Aggregate G-Buffer Anti-Aliasing

-Extended Version-

Cyril Crassin
NVIDIA

Morgan McGuire
NVIDIA / Williams College

Kayvon Fatahalian
Carnegie Mellon University

Aaron Lefohn
NVIDIA

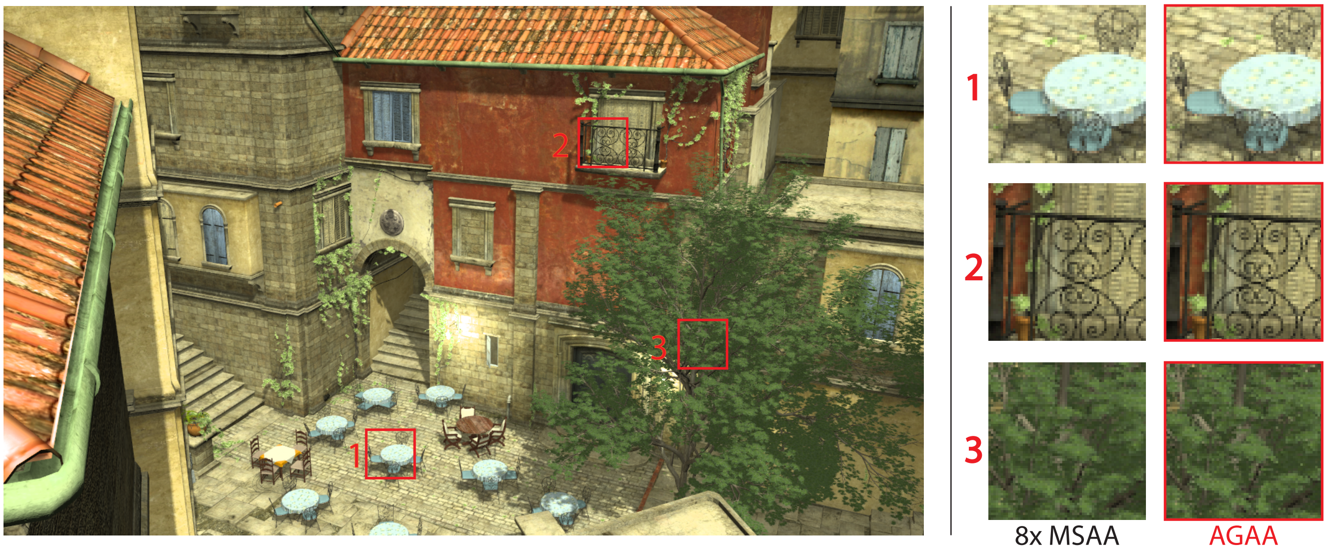


Figure 1: *Large image rendered with Aggregate G-Buffer Anti-Aliasing (AGAA). The AGAA results (red outlines) shaded only twice per pixel give comparable results to the MSAA reference shaded eight times per pixel, and use 37% less memory. AGAA reduces aliasing by prefiltering the scene's sub-pixel geometric detail (foliage, thin railings, etc.) into a small set of statistical aggregates per-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. Our goal is to make high per-pixel sampling rates practical for real-time applications by substantially reducing shading costs and per-pixel storage compared to traditional deferred shading. AGAA uses the rasterization pipeline to generate a compact, pre-filtered geometric representation inside each pixel. We shade this representation 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 aggregates per-pixel generates results comparable to 32× MSAA, but requires 54% less memory and is up to 2.6× faster (−30% memory and 1.7× faster for 8× MSAA).

Keywords: anti-aliasing, graphics pipelines, pre-filtering, shading

IEEE Transactions on Visualization and Computer Graphics (TVCG) - October 2016 - Authors Version

1 Introduction

Developing efficient strategies for high quality antialiasing of massive amounts of sub-pixel geometrical details is one of the most important challenge for reaching cinematic image quality in real-time. Offline high-quality renderers enjoy excellent anti-aliasing quality by sampling geometrically complex environments, such as those containing foliage, fur, or intricate geometry (e.g., the tables and furniture details in figure 1), at very high rates in order to properly capture all sub-pixel details. These environments are challenging for any rendering system, but are particularly difficult for real-time systems. Real-time applications traditionally rely on relatively simple geometry, with surface details encoded inside texture maps, which can be pre-filtered as a MIP-map hierarchy. However, there is a global trend in favor of using increasingly complex geometries for improving overall quality, displaying more detailed scenes, as well as simplifying authoring processes.

This poses multiple difficulties. 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. In addition, shaders evaluating lighting generally access the G-buffer many times per frame, incurring high bandwidth cost.

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 [NVI 2014]. Such techniques drastically under-sample geometry and shading when rendering complex geometry. As a result, they 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 computed, 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 texture-based and voxel-based pre-filtering techniques (cf. section 2). We create a new GPU-based 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 (AG-buffer)*. It compactly encodes the distribution of depths, normals, and other material and geometric attributes 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.

Generally, we see this technique as a path forward for driving-up geometric sampling rate in real-time applications.
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.

Throughout the paper, we use the following standard terms: A *primitive* is a planar polygon input to rasterization, typically a triangle but may be a line, quad, or point sprite. A *fragment* is the portion of a primitive that lies within a pixel. A *sample* is a location (or the values stored at it) within a pixel, which may be *covered* by a fragment. A geometry buffer (*G-buffer*) is a set of textures into which the shading input (e.g., shading normal, BSDF coefficients) is written during a *G-buffer generation pass*. Subsequent *deferred shading passes* combine lighting with the G-buffer to produce an image.

2 Related Work

Decoupling shading rate from visibility sampling rates is a key idea to reduce shading costs in real time rendering. It is used in the context of both forward [Akeley 1993; Fatahalian et al. 2010] 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] or for adaptive shading of complex acquired materials [Bagher et al.]. 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 to ensure full scalability, reuse is applied across multiple (potentially disconnected) primitives.

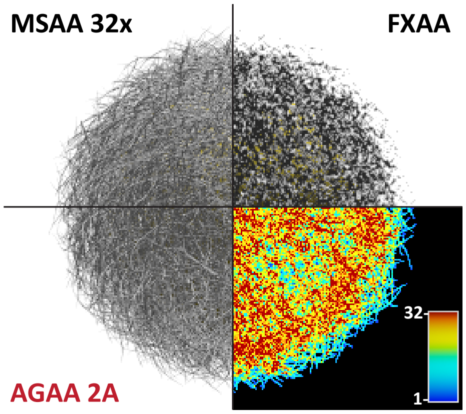


Figure 2: *Complex objects like the Fur Ball exhibit significant sub-pixel details (up to 32 triangles per pixel here) and cannot be anti-aliased using post-process screen-space anti-aliasing techniques like FXAA [Lottes 2009] (top-right). In contrast, our technique allows capturing these sub-pixel details,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

A simple way to reduce shading work in a deferred shading pipeline is to generate a full supersampled G-buffer, but to adaptively select only a subset of the samples to shade for each pixel. Lauritzen [2010] shades only once for *simple* pixels, which contain only a single surface, and for every samples in *complex* pixels, with multiple surfaces. In the same spirit, Hollander et al. [2013] shade a more adaptive number of samples, using simple metrics based on geometric sample attributes and luminance values. These schemes do not reduce memory requirements and speed-up rendering only when triangles are large, and the overall per-pixel complexity is low.

