High Performance Graphics (2014)
Jonathan Ragan-Kelley and Ingo Wald (Editors)

Streaming G-Buffer Compression for Multi-Sample Anti-Aliasing

E. Kerzner[1,2†] and M. Salvi[3‡]

[1]SCI Institute [2]School of Computing [3]Intel Corporation

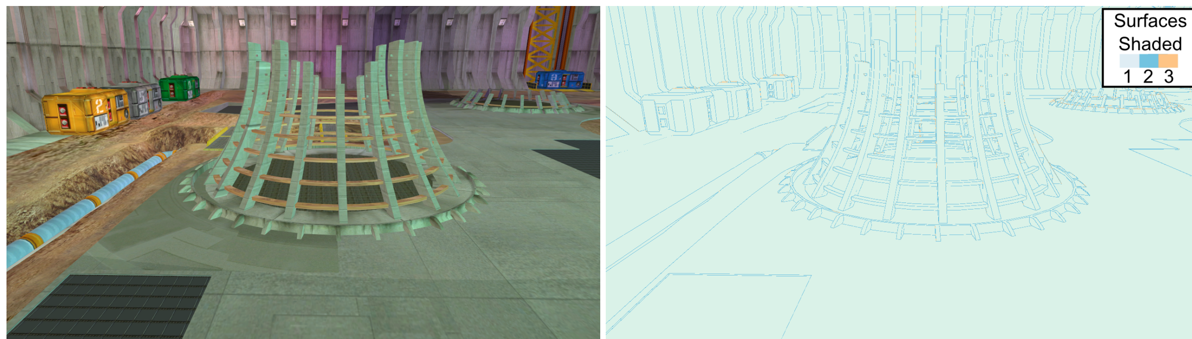

Figure 1: *Our streaming compression algorithm reduces the memory usage and shading costs associated with multi-sample anti-aliasing (MSAA) coupled to deferred shading. Here, a scene rendered with our algorithm using 8x MSAA (left) reduces memory usage by 50% and total running time by 30% when compared to an optimized deferred shading implementation. In most cases we shade once per pixel (right) even when multiple geometric primitives cover it.*

Abstract

We present a novel lossy compression algorithm for G-buffers that enables deferred shading applications with high visibility sampling rates. Our streaming compression method operates in a single geometry rendering pass with a fixed, but scalable, amount of per pixel memory. We demonstrate reduced memory requirements and improved performance, with minimal impact on image quality.

1. Introduction

Computing images with high visibility sampling rates is a long standing problem in real-time rendering due to the much increased demand for shading, storage, and memory bandwidth. Shading costs can be lowered with methods that decouple visibility determination from shading, such as multi-sample anti-aliasing (MSAA), while color and depth compression techniques employed by modern graphics hardware can effectively reduce memory bandwidth, but do not reduce storage requirements. Moreover, applications that defer shading further increase the amount of memory necessary to store to-be-shaded-attributes (i.e. the G-buffer) and cannot directly take advantage of hardware support for decoupling visibility and shading.

Our lossy streaming compression algorithm reduces the memory usage and shading costs of MSAA when coupled to deferred shading, with little impact on image quality. We exploit hardware support for MSAA to compute each fragment's coverage mask, but rather than storing shading data per sample or per fragment we coalesce fragments originating from the same geometric feature into a single *surface* data structure. This process makes it possible to render a compressed representation of the G-buffer in a single ren-

† kerzner@sci.utah.edu
‡ marco.salvi@intel.com

© The Eurographics Association 2014.

DOI: 10.2312/hpg.20141088

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

dering pass. We further increase performance by shading per surface instead of per sample.

2. Previous Work

Multi-sampling anti-aliasing (MSAA) first introduced the idea of mapping a single shading sample to all samples covered by a primitive within a pixel [Ake93]. This technique is effective at reducing shading computations as long as geometric primitives cover many visibility samples. In the limit case when each visibility sample is covered by a different primitive, performance degrades to super-sample anti-aliasing (SSAA) where visibility and shading are computed at the same rate [FGH*85, DWS*88, Mam89, HA90]. Fatahalian et al. [FBH*10] address this case in a tessellation based pipeline by merging pixel quads from adjacent primitives in the same patch prior to shading. These techniques, if implemented in a single rendering pass, store n visibility and shading samples per pixel, which can significantly impact memory and bandwidth requirements when n is *large*.

