c) 1× Shading + MLAA (d) 1× Shading + New SRAA (e) 16384× Shading Reference

Figure 2: *Morphological Antialiasing (MLAA) overblurs some edges because it lacks geometric information. SRAA cannot reconstruct undersampled geometry. We hypothesize as future work that merging the algorithms will be superior to either alone.*

We designed SRAA specifically for games and other real-time applications that render on modern GPUs. The inputs are the screen-resolution 1× shaded image, a superresolution depth buffer, and an optional superresolution normal buffer. The superresolution buffers can be rendered in a single MSAA forward-rendering pass or with multiple 1× forward-rendering passes; we find either method requires a small fraction of the total shading cost. The algorithm may work on tiles to reduce the peak memory requirements and interoperate with tiled shading algorithms. This paper contributes:

- An efficient algorithm for antialiasing images as a post-process.
- A set of simplifications allowing implementers to increase performance for small quality reductions.
- Detailed analysis of the algorithm and comparison to MLAA [Reshetov 2009], the current "best in class" approach for antialiased deferred shading.
- Evaluation on real game scenes including texture, specular reflection, geometric aliasing, emission, and bloom.

2 Related Work

Multisample antialiasing (MSAA) was designed for forward rendering and encounters severe drawbacks when used in conjunction with deferred shading algorithms [Koonce 2007; Shishkovtsov 2005]. During forward rendering, MSAA shades once per *fragment*, the portion of a primitive intersecting the boundary of a pixel. MSAA then writes the computed color to all samples in the pixel covered by that fragment. Since fragments are planar and often small in world space, the shading across a fragment can be approximated as constant in many cases and thus MSAA gives quality comparable to supersampling at a fraction of the cost. REYES [Cook et al. 1987] uses a similar strategy, but with micropolygons even smaller than the typical GPU fragment.

Deferred shading performs shading on the final samples in the framebuffer, when fragments are no longer available. In particular, there is no longer any information about which samples originate from the same surface, requiring the algorithm to shade at every sample. In this case, MSAA degenerates into brute-force supersampling and its benefits are lost. One can imagine various ad hoc strategies for *guessing* which samples come from the same surface during deferred shading. Such strategies will incur the overhead of that guess, the warp incoherence of branching independently at each pixel, and the quality cost of sometimes guessing wrong.

Like MLAA [Reshetov 2009; Jimenez et al. 2011], our algorithm requires only one shading sample per pixel and uses shading from adjacent pixels to increase apparent resolution. But MLAA relies only on final colors, so it applies heuristic rules that can fail to identify some edges requiring antialiasing. Because SRAA is guided by geometric samples, it is independent of edge orientation and avoids certain kinds of overblurring. Since geometric samples are inexpensive to compute compared to the full shading, its performance is comparable to MLAA and 1× shading with no antialiasing. Even though MLAA works only on the final image, the runtime is not constant but varies with the number of edges. This makes MLAA difficult to use in games which require fixed time budgets for post-processing effects.

SRAA is also similar to Coverage Sampled Antialiasing (CSAA) [Young 2006], which takes advantage of additional visibility samples to improve edges. However, CSAA only works with forward rendering because the fragments corresponding to the coverage masks are no longer available when a deferred shading pass occurs.

A key operation in SRAA is joint bilateral upsampling [Kopf et al. 2007]. Many lighting algorithms operate at low resolution and then use upsampling to reconstruct a final image at screen resolution [Sloan et al. 2007; Bavoil et al. 2008; Shopf 2009; McGuire 2010]. Such approaches face two main problems: undersampling and temporal coherence. If a feature gets missed or undersampled at the very low source resolution, the resulting screen-space artifacts can become very large. Undersampling also makes these algorithms prone to temporal coherence issues, which are typically resolved by applying stronger blur. These methods tend to work best on smooth, low-frequency input like indirect illumination.

A good filter for forward antialiasing is Konstantine et al.'s directionally adaptive filter [2009]. That takes multisampled shading information and reconstructs it using geometric information from neighboring pixels. Konstantine's filter cannot be directly applied to deferred shading.

