Decoupled Coverage Anti-Aliasing

Yuxiang Wang
UC Santa Barbara

Chris Wyman
NVIDIA

Yong He
Carnegie Mellon University

Pradeep Sen
UC Santa Barbara

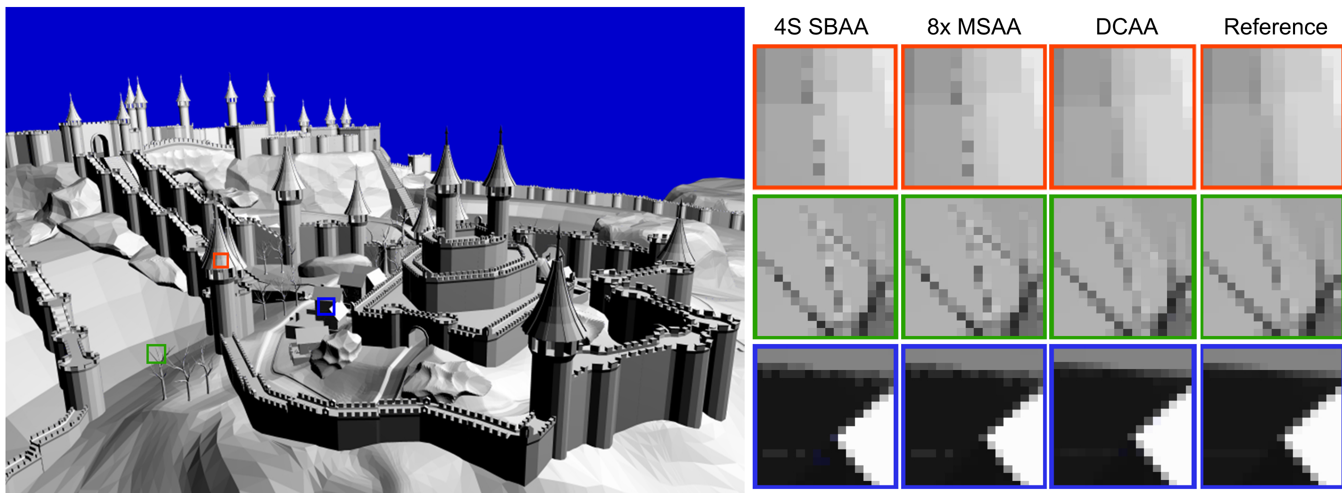

Figure 1: (left) *The* CITADEL *scene rendered with the proposed Decoupled Coverage Anti-Aliasing (DCAA) method.* **(right)** *Insets comparing various algorithms for geometric anti-aliasing. Although* $8\times$*MSAA still has visible aliasing in regions with fine-scale geometry (e.g., the tree branches or building detail), our approach produces results of comparable quality to the reference, which is computed by correctly downsampling a* $64\times$ *larger image computed with* $8\times$*MSAA (effectively* 512 *visibility samples per pixel).*

Abstract

State-of-the-art methods for geometric anti-aliasing in real-time rendering are based on Multi-Sample Anti-Aliasing (MSAA), which samples visibility more than shading to reduce the number of expensive shading calculations. However, for high-quality results the number of visibility samples needs to be large (e.g., 64 samples/pixel), which requires significant memory because visibility samples are usually 24-bit depth values. In this paper, we present Decoupled Coverage Anti-Aliasing (DCAA), which improves upon MSAA by further decoupling *coverage* from *visibility* for high-quality geometric anti-aliasing. Our work is based on the previously-explored idea that all fragments at a pixel can be consolidated into a small set of visible surfaces. Although in the past this was only used to reduce the memory footprint of the G-Buffer for deferred shading with MSAA, we leverage this idea to represent each consolidated surface with a 64-bit binary mask for coverage and a single decoupled depth value, thus significantly reducing the overhead for high-quality anti-aliasing. To do this, we introduce new surface merging heuristics and resolve mechanisms to manage the decoupled depth and coverage samples. Our prototype implementation runs in real-time on current graphics hardware, and results in a significant reduction in geometric aliasing with less memory overhead than $8\times$MSAA for several complex scenes.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Antialiasing I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: Anti-aliasing, real-time rendering, graphics hardware