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]. Decoupling shading from visibility sampling has been explored in order to reduce shading cost in the context of stochastic rendering [Clarberg et al. 2013; Liktor and Dachsbacher 2012; Ragan-Kelley et al. 2011]. However, these techniques do not allow amortizing shading over multiple primitives, which limits their usability for antialiasing. They also rely on complex screen-space data-structures, which suffer from the lack of hardware acceleration, and prevent them from reaching realistic performance for real-time applications.

Our work improves on the approach of Salvi et al. [2012] (SBAA), which analyses the results of dense geometry sampling during rasterization to identify and retain exactly one fragment for each of 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 SBAA by designing a single-pass rendering algorithm (while SBAA requires a geometric depth pre-pass for visibility estimation) which also allows merging shading attributes belonging to similar non-intersecting planes, at the cost of a software evaluation of visibility using an interlocked fragment shader. In the context of forward shading, Quad-fragment

Merging [Fatahalian et al. 2010] performs a similar merging *on the fly*, prior to fragment shader execution and in a more opportunistic way. Additional triangles' adjacency information is used and only one set of attributes from the merged input is also retained for shading.

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 to maintain surface details [Cohen et al. ; Yoon et al. 2006; Cook et al. 2007]. Similarly, sprites and billboard clouds [Décoret et al. 2003; Lacewell et al. 2006] allow complex unconnected geometries to be simplified and encoded into a limited number of textured quads. These approaches are attractive in that they also reduce the cost of geometry processing during rendering, while our work, like other dynamic filtering approaches, requires additional geometry processing. However, they require heavy preprocessing and incur large storage cost for animated objects. In addition, due to the intrinsically flat (surface-based) nature of the simplified representation, they also fail to accurately preserve the appearance of the geometry in the general case, when the simplified details are truly 3D, coming from multiple surfaces (possibly disjoint), and scattered in space.

Our approach to modeling aggregate geometry in a pixel largely takes inspiration from previous works on appearance-preserving prefiltering of micro-scale surface details into 2D texture maps [Fournier 1992a; Fournier 1992b; Olano and North 1997; Toksvig 2005; Han et al. 2007; Olano and Baker 2010; Bruneton and Neyret 2011; Dupuy et al. 2013], as well as volumetric geometric prefiltering schemes [Decaudin and Neyret 2004; Christensen and Batali 2004; Crassin et al. 2009; Crassin et al. 2011; Heitz and Neyret]. Voxel-based prefiltering aims at unifying geometric and material descriptions into a unique multi-scale representation, which is pre-computed and can be queried at any resolution, and that scalably adapts to the sampling rate used for rendering. However, such representation currently suffers from several drawbacks which limits its usability, especially with dynamic scenes. In particular, it requires a costly pre-process to be built, and induces important storage costs.

As a preprocess, these approaches pre-filter and tabulate high-resolution surface details, inside a world-space data structure which needs to be re-sampled and traversed for visibility estimation at runtime. Instead, our work leverage rasterization to dynamically sample and aggregate per-pixel statistics from high-resolution surfaces, directly inside a screen-space structure, at the required resolution. While accounting for visibility and masking effects is generally a problem for appearance-preserving pre-filtering techniques, our approach dynamically resolves visibility on a per-sample granularity, prior to aggregation, which greatly improves the quality of the reconstruction.

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 4:

1. **Dense Visibility Sampling** (depth + compressed normals *prepass*): render geometry, storing depth and compressed normals using n visibility samples per pixel.

2. **Aggregate Definition**: 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] (which also requires a geometry pre-pass), nevertheless because the two algorithms make different assumptions, the actual algorithm implemented by each stage is quite different. This rendering pipeline supports both opaque and translucent geometry through alpha-to-coverage.

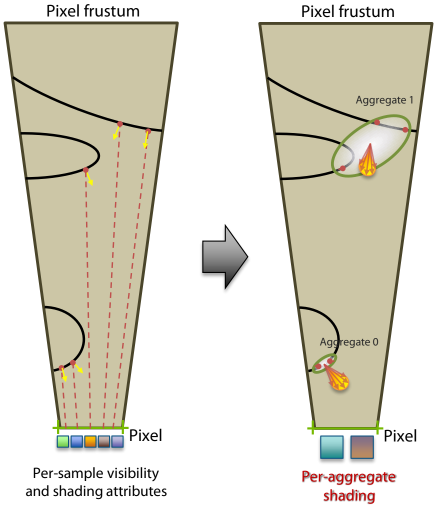


Figure 3: *AGAA samples sub-pixel geometry and shading attributes at high frequency, like what MSAA would do (left figure), but shades at a much lower frequency, on a few statistical geometric aggregates per-pixel (right figure), while preserving appearance.*

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 7) for each sample. We use a high multi-sampling rate (e.g., 8× 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

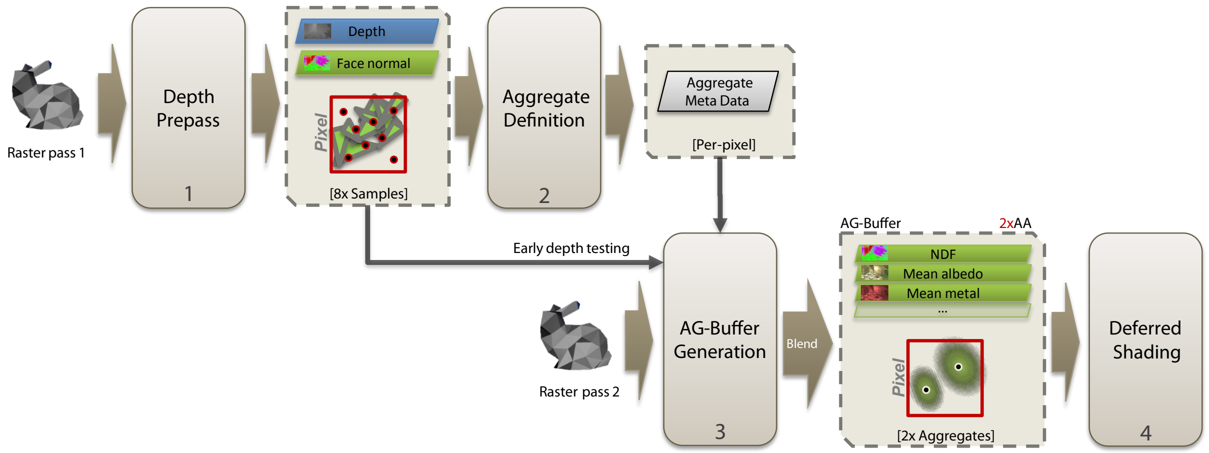


Figure 4: *Functional view of the 4 processing stages of the execution pipeline of the technique, together with the output of each stage and their storage frequency.*

requires to output primitive's depth values (generated by the hardware rasterizer) as well as the primitive's normal (from the plane, no normal map perturbation). In the case of transparent 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 multisampled 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 7). 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 Definition

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 definition pass is mapping of samples to clusters (that we call *aggregates metadata*). We encode this mapping using $d = n \times log_2(c)$ bits, with n the MSAA rate used for rasterizing the geometry. Thus, in a 4-cluster configuration, the mapping requires two-bytes per pixel when using 8x-MSAA (two bits per sample), as shown in Figure 5.