Yang et al. [2008] pioneered the use of cross bilateral filtering to upscale shading information. We extend their ideas to address subpixel samples, which they describe as a limitation of their work: "scenes with very fine geometric detail may not be adequately reproduced in the low resolution buffer, resulting in artifacts." [Yang et al. 2008]

In independent work, Smash proposes a similar scheme [2009] for a demoscene project, but does not report further details or analysis.

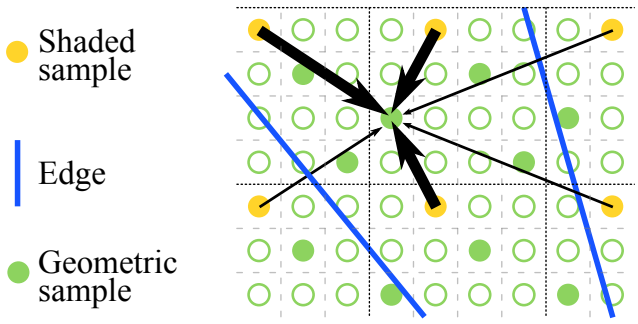


Figure 3: *Reconstructing subpixel shading with SRAA. Dotted lines show pixel boundaries; within each, there are four geometric samples (colored disks) on a* 4×4 *sample grid. One of those contains shading information (yellow disks). At each sample, we compute bilateral weights from neighboring shading samples (arrows). A neighboring sample with significantly different geometry is probably across a geometric edge (blue line), and receives a low weight.*

Some more recent approaches lower the shading rate by adapting it based on the geometry and shading, for instance in Nichols et al.'s screen-space multiresolution gather methods[Nichols et al. 2010]. While these methods also achieve low shading rates, they are focused on low-frequency phenomena.

3 Algorithm

3.1 Overview

SRAA exploits the fact that shading often changes more slowly than geometry in screen space to perform high-quality antialiasing in deferred rendering by sampling geometry at higher resolution than shading. We refer to *geometry samples*, which capture surface properties–in our case, the normal and position of a surface fragment–and *shading samples*, which contain a color. By upsampling shading onto a "superresolution" image of geometric data, SRAA creates high-resolution shading data suitable for filtering back down to screen resolution.

SRAA requires two modifications to a standard rendering pipeline. First, applications must generate normal and position information at subpixel resolution. See Section 3.4 for details.

Second, applications must perform a reconstruction pass after shading and before post-processing. This pass refines the results from shading using the G-buffer information. The output of this reconstruction step is a screen-resolution, antialiased shading buffer which can then be post-processed as normal. The shading buffer resolution is usually the same as the screen or slightly higher, but much lower than the geometry buffers.

Our reconstruction is a modified cross-bilateral filter similar to the metric used by irradiance caching [Ward and Heckbert 1992]. The filter accounts for differences in normal and position and is explained in more detail in Section 3.2.

In figure 3 we can see how our algorithm reconstructs one subpixel. All shaded neighbors in a fixed radius are considered and interpolated using the bilateral filter weights. After each sub-sample has been reconstructed, we combine them all together using a box filter for the final value of that pixel. More sophisticated multi-filters, for instance a triangle kernel, could be employed for reconstruction. However, more complex filters have to be carefully tuned to work well with the extremely small number of samples in the filter support range.

Notice that due to the fixed radius of the filter support, a variable number of shaded samples are used to reconstruct a subpixel sample. This reduces the total number of G-buffer loads and allows us to re-order the instructions to improve cache hit rate.