1 Introduction

Aliasing artifacts have long plagued rendering systems, as sampling the scene to produce an image introduces aliasing wherever the frequency content of the scene is sufficiently high [Shannon 1949]. A common technique for anti-aliasing (addressing geometric, shader, and texture aliasing) is super-sampling, which increases the number of samples in each pixel [Fuchs et al. 1986; Mammen 1989; Haeberli and Akeley 1990]. However, high-quality super-sampling requires a large number of shading and visibility calculations which, although feasible for high-end offline rendering systems, are often too expensive for real-time rendering.

Therefore, researchers have explored anti-aliasing solutions specifically for real-time rendering. For example, real-time texture anti-aliasing is usually handled through mipmaps [Williams 1983], and there has also been work on real-time shader anti-aliasing [Olano et al. 2003]. For geometric anti-aliasing (i.e., aliasing due to edges in the scene geometry), the most effective solutions are based on Multi-Sample Anti-Aliasing (MSAA) which decouples shading and visibility by sampling visibility at multiple locations for each pixel but shading only once [Akeley 1993]. MSAA is based on the observation is that while shading/textures can be anti-aliased through other means, geometric anti-aliasing can only be addressed through more samples. Therefore, by limiting the number of shading samples, the memory bandwidth required is considerably reduced as compared to standard super-sampling.

However, geometric anti-aliasing methods based on MSAA have two general problems. First, they require a significant number

of visibility samples (e.g., 64 samples/pixel) to get a high-quality anti-aliased result for complex geometry with fine detail, as seen in Fig. 1. Since each visibility sample is usually stored as a 24-bit depth value, high-quality MSAA can consume an impractical amount of video memory (e.g., 1GB for 1920×1080 resolution).

Second, MSAA methods require even more memory for *deferred shading* rendering systems which are a popular choice for video game engines today [Mittring 2011; Tatarchuk et al. 2013]. These systems improve performance by deferring the typically expensive shading pass until the end, so that only visible surfaces are actually shaded. This requires storing all shader inputs for the nearest scene object at every sample in a structure known as a G-Buffer [Saito and Takahashi 1990], which for MSAA would grow linearly with the number of samples since each sample could come from a different object. For example, switching a $64\times$MSAA system at 1920×1080 resolution from regular forward rendering to deferred shading would increase the memory footprint from 1GB to 2GB for a typical G-Buffer configuration. Although researchers have begun to explore approaches to compress the G-Buffer by storing a small set of aggregate surfaces [Salvi and Vidimče 2012; Kerzner and Salvi 2014; Crassin et al. 2015], these approaches do not address the first problem with the large size of the visibility map for high-quality anti-aliasing.

In this paper, we propose a novel algorithm that addresses both problems with MSAA simultaneously and therefore enables high-quality, real-time anti-aliasing of complex scenes with fine geometric detail. Our key insight is that we can leverage existing work on G-buffer compression that aggregates samples into a few consolidated surfaces [Kerzner and Salvi 2014] to further decouple coverage from visibility (depth) calculations, thereby significantly reducing the size of the necessary G-Buffer to make it more practical. This requires the introduction of new rules for merging and discarding fragments to deal with the small geometric details captured by the coverage samples, but enables us to use a high-resolution, 64 sample/pixel coverage mask which allows us to accurately compute the weights of each color surface for high-quality geometric anti-aliasing. We demonstrate our algorithm running at real-time rates on current graphics hardware, and note that with dedicated hardware support the algorithm would be even faster. We now begin with a survey of related work in geometric anti-aliasing.

2 Related Work

As discussed earlier, the idea of decoupling shading and visibility sampling during anti-aliasing to reduce expensive shading computation was first proposed in Multi-Sample Anti-Aliasing (MSAA) [Akeley 1993]. There have been other methods proposed to decouple shading and visibility. To address problems with deferred shading, Ragan-Kelley et al. [2007] use an indirect framebuffer to preserve the relationship between shading and visibility samples, achieving a reduced shading rate similar to MSAA. Lauritzen et al. [2010] analyzes planar features that are shared in a multi-sampled G-Buffer and adaptively shades the pixel, lowering the shading cost. A similar idea is also applied for stochastic rasterization [Clarberg et al. 2013; Liktor and Dachsbacher 2012]. However, these approaches cannot address the memory footprint issue described earlier, as their samples still contain sizeable depth information and a large amount of per-sample data is still needed.