Note that because many scenes contain an emissive skybox that does not require shading, we exclude samples at the maximum depth value from aggregate definition. 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

The goal of this step is to segregate c-modal distributions of geometry contributing to a pixel into c aggregates. 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 values taken by the different attributes are *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. This is a standard hypothesis for all pre-filtering techniques, including texture MIP-mapping. An example of such a correlation is illustrated in section 6 (Figure 17-(a)). 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.

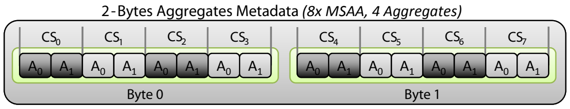

2-Bytes Aggregates Metadata (8x MSAA, 4 Aggregates)
CS_0
CS_1
CS_2
CS_3
CS_4
CS_5
CS_6
CS_7
A_0 A_1 A_0 A_1 A_0 A_1 A_0 A_1
A_0 A_1 A_0 A_1 A_0 A_1 A_0 A_1
Byte 0
Byte 1

Figure 5: *Memory layout description for the 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 distance d between surface samples a and b is given by:

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

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 definition 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]. 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 a second time at $n\times$ MSAA, using the depth buffer generated during the visibility prepass (1). The depth-test is set to EQUAL and early depth testing is enabled (evaluated *before* fragment shading) to ensure that only the samples visible in the prepass generate fragment coverage and shader evaluation in a pixel. 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. In order to maximize the performance of this step, and allow flexible storage formats, we rely on the hardware color blending to perform the additive accumulation of the attributes. 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 parameters 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.

Depending on the shading model (cf. Section 3.5), we handle normal distributions using either the Toksvig [2005] approximation (for isotropic normal distributions) or for anisotropic normal distributions, the LEAN mapping distribution [Olano and Baker 2010] or the SGGX (Symmetric GGX) distribution [Heitz et al. 2015]. 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. A position distribution is used to define the shape of the aggregate. It is specified as a mean depth and a variance, and it is necessary for taking into account the variance of light source directions during local shading estimation (particularly important in case of aggregates elongated in depth, and close light sources), as well as for filtering the shadowing term of the shading function (ie. filtering the shadow map), as explained later. In practice, this is re-constructed from the depth value and sub-pixel position of the samples inside an aggregate.

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 in the worst case AGAA performs n/c fewer surface shader evaluations than a system using supersampled shading, these evaluations are more costly. (see Section 5.3). 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 Shading models

The algorithm is independent of the shading model. For most of our experiments, we use the Blinn-Phong shading model, and we compute the pre-filtered shading for the Diffuse (Lambertian) and Specular components separately. We also tested the GGX specular distribution [Walter et al. 2007], which is becoming the industry standard for real-time rendering engines [Karis 2013; Lagarde and De Rousiers 2014; Schulz 2014]. Other analytic microfacets

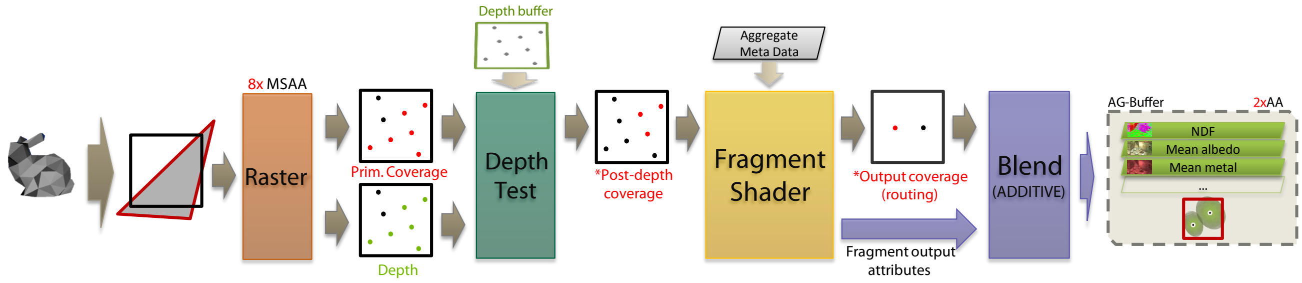


Figure 6: *Functional view of the hardware target-independent rasterization pipeline used for aggregate G-buffer generation. In this example, the rasterizer generates 8 coverage and depth samples, and the resulting AG-buffer contains two samples, one for each aggregate. Elements marked with a star require specific Maxwell features.*

distributions and BRDF models could also be used, as long as pre-filtering schemes exists, and their input parameters can be linearly pre-filtered [Bruneton and Neyret 2011; Olano and Baker 2010].

3.5.2 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 ; Crassin et al. 2009], there is no need for filtering visibility since we rely on the geometry pre-pass to aggregate visible attributes.

We approximate pre-filtered shading for Blinn-Phong or GGX specular reflectance using the Toksvig [2005] Normal distribution, because of its compactness and evaluation efficiency. Toksvig's distribution approximates the variance in the normal directions from the length of the stored 3D normal vector (length varies inversely with the variance of the orientation of the surfaces). We follow Toksvig in using that variance as an effective way to filter specular reflectance by increasing the roughness, e.g., lowering the Blinn-Phong's glossy exponent term.

Toksvig's approach was designed only for the Blinn-Phong reflectance. We apply the following simple steps in order to apply it to the filtering of GGX specular reflectance: (1) Convert the original GGX roughness r into Phong specular exponent $s = \frac{2}{r^2} - 2$. (2) Compute Toksvig factor $f_t = |\mathbf{Na}|/(|\mathbf{Na}| + s(1 - |\mathbf{Na}|))$ using the Toksvig vector $\mathbf{Na}$ stored for an aggregate. (3) Update the Phong specular exponent $s' = f_t * s$. (4) Convert specular exponent back to a GGX roughness $r' = \sqrt{2/(s'+2)}$. (5) Finally use updated r' to compute specular shading for the aggregate.

Note that for filtering specular reflectance, we also tested the anisotropic LEAN [Olano and Baker 2010] and the SGGX [Heitz et al. 2015] distributions for GGX specular reflectance. Both propose an analytic form for evaluating the filtered specular reflectance, and provide higher quality in case of important anisotropy in the surface orientations (cf. Section 5.1).

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 isotropic distribution. We also tested the simple numerical evaluation proposed with the SGGX anisotropic distribution(cf. Section 5.1). 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).

3.5.3 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 is 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 (reconstructed from the depth value and sub-pixel location of the 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 n is large, or 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; McGuire et al. 2010] and hardware alpha-to-coverage conversion [Kirkland et al. 1999]. Our implementation rasterizes fine-detail geometry modelled using alpha-textured polygons (e.g., the leaves of the trees in Figure. 1) as well as translucent primitives, using alpha-to-coverage. Since visibility is determined during the geometric prepass (cf. Section 3.2), sampling of the alpha texture and coverage generation only need to be performed during this pass. Aggregate definition (*clustering*) is performed similarly for stochastically generated samples and samples from standard opaque geometry. During AG-buffer generation, depth-testing (which uses the depth-buffer from the prepass) ensures that the right coverage, generated in the prepass, is used for aggregation. Similarly to deferred shading of aggregate MSAA samples from solid objects, aggregate deferred shading from stochastic sampling of semi-transparent primitives allows greatly reduced shading rate while preserving good image quality (cf. Section 5.1).