To improve performance modern graphics hardware stores and transmits color and depth data using proprietary compression formats. These lossless algorithms lower memory bandwidth but do not reduce the size of the frame buffer. Exceptions are coverage sampling anti-aliasing (CSAA) and enhanced quality anti-aliasing (EQAA), lossy extensions to MSAA where color data are decoupled from coverage. The latter is sampled by the rasterizer at higher rate than both color and visibility [You07, AMD12], which increases image quality on primitive edges without requiring more color and visibility samples.

The Z^3 algorithm [JC99] samples coverage and visibility at the same rate but allocates a small and fixed a priori number of fragments per pixel. If this per-pixel buffer overflows fragments are merged while trying to minimize image artifacts. Lee et al. [LK00] replace the fixed size buffer of Z^3 with a dynamically allocated linked list of fragments, similar to an A-buffer [Car84], and augment fragment data with object tags to reduce the likelihood of merging fragments that belong to different objects.

If shading is performed after geometry is rendered the one-to-many mapping from shading samples to visibility samples provided by MSAA is lost. Methods that defer the bulk of their shading computations will therefore inefficiently sample shading and visibility at the same rate, similar to SSAA. These applications can still take advantage of lossless bandwidth compression techniques employed by GPUs, but memory usage is negatively impacted since deferred shading techniques significantly increase the amount of data stored for each sample, the so-called G-buffer. These additional costs cannot be reduced by applying some of the aforementioned lossy compression methods, since their merge heuristics use color information, which is not available prior to shading.

To address these issues numerous image anti-aliasing post-processing techniques that do not require MSAA have been proposed [JGY*11]. These were first pioneered by Reshetov's work on morphological anti-aliasing [Res09]. Such methods typically provide high performance and easy integration with rendering engines but cannot address aliasing introduced by sub-pixel features, which often generate temporal artifacts. More advanced post-processing methods can reduce these artifacts by sampling visibility at higher rate while still shading only once per pixel [CML11,Res12].

Ragan-Kelley et al. [RKKS*07] propose to preserve the relation between visibility and shading samples with an indirect frame buffer that explicitly associates each shading sample to one or more visibility samples, allowing reduced shading rates similarly to MSAA. To further save memory the indirect frame buffer can be replaced by a visibility buffer that encodes triangle and instance IDs which are later used to retrieve and shade geometry associated to the visible samples, thus eliminating the G-buffer [BH13]. Sort-based deferred shading augments the visibility buffer by storing shading coordinates. The samples are then sorted on a per-screen-tile basis to extract a list of to-be-shaded primitives and relative shading locations [CTM13]. This lowers shading requirements as many visibility samples tend to map to the same shading location.

The method introduced by Lauritzen [Lau10] analyzes the G-Buffer content prior to shading to discover pixels covered entirely a single geometric feature and adaptively shades per pixel or per sample. This can reduce the overall shading costs but does not change the memory requirements. Surface based anti-aliasing (SBAA) analyzes the result of a first simplified MSAA rendering pass to discover which fragments could be merged and stored in the G-buffer in a subsequent rendering pass. By allocating a small number of G-buffer samples per pixel (e.g. 2 or 3) SBAA acts as a compression algorithm, reducing both the size of the G-Buffer for high MSAA rates and the number of to-be-shaded samples [SV12].

Similar to SBAA the method presented in this paper analyzes the stream of incoming fragments to generate an on-the-fly lossily compressed representation of the G-buffer but unlike SBAA our method acts in a single rendering pass. The compressed G-buffer requires two or three samples per pixel and it is particularly advantageous at high visibility sampling rates (e.g. 8 samples per pixel or more) where it can reduce memory and shading requirements compared to many deferred shading techniques based on MSAA.

