Aggregate G-Buffer Anti-Aliasing

Cyril Crassin
NVIDIA

Morgan McGuire
NVIDIA / Williams College

Kayvon Fatahalian
Carnegie Mellon University

Aaron Lefohn
NVIDIA

Figure 1: *Large image rendered with Agregate G-Buffer Anti-Aliasing (AGAA). The AGAA results (red outlines) shade only twice per pixel, give comparable results to the MSAA reference image shaded eight times per pixel, and use 33% less memory. AGAA reduces aliasing by prefiltering the scene's sub-pixel geometric detail (foliage, thin railings, etc.) into an aggregate G-buffer that models the distribution of geometry projecting into each pixel.*

Abstract

We present Aggregate G-Buffer Anti-Aliasing (AGAA), a new technique for efficient anti-aliased deferred rendering of complex geometry using modern graphics hardware. In geometrically complex situations, where many surfaces intersect a pixel, current rendering systems shade each contributing surface *at least* once per pixel. As the sample density and geometric complexity increase, the shading cost becomes prohibitive for real-time rendering. Under deferred shading, so does the required framebuffer memory. AGAA uses the rasterization pipeline to generate a compact, pre-filtered geometric representation inside each pixel. We then shade this at a fixed rate, independent of geometric complexity. By decoupling shading rate from geometric sampling rate, the algorithm reduces the storage and bandwidth costs of a geometry buffer, and allows scaling to high visibility sampling rates for anti-aliasing. AGAA with 2 aggregate surfaces per-pixel generates results comparable to 8x MSAA, but requires 30% less memory (45% savings for 16x MSAA), and is up to 1.3x faster.

Keywords: anti-aliasing, GPU architecture, graphics pipelines, shading, decoupled shading, pre-filtering

1 Introduction

High-quality renderers sample geometrically complex environments, such as those containing foliage, fur, or intricate geometry (e.g. the tables and furniture details in figure 1) at high rates to capture sub-pixel detail. These environments are challenging for any rendering system, but are particularly difficult for real-time systems, especially those based on deferred shading, a technique frequently employed by games.

First, despite the high performance of modern GPUs, evaluating the shading function at high sampling rates remains too costly for real-time applications. Second, because a deferred shading system delays all shading computations until after geometric occlusions have been resolved, it must buffer shading inputs for all samples in the renderer's G-buffer. At high sampling rates, the storage and memory bandwidth costs of generating and accessing this buffer become prohibitive. For example, a 1920×1080 G-buffer holding 16 samples per pixel encoded using a typical 20-bytes-per-sample layout requires over 600 MB of storage.

To reduce these costs, game engines typically provision storage for, and limit shader evaluation to, only a few samples per pixel (e.g. four [Tatarchuk et al. 2013]). Post-process anti-aliasing techniques [Chajdas et al. 2011; Lottes 2009] increase image quality using neighboring pixels or temporally re-projected sample information from previous frames. Such techniques generally introduce blur and fail to capture the appearance of sub-pixel details, as illustrated in figure 2.

In this paper, we focus on efficiently shading scenes with many distinct geometric elements contributing to the appearance of a single pixel, in the context of real-time deferred rendering systems. The core idea of our technique is to decouple the rate at which lighting

is sampled, which we want to keep as low as possible, from the sampling rate of geometry and materials. Our goal is to perform this decoupling while preserving the appearance of high frequency details in the image.

We achieve this goal by taking inspiration from surface-based pre-filtering and voxel-based pre-filtering techniques (cf. section 2). We create a new deferred shading pipeline that dynamically generates and shades compact per-pixel aggregates of statistically defined attributes, instead of samples from individual scene surfaces. We call this new data structure an *aggregate G-buffer*. It compactly encodes the distribution of depths, normals, and other shading quantities needed for shading.

We find that only two to three shader evaluations per pixel are required to achieve image quality (even under motion) commensurate with densely point-sampled results. Because the proposed method operates on the outputs of the rasterizer, it is highly general, avoids analyzing and storing statistics for off-screen or occluded geometry, and supports dynamic scenes.

The key contributions of this work are:

- A new deferred rendering pipeline that dynamically generates and shades pre-filtered shading attributes.
- A clustering scheme which distributes geometric samples among aggregates in order to maximize shading quality.
- A screen-space pre-filtering technique that dynamically filters attributes from potentially disjoint primitives.
- A shading scheme which operates directly on pre-filtered attributes and handles shadowing correctly.

2 Related Work

Decoupling shading rate from visibility sampling rates is a key idea in real time rendering, and is used in the context of both forward [Akeley 1993] and deferred rendering [Lauritzen 2010] as well as in the context of stochastic rasterization [Clarberg et al. 2013; Liktor and Dachsbacher 2012; Ragan-Kelley et al. 2011]. The key idea of each of these approaches is to reuse shading results across visibility samples from the *same surface*. Our work is based on the same reuse principle, but reuse is applied across multiple (potentially disconnected) primitives.

A simple way to reduce shading work in a deferred shading pipeline is to shade once per pixel if it contains only one surface, or to shade every sample in all other cases [Lauritzen 2010]. This scheme does not reduce memory requirements and speeds up rendering only when triangles are large, and there is only one triangle covering most pixels. Other techniques hallucinate additional detail through data-dependent resampling of shading results in adjacent pixels [Reshetov 2009; Chajdas et al. 2011; Reshetov 2012] or reprojection of results from prior frames [NVI 2014; Herzog et al. 2010].

Our work improves on the approach of Salvi et al. [2012], which analyses the results of dense geometry sampling during rasterization to identify and retain exactly one fragment from the n "most important" surfaces per pixel. However their method discards information from all other surfaces, which leads to aliasing in situations where many surfaces contribute to a pixel's appearance. Kerzner and Salvi [2014] improve on this technique by designing a single-pass rendering algorithm which allows merging shading attributes belonging to similar non-intersecting planes, at the cost of a software evaluation of visibility using an interlocked fragment shader.

Our efforts to maximize image quality given fixed G-buffer storage also bare similarity to K-buffer-based schemes that merge shading outputs for anti-aliasing and transparency [Jouppi and Chang 1999; Bavoil et al. 2007].

Our approach to modeling the distribution of rasterized geometry in a pixel takes inspiration from prior efforts to encode and prefilter geometric detail stored in 2D or 3D textures [Olano and Baker 2010; Toksvig 2005; Dupuy et al. 2013; Han et al. 2007; Bruneton and Neyret 2011; Olano and North 1997; Fournier 1992a; Fournier 1992b]. However, rather than tabulate and prefilter geometric detail for each surface in a rendering preprocess, our work leverages rasterization to dynamically aggregate per-pixel statistics across multiple surfaces. Unlike dynamic voxelization-based approaches [Christensen and Batali 2004] which perform similar aggregation, our system operates in screen space using a compact, fixed amount of storage, while guaranteeing per-pixel detail.

Techniques such as sprites and billboard clouds [Décoret et al. 2003; Lacewell et al. 2006] allow reflectance from complex geometries to be properly pre-filtered and renderer at low cost but betray their lack of 3D information when viewed up close. Further, they do not encode surface attributes for dynamic shading and incur large storage cost for animated objects.

Last, an alternative approach to reducing rendering costs for geometrically complex scenes is to simplify input geometry prior to rasterization by approximating it with a lower resolution model (see Luebke et al. [2002] for a survey of techniques). Simplification techniques seek to discard a subset of scene elements while adjusting surface material properties using aggregate statistics to preserve overall object appearance [Cohen et al. 1998; Yoon et al. 2006; Cook et al. 2007]. These approaches are attractive in that they also reduce that cost of geometry processing during rendering, while our work, like other dynamic filtering approaches, requires additional geometry processing. However, without fixed-shading rate guarantees they do not necessarily reduce the cost of shading or G-buffer storage, which are our primary concerns.