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 definition

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. The cost of this algorithm scales linearly with the number of samples n. This can be excessively expensive when n is large, but the actual geometric complexity of a pixel is low.

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 such a form that directly represents the $p \leq c$ triangles in a tile, aggregate definition can be accelerated by operating directly on the compressed representation, 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. For all other tiles, we rely on the clustering algorithm described in section 3.3, using per-sample depth and low-precision normal information. This optimisation is particularly useful for high MSAA rates, above 8× MSAA.

4.2 Target-independent rasterization into the aggregate G-buffer

The aggregate G-buffer generation algorithm, described in Algorithm 2 and illustrated in Figure 6, 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 to discard occluded samples. Second, we configure the input coverage mask provided to the fragment shader (into which we perform the 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

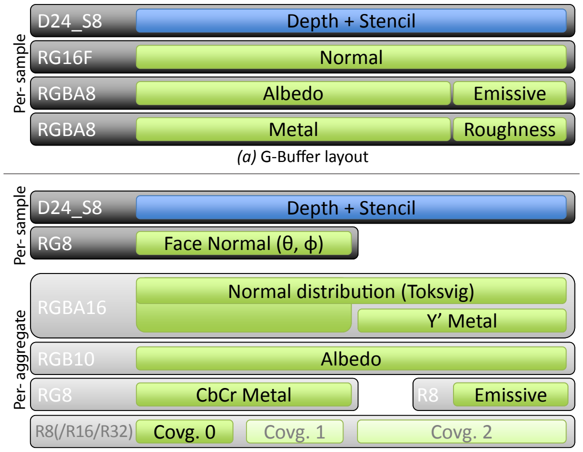
Because aggregate values will need to be re-normalized by the number of samples in their effective coverage mask before shading, it is important to ensure that no more than one primitive contribute to the same sample. Even with the depth-test of the generation pass set to EQUAL, such a situation can 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. This is done simply by incrementing stencil value by one on depth-pass, and discarding on $stencil \geq 1$.

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 ≈ 20 bytes apparently the most common on PC.

For the sake of our feasibility demonstration, and in the case of Blinn-Phong shading, we chose to encode attributes corresponding to the 16 bytes layouts presented in figure 7-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



(a) G-Buffer layout

(b) AG-Buffer layout

Figure 7: *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.*

to represent as filtered attributes for each aggregate (cf. section 3.5). We use the aggregate G-buffer layout presented in figure 7-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.

4.4 Hardware Emulation of High MSAA rates

Most current GPUs only natively support MSAA rates up to 8× MSAA. In order to reach higher sampling rates, we designed an hardware-based emulation scheme which allows generating more than 8 MSAA samples within a single geometric rendering pass. The general idea is to use hardware to rasterize each triangle multiple times at 8× MSAA, offsetting screen-space position at each rasterization pass in order to sample different locations on the primitives. For instance, we emulate 32× MSAA with 4 rasterization passes at 8× MSAA, as illustrated in figure 8. In contrast to rasterizing at higher pixel resolution (*supersampling*), this ensures that generated fragments get correct screen-space derivatives, and that texture samples exactly match those of a native MSAA implementation at the emulated rate.

We perform this rendering in a single geometric pass, and avoid using a costly *Geometry Shader* (required to duplicate each primitive), by taking advantage of *viewport multicasting*, a feature available on NVIDIA Maxwell (and newer architectures) through the `NV_viewport_array2` OpenGL extension [NVIDIA 2014]. This allows a single primitive to be transformed once by the vertex pipeline, and to be broadcast automatically, right before rasterization, into multiple layers of a *layered framebuffer*, and to be rasterized independently into each of them. The *layered framebuffer* contains 2D *array textures* (stack of 2D layers) instead of simple 2D textures, and each *layer* stores the depth, geometric normal (for the visibility *prepass*) or aggregate values for one of the 8× MSAA rasterization pass. After using this emulation during the AG-buffer generation pass, these multiple values generated for each aggregate are merged together, before executing the deferred shading step.

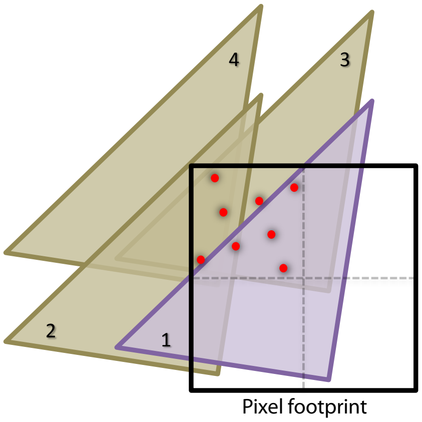


Figure 8: *Emulation of* 32× *MSAA rasterization for a single triangle. Original triangle (in purple) is rasterized 4 times using* 8× *MSAA (red dots), with fractional offset viewport positions. Programmable sample location is used to position samples in the top-left quadrant of each pixel.*

In order for the all the samples rasterized through the multiple passes to be uniformly spread inside each pixel, thus capturing different geometric information, sample locations need to be changed at each pass. These locations cannot be set independently for each rasterization pass during *viewport multicasting*. However, each *layer* has a separate *viewport definition* associated, which can specify a fractional viewport position used to compute *Normalized Device Coordinates* during the *viewport transform* stage. This allows us to specify a per-*layer* sub-pixel offset to the primitives, resulting in an equivalent (negative) offset of the base hardware sample positions.

Base hardware sample positions are chosen such as this per-pass offsetting properly distribute samples inside each pixel's footprint. We use the new *programmable sample location* feature, exposed in `NV_sample_locations` [NVIDIA 2014], to define the subsamples positions used for each pixel. In the example shown in figure 8, we emulate 32× MSAA by positioning the 8 MSAA samples inside the top-left quadrant of each pixel, while offsetting the *layers*' viewports by half a pixel in each direction. Other sample location patterns and viewport offsets could be used, in order to provide better quality sample distributions. For good quality, samples must be distributed as much uniformly as possible, avoid alignments resulting in correlation artefacts, and viewport offsets must keep samples inside each pixel's footprint.