3. Algorithm

As already noted in previous work [Lau10,SML11] it is possible to exploit groups of primitives forming geometric features with little or no curvature (i.e. *surfaces*) to locally reuse shaded samples and thus reduce the shading rate. We

© The Eurographics Association 2014.

```
struct surface {
    uint  depth;
    half2 depth_dxdy;
    uchar coverage;
    uchar depthResolvedCoverage;
    struct sample
    {
        uint albedo;
        uint normal;
        ...
    } GBufferData;
} SurfaceData;
```

Figure 2: *Surface data structure. Every pixel in the compressed G-bufffoccler stores a fixed length array of surfaces.*

take advantage of this observation by merging to-be-shaded fragments into surfaces in a streaming fashion. Unlike previous deferred shading methods we do so *in a single rendering pass* (that we call a *compression pass*) as primitives are rendered into our compressed G-buffer.

Our compressed G-buffer uses a new per-pixel data structure that encodes a fixed length array of surfaces. Each surface consists of a G-buffer sample and additional information used in our compression algorithm, such as depth (computed at the pixel center), depth derivatives (with respect to screen-space), and two coverage masks (that we discuss in Section 3.2.1). An example structure is in Figure 2. Although one pixel may be covered by up to eight sub-pixel samples we have found that three sub-pixel surfaces provide adequate image quality.

After our streaming compression pass we output pixel color from the compressed G-buffer. We achieve this by averaging the color contribution of each shaded surface weighted by the number of samples it covers.

3.1. Merge Metrics

During our streaming compression pass we merge fragments belonging to planar geometric features. We use the following merge metric to determine when fragments belong to the same surface. It consists of three conditions that must be mutually satisfied:

- aligned normals,
- overlapping depth intervals, and
- mutually exclusive coverage masks.

Next, we more clearly define these conditions.

If two surfaces have normals n_0 and n_1 then these normals are aligned only if:

$$n_0 \cdot n_1 > cos(\alpha_\varepsilon)$$

We found $\alpha_\varepsilon = \frac{\pi}{4}$ to provide high quality images while reducing the shading costs by merging surfaces.

We compute surface depth intervals within a pixel using the depth derivatives with respect to screen space position. Here is an example of computing a surface's minimum depth:

```
float dz_dx = surface.depth_dxdy.x;
float dz_dy = surface.depth_dxdy.y;
float z_min = surface.depth
            - abs(dz_dx)
            - abs(dz_dy);
```

This estimates minimum depth at one of the pixel's corners. We use the same process to compute maximum depth. If two surfaces have depths z_{min_0}, z_{max_0}, z_{min_1}, and z_{max_1}, these ranges overlap only if:

$$z_{min_0} \leq z_{max_1} \;\; \&\& \;\; z_{min_1} \leq z_{max_0}$$

This technique estimates surface depth range and allows merges to occur when there is potential for depth range overlap. We also require coverage masks to be mutually exclusive. Two surfaces with coverage C_0 and C_1 are mutually exclusive only if:

$$C_0 \cap C_1 == \emptyset$$

This is based on our observation that geometric features seldom contain overlapping primitives.

We demonstrate these three conditions through examples shown in Figure 3. Particularly, we show cases where artifacts occur when skipping any one of three conditions.

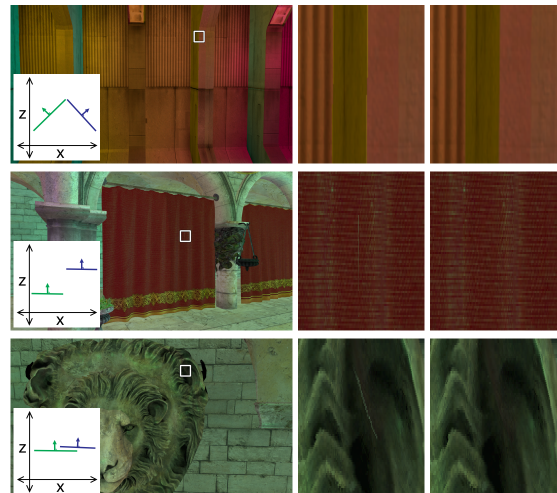

Figure 3: *Skipping any one of our three merge metric conditions causes artifacts due to incorrect merges. Removing conditions of aligned normals (top), overlapping depth intervals (middle), and mutually exclusive coverage masks (bottom) leads to incorrect merges. Each example shows the artifact context and its cause (left), a close-up of the artifact (center), and the correct image using all three conditions (right).*

© The Eurographics Association 2014.