Figure 4: *Quality comparison between estimating the distance using normals only, depth only, and both. Some edges can be detected using only depth, while others require normals. Scene courtesy of DICE from the Frostbite 2 game engine.*

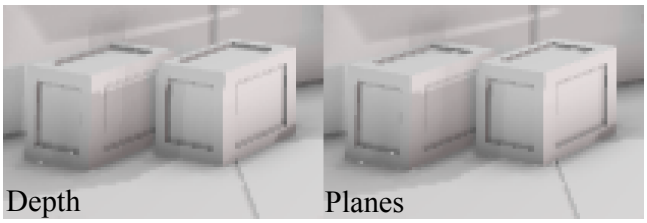


Figure 5: *Quality comparison between position estimates based depth only versus plane equations. The plane distance metric captures the small insets on the boxes slightly better.*

Typically, the filter radius is extremely narrow to avoid blurring and keep the number of texture lookups at a reasonable level. A larger filter radius increases the reconstruction quality in theory, but it also increases the worst-case error. We thus use only shaded samples which are directly adjacent, allowing us to guarantee a screen-space error of one pixel.

3.2 Distance metric

We take both position and normal change into account when computing distance. For the position change, we can estimate the difference by using plane equations. For a source sample represented by a plane with normal $\vec{n_s}$ and a point p_s and a target sample p_t, the position difference is $\delta_p = \sigma_p |(p_t - p_s) \cdot \vec{n_s}|$. We combine this with the normal change term $\delta_n = 1 - (\vec{n_s} \cdot \vec{n_t})$, giving us the total weight $w = \exp(-\tau * \max(\delta_n, \delta_p))$.

In practice, we have to scale the depth by σ_p to account for different depth ranges such that actual discontinuities result in values ≥ 1. The τ factor determines how quickly the weights fall off and allows us to easily increase the importance of the bilateral filter. We used the same τ value (500) in all our testing.

Our algorithm allows several performance/quality trade-offs by changing how the filter weights are computed. We can estimate distance between source and target samples by simply comparing depth values rather than evaluating the plane equation. This approximation loses some ability to correctly resolve differences in tight corners or grooves (see Figure 5), but can significantly improve performance.

We can also remove the δ_n term, which accounts for change in normal, from the distance metric. Removing normals has the greatest impact on performance, reducing the read bandwidth by 50% and simplifying much computation. We have found that using only depth has a minor quality impact, and seems like a good trade-off for games; see Figure 4 for a comparison.

Table 1 summarizes the performance of different optimizations using CUDA 3.2. Our timings do not include the cost of transfering data between graphics and compute modes, which can vary widely across APIs (Direct3D, DirectCompute, OpenGL, OpenCL, CUDA C/C++, etc). Unless otherwise noted, all results use both depth and normals.

Method	Output Resolution 1280×720	1920×1080
Planes & Normals	1.86 ms	4.15 ms
Depth & Normals	1.14 ms	2.53 ms
Normals only	0.95 ms	2.11 ms
Depth only	0.55 ms	1.23 ms

Table 1: *SRAA performance for various distance metric optimizations, using 4× geometry samples on an NVIDIA Geforce GTX 480.*

3.3 Assumptions and Limitations

We assume that the shading cost dominates the total render time. In particular, generating the additional buffers must not introduce a significant cost or the overhead of G-buffer generation will dominate the reconstruction time. For example, we found that at 1280×720 it takes 1.4 ms to generate the additional samples to bring the G-buffer up to 2560×1440 and 1.1 ms to perform the antialiasing pass. The main "fat" G-Buffer took 1.1 ms in this case. If the renderer spends more than 0.75 ms in lighting, the reduction in shading computation pays for itself.

Our algorithm has one main limitation: in uniform regions it will introduce blur to the output. This is because in areas with no normal & depth variation, our filter weights become all equal and the reconstruction thus degrades into an extremely narrow blur. We reduce the blur by adding a strong screen-space falloff on the filter weights, but some blurring is inevitable as we filter across pixels. This is a common problem for filters which reconstruct by sampling multiple pixels. This could be possibly fixed by identifying such regions up front and masking them out, taking the noisy but sharp underlying shading information.

3.4 Generating Latin Square G-buffers

We suggest two methods to create the auxiliary geometry buffers. Our results were all produced by rendering four 1× G-buffers with subpixel offsets applied to the projection matrix (Figure 6). A better alternative would render one 4× rotated-grid MSAA render target, incurring less overhead than our 3 additional G-buffer passes. Implementing the MSAA pass would add implementation complexity that is probably justified for a production setting but would not affect the trends observed in our experiments.