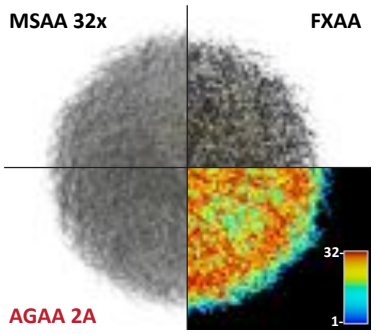

Figure 2: *Complex objects like the Fur Ball exhibit significant sub-pixel details (up to 32 triangles per pixel here) and cannot be anti-aliased using post-process screen-space anti-aliasing techniques like FXAA [Lottes 2009] (top-right). In contrast, our technique allows capturing these sub-pixel details, while shading only twice per pixel (bottom-left).*

3 Algorithm

Our method analyzes post-projection geometry and represents the collection of distinct geometric primitives visible in each pixel using a small, fixed number of *geometry aggregates*. Each aggregate corresponds to a subset of the primitives visible in the pixel, including their coverage, the mean and standard deviation of primitive

depths and normals, and mean values of relevant surface attributes.

The following subsections describe a four-step process for generating and storing per-pixel geometry aggregates in an aggregate G-buffer, and then using the aggregates for efficient deferred shading of aggregate detail.

3.1 Overview

Our technique operates within a four-stage renderer illustrated in figure 3:

1. **Dense Visibility Sampling** (depth + compressed normals *prepass*): render geometry, storing depth and compressed normals using n visibility samples per pixel.
2. **Aggregate Assignment**: group surface samples visible in a pixel into c aggregates by analysing per-sample depth + normal buffer.
3. **Aggregate G-buffer generation**: generate aggregate G-buffer by rendering geometry and accumulating shading inputs into c aggregates per pixel.
4. **Aggregate deferred shading**: screen-space *deferred lighting* pass(es) using the aggregate shading inputs.

These steps look very similar to the 4 stages of SBAA [Salvi and Vidimče 2012], nevertheless because the two algorithms make different assumptions, the actual algorithm implemented by each stage is quite different.

3.2 Dense Visibility Sampling

The goal of the first step is to determine geometric visibility at per sample granularity, as well as generating a per-sample normal information that will be used for clustering samples into aggregates in the subsequent stage. We do so by rasterizing scene geometry into a screen-space geometry buffer storing depth (standard *depth buffer*) and a low-precision surface normal information (cf. layout in figure 5) for each sample. We use a high multi-sampling rate (e.g., $8\times$ MSAA which is natively supported by the GPU, and up to 32 samples per pixel with emulation) to ensure the geometry buffer captures the fine-scale geometric details.

In practice, the cost of the dense visibility pass is lower than the subsequent full geometry pass (sampling all the attributes), since it only requires generation of surface depth and normals (we store flat triangle normals and do not evaluate normal maps in this pass). In the case of transparency surfaces, this pass does also sample the alpha map to determine coverage (cf. section 3.6).

The outputs of this pass are a multisampled depth-buffer and a low precision normal buffer. Normals are encoded using (θ, ϕ) spherical coordinates in pixel-space, and stored inside two 8-bit color components (`RG8`, cf. figure 5). Because we are using actual primitive normals, and primitives are back-face culled, then only the visible hemisphere of normal directions needs to be represented.

3.3 Aggregate Assignment

The second step is to assign each of the n visibility samples to one of the c aggregates (e.g., $c = 2, 3, 4$) using a clustering algorithm. This is done within a compute shader pass (cf. implementation details in section 4) by using the per-sample depth and low-precision normal information generated in the previous stage. The output of the aggregate assignment pass is mapping of samples to clusters (that we call *aggregates metadata*). In a 4-cluster configuration, we encode this mapping using two bits per sample as shown in Figure 4. Thus the mapping requires two-bytes per pixel when using 8x-MSAA visibility sampling.

Note that because many scenes contain an emissive skybox that does not require shading, we exclude samples at the maximum depth value from aggregate assignment. Thus, the aggregate sample counts sum to less than n and measure the fractional coverage by objects at finite distance from the camera.

3.3.1 Grouping criteria

In contrast to previous techniques like SBAA [Salvi and Vidimče 2012] or Streaming G-Buffer Compression [Kerzner and Salvi 2014], which group samples which belong to similar surfaces (with similar plane equations), our goal is to minimize errors dues to aggregation of samples with correlated attributes [Bruneton and Neyret 2011]. Consequently, we have no restriction of minimum similarity between primitives' support planes, and we support aggregating samples from different disjoint surfaces.

The shading model (based on pre-filtered attributes) accurately estimates the full lighting computation only when the value taken by the different attributes must be *statistically independent* [Bruneton and Neyret 2011], meaning there should be no link between the probability of occurrence of one attribute and the probability of another one. For instance, if within a pixel there is a set of blue samples which are in shadow and another set of red samples which are lit, then the correlation between the shadowing and the albedo input parameters of the shading equation will produce inaccurate results when filtering them.

Our goal is to assign samples to aggregates in a manner that reduces the likelihood of highly correlated attributes. In practice, most issues arise from correlation between the surface orientations (which determines shading), as well as the shadowing, and the other attributes. In addition, because the simple normal distribution model that we use is uni-modal and isotropic (cf, section 4), a few dissimilar normals can't be represented precisely in the same aggregate and we aim at avoiding this case.

Consequently, we designed the clustering algorithm to favor both shadowing-based and orientation-based grouping of samples. Because the shadowing information is not available at cluster creation time, our algorithm favour distance-based grouping of samples, based on the assumption that shadowing discontinuities are low enough frequency to be captured by spatial locality.

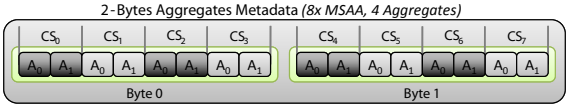

Figure 4: *Illustration of the memory layout description for per-pixel aggregates Metadata information used for per-fragment aggregate selection. In this example we use $8\times$ MSAA rasterizaton and $c = 4$ aggregates. $CS_0 - CS_7$ indicates the two bits (A_0, A_1) used to encode the aggregateID associated to each of the 8 coverage samples.*

3.3.2 Clustering scheme

We cluster surface samples into aggregates using a fast $\mathrm{O}(n \cdot c^2)$ algorithm that can be viewed as a crude approximation to principle component analysis (see Algorithm 1). In this algorithm, the

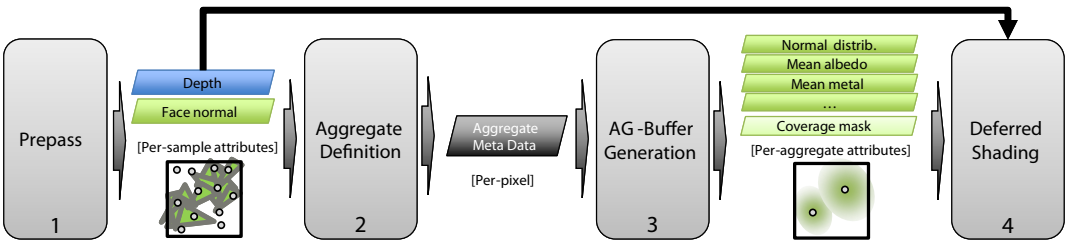

Figure 3: *Functional view of the 4 processing stages of the execution pipeline of the technique, together with the output of each stage and their storage frequency.*

distance d between surface samples a and b is given by:

$$d(xyz_a, xyz_b, \hat{n}_a, \hat{n}_b) = |(xyz_a - xyz_b)/k|^2 + \frac{(1 - \hat{n}_a \cdot \hat{n}_b)}{2}, \quad (1)$$

where constant k is the characteristic length of the scene. It cancels the distance units and specifies the largest scale at which one expects important local detail, i.e., at which orientation differences should give way to position differences. We used $k = 10$ cm for our experiments. This distance function extends meaningful semantics to the scale factor in Chajdas et al.'s [2011] and Reshetov's [2012] *post*-shading aggregating metrics.