3.2. Compression

We now detail our G-buffer compression algorithm.

For each frame rendered we store the first fragment covering a pixel as the first element in the surface array. Specifically, we store its associated G-buffer data, depth, depth derivatives and coverage mask (Figure 2).

We attempt to merge each subsequent fragment covering the pixel with all existing surfaces in the pixel surface array. If the merge is successful we combine their coverage masks and average the rest of the G-buffer and surface data. If the merge fails and the pixel array is not full, we insert a new surface, keeping the array in front-to-back depth order. Otherwise, if the surface buffer is full we discard either an existing surface or the incoming fragment.

Discarding information potentially introduces visible artifacts to the image. Although we do not have final color information at G-buffer compression time, we eventually compute it as an average of surface colors weighted by their coverage. It follows that the surface or fragment with the smallest coverage is likely to have a small impact on final pixel color. Thus, we use coverage as a heuristic for discarding fragments: we discard the surface with the smallest coverage. While our coverage heuristic may not minimize overall error, in general it does not introduce noticeable artifacts into the final image.

3.2.1. Coverage Determination

In current graphics APIs the coverage mask available as input to the fragment shader is not affected by the depth test. Moreover, subsequent fragments can affect the coverage data previously stored in the surface array and even entirely occlude surfaces. Therefore, we must account for occlusion to approximate coverage for all stored surfaces and the incoming fragment. We do so by *fusing* occluders (i.e. per-pixel surfaces and the incoming fragment) in front-to-back order; fusing the first occluder with the second, the resulting occluder with the third, and so on, updating the *depth-resolved* coverage mask for each occluder. Note that the depth-resolved coverage is stored in the surface data along with the standard coverage (Figure 2). The former is used to determine which surfaces must be discarded while the latter is more accurate when merging fragments.

Since occluders may be inter-penetrating we use depth derivatives to compute the per-pixel depth intervals of two neighboring occluders. Our depth-resolved coverage computation depends on whether these depth intervals are disjoint. If they are disjoint then the further surface may be partially or entirely occluded and we save this information by removing occluded samples from its depth-resolved coverage mask. Otherwise, if they are overlapping we leave the depth-resolved coverage unchanged and continue with the occluder fusion.

Following occluder fusion we have the approximate depth-resolved coverage of each occluder. We safely discard occluders that have depth-resolved coverage of zero as they do not contribute to the final pixel color. If no occluder has zero coverage we discard the one with the smallest coverage. This step may introduce artifacts, especially when the discarded samples remain uncovered as we treat uncovered samples as the background color in pass. To avoid this problem we use a flag to mark pixels that discard occluders with non-zero coverage. This flag is used at resolve time to ignore uncovered samples and prevent them from aliasing to the background (see Section 4).

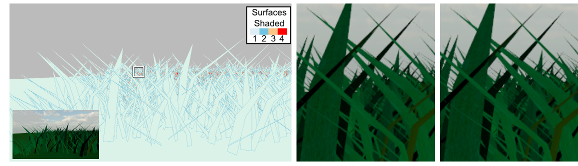

Figure 4: *Three surfaces adequately represent pixel color. We implemented our algorithm using four surfaces per pixel (left); black pixels show where four surfaces contribute to the final pixel color. Even in these cases our three surface implementation (center) is indistinguishable from a reference 8x MSAA image (right).*

3.3. Resolve

The resolve pass computes final pixel color by accumulating each surface's color contribution. For each pixel we compute the surface weights by counting each surface's unoccluded samples. Next, we shade surfaces with non-zero weights. Finally, we output the weighted average of surface colors.

To determine surface weights we compute the depth at each sample covered by each surface. We evaluate sample depth similar to the depth interval estimation described in Section 3.1, however, instead of computing the depth at pixel corners we use each sample's sub-pixel location. We resolve sample visibility within a pixel using each sample's estimated depth. We iterate over the surfaces in front-to-back order while storing the closest depth of each sample and the surface index covering that sample in two arrays. As we process surfaces we update these arrays to maintain depths and surface indexes closest the viewer. After processing all surfaces we compute surface weights by counting their unoccluded samples. This process correctly resolves coverage for inter-penetrating surfaces.