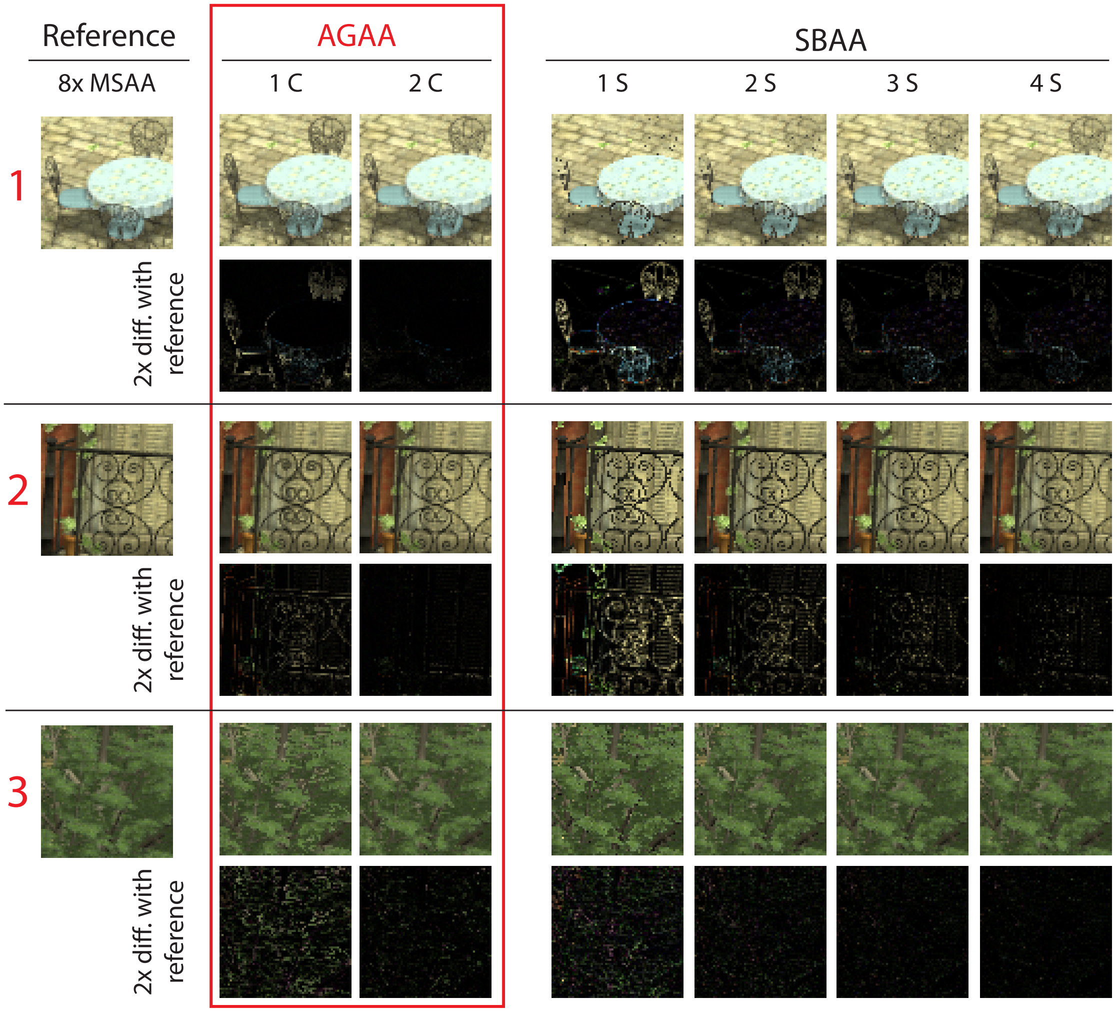


Figure 9: *Image quality comparison between Aggregate G-buffer Anti-Aliasing (AGAA) and Surface-Based Anti-Aliasing (SBAA) [Salvi and Vidimče 2012] for 1 to 4 surfaces per pixel. Each zoomed picture correspond to one of the crops in Figure1. Note that AGAA with 2 aggregates exceeds the quality of SBAA with 4 surfaces.*

5 Evaluation

We evaluate the performance and quality of AGAA on five scenes (Figure 15), 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 16-top) exhibits fine-scale geometry far beyond that used in video games today. Finally, the *metal* scene (Figure 16-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 1280x720 on an NVIDIA GTX 980 graphics processor (Maxwell GM204) using an OpenGL implementation of the algorithm described in Section 3. Unless stated otherwise, we used the Blinn-Phong shading model and a Toksvig Normal distribution for these experiments. 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.

The presented technique runs natively on current Maxwell hardware. However, for optimal performance above 8× MSAA, we rely on the ability to access the compressed representation of the depth buffer presented in Section 4.1, 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 16× and 32× MSAA results. 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 9 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 (SBAA) method of Salvi et al. [2012] configured to use its highest quality “merge” clustering predicate. 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 16 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 are 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 16-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. Generally, AGAA specular quality and temporal stability is dependent on the precision of the Normal Distribution Function, especially for complex arrangements of primitives, which can lead to anisotropic or multi-modal distributions.

5.1.3 GGX specular reflectance

We also evaluated image quality obtained when using the GGX specular reflectance model (instead of Blinn-Phong, cf. Section 3.5.1), which is widely used in recent game engines. Figure 10 shows image quality obtained with AGAA, using the Toksvig and the SGGX distributions as described in Section 3.5.2, as well as difference with per-sample shading ground-truth. Similarly to Blinn-Phong BRDFs, our approach approximates lighting of thin, high-curvature elements with GGX BRDFs relatively well when compared to the super-sampled reference. In presence of very elongated specular elements with anisotropic curvature, such as the tubes of metallic furnitures, the SGGX distribution gives better results than the isotropic Toksvig distribution, thanks to its anisotropic representation.

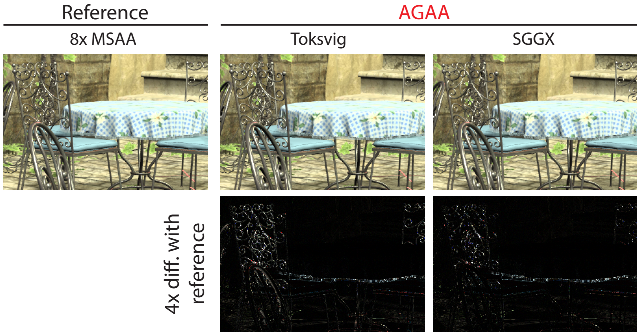


Figure 10: *Quality comparison for GGX specular reflectance using the isotropic Toksvig normal distribution function and the anisotropic SGGX distribution.*

5.1.4 Diffuse reflectance using SGGX distribution

As described in Section 3.5.2, we also pre-filter diffuse shading per-aggregate using the Normal Distribution Function. We compared the image quality for diffuse shading when using the isotropic Toksvig normal distribution function [Baker and Hill 2012], with the quality obtained with the anisotropic SGGX distribution using the numerical integration scheme proposed in [Heitz et al. 2015] (cf. Section 3.5.2). Image comparison with amplified per-pixel difference is shown in Figure 11. As expected, SGGX provides higher quality results than Toksvig where geometry exhibits important anisotropy. However, SGGX tends to generate slightly smoother results than the reference in other regions, which leads to a slight loss of contrast.

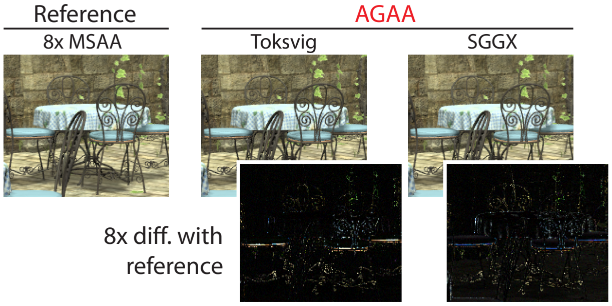


Figure 11: *Quality comparison for diffuse shading using isotropic Toksvig distribution and anisotropic SGGX distribution on elements with a high anisotropic curvature.*

5.1.5 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 12 compares the quality of AGAA shading to the baseline as the visibility sampling rate is increased from 4 to 32 samples per pixel. These results were produced using the emulation presented in Section 4.4. 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 Shading rate reduction

In Figure 13, we show the reduction of the average number of shading events executed per-frame (independently of the cost of these events) provided by AGAA (with $c = 2$ and dynamic number of aggregates) on the Old City scene, in comparison to the simple/complex approach [Lauritzen 2010] and SBAA [Salvi and Vidimče 2012] (with $S = 6$).