Algorithm 1 Aggregate assignment algorithm

1. Define c aggregates
   (a) Read depth and normal for each screen-space sample. Convert depth to position.
   (b) Compute average position and normal of all samples
   (c) Define first aggregate as sample, s_0, that is *farthest* from average (using distance metric based on position + normal) using Equation 1 to compute distance.
   (d) Define second aggregate as sample, s_1, that is farthest from s_0.
   (e) Define each additional aggregate by finding the sample with the largest sum of square distances from the existing aggregates.
2. Assign remaining visible samples to aggregates
   (a) Assign each sample to the closest aggregate.
3. Store a sample mask for each aggregate

Note that in order to reduce shading workload to its minimum, we create aggregates only if they are separated by a minimum distance t from previously defined aggregates. Once the clusters are defined, the algorithm classifies each sample as belonging to the nearest cluster using distance function d.

3.4 Aggregate G-buffer Generation

The third step of the algorithm generates the aggregate G-buffer by rasterizing scene geometry a second time, evaluating material shader inputs at each visibility sample and combining these values to compute a statistical model of the attribute's value for each of the c aggregates per pixel. Similarly to MIP-mapping based techniques, which pre-filter the parameters of the shading function on a surface, our technique assumes separability of the terms of the shading equation in order to average these attributes separately [Bruneton and Neyret 2011; Heitz and Neyret 2012]. This requires that the inputs of the shading function can be factored into linearly combinable terms, as we will discuss in section 4.3.

We compute aggregate's values efficiently by rasterizing the scene with $n\times$ MSAA coverage, using early depth testing to ensure only visible samples generate fragment coverage in a pixel (thanks to the depth buffer generated during the prepass). The resulting fragment uses the *aggregates metadata* information to select the aggregate the current fragment contributes to, then blends its contribution into the frame-buffer element corresponding to the aggregate. Pseudocode for the G-buffer generation pass is given in Algorithm 2, and further details about it's implementation on modern GPUs is discussed in Section 4.2.

3.4.1 Pre-filtered attributes

G-buffer shading parameter are application-specific. We build statistical distribution information for each shading attribute, in order to account for the discrepancy during the shading (cf. next section). We chose to model this statistical information as the first (mean) and second (variance) moments of a Gaussian distribution. In practice, we only construct these distributions for the normal directions and the sub-pixel position of each aggregate, and we only retain the first moment (average) of the other attributes.

We handle normal distributions using either the Toksvig [2005] approximation (for isotropic normal distributions) or a LEAN mapping distribution [Olano and Baker 2010] for anisotropic normal distributions. Other distribution schemes such as cLEAN [Baker 2011] and LEADR [Dupuy et al. 2013] could also be used. While the standard usage of such normal distribution is to model micro-geometry, we use it to model both micro- and meso- scale geometric distributions, coming from texture details as well as real triangle-based geometry.

We don't explicitly store the roughness (or Blinn-Phong's specular exponent), but instead rely on the Toksvig representation to encode it directly, which also has the advantage of being linearly filterable.

3.5 Deferred shading

The deferred shading stage can be implemented using any screen-space deferred lighting technique (full-screen pixel shader, GPU compute shader, per-light bounding box rasterization, etc.). In contrast to traditional G-buffered shading, which performs shading once per pixel, or once per sample, our system shades once per aggregate.

Although AGAA performs n/c fewer surface shader evaluations than a system using supersampled shading, these evaluations are more costly. (see Section 5.2). After shading, the result shaded color for each aggregate is weighted by its relative sample count, and then all shading results are composited together and over the background image.

3.5.1 Aggregate shading

Shading an aggregate is very similar to shading a MIP-mapped and bilinearly-filtered sample from a single surface and material [Olano and Baker 2010; Han et al. 2007; Olano and North 1997; Fournier 1992b; Fournier 1992a]. Similar to this use case, and in contrast with volumetric pre-filtering [Heitz and Neyret 2012; Crassin et al. 2009], there is no need for filtering visibility since we rely on the geometry pre-pass to aggregate visible attributes.

The algorithm is independent of the shading model. For our experiments, we use the Blinn-Phong shading model, and we compute the pre-filtered shading for the Lambertian and Specular components separately. Other analytic BRDF models (based on Beckmann or GGX microfacet distributions for instance) could also be used, as long as their parameters can be linearly pre-filtered [Bruneton and Neyret 2011; Olano and Baker 2010].

There are three important differences with surface-based filtering in our case. First, filtering the specular reflectance is not enough, since variations in the sub-aggregate surfaces orientations can also lead to important differences in the diffuse shading as shown in [Dupuy et al. 2013]. We follow [Baker and Hill 2012] analytic approximation for filtering the diffuse component from the Toksvig normal distribution. Second, because the support geometry for these attributes can spread large depth extents per-pixel, the shadowing term must also be filtered. This problem will be discussed in the next section. Finally, very large depth discrepancy within an aggregate can induce potentially important light direction discrepancy in case of nearby light sources. This case can be accounted for by re-injecting, for each light source, the variance of light directions as additional variance in the normal distribution. However in practice, this effect appear very limited, thanks to our clustering scheme which tends to avoid such elongated aggregates (cf. section 3.3).

3.5.2 Shadowing

Among local shading terms, shadowing also needs to be filtered in order to account for differences of light visibility within a given aggregate. This is done independent of the initial number of visibility samples included within each aggregate. The idea is to sample the visibility within the shadow-map inside the shape of the aggregate, which we statistically defined by the mean and variance of the depth value. In practice, we reconstruct the world-space 3D position and variance vector, and project them inside the shadowmap to sample within this footprint using a fixed number of samples (usually 3-4 taps, or using hardware anisotropic filtering). Even though it is generally not necessary in practice, a more precise filtering can be obtained by reconstructing the anisotropic ellipsoid shape of the aggregate from the 3D world-space position of each sample. This can be done by computing the 3D covariance matrix representing the statistical distribution of positions within the aggregate.

Instead of numerically sampling the shadowing term, shadow-map pre-filtering techniques [Donnelly and Lauritzen 2006] could also be used in order to de-correlate even more the cost of shadowing from the extent of the aggregate that we are shading. We haven't explored this direction, but this is definitely an interesting future work.

In case of strong correlation between shadowing and other parameters, it is also possible to evaluate the shadowing per-sample, during the G-buffer generation pass, and pre-multiply the per-sample albedo and specular coefficients by the shadowing term before aggregating them. However, such an approach makes the shadowing cost scaling with the number of samples, which is not desirable, especially when many lights need to be evaluated.