Finally, we shade surfaces with non-zero weights and compute a weighted average of their colors. This eliminates unnecessary per-sample shading computations common to MSAA G-buffer implementations.

4. Implementation

Our algorithm updates the compressed G-buffer in a streaming fashion via read-modify-write memory operations. We

© The Eurographics Association 2014.

note that simply using atomic operations cannot avoid data races caused by concurrently shaded fragments accessing the same pixel data. A per-pixel critical section could eliminate data races, but it would still cause temporal artifacts due to our lossy compression scheme not operating with deterministic ordering. To guarantee data race free updates in primitive submission order we use Intel's PixelSync extension for DirectX 11 applications [Sal13].

Our implementation resembles a standard G-buffer, but we replace G-buffer construction with our streaming compression and we substitute per-sample (or per-pixel) G-buffer shading with per-surface shading. Next, we discuss four main points of our implementation.

We leverage a multi-sampled depth buffer for early-z rejection during our compression pass. Early-z testing avoids unnecessary shader executions. This significantly benefits performance as we describe in Figure 5.

In our surface buffer we store a small fixed number of surface structures per pixel. We organize this buffer such that shaders for each pixel access only their list of surfaces. We found storing surfaces in a tiled fashion achieves the best performance by exploiting spatial and temporal locality of pixel shader memory access. We demonstrate that three surfaces per pixel reduces shading costs and memory usage with minimal impact on image quality. In Figure 4, we show that even in places where four surfaces would contribute to pixel color, our three surface implementation is indistinguishable from standard 8x MSAA.

To reduce the amount of data transferred by memory operations we keep information about each pixel's surfaces in a 2D texture with 4 bytes available to each pixel. In this so-called *count texture* we maintain per pixel surface data: surface count (2 bits), depth-ordered list of surface indexes (6 bits), and the discarded sample flag (1 bit). (Although each pixel uses only 9 bits of the 4 byte count texture, DirectX requires that shaders write to textures with an element size of at least 4 bytes.) By using a depth-ordered list of indexes, we minimize global memory operations and avoid dynamic array access which may not be supported by hardware.

We also use the count texture to reduce the cost of clearing memory between frames. Although we always have to clear the depth buffer between frames, we avoid clearing the surface buffer by setting each pixel's surface count to zero. In the following frame all data in the surface buffer is overwritten.

5. Results

Our technique offers a 50% reduction of memory usage for deferred shading applications coupled to high visibility sampling rates. We also offer increased performance when compared to other MSAA G-Buffer techniques, such as *Lauritzen's* [Lau10] algorithm that adaptively shades at pixel or

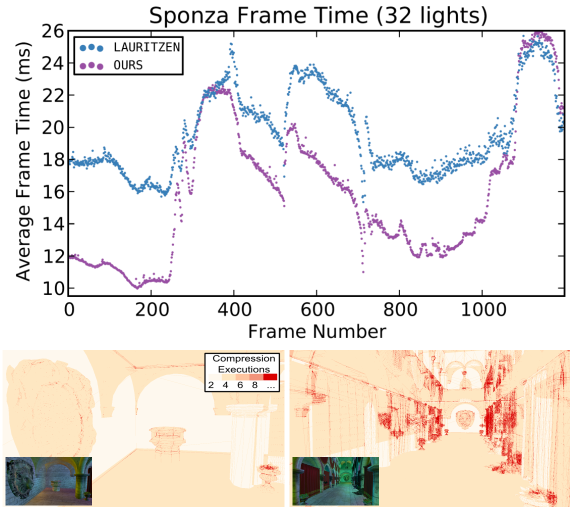

Figure 5: *The performance of* Our *algorithm compared to* Lauritzen *while rendering the Sponza with 32 lights.* Our *performance is directly related to the number of fragments processed during G-buffer construction. When relatively few fragments are processed the compression costs about 4.0ms of the 11.3ms total frame (Frame No. 151, bottom left). In contrast, as depth complexity increases, the cost of compression balloons to 20.7ms of the 26ms total frame time (Frame No. 1142, bottom right).*

sample resolution. We measured performance of *Our* and *Lauritzen's* algorithms using DirectX 11 implementations. All of our metrics were gathered on an Intel Iris Pro (Core i7 @ 2.0 Ghz) with 8Gb RAM running Windows 7.