We *do not* recommend rendering the G-buffers at 16× resolution using an interleaving mask [Kircher and Lawrance 2009]. Current hardware does not take advantage of the interleaving because it is forced to render in 2×2 pixel blocks to compute derivatives for texture filtering. Furthermore, at 1280×720, a 16× G-buffer would require 88 MB, of which 75% would be wasted because it is never read by our algorithm.

3.5 Implementation Details

The filter as described is heavy on texture reads and arithmetic operations. For a 4× SRAA reconstruction, we read the G-buffers 58× per pixel (194 for 16×.) The filter is evaluated 25 (81) times. We thus rely on the L1 and texture caches provided by NVIDIA's Fermi architecture to reduce the necessary bandwidth. To maximize cache performance, we split our input data so it is read from both L1 and texture caches, giving the aggregate bandwidth and size of both caches. Especially with "fat" G-buffer formats, we have found this to be extremely beneficial.

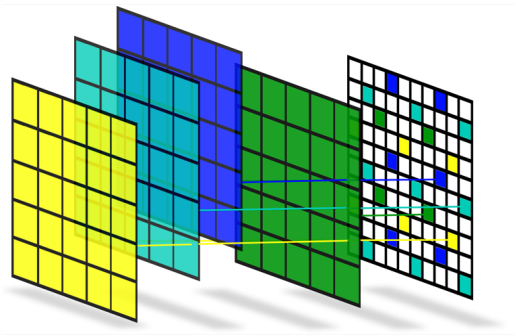


Figure 6: *Interleaving 4 screen-resolution G-buffers rendered with subpixel offsets to form one superresolution latin square pattern.*

Tight kernels like our reconstruction filter are easily expressed as a set of highly nested loops with a few branches, but we used a small custom code generator to emit optimal CUDA C code for our kernels, traversing the call graph during code generation. The resulting code consists of a single basic block, giving the optimizer maximum opportunity for reordering and other improvements.

4 Results

Figure 7 shows a scene with high geometric complexity processed using our algorithm. All the detailed geometry along the roof or the stair railing is actual geometry and can be thus processed using SRAA. This image uses a an ordered grid for supersampling, meaning the G-buffers were simply generated at twice the resolution. Still, our algorithm is able to provide high-quality antialiasing.

In Figure 8 we show how our algorithm degrades gracefully to regular sampling once the shading frequency becomes too high. In particular, the lighting on the stairs is subsampled even at 16× supersampling and as such our algorithm has no chance to resolve them properly. We fall back to a regular aliasing pattern in this case. Notice however how SRAA reconstructs the banister perfectly, which has a low shading frequency.

Figure 9 highlights some interesting cases with high texture detail and alpha-tested geometry. SRAA does not add excessive blur to the image, as can be seen on the concrete, while also working correctly on alpha-tested geometry like the fences. On most geometry, we get near 16× supersampled quality.

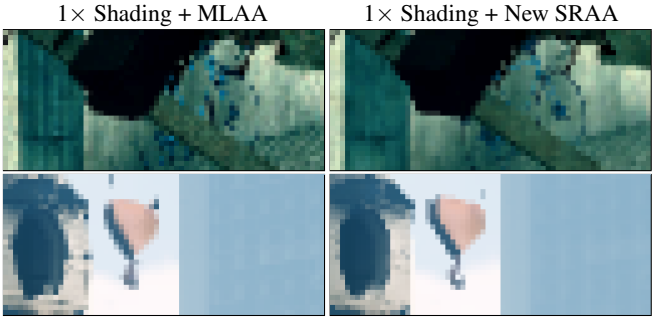


Figure 10: *Comparison with MLAA for small detail geometry. The thin features on the bicycle as well as an the hook are undersampled, so there is no continuous edge in the 1x input. MLAA thus removes the edge. However, the G-buffers sparsely capture that information, so the edge gets correctly reconstructed by SRAA. Scene courtesy of DICE from the Frostbite 2 game engine.*