Various post-processing methods have been proposed that perform anti-aliasing by reusing information from adjacent pixels [Reshetov 2009; Lottes 2009; Reshetov 2012; Jimenez et al. 2011] or adjacent frames [NVIDIA 2014b], thus saving computation. These methods are typically fast and easy to integrate into a rendering engine, and have become a popular choice for modern video games. However, they have their own limitations, such as temporal artifacts for sub-pixel details or blurriness.

The A-Buffer algorithm [Carpenter 1984] builds an linked list for multiple fragments in a pixel in order to address transparency and anti-aliasing problem. The Z^3 algorithm [Jouppi and Chang 1999], as a hardware variation of the A-Buffer algorithm, stores z-slope and coverage map for each fragment, and save them in a fixed length list, which is similar to our algorithm. However, the Z^3 algorithm shades for every input fragment, which is more towards forward rendering; when the number of fragments exceeds the fixed size of list, the Z^3 algorithm forces to merge closest fragments together only according to weighted depth values, which will introduce more artifacts, e.g., merging fragments with large normal difference, than our merge and discard heuristics.

To reduce the storage overhead of many visibility samples per pixel, some algorithms, like our own, have proposed decoupling coverage from visibility. The CSAA/EQAA algorithms [NVIDIA 2010; AMD 2012] sample coverage more than visibility, and associate multiple coverage samples with each visibility sample. However, this association can change in unpredictable ways depending on fragment ordering, sometimes leaving covered samples with no corresponding visibility sample.

To reduce the size of the G-Buffer, Surfaced-Based Anti-Aliasing [Salvi and Vidimče 2012] introduced the notion of consolidating fragments into a small set of surfaces that could be independently shaded. To do this, they analyze the result of a MSAA prepass, then find the most important n fragments to store in the G-Buffer. This allows similar fragments to be merged together and limits the number of G-Buffer samples per pixel needed. The Streaming G-Buffer compression algorithm by Kerzner and Salvi [2014] improves upon SBAA by merging and discarding fragments on-the-fly in a streaming fashion, making it a single pass algorithm. In order to reduce the memory overhead, both approaches will sometimes need to discard fragments, which leads to artifacts in cases when many fragments contribute to a pixel. Aggregate G-Buffer Anti-Aliasing (AGAA) [Crassin et al. 2015] filters geometric information and compresses it into few aggregate surfaces, then shades each only once. However, all three approaches rely on MSAA for visibility sampling, and can achieve no better quality than the corresponding MSAA quality they use in their process.

Note that the Kerzner and Salvi [2014] method, like our own, only stores binary coverage and a single depth at the aggregated surfaces. However, they do this by computing visibility and then discarding the depth values, since hardware does not support decoupled depth and coverage sampling rates. Therefore, in their approach the memory footprint still includes the depth value per sample, which can be expensive at high sampling rates. In contrast, our method generates coverage maps results without depth information by using a shader, reducing the overall memory footprint.

Coverage determination is another important issue for anti-aliasing. Modern graphics hardware can use up to 8 samples per fragment for coverage and visibility determination, but increasing the number beyond that would be expensive. Researchers have studied ways this could be increased. For example, Waller et al. [2000] present a sub-pixel coverage map based on lookup table, which yields high quality anti-aliasing results when using an ASIC. Wyman et al. [2015] use a high quality coverage map to sample an irregular z-buffer to render anti-aliased hard shadows, enabling sub-pixel accuracy with little speed overhead.

Motivated by ideas such as the Streaming G-Buffer, we propose to address the problem of high-quality geometric anti-aliasing using a lossy compressed G-Buffer with a coverage map with 64 samples per pixel.