3.6 Handling transparency

Because it supports high sampling rate visibility, our technique is compatible with stochastic rasterization techniques [Akenine-Möller et al. 2007] and hardware alpha-to-coverage conversion [Kirkland et al. 1999]. Our implementation rasterizes fine-detail geometry modelled using alpha-textured polygons (e.g., leaves), as well as translucent primitives, using alpha-to-coverage. Because the visibility is determined during the geometric prepass (stage 1), the alpha texture sampling and coverage generation only need to be performed during this pass.

4 GPU Implementation and Optimizations

In this section, we discuss some important details for the efficient implementation of this algorithm on the GPU.

4.1 Accelerating aggregate assignment

The clustering algorithm in Section 3.3 assumes that surface depth and normal information is stored per sample as a result of rasterization in pass 1. However, many modern GPUs implement depth-buffer compression mechanisms which store plane equations and coverage masks for visible triangles within a screen tile, as opposed to explicit depth samples (See Hasselgren and Akenine-Möller [2006] for a good description of a modern depth compression implementations). When the depth buffer is stored in a form that directly represents the $T < c$ triangles in a tile, aggregate assignment can be accelerated by operating directly on the stored triangles, instead of individual samples. That is, when geometric complexity in a screen region is low, the cost of constructing aggregates for this region can be reduced.

4.2 Target-independent rasterization into the aggregate G-buffer

The aggregate G-buffer generation algorithm, described in Algorithm 2, rasterizes the scene using n MSAA samples, while the filtered attributes associated to the c per-pixel aggregates are stored in a set of color render targets with c MSAA samples. We rely on *target independent rasterization*, a feature available through the `NV_framebuffer_mixed_samples` OpenGL extension [NVIDIA 2014] to enable the GPU to rasterize and perform depth-testing at higher sampling rate than the destination color targets. For each fragment shader execution, only one of the c target aggregates is selected using the aggregate selection scheme (Algorithm 2), in order to accumulate the fragment's attributes. This is achieved by routing the shader output to a given aggregate by modifying its coverage mask (see the `NV_sample_mask_override_coverage` OpenGL extension [NVIDIA 2014]).

4.2.1 Visibility-based scaling of attributes

To correctly account for visibility we must accumulate the fragment shader's attributes after scaling them by the number of visible samples. We do so by first configuring the graphics pipeline to perform an early depth-test, exploiting the depth buffer generated during the prepass. Second, we set up the input coverage mask provided to the fragment shader (which is used for scaling) to only contain the samples passing the depth test (see the `EXT_post_depth_coverage` extension [NVIDIA 2014]).

Algorithm 2 G-buffer generation algorithm

1. Set rendering states:
   (a) Disable depth writes and set depth test to EQUALS
   (b) Enable early depth-test and post-depth coverage
   (c) Enable stencil test to keep only the first sample passing depth test
   (d) Enable additive blending on G-buffer storage buffers
2. Render scene. For each fragment, find its aggregate and visible samples:
   (a) Read M_f, coverage mask of fragment's visible samples
   (b) Read D_a, aggregates meta-data for the pixel
   (c) Find $AggregateID$ by :
      i. finding $S_{id} = firstNonZeroBit(M_f)$
      ii. $AggregateID = (D_a \gg (S_{id} * MAX\_BITS\_AGGREGATE\_ID)) \;\&\; (MAX\_NUM\_AGGREGATES - 1)$
3. Compute G-buffer terms identical to traditional deferred rendering. Optionally compute LEAN mapping terms for normals.
4. Weight G-buffer terms by fractional coverage from M_f.
5. Route G-buffer results to the $AggregateID$ sample in output color buffers.

4.2.2 Enforcing one primitive value per sample

Because aggregate values will need to be re-normalized by the number of samples in their effective coverage mask before shading, it is important to ensure that no more than one primitive contribute to the same sample. Even with the depth-test of the generation pass set to EQUAL, such a situation can happen in case of Z-fighting, when more than one fragment's depth value pass the depth test for a given sample (because they are the same). This would produce noise artefacts and it can be avoided in a consistent way by using the stencil test to only keep the first sample value passing the depth test.

4.3 Aggregate G-buffer memory layout

Current practice in the video games industry tends to use a Blinn-Phong shading model for deferred shading, storing at least an RGB *albedo* value for the diffuse term, one for the specular term (*metal*), a *roughness* coefficient (reciprocal of the Phong's glossy exponent), a scalar emissive coefficients and a normal encoded as 2D spherical coordinates. These G-buffer layouts range from 12 bytes to 41 bytes per sample (including depth) [Filion and McNaughton 2014; Tatarchuk et al. 2013; Mittring 2012; Andersson 2011; Coffin 2011; Kasyan et al. 2011; Filion and McNaughton 2008; Valient 2007], with $\approx$ 20 bytes apparently the most common on PC. For the sake of our feasibility demonstration, we chose to encode attributes corresponding to the 16 bytes layouts presented in figure 5-top, which we consider as representative of a real game engine scenario (within the lower bound of the memory requirements).

For AGAA, we rely on the same set of parameters, which we need to represent as filtered attributes for each aggregate (cf. section 3.5). We use the aggregate G-buffer layout presented in figure 5-bottom. It encodes the normal distribution using Toksvig's normal vector as RGB16 (normalized, fixed point), the material albedo as RGB10, the specular coefficient in Y'CbCr color space, using 16b Y' and 8b Cb and Cr, and the emissive coefficient as R8.

We chose to use fixed-point color formats for increased precision. In order for the additive blending accumulation to work, all accumulated values (generated per-fragment) must be pre-normalized in the fragment shader by the total number of samples per-pixel. One could also use floating point color formats.

Unlike traditional G-buffers, ours do not store explicit roughness (i.e., the BRDF's glossy exponent term) directly but instead inject it as additional variance inside the normal distribution. We save G-buffer memory by not explicitly storing the distribution of positions. This is instead computed per aggregate from the multi-sampled depth buffer during the deferred shading pass.

Because there can be mismatches between per-sample clustering (maintained by the *aggregates metadata*) and the fragment values actually accumulated, a sample counter must also be maintained to allow the re-normalization of the attributes. We keep this information as a per-aggregate coverage mask, which is also used for reconstructing the distribution of positions.

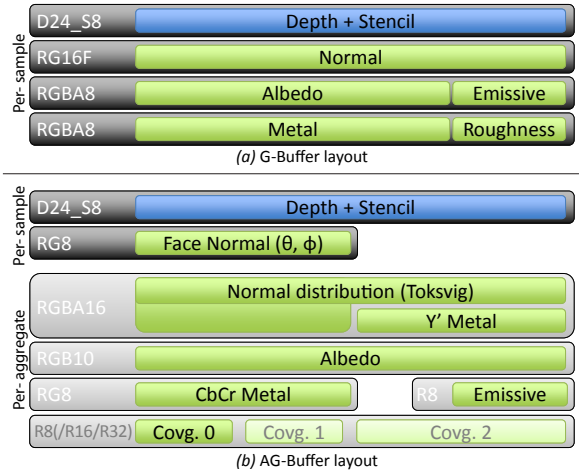

Figure 5: *G-buffer layouts we use for classical deferred MSAA (top, 16 bytes per sample) and AGAA (bottom, 16 bytes per aggregate + 6 bytes per sample). Actual number of bits used for coverage information depends on MSAA rate used for rasterization.*

5 Evaluation

We evaluate the performance and quality of AGAA on five scenes (Figure 9), chosen to challenge both our algorithm and prior work. *Old City* is a game-like scene that has intricate railings, furniture, and complex foliage. *Foliage* is a scene from a Epic Unreal Engine 3 demo. The foliage in these two scenes is composed of geometric and translucent alpha-mapped parts (cf. section 3.6). *Furball* (Figure 7-middle) exhibits fine-scale geometry far beyond that used in video games today. Lastly, the *metal* scene (Figure 7-bottom) contains many thin metal tubes (highly tessellated geometry). It presents the challenge of building suitable geometric aggregates for highly-curved specular surfaces.