Table 1 contains detailed memory usage and shading costs of our algorithm compared to *Lauritzen*. When running with eight visibility samples per pixel, compared to *Lauritzen* we use 50% less memory. We note that the memory usage of *Lauritzen's* implementation includes eight 20 byte G-buffer samples and an 8x multi-sampled frame buffer for performing intermediate lighting computations. Our algorithm also

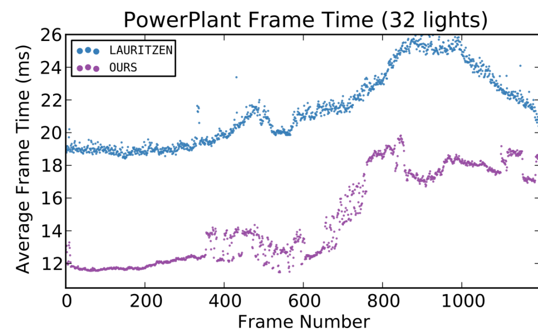

Figure 6: *The performance of* Our *algorithm compared to* Lauritzen *while rendering the PowerPlant with 32 lights.*

© The Eurographics Association 2014.

Technique	Bytes/Pixel	% *Lauritzen* Mem.
Lauritzen 8x (4x)	192 (96)	100 (100)
Ours 8x (4x)	96 (80)	50.0 (83.3)

Scene	*Lauritzen* (#)	*Ours* (%)
Sponza 8x (4x)	1.23M (1.03M)	77.8 (92.2)
PowerPlant 8x (4x)	1.39M (1.09M)	70.5 (88.7)
Grass 8x (4x)	2.55M (1.49M)	44.9 (72.9)

Table 1: *A comparison of* Our *and* Lauritzen *with respect to memory usage (left) and shading executions (right).* Lauritzen's *memory usage includes per sample G-buffer data and a multi-sampled intermediate buffer for performing lighting computations.* Our *execution count is expressed as a percent of* Lauritzen's *execution count.*

significantly decreases the number of surfaces (or samples) that need to be shaded: in some scenes we perform 44.9% of *Lauritzen's* shading computations.

When comparing frame time, we often outperform *Lauritzen* (see Figures 5, 6). Even in cases where we perform on par with *Lauritzen* we reduce memory usage by as much as 50%. In these cases, the compression pass is the bottleneck of our algorithm as shown in Figure 5. In Figure 7 we compare image quality across three methods: *Lauritzen*, *Our*, and a *Reference* 8x MSAA forward shading implementation. Both *Our* and *Lauritzen's* algorithm generate images that are often indistinguishable from the *Reference* images.

Although we could further reduce memory consumption and improve performance for our method by storing only two surfaces per pixel, we found that at least three are required to obtain acceptable image quality in the vast majority of cases. For instance, corners where three walls connect always appear aliased when using two surfaces.

5.1. Failure Cases

False positives of our merge metric results in aliasing as shown in Figure 8. The incorrect merge causes aliasing along the boundary of light grey and black rectangle as we average their G-buffer data. This problem can be resolved by tightening the α_ε used in our merge metric.

6. Conclusion

We demonstrated a novel streaming compression algorithm for hardware multi-sampled G-buffers. Our method significantly reduces memory requirements with minimal impact on image quality and scales very well as the number of visibility samples increases. The cost of each new visibility sample is independent from the amount of information stored in a G-buffer sample, requiring only 36s bit of memory (e.g. one 32 bit depth sample and two coverage bits). We believe our method will provide even better results on future graphics hardware supporting higher MSAA rates (e.g. 16x or more). Also the memory usage of our technique could be further reduced by having read/write access to the depth buffer in the compression pass. In the future we plan to investigate simpler fragment merging and discarding schemes to further improve performance.