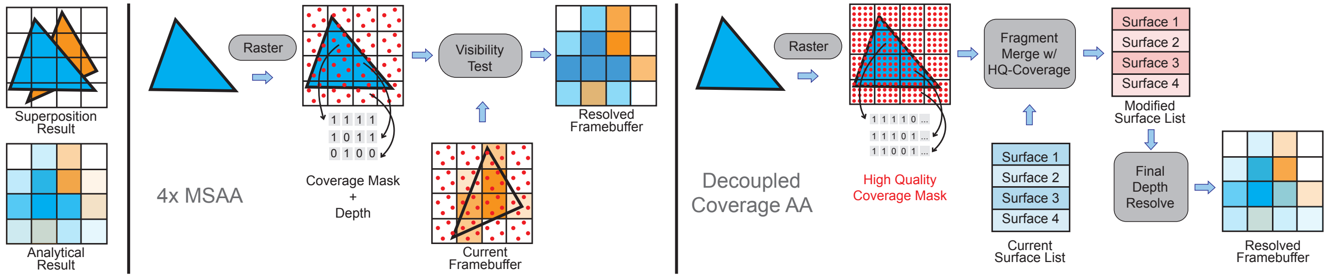

Figure 2: *Algorithm overview.* **(left)** *The superposition of two triangles, and the analytical framebuffer based on the covered area of each pixel (ground truth anti-aliasing).* **(middle)** *The* $4\times$*MSAA process averages the visible color of each sample to produce the resolved result. However, the small number of MSAA samples cannot provide precise result.* **(right)** *By decoupling coverage and visibility, our DCAA algorithm can generate high quality coverage map, and uses a new set of rules to merge the new fragment with existing surfaces. After the final depth resolve, the resolved framebuffer is more precisely weighted and is comparable to the analytical result.*

3 Algorithm

As shown in Fig. 2, our algorithm uses post-projection geometry information to get the explicit representation of triangles edges, and uses them to generate fragment coverage maps based on the triangle edge equations. We use the coverage map along with other information to aggregate surfaces, and merge or discard fragments based on a new set of rules we will discuss later. Finally we do a depth resolve to calculate the contribution of each surface. Specifically, for every fragment our algorithm takes the following three steps:

1. Generate coverage map (Sec. 3.1): we render the scene using a conservative rasterizer and use a shader to generate a high quality coverage map for each fragment.

2. First Merge Attempt (Sec. 3.2 and Alg. 1): we merge similar fragments together into aggregate surfaces stored in the G-buffer.

3. Second Merge Attempt (Sec. 3.3 and Alg. 1): If a fragment cannot be merged with an existing surface and the list of surfaces is full, we must discard the surface with the least visual contribution. However, before this happens we attempt a second merge with relaxed rules to try and preserve the information.

After all fragments are processed, we do a final depth resolve to determine the contribution of each surface (Sec. 3.4). We now discuss in detail each step of our algorithm.

3.1 Coverage Map Generation

A key aspect of our algorithm is decoupling coverage from visibility, which reduces the storage of each sample and thereby enables much higher coverage rates than MSAA. This, in turn, results in higher-fidelity weights for combining shaded samples, which improves anti-aliasing quality. Therefore, unlike the 24-bit depth used to represent coverage in MSAA, we use a bitmask with one bit per sample to identify which coverage samples belong to a specific surface, and associate a single depth value to each surface.

Since current graphics hardware generates at most 8 coverage samples per pixel, we use conservative rasterization and a custom fragment shader to perform denser coverage sampling for each triangle. For each fragment, we project its triangle edges onto the pixel and use a lookup table (LUT) to identify which samples in our bitmask are covered. Researchers have previously used similar LUTs (e.g., Waller et al. [2000]). The basic idea is to project each triangle edge independently, lookup the half-plane coverage in the LUT, and perform a binary AND of the contributions from all three triangle