Figure 7: *This scene contains many sources of geometric aliasing with low shading frequency, which are particularly well suited for SRAA. For instance, it is able to correctly reconstruct the brick structure along the roof, as well as the bent fence at the far end. The image has been reconstructed using 4 subsamples on an ordered grid to a target resolution of 1920×1080 pixels in 2.5 ms. Scene courtesy of Crytek from Crysis 2.*

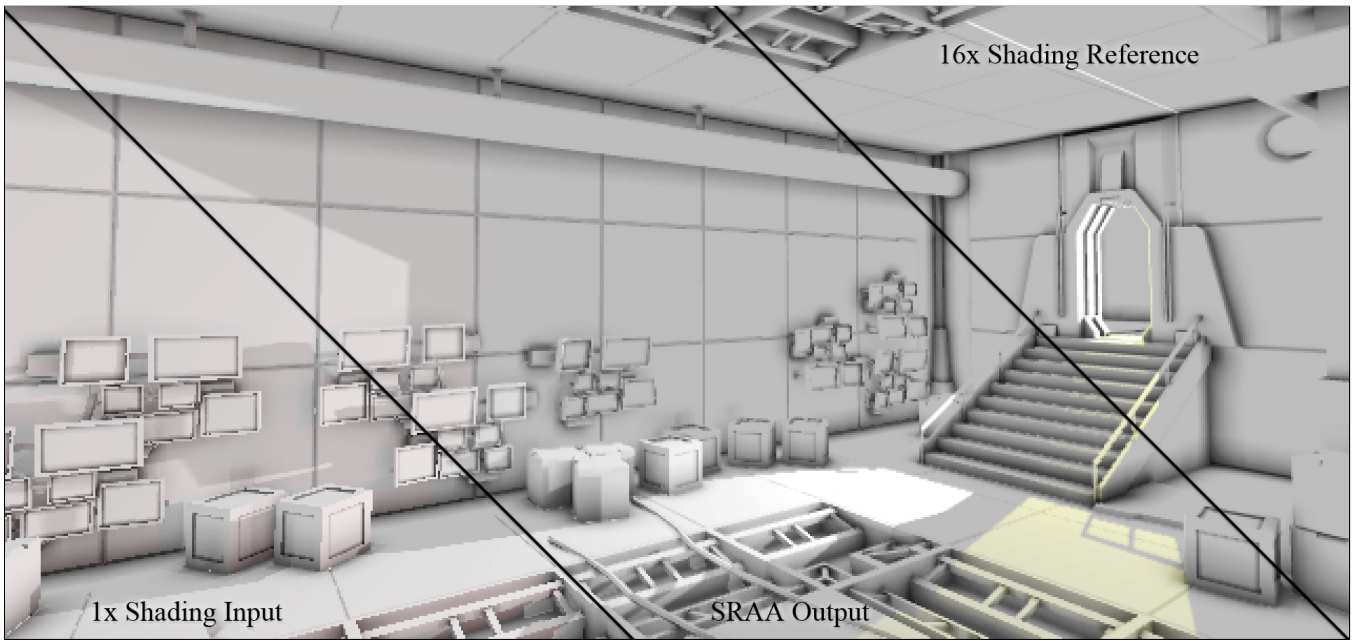


Figure 8: *Full scene from Figure 1 shown with the 1× shading input, output from 1× shading + SRAA, and a 16× shaded reference image. SRAA quality is best at the edges between wide features, such as on the ceiling, pipe, and stairway railing. SRAA quality is lowest at features that are below the Nyquist rate in the 1× shading input, such as the very thin insets on the crates and the inside corners of the stairs. The output resolution is 640×360, reconstruction time was < 1 ms.*

Figure 9: *SRAA reconstructs subpixel edges while preserving sharp texture and shading features. The large image was processed by SRAA. The details compare the 1x input, the SRAA output, and the 16x shaded reference. Scene courtesy of DICE from the Frostbite 2 game engine. Zoom the electronic version of this paper to see pixel-scale detail. The reconstruction at the target resolution (1920×1080) took 2.5 ms.*

SRAA can also resolve several cases where standard MLAA fails. Figure 10 shows that MLAA has problems with pixel-sized geometric features that are correctly handled by SRAA.