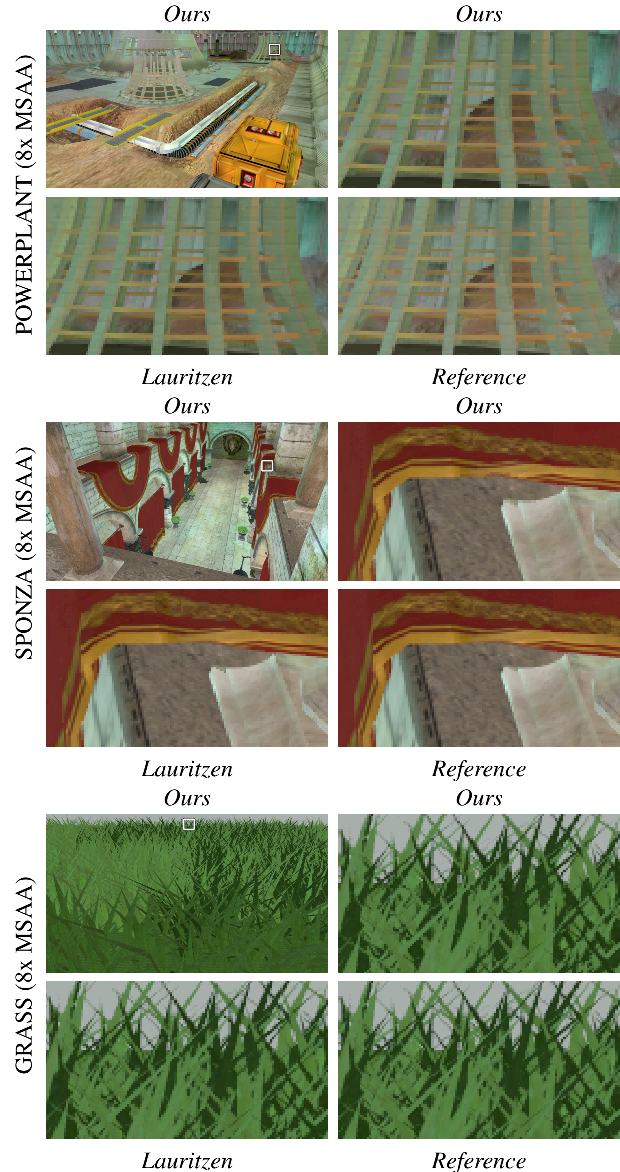

Figure 7: *Comparison of* Our *algorithm with* Lauritzen *and* Reference.

© The Eurographics Association 2014.

Figure 8: *Our algorithm may introduce artifacts due to incorrectly merging surfaces (overview left and zoomed center). These artifacts can be resolved by tightening the normal alignment condition (right).*

7. Acknowledgments

We thank Sungkil Lee for the GRASS scene, Nina McCurdy for her help creating diagrams, and Miriah Meyer for her encouragement throughout this project. Thanks to Karthik Vaidyanathan and the rest of Intel's Advanced Rendering Technology Team for their contributions and support. We thank Chuck Lingle, Tom Piazza and David Blythe, also at Intel, for supporting this research. Ethan Kerzner was supported in part by an internship and hardware donations from Intel.

References

[Ake93] AKELEY K.: RealityEngine Graphics. In *Proceedings of SIGGRAPH 93* (1993), ACM, pp. 109–116. 2

[AMD12] AMD: *EQAA Modes for AMD 6900 Series Graphics Cards*. Tech. rep., AMD, 2012. 2

[BH13] BURNS C. A., HUNT W. A.: The visibility buffer: A cache-friendly approach to deferred shading. *Journal of Computer Graphics Techniques (JCGT) 2*, 2 (August 2013), 55–69. URL: http://jcgt.org/published/0002/02/04/. 2

[Car84] CARPENTER L.: The A-buffer, an Antialiased Hidden Surface Method. In *Computer Graphics (Proceedings of SIGGRAPH 84)* (1984), vol. 18, ACM, pp. 103–108. 2

[CML11] CHAJDAS M. G., MCGUIRE M., LUEBKE D.: Subpixel Reconstruction Antialiasing for Deferred Shading. In *Symposium on Interactive 3D Graphics and Games* (2011), ACM, pp. 15–22. 2

[CTM13] CLARBERG P., TOTH R., MUNKBERG J.: A Sort-Based Deferred Shading Architecture for Decoupled Sampling. *ACM Transactions on Graphics, 32*, 4 (2013), 141:1–141:10. 2