On average, the actual shading rate achieved by AGAA is less than 25% of the simple/complex approach, and 45% of SBAA. SBAA requires up to 6 surfaces per-pixel to reach similar image quality, which also results in almost $2\times$ the memory consumption of AGAA. The per-pixel analyse shows that SBAA is relatively inefficient in terms of shading execution on the GPU. This is due to the large pixel-to-pixel discrepancy in the shading rate (1-6 shading executions), which entails a worst case execution at the SIMD granularity. As shown in Figure 13, AGAA behaves much better in this respect, thanks to the much lower worst case pixel shading rate.

Time (ms)	**AGAA** - 2A					Reference				**Speedups**	
8×MSAA	Geom-prepass	Aggregate Def.	Gen.	Shading	**Total**	Simple/ Complex	Gen.	Shading	**Total**	**Shading**	**Frame**
Old City	2.61	0.60	3.80	3.60	**10.61**	5.73	0.41	10.24	**16.38**	2.84×	**1.54×**
UE3 FoliageMap	2.00	0.60	2.47	3.67	**8.74**	4.35	0.41	10.45	**15.21**	2.85×	**1.74×**
Bamboo	3.75	0.88	4.14	4.26	**13.03**	4.69	0.39	14.9	**19.98**	3.50×	**1.53×**
Metal	2.02	0.73	2.43	0.99	**6.17**	3.51	0.45	2.44	**6.40**	2.46×	**1.04×**
Fur Ball	1.90	0.71	1.84	0.39	**4.84**	3.56	0.49	0.81	**4.86**	2.08×	**1.00×**
32×MSAA											
Old City	10.38	2.74	15.80	4.80	**33.72**	30.80	3.70	31.10	**65.60**	6.48×	**1.95×**
UE3 FoliageMap	9.21	2.16	13.24	5.02	**29.63**	24.70	3.70	46.34	**74.74**	9.23×	**2.52×**
Bamboo	9.06	3.53	13.94	12.21	**38.74**	19.89	3.36	74.9	**98.15**	6.13×	**2.53×**
Metal	6.84	1.87	8.61	1.54	**18.86**	10.70	2.90	6.54	**20.14**	4.25×	**1.07×**
Fur Ball	8.26	1.84	10.48	0.87	**21.45**	16.54	2.92	1.94	**21.40**	2.23×	**1.00×**

Table 1: *Runtime performance (in milliseconds) for the main steps of AGAA ($c = 2$) at 8× MSAA (top) and 32× MSAA (bottom) 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.*

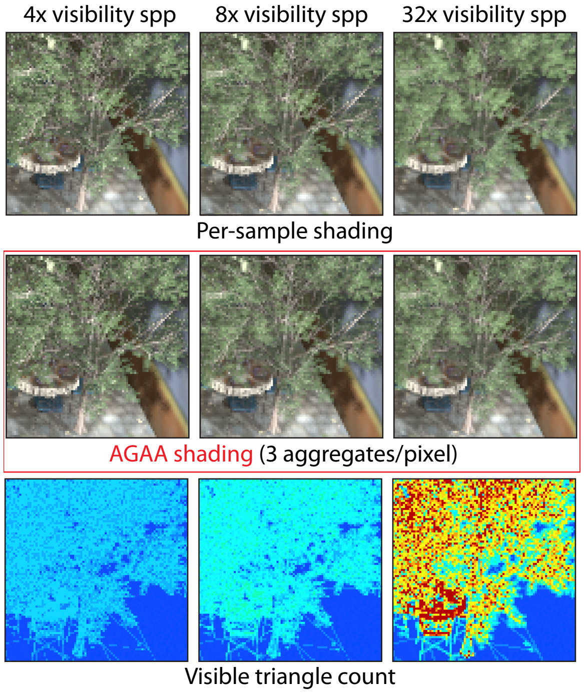


Figure 12: *Increase of visibility sampling rate allows capturing more triangles per-pixel (bottom row), which benefits to both traditional super-sampled rendering (top) and aggregate G-buffer rendering (middle), with more accurate and stable results.*

5.3 Execution performance

Table 1 shows execution performance numbers of AGAA for various scenes we have tested (Fig. 15), as well as speed-ups relative to the Simple/Complex approach [Lauritzen 2010] (setup for no quality degradation). We show performance results for both 8× and 32× MSAA and a maximum of two aggregates ($c = 2$). 8× MSAA is the highest MSAA rate natively supported by current GPU hardware. For 32× MSAA, we rely on the emulation procedure detailed in Section 4.4. 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. We used the Blinn-Phong shading model for these experiments. We observed $\sim 10\%$ higher speed-ups on the deferred shading pass when using the GGX specular shading model with the Toksvig distribution.

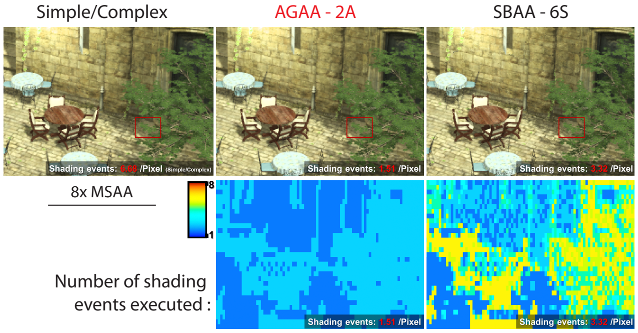

Simple/Complex
AGAA - 2A
SBAA - 6S
Shading events: 6.68 /Pixel (Simple/Complex)
Shading events: 1.51 /Pixel
Shading events: 3.32 /Pixel
8x MSAA
8
1
Number of shading events executed :
Shading events: 1.51 /Pixel
Shading events: 3.32 /Pixel

Figure 13: *Shading events measurement and comparison at iso-quality between per-sample Simple/Complex deferred shading (reference), AGAA ($c = 2$) and SBAA ($S = 6$), using 8× MSAA.*

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 speed-up of our technique comes from the shading pass. For the scenes we tested, it is at least 2× faster and up to 3.5× faster than the Simple/Complex per-sample shading at 8× MSAA. At 32× MSAA, the speed-up for the shading pass reaches 9.2×. The execution time of the shading pass is increased 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. On the entire frame, we observe up to 74% speed-up at 8× MSAA, and up to 156% at 32× MSAA.