5 Discussion & Future Work

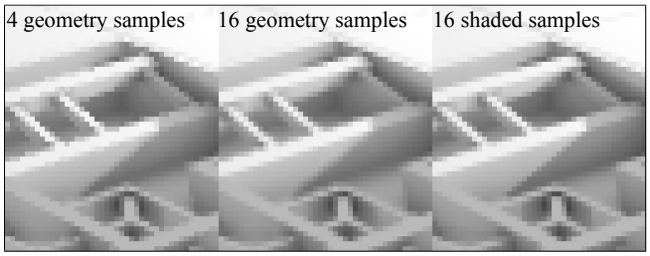


Figure 11: *The left two images use SRAA with one shaded sample and a varying number of geometric samples. With 16× geometric samples, SRAA approaches the 16× SSAA reference on the rightmost image.*

We present a new antialiasing algorithm which exploits subpixel geometry information to reconstruct subpixel shading for antialiasing. SRAA runs in a few milliseconds on today's GPUs, which makes it practical for real-time rendering applications. It will also scale to take advantage of increased bandwidth and computation on future GPUs. The algorithm can work with any underlying sampling pattern and number of shading samples. For example, it scales with more geometric samples (as seen in Figure 11) and shading samples. Unlike MLAA, the algorithm's time and space cost is independent of the scene, except for rendering the G-Buffers where the time increases with scene complexity.

We believe the next step is to combine ideas from SRAA and MLAA. SRAA uses relatively inexpensive geometric information to improve expensive shading results and is able to produce very good edge antialiasing. MLAA's heuristics often produce overblurring (Figure 2), but it is able to resolve *shading* edges that SRAA cannot. These include texture, shadow, and specular highlight boundaries. An algorithm that combines heuristic shading weights with accurate geometric weights may be able to achieve higher quality than either alone in practice.

Acknowledgements

We are grateful for assistance from Johan Andersson, Rendering Architect at DICE, Anton Kaplanyan, Lead Researcher at Crytek, additional data from Vicarious Visions, support and feedback from Rüdiger Westermann and NVIDIA colleagues David Tarjan, Timothy Farrar, Evan Hart, Eric Enderton, and Peter Shirley.

	1×	***New SRAA***	*16×*
G-buffer size	5.5 MB	**22.1 MB**	88.4 MB
G-buffer render time	1.1 ms	**2.5 ms**	5.0 ms
Shading time	10.0 ms	**10.0 ms**	160.0 ms
AA filter time	0.0 ms	**1.9 ms**	2.0 ms
Net shading time	11.1 ms	**14.4 ms**	167.0 ms

Table 2: *SRAA adds minimal overhead compared to supersampling. The table shows the cost of deferred shading at* 1280×720 *screen resolution under the schemes shown in Figure 1.*

References

BAVOIL, L., SAINZ, M., AND DIMITROV, R. 2008. Image-space horizon-based ambient occlusion. In *SIGGRAPH '08: ACM SIGGRAPH 2008 talks*, ACM, New York, NY, USA, 1–1.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The reyes image rendering architecture. *SIGGRAPH Comput. Graph. 21*, 4, 95–102.

IOURCHA, K., YANG, J. C., AND POMIANOWSKI, A. 2009. A directionally adaptive edge anti-aliasing filter. In *High Performance Graphics 2009*, ACM, New York, NY, USA, 127–133.

JIMENEZ, J., MASIA, B., ECHEVARRIA, J. I., NAVARRO, F., AND GUTIERREZ, D. 2011. Practical Morphological Anti-Aliasing. *GPU Pro 2*. to appear.

KIRCHER, S., AND LAWRANCE, A. 2009. Inferred lighting: fast dynamic lighting and shadows for opaque and translucent objects. In *SIGGRAPH Symposium on Video Games*, ACM, New York, NY, USA, 39–45.

KOONCE, R. 2007. Deferred Shading in Tabula Rasa. *GPU Gems 3*, 429–457.