All results have been produced at the resolution of 1920x736 on an NVIDIA GTX 980 graphics processor (Maxwell GM204) and using an OpenGL implementation of the algorithm described in Section 3. We compare our technique to the simple/complex optimization of Lauritzen et al. [2010] for deferred shading, which we configured to ensure no quality degradation compared to brute-force per-sample shading. For the optimal implementation of this technique, we rely on the ability to access the compressed representation of the depth buffer, which is not currently exposed by OpenGL. Our implementation emulates this ability, and therefore

does not account for the cost of this emulation in the results. As discussed in Section 4.1, if this feature were supported natively, using the depth plane information would not add any additional runtime cost.

5.1 Image Quality

5.1.1 Video game and artificial scene

Figure 6 compares the rendered output of AGAA against that of two alternatives: super-sampled shading of each visibility sample (which we consider a high-quality baseline) and the surface-based anti-aliasing method of Salvi et al. [2012] configured to use its highest quality "merge" clustering predicate (SBAA). We plot the per-pixel differences between AG-buffer and SBAA renderings (magnified by a factor of 2) against that of the baseline per-sample shading. Figure 7 provide a similar analysis on more artificial, but higher complexity scenes and higher sampling rates, with the number of shading computations per pixel (number of Aggregates/Surfaces) varying from 1 to 4.

Generally AGAA provides higher image quality than SBAA when using the same number of shading events per pixel, and it is not unusual that even with 4 surfaces per pixel, SBAA quality is lower than AGAA with 2 surfaces per pixel. As expected, the image quality of both approximations increases with the number of shaded aggregates (surface clusters in the case of SBAA), but we find that the AGAA results more closely match those of the baseline.

Our experiments indicate that when rendering intricate geometry such as foliage, hair, or the detailed furniture forms in Old City, two aggregates per pixel is sufficient to produce visually pleasing results. We also found that even though the AGAA results may not match that of the baseline implementation, the results generally exhibit more temporal stability than the SBAA results. We invite the reader to inspect the accompanying video to further assess AGAA temporal stability.

5.1.2 Highly specular surfaces

The benefits of aggregating statistics from all elements contributing to a pixel, as opposed to a select few, is particularly apparent when rendering specular surfaces (Figure 7-bottom). By modeling the distribution of normals featured on the scene's thin, high-curvature metal rods, shading using the AG-buffer is able to approximate the specular highlight well. The SBAA output exhibits severe aliasing, even when shading is evaluated four times per pixel. Note that for this scene, we used the anisotropic LEAN normal distribution instead of Toksvig.

5.1.3 High sampling rates

Although the aggregate G-buffer enables the renderer to evaluate shading more sparsely while still preserving image quality, it does not eliminate the need for dense sampling of scene visibility. Figure 8 compares the quality of AGAA shading to the baseline as the visibility sampling rate is increased from 4 to 32 samples per pixel. Although the AGAA shading rate stays constant (3 aggregates/pixel), the output quality of the AGAA images improves with the visibility sampling rate, because dense sampling results in more small primitives captured and contributing to each pixel.

5.2 Execution performance

Table 1 shows execution performance numbers of AGAA ($8\times$ MSAA, $C = 2$) for various scenes we have tested (Fig. 9), as well as speed-ups relative to Simple/Complex [Lauritzen 2010].

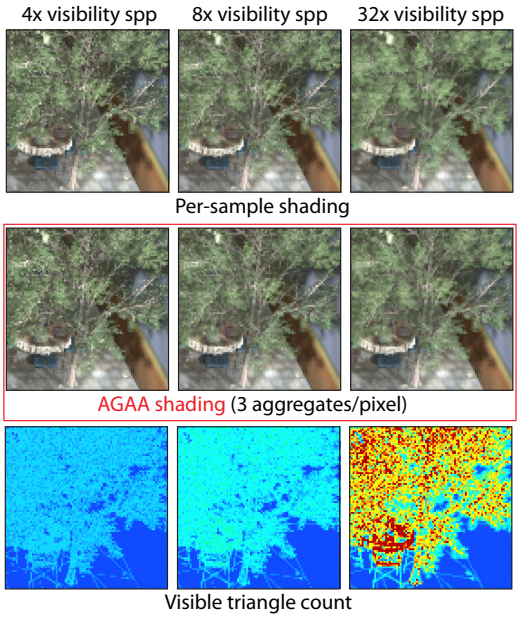

Figure 8: *As the visibility sampling rate of rendering is increased, more triangles contribute coverage to each pixel (bottom row). Although these triangles are not individually shaded in the aggregate G-buffer renderer, these triangles contribute to the computation of more accurate and more temporally stable aggregates. Therefore, both traditional super-sampled rendering (top) and aggregate G-buffer rendering (middle) benefit from high-rate visibility sampling.*

We limit our performance experiments to $8\times$ MSAA, the highest MSAA rate natively supported by GPU hardware. All scenes feature one main shadowed light as well as 16 secondary point light sources, which we consider as a realistic number for representing the workload exercised by a modern game engine. The only exception is the Furball scene which uses only one non-shadowed light source.

AGAA is constantly faster than Simple/Complex, despite the cost of the additional Z-prepass that we need to perform. This cost is mostly compensated by a faster geometric generation pass. This pass benefits from both the early depth culling, and the bandwidth reduction to the video memory made possible by the reduction of per-pixel data in the aggregate G-buffer.

Because it performs at most two (even though more costly) shading events per pixel, most of the speedup of our technique comes from the shading pass, which for most scenes is at least $2\times$ faster than the Simple/Complex per-sample shading. This execution time of the shading pass is impacted by the computation of the depth distribution information used by the pre-filtered shading. This information could also be aggregated on the fly like other attributes, at the cost of a slightly higher memory consumption.

5.3 Memory consumption

We analyzed the memory requirement of AGAA relative to a standard G-buffer implementation and to SBAA. Results are shown in figure 10. For AGAA, we used the 16B/Aggregate + 6B/sample AG-Buffer layout described in section 4.3, with the corresponding 16B/sample layout for classical G-buffer. We believe that this is somehow representative of what a modern game engine would use. In addition to the AG-Buffer layout, AGAA requires $n \times log_2(c)$ (n MSAA rate, c number of aggregates) additional bits per pixel as