At 32× MSAA, because of the emulation, the AG-buffer generation pass generates, for each of the final aggregates, 4 separate sets of sub-aggregate values that need to be merged together before shading. In addition to the cost of the emulation (measured inside the geometric pre-pass and the AG-buffer generation pass),

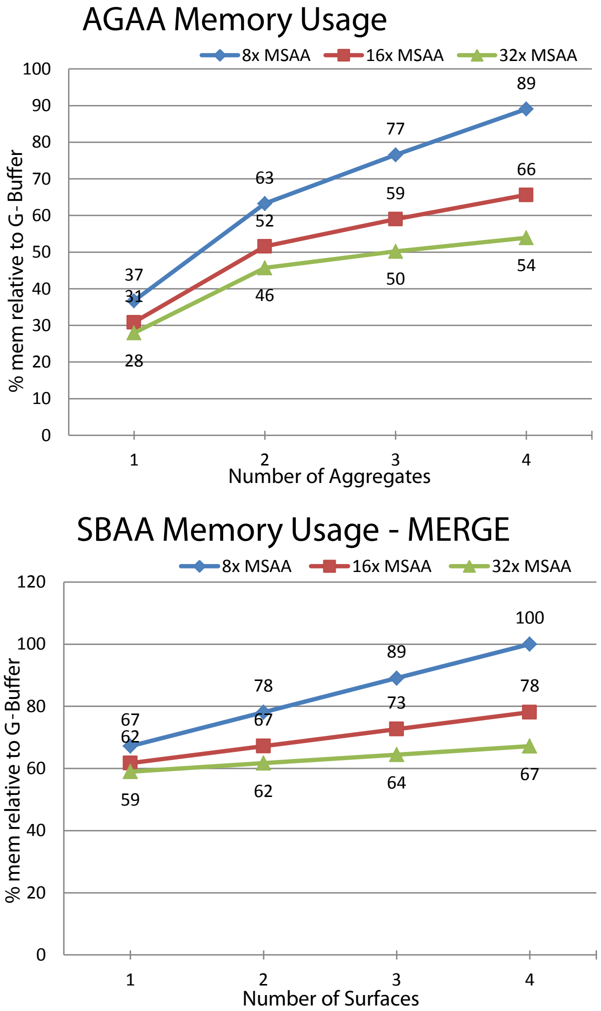


Figure 14: *Memory consumption of AGAA and SBAA in MERGE mode in percent relative to full multisampled G-buffer, depending on number of aggregates/surfaces and for* $8\times/16\times/32\times$ *rendering. Includes all required per-sample and per-aggregate storages.*

the cost of this merging procedure represents around 5% of the total frame time, and is integrated into the shading pass. Native support for $32\times$ MSAA would save both of these overheads, resulting in higher speed-ups.

5.4 Memory consumption

We analyzed the memory requirement of AGAA relative to a standard G-buffer implementation and to SBAA. Results are shown in figure 14. 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 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 9) as SBAA with $S = 4$, which corresponds to a $\sim$37% memory reduction.

6 Limitations

Similar to other pre-filtering techniques ([Bruneton and Neyret 2011]), our algorithm doesn't produce accurate results in the presence of important correlation between the values taken by independently filtered parameters inside the same aggregate. Figure 17 *(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, or if the sets of correlated samples were split into separate aggregates (here only one aggregate is used). From our experience, this kind of correlation effect is rarely visible when using at least two aggregates per pixel.

In some situations, our technique also fails to accurately model and preserve appearance of sub-pixel geometry inside the aggregates. This is true in case of long and thin geometry, like the thin hairs in Figure 17 *(b)*, which at some scales require an anisotropic normal distribution, like SGGX or LEAN, which are more costly to store and evaluate. This is more of an issue in presence of structured geometry, which generates multi-modal normal distributions that we can't represent precisely, like the railings in Figure 17 *(c)*. This problem appear only in presence of a few (2-3) lobes in the normal distribution, and can generate high-intensity noise. This issues is also rarely visible in practice when using at least two aggregates per pixel, thanks to our aggregate definition scheme (Section 3.3) which limit this effect by taking into account the normal direction for clustering. However, the problem of designing efficient and filterable multi-modal normal distribution functions remains open for future works.

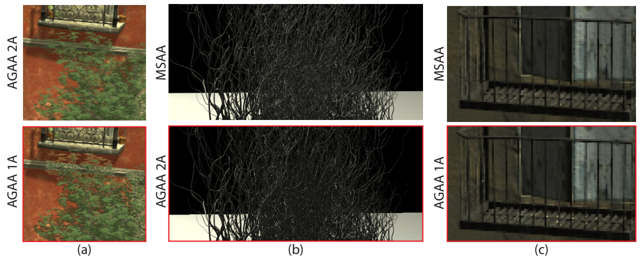


Figure 17: *Three failure cases for our filtering technique.*

7 Conclusion

This paper introduced Aggregate G-buffer Anti-Aliasing (AGAA), a technique to improve anti-aliasing of fine geometric details in deferred renderers, by allowing high geometric sampling rates. 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 demonstrated 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.

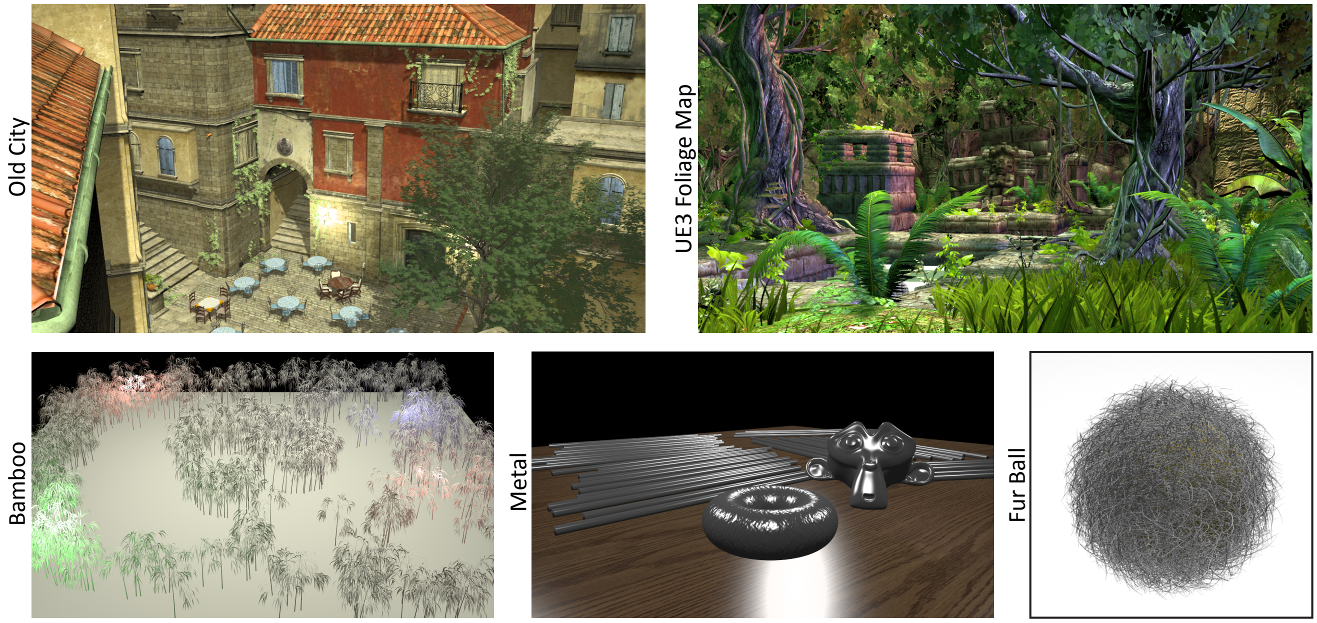


Figure 15: *The five test scenes rendered at* 1280×720 *: Old City, UE3 Foliage Map (Courtesy Epic Games), Bamboo, Metal and Fur Ball.*

The benefits of our technique, both in terms of memory saving and shading time, furthermore *increase* with as the geometric sampling rate increases. 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 and Louis Bavoil for the helpful discussions, Nir Benty for his help with the video results and editing, as well as the anonymous reviewers for their suggestions. We also want to thank the G3D Innovation Engine team for their great 3D engine that we used for prototyping, Epic Games for the UE3 Foliage Map scene, Anjul Patney for the Metal scene, Samuli Laine and Tero Karras for the Fur Ball model, Guillermo M. Leal Llaguno for the furniture and tree models, and we purchased the Old City model from TurboSquid.