KOPF, J., COHEN, M., LISCHINSKI, D., AND UYTTENDAELE, M. 2007. Joint bilateral upsampling. *ACM Transactions on Graphics (TOG) 26*, 3, 96.

MCGUIRE, M. 2010. Ambient occlusion volumes. In *Proc. of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA.

NICHOLS, G., PENMATSA, R., AND WYMAN, C. 2010. Interactive, multiresolution image-space rendering for dynamic area lighting. *Computer Graphics Forum* (June), 1279–1288.

RESHETOV, A. 2009. Morphological antialiasing. In *Proceedings of the 2009 ACM Symposium on High Performance Graphics*.

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph. 24*, 4, 197–206.

SHISHKOVTSOV, O. 2005. Deferred shading in S.T.A.L.K.E.R. *GPU Gems 2*, 143–545.

SHOPF, J., 2009. Mixed resolution rendering, March. http://developer.amd.com/gpu_assets/ShopfMixedResolutionRendering.pdf.

SLOAN, P.-P., GOVINDARAJU, N. K., NOWROUZEZAHRAI, D., AND SNYDER, J. 2007. Image-based proxy accumulation for real-time soft global illumination. In *Pacific Conference on Computer Graphics and Applications*, IEEE Computer Society, Washington, DC, USA, 97–105.

SMASH, 2009. deferred rendering in frameranger, November. Blog posting http://directtovideo.wordpress.com/2009/11/13/deferred-rendering-in-frameranger/.

WARD, G., AND HECKBERT, P. 1992. Irradiance gradients. In *Third Eurographics Workshop on Rendering*, 85–98.

YANG, L., SANDER, P. V., AND LAWRENCE, J. 2008. Geometry-aware framebuffer level of detail. *Comput. Graph. Forum 27*, 4, 1183–1188.

YOUNG, P. 2006. Coverage sampled antialiasing. Tech. rep., NVIDIA, Oct. http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/Direct3D9/src/CSAATutorial/docs/CSAATutorial.pdf.

Code Listing

This is a CUDA implementation of SRAA for 4× Latin-square G-bufffers, using depth and normals in the bilateral filter. In practice we statically evaluate all branches and loops in our code generator.

```
float3 normal(int x, int y)
{ return normalBuffer.Get(x, y) * 2.0 -
         make_float3(1, 1, 1); }

float depth(int x, int y)
{ return depthBuffer.Get(x, y); }

float bilateral(float3 centerN, float centerZ,
                float3 tapN,    float tapZ) {
  return exp(-scale * max(
   (1.0 - dot(centerN, tapN)),
   depthScale * abs(centerZ - tapZ)));
}

// Iterate the ``center'' (cx, cy) of the filter
// over the samples in the pixel at (x, y)
float weights[9] = {0};
for (int cy = y; cy < (y + 2); ++cy) {
   for (int cx = x; cx < (x + 2); ++cx) {

    float3 N = normal(cx, cy);
    float  Z = depth(cx, cy);

    float tmpWeights[9] = {0};
    float sum = 0.0f;

    // Iterate over the neighboring samples
    for (int j = 0; j < 3; ++j) {
      for (int i = 0; i < 3; ++i) {

        // If inside filter support
        if ((abs(i - 1 - cx)) <= 1) &&
            (abs(j - 1 - cy)) <= 1)) {
          int tapX = x + i - 1;
          int tapY = y + j - 1;

          // Compute the filter weight
          float w = bilateral(N, Z,
            depth(tapX, tapY), normal(tapX, tapY)));

          tmpWeights[i + j * 3] = w;
          sum += w;
        }
      }
    }
    for (int t = 0; t < 9; ++t)
      weights[t] += tmpWeights[t] / sum;
  }
}

// Apply the filter
float3 result = make_float3 (0, 0, 0);
for (int j = 0; j < 3; ++j)
   for (int i = 0; i < 3; ++i)
      result += weights[i + j * 3] * 0.25 *
          colorBuffer.Get(x + i - 1, y + j - 1);
```