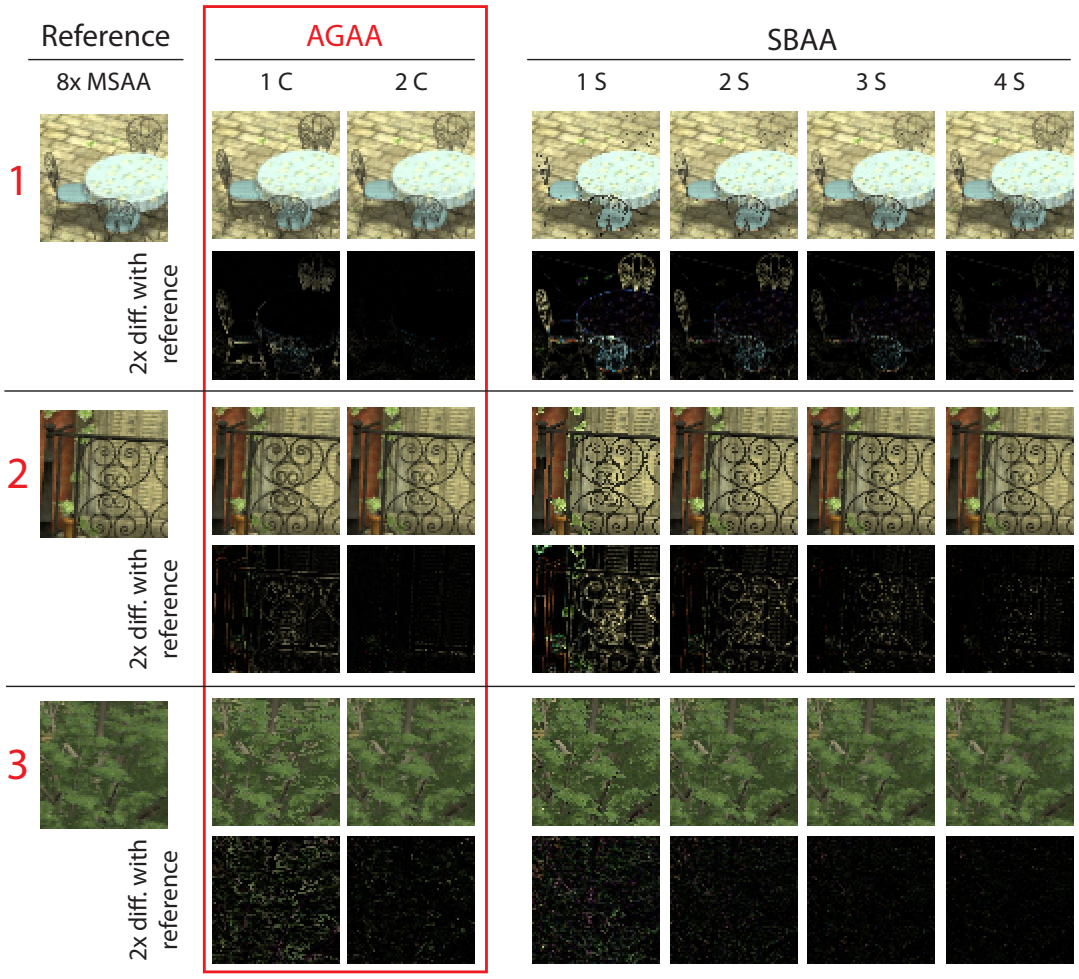

Figure 6: *Image quality comparison between Aggregate G-buffer Anti-Aliasing (AGAA) and Surface-Based Anti-Aliasing (SBAA) [Salvi and Vidimče 2012] for 1 to 4 surfaces per pixel. Each zoomed picture correspond to one of the crops in Figure1. Note that AGAA with 2 aggregates exceeds the quality of SBAA with 4 surfaces.*

metadata for clustering (cf. section 4.1). SBAA requires a 1B additional primitive ID per sample, plus 2B of surface data per surface.

Globally, with $C = 2$, the benefit of AGAA in terms of memory is a little under 40% at $8\times$ MSAA, and almost 50% at $16\times$ MSAA. AGAA requires also $\sim$20% less memory than SBAA for two aggregates and two surfaces. In addition, in the relatively complex scenes we analysed, AGAA with $C = 2$ achieves nearly the same image quality (cf. section 6) as SBAA with $S = 4$, which corresponds to a $\sim$37% memory reduction.

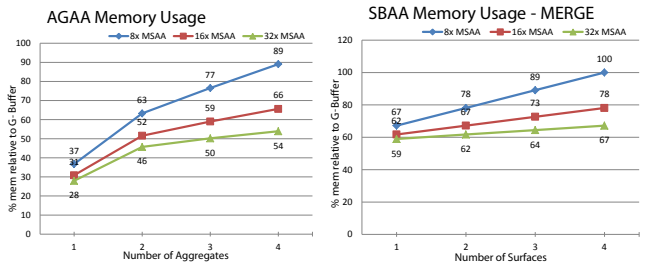

Figure 10: *Memory consumption of AGAA (top) and SBAA in MERGE mode (bottom) in percent relative to a full multisampled G-buffer, depending on the number of aggregates/surfaces and for $8\times$/$16\times$/$32\times$ multisampled rendering. This includes all required per-sample and per-aggregate storages. In the case of $8\times$ MSAA and two aggregates, AGAA saves 33% of memory relative to a classical G-buffer. AGAA requires consistently less memory than SBAA for the same number of shaded aggregate surfaces.*

6 Limitations

Figure 11 shows three main failure cases of our technique. Similar to other pre-filtering techniques ([Bruneton and Neyret 2011]), our algorithm doesn't produce accurate results in the presence of important correlation between independently filtered parameters. Figure 11 *(a)* is a typical manifestation of this issue when using only one aggregate per-pixel. Halos are visible around the leaves of the tree because samples from the red wall, which is mostly in shadow, are filtered together with samples from the leaves of the tree, which are mostly lit. This improperly induces the shading of a yellowish average material which is semi-shadowed. Note that this issue would not arise here without shadowing.

From our experience, this kind of correlation effect is rarely visible when using at least two aggregates per pixel, except in very specific configurations in presence of structured geometry which

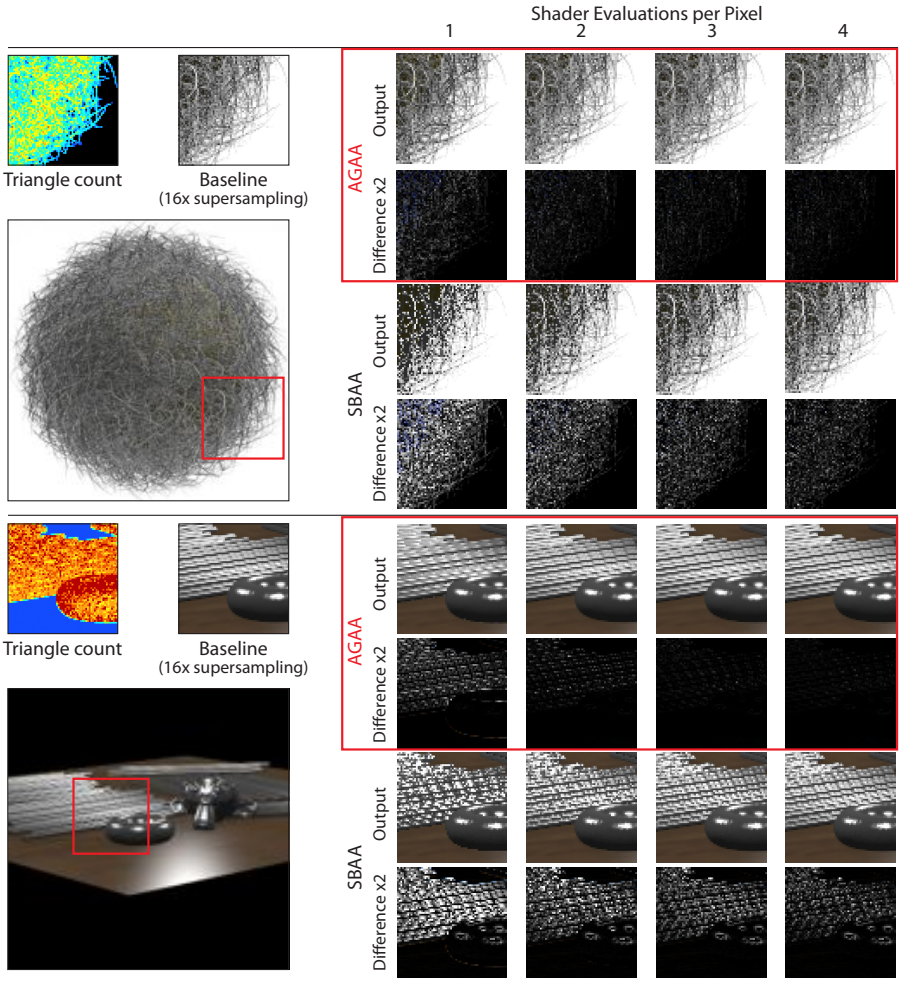

Figure 7: *Shading only a few aggregates stored in an aggregate G-buffer often closely approximate the results of super-sampled shading (shown here compared to 16× super-sampled shading). Image quality is noticeably better than that of Surface-Based Anti-Aliasing (SBAA). The improvement over SBAA is even more pronounced under motion.*