[DWS*88] DEERING M., WINNER S., SCHEDIWY B., DUFFY C., HUNT N.: The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics. In *Computer Graphics (Proceedings of SIGGRAPH 88)* (1988), vol. 22, ACM, pp. 21–30. 2

[FBH*10] FATAHALIAN K., BOULOS S., HEGARTY J., AKELEY K., MARK W. R., MORETON H., HANRAHAN P.: Reducing Shading on GPUs using Quad-Fragment Merging. *ACM Transactions on Graphics, 29*, 4 (2010), 67:1–67:8. 2

[FGH*85] FUCHS H., GOLDFEATHER J., HULTQUIST J. P., SPACH S., AUSTIN J. D., BROOKS JR. F. P., EYLES J. G., POULTON J.: Fast Spheres, Shadows, Textures, Transparencies, and Imgage Enhancements in Pixel-Planes. In *Computer Graphics (Proceedings of SIGGRAPH 85)* (1985), vol. 19, ACM, pp. 111–120. 2

[HA90] HAEBERLI P., AKELEY K.: The Accumulation Buffer: Hardware Support for High-Quality Rendering. In *Computer Graphics (Proceedings of SIGGRAPH 90)* (1990), vol. 24, ACM, pp. 309–318. 2

[JC99] JOUPPI N. P., CHANG C.-F.: Z^3: An Economical Hardware Technique for High-Quality Antialiasing andTransparency. In *Graphics Hardware* (1999), HWWS '99, ACM, pp. 85–93. 2

[JGY*11] JIMENEZ J., GUTIERREZ D., YANG J., RESHETOV A., DEMOREUILLE P., BERGHOFF T., PERTHUIS C., YU H., MCGUIRE M., LOTTES T., MALAN H., PERSSON E., ANDREEV D., SOUSA T.: Filtering approaches for real-time antialiasing. In *ACM SIGGRAPH Courses* (2011). 2

[Lau10] LAURITZEN A.: Deferred rendering for current and future rendering pipelines. Beyond Programmable Shading course, SIGGRAPH 2010. http://bps10.idav.ucdavis.edu/, 2010. 2, 5

[LK00] LEE J.-A., KIM L.-S.: Single-pass full-screen hardware accelerated antialiasing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 2000), HWWS '00, ACM, pp. 67–75. URL: http://doi.acm.org/10.1145/346876.348225, doi:http://doi.acm.org/10.1145/346876.348225. 2

[Mam89] MAMMEN A.: Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Comput. Graph. Appl. 9* (1989), 43–55. 2

[Res09] RESHETOV A.: Morphological Antialiasing. In *Proceedings of High Performance Graphics 2009* (2009), ACM, pp. 109–116. 2

[Res12] RESHETOV A.: Reducing aliasing artifacts through resampling. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2012), EGGH-HPG'12, Eurographics Association, pp. 77–86. URL: http://dx.doi.org/10.2312/EGGH/HPG12/077-086, doi:10.2312/EGGH/HPG12/077-086. 2

[RKKS*07] RAGAN-KELLEY J., KILPATRICK C., SMITH B. W., EPPS D., GREEN P., HERY C., DURAND F.: The Lightspeed Automatic Interactive Lighting Preview System. *ACM Transactions on Graphics, 26*, 3 (2007), 25:1–25:11. 2

[Sal13] SALVI M.: Pixel Synchronizaton: Solving Old Graphics Problems with New Data Structures. SIGGRAPH 2013 Advances in Real-Time Rendering in Games course, 2013. URL: http://advances.realtimerendering.com/s2013/. 5

[SML11] SALVI M., MONTGOMERY J., LEFOHN A.: Adaptive transparency. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG '11, ACM, pp. 119–126. URL: http://doi.acm.org/10.1145/2018323.2018342, doi:10.1145/2018323.2018342. 2

[SV12] SALVI M., VIDIMČE K.: Surface based anti-aliasing. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2012), I3D '12, ACM, pp. 159–164. URL: http://doi.acm.org/10.1145/2159616.2159643, doi:10.1145/2159616.2159643. 2

[You07] YOUNG P.: *Coverage Sampled Anti-Aliasing*. Tech. rep., NVIDIA Corporation, 2007. URL: http://news.developer.nvidia.com/2007/01/coverage_sampli.html. 2

© The Eurographics Association 2014.