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/`.

BAGHER, M. M., SOLER, C., SUBR, K., BELCOUR, L., AND HOLZSCHUCH, N. Interactive rendering of acquired materials on dynamic geometry using bandwidth prediction. In *Proceedings of I3D '12*, ACM, 127–134.

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*.

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. Appearance-preserving simplification. In *Proceedings of ACM SIGGRAPH '98*.

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.

CRASSIN, C., NEYRET, F., SAINZ, M., GREEN, S., AND EISEMANN, E. 2011. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum (Proceedings of Pacific Graphics 2011) 30*, 7 (sep).

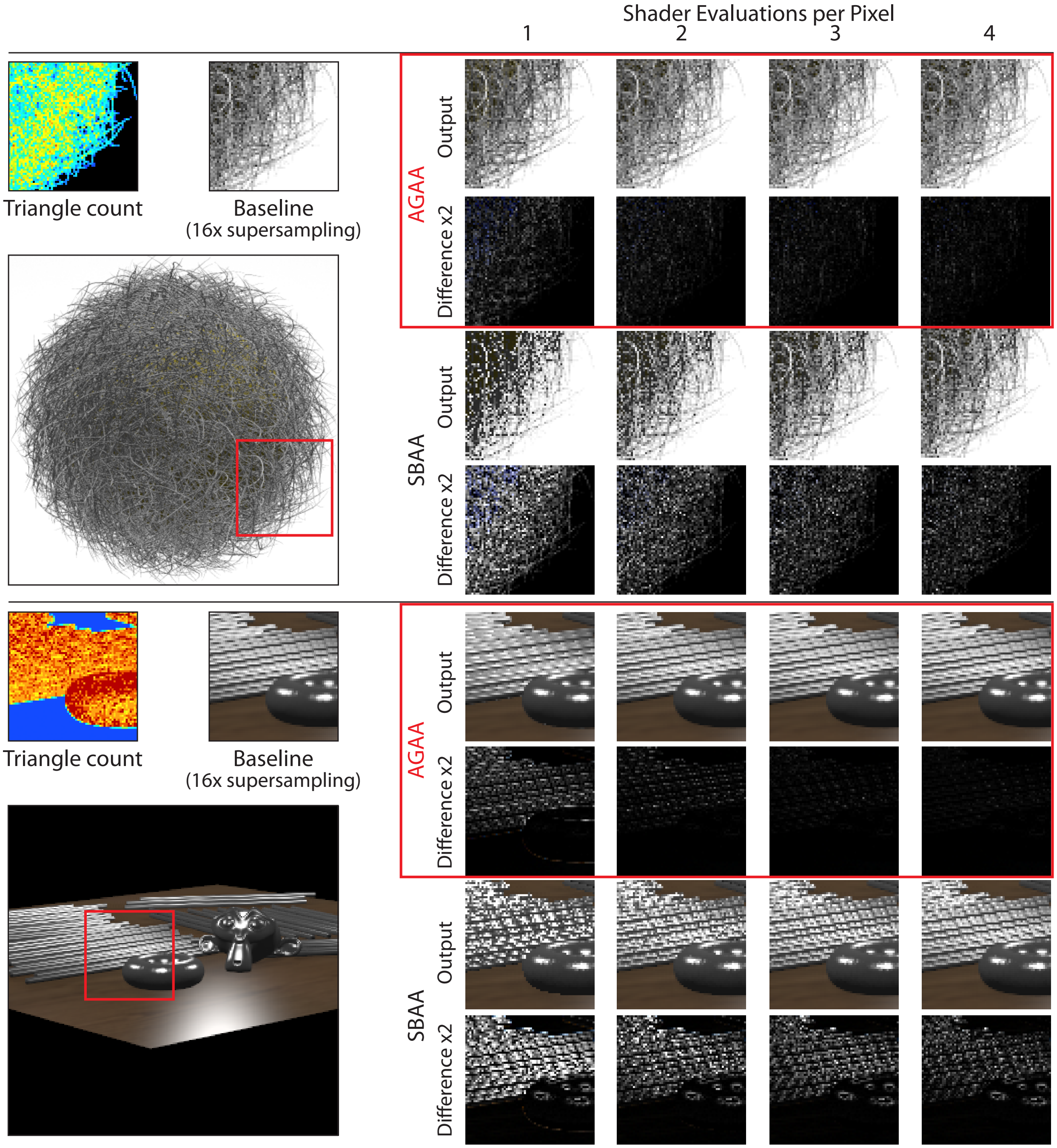


Figure 16: *Shading only a few aggregates stored in an aggregate G-buffer often closely approximate the results of super-sampled shading (shown here compared to 16× super-sampled shading). Image quality is noticeably better than that of Surface-Based Anti-Aliasing (SBAA). The improvement over SBAA is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

DECAUDIN, P., AND NEYRET, F. 2004. Rendering forest scenes in real-time. In *Rendering Techniques (EGSR)*, 93–102.

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.

FATAHALIAN, K., BOULOS, S., HEGARTY, J., AKELEY, K., MARK, W. R., MORETON, H., AND HANRAHAN, P. 2010. Reducing shading on gpus using quad-fragment merging. In *SIGGRAPH*, ACM.

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 '06*, ACM, 103–110.

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

HEITZ, E., DUPUY, J., CRASSIN, C., AND DACHSBACHER, C. 2015. The sggx microflake distribution. *ACM Trans. Graph. 34*, 4 (July), 48:1–48:11.

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.

HOLLANDER, M., BOUBEKEUR, T., AND EISEMANN, E. 2013. Adaptive supersampling for deferred anti-aliasing. *Journal of Computer Graphics Techniques 2*, 1, 1–14.

KARIS, B. 2013. Real shading in unreal engine 4. In *Physically Based Shading in Theory and Practice*, ACM, SIGGRAPH '13 Courses.

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.

LAGARDE, S., AND DE ROUSIERS, C. 2014. Moving Frostbite to PBR. In *Physically Based Shading in Theory and Practice*, ACM, SIGGRAPH '14 Courses.

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.

MCGUIRE, M., ENDERTON, E., SHIRLEY, P., AND LUEBKE, D. 2010. Real-time stochastic rasterization on conventional gpu architectures. In *HPG*.

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.

SCHULZ, N. 2014. The rendering technology of ryse. In *Advances in Real-time Rendering in Games*, ACM, SIGGRAPH '14 Courses.

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.

WALTER, B., MARSCHNER, S. R., LI, H., AND TORRANCE, K. E. 2007. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, EGSR'07, 195–206.

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.