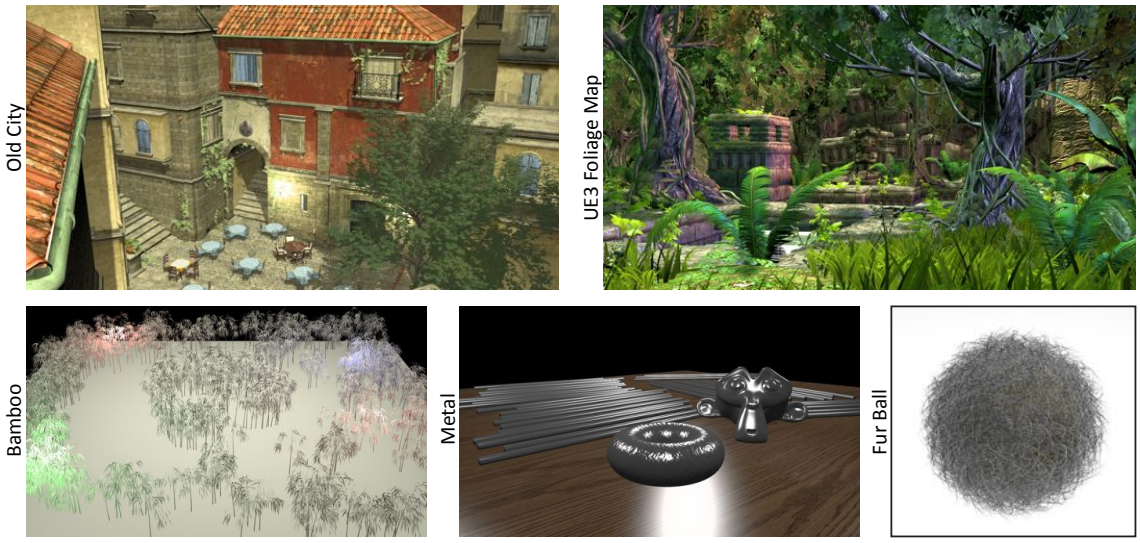

Figure 9: *The five scenes used to test our system, rendered at* 1920×736 *: Old City, UE3 Foliage Map (Courtesy Epic Games), Bamboo, Metal and Fur Ball.*

Time (ms)	AGAA					Reference				Speedups	
Scene	Z-Prepass	Aggregate Def.	Gen.	Shading	**Total**	Simple/ Complex	Gen.	Shading	**Total**	**Shading**	**Frame**
Old City	1.43	0.74	2.34	2.3	**6.81**	0.33	3.07	4.6	**8**	2×	**1.17×**
UE3 FoliageMap	1.76	0.5	1.79	2.45	**6.5**	0.38	3.18	5.12	**8.68**	2.09×	**1.34×**
Bamboo	3.75	0.99	4.28	4.28	**13.3**	0.34	5.01	9.25	**14.6**	2.16×	**1.10×**
Metal	1.43	0.54	1.31	0.93	**4.21**	0.31	2.45	1.61	**4.37**	1.73×	**1.04×**
Fur Ball	1.5	0.58	1.65	0.14	**3.87**	0.34	3.42	0.31	**4.07**	2.21×	**1.05×**

Table 1: *Runtime performance (in ms) for the main steps of AGAA ($C = 2$) at $8\times$ MSAA for various scenes, compared to the Simple/Complex deferred-shading technique used as a reference. The most right columns show the speed-ups provided by AGAA on the shading pass only and on the entire frame.*

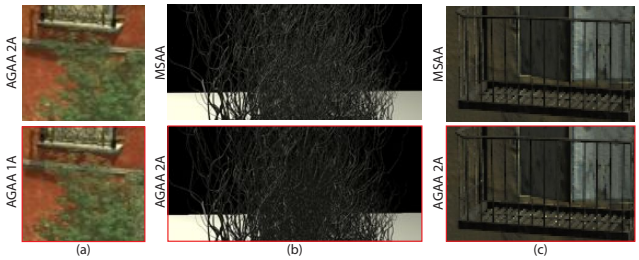

Figure 11: *Illustration of three failure cases for our filtering technique.*

exhibit high degrees of correlation, like the grid in Figure 11 *(c)* which shows some high-intensity noise.

In some situations, our technique also fails to accurately model the sub-pixel geometry inside the aggregates. This is especially true in case of long and thin geometry, like the thin hairs in Figure 11 *(b)*, which at some scales produces highly anisotropic normal distributions that our simple anisotropic NDF model (Toksvig) can not model. This tends to decrease the shaded intensity for such structures. In order to solve this problem, an anisotropic normal distribution defined on the entire sphere would be needed. This is a problem that remains open for future works.

7 Conclusion

This paper introduces aggregate G-buffer anti-aliasing (AGAA), a technique to improve anti-aliasing of fine geometric details in deferred renderers. It is based on a new mechanism to decouple light shading rate from the geometric sampling rate.

The primary contribution is a fully dynamic screen-space algorithm that efficiently aggregates material properties across disjoint surfaces. We demonstrate that storing and shading only 2 to 3 aggregates per pixel is sufficient for a wide range of scenes, irrespective of the number of visibility samples per pixel (i.e., the MSAA rate). Our technique approaches the quality of super-sampled shading at a substantially lower memory and compute cost, especially for effects such as specular highlights, by pre-filtering shader inputs during aggregate generation.

The benefits of our technique, both in terms of memory saving and shading time, improve greatly with the increase of the geometric sampling rate. This is limited to $8\times$ MSAA on current GPUs. Looking forward, our technique makes much higher MSAA rates affordable, motivating GPU hardware support for coverage estimation and depth testing above 8 samples per pixel.

In order to improve quality and reduce shading rate even more, future work will design new normal distribution functions and associated shading models adapted to our representation, which would provide a more precise description of filtered surfaces inside aggregates.

Acknowledgements

We thank David Luebke for the helpful discussions, Nir Benty for his help with the video results and editing, as well as the anonymous reviewers for their suggestions.

References

AKELEY, K. 1993. Reality engine graphics. In *Proceedings of SIGGRAPH '93*, ACM, 109–116.

AKENINE-MÖLLER, T., MUNKBERG, J., AND HASSELGREN, J. 2007. Stochastic rasterization using time-continuous triangles. In *Graphics Hardware*, 7–16.

ANDERSSON, J., 2011. Shiny PC graphics in Battlefield 3. GeForce LAN. `http://www.slideshare.net/fullscreen/DICEStudio/shiny-pc-graphics-in-battlefield-3/`.

BAKER, D., AND HILL, S. 2012. Rock-solid shading. In *Advances in Real-Time Rendering in 3D Graphics and Games. SIGGRAPH Course*.

BAKER, D. 2011. Lean and clean specular highlights. In *Game Developer Conference*.

BAVOIL, L., CALLAHAN, S. P., LEFOHN, A., COMBA, J. A. L. D., AND SILVA, C. T. 2007. Multi-fragment effects on the gpu using the K-buffer. In *Proceedings of I3D 2007*, ACM, 97–104.

BRUNETON, E., AND NEYRET, F. 2011. A survey of non-linear pre-filtering methods for efficient and accurate surface shading. *IEEE TVCG*.

CHAJDAS, M. G., MCGUIRE, M., AND LUEBKE, D. 2011. Subpixel reconstruction antialiasing for deferred shading. In *Proceedings of I3D 2011*, ACM, 15–22.

CHRISTENSEN, P. H., AND BATALI, D. 2004. An irradiance atlas for global illumination in complex production scenes. In *Proceedings of EGSR'04*, Eurographics Association, 133–141.

CLARBERG, P., TOTH, R., AND MUNKBERG, J. 2013. A sort-based deferred shading architecture for decoupled sampling. *ACM Trans. Graph. 32*, 4 (July), 141:1–141:10.

COFFIN, C., 2011. SPU-based deferred shading for Battlefield 3 on Playstation 3. Game Dev. Conf.

COHEN, J., OLANO, M., AND MANOCHA, D. 1998. Appearance-preserving simplification. In *Proceedings of ACM SIGGRAPH '98*, ACM, 115–122.

COOK, R. L., HALSTEAD, J., PLANCK, M., AND RYU, D. 2007. Stochastic simplification of aggregate detail. In *Proceedings of ACM SIGGRAPH '07*, ACM.

CRASSIN, C., NEYRET, F., LEFEBVRE, S., AND EISEMANN, E. 2009. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of I3D '09*, ACM, 15–22.

DÉCORET, X., DURAND, F., SILLION, F. X., AND DORSEY, J. 2003. Billboard clouds for extreme model simplification. *ACM Trans. Graph. 22*, 3 (July), 689–696.

DONNELLY, W., AND LAURITZEN, A. 2006. Variance shadow maps. In *Proceedings of I3D 2006*, ACM, 161–165.

DUPUY, J., HEITZ, E., IEHL, J.-C., POULIN, P., NEYRET, F., AND OSTROMOUKHOV, V. 2013. Linear efficient antialiased displacement and reflectance mapping. *ACM Transactions on Graphics 32*, 6 (Nov.), Article No. 211.

FILION, D., AND MCNAUGHTON, R., 2008. Chapter 5: Starcraft ii effects and techniques. SIGGRAPH 2008 Advances in Real-Time Rendering in 3D Graphics and Games Course.

FILION, D., AND MCNAUGHTON, R., 2014. How inFAMOUS: Second son used the PS4. Dual Shockers online article. http://www.dualshockers.com/2014/04/02/how...

FOURNIER, A. 1992. Filtering normal maps and creating multiple surfaces. In *Technical report*, University of British Columbia.

FOURNIER, A. 1992. Normal distribution functions and multiple surfaces. In *Graphics Interface*, 45–52.

HAN, C., SUN, B., RAMAMOORTHI, R., AND GRINSPUN, E. 2007. Frequency domain normal map filtering. *SIGGRAPH 26*, 3, 28:1–28:12.

HASSELGREN, J., AND AKENINE-MÖLLER, T. 2006. Efficient depth buffer compression. In *Graphics Hardware*, ACM, GH '06, 103–110.

HEITZ, E., AND NEYRET, F. 2012. Representing Appearance and Pre-filtering Subpixel Data in Sparse Voxel Octrees. In *HPG '12*, ACM/Eurographics, 125–134. Best Paper HPG 2012.

HERZOG, R., EISEMANN, E., MYSZKOWSKI, K., AND SEIDEL, H.-P. 2010. Spatio-temporal upsampling on the gpu. In *Proceedings of I3D 2010*, ACM, 91–98.

JOUPPI, N. P., AND CHANG, C.-F. 1999. Z3: An economical hardware technique for high-quality antialiasing and transparency. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, ACM, HWWS '99, 85–93.

KASYAN, N., SCHULZ, N., AND SOUSA, T., 2011. Secrets of CryENGINE 3 graphics technology. SIGGRAPH 2011 Advances in Real-Time Rendering in 3D Graphics and Games Course.

KERZNER, E., AND SALVI, M. 2014. Streaming g-buffer compression for multi-sample anti-aliasing. In *HPG2014*, Eurographics Association.

KIRKLAND, D., ARMSTRONG, B., GOLD, M., LEECH, J., AND WOMACK, P., 1999. ARB_Multisample OpenGL extension specification. `http://www.opengl.org/registry/specs/ARB/multisample.txt`.

LACEWELL, J. D., EDWARDS, D., SHIRLEY, P., AND THOMPSON, W. B. 2006. Stochastic billboard clouds for interactive foliage rendering. *J. Graphics Tools 11*, 1, 1–12.

LAURITZEN, A., 2010. Deferred rendering for current and future rendering pipelines. SIGGRAPH Course. Beyond Programmable Shading.

LIKTOR, G., AND DACHSBACHER, C. 2012. Decoupled deferred shading for hardware rasterization. In *Proceedings of I3D '12*, ACM, 143–150.

LOTTES, T., 2009. Fast Approximate Anti-Aliasing (FXAA). `http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf`.

LUEBKE, D., WATSON, B., COHEN, J. D., REDDY, M., AND VARSHNEY, A. 2002. *Level of Detail for 3D Graphics*. Elsevier Science Inc.

MITTRING, M., 2012. The-technology-behind-the-elemental-demo. SIGGRAPH 2012 Advances in Real-Time Rendering in 3D Graphics and Games Course.

NVIDIA. 2014. *TXAA Technology Documentation*. Available at `http://www.geforce.com/hardware/technology/txaa/technology`.

NVIDIA, 2014. NVIDIA OpenGL Extensions Specifications. `https://developer.nvidia.com/nvidia-opengl-specs`.

OLANO, M., AND BAKER, D. 2010. Lean mapping. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, I3D '10, 181–188.

OLANO, M., AND NORTH, M. 1997. Normal distribution mapping. Tech. rep.

RAGAN-KELLEY, J., LEHTINEN, J., CHEN, J., DOGGETT, M., AND DURAND, F. 2011. Decoupled sampling for graphics pipelines. *ACM Trans. Graph. 30*, 3 (May), 17:1–17:17.

RESHETOV, A. 2009. Morphological antialiasing. In *Proceedings of HPG '09*, ACM, 109–116.

RESHETOV, A. 2012. Reducing aliasing artifacts through resampling. In *Proceedings of HPG'12*, Eurographics Association, 77–86.

SALVI, M., AND VIDIMČE, K. 2012. Surface based anti-aliasing. In *I3D'12*, ACM, 159–164.

TATARCHUK, N., TCHOU, C., AND VENZON, J., 2013. Destiny: From mythic science fiction to rendering in real-time. SIGGRAPH 2013 Advances in Real-Time Rendering in 3D Graphics and Games Course.

TOKSVIG, M. 2005. Mipmapping normal maps. *Journal of GraphicsTools 10*, 3, 65–71.

VALIENT, M., 2007. Deferred rendering in Killzone 2. Game Developers Conference.

YOON, S.-E., LAUTERBACH, C., AND MANOCHA, D. 2006. R-lods: Fast lod-based ray tracing of massive models. *Vis. Comput. 22*, 9 (Sept.), 772–784.