Algorithm 1: Merge and discard algorithm

```
Input: s_in – input fragment, θ_t – normal angle threshold
 1: for all surfaces s_i in list S do    // first merge attempt (Sec. 3.2)
 2:   if n_i · n_in > cos θ_t and DepthOverlap(s_i, s_in) then
 3:     Merge(s_i, s_in);
 4:     merged = true;
 5:   end if
 6: end for
 7: if !merged then
 8:   if list is not full then
 9:     Store(s_in, S);
10:   else                              // second merge attempt (Sec. 3.3)
11:     CoarseDepthResolve(S, s_in);
12:     s_d = FindSmallestCoverage(S, s_in);
13:     for all surfaces s_i in {S, s_in} do
14:       if DepthOverlap(s_i, s_d) then
15:         Merge(s_i, s_d);
16:       end if
17:     end for
18:     if s_d != s_in then
19:       Discard(s_d, S);
20:       Store(s_in, S);
21:     end if
22:   end if
23: end if
```

edges to get the final coverage map. Specifically, for each LUT entry there is a corresponding directed line. When building the LUT, we test every coverage sample location in the pixel to see if it is located on the left of the line, which is treated as covered; then we save the coverage bitmask in the LUT. To use the LUT for coverage map, we find the corresponding LUT entries for three projected triangle edges, and then do bitwise AND for all three coverage bitmasks for the final coverage map, which is done in fragment shader. Fig. 3 gives an outline of this process.

One advantage of using a LUT is the ability to adjust sampling locations and the number of samples. In our implementation we use 64 coverage samples per pixel, but this can easily be increased, albeit with additional storage cost for a larger coverage mask.

3.2 Merging Heuristics and First Merge Attempt

As with some previous approaches [Salvi and Vidimče 2012; Kerzner and Salvi 2014], our algorithm exploits the fact that we cannot distinguish many distinct surfaces within a pixel, by merging similar fragments together to reduce the number of per-sample

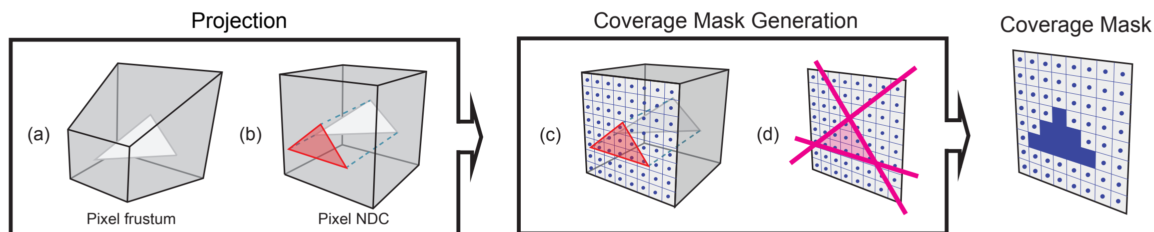

Figure 3: *We generate a 64-bit coverage mask for each fragment by: (a,b,c) projecting the triangle onto the pixel plane in either perspective space or normalized device coordinates, (d,e,f) using each edge to index into a lookup table to determine coverage for a half plane, and bitwise AND the three edge coverages to get the fragment's sub-pixel coverage map.*

shader inputs that need to be stored. However, compressing storage to our target of four per-pixel surfaces requires a set of merging heuristics. Since we want to merge surfaces that are close together and have similar orientations (e.g., adjacent triangles in a mesh), we use two common heuristics from prior surface-based approaches:

Aligned normal: The normals of two merge candidates should be close to each other. We found that a difference angle of $\theta_t = \pi/16$ worked robustly.

Overlapping depth: The two merge candidates should be similar in depth. To test this, we approximate a fragment bounding box using depth derivatives $\frac{dD}{dx}$ and $\frac{dD}{dy}$ over the pixel extent. These bounding boxes must overlap in depth in order to merge the fragments.

Unlike the work of Kerzner and Salvi [2014], we do not consider the overlap of surface coverage when merging. In fact, we found that the addition of this third rule resulted in artifacts in some cases. In a tree, for instance, it may be better to merge two slightly overlapping branches than to discard one because coverage samples overlap.

When a new fragment is rasterized, we apply the aligned normal and overlapping depth metrics to determine which existing surface provides the best merge candidate (i.e., closest normal and maximal depth overlap). We then merge the fragment and its closest surface, averaging normals, depths, and material parameters based on a weighted average of the fragment's and surface's coverage. We then combine coverage with a binary OR operation.

If no existing surface provides a satisfactory match, the incoming fragment becomes a new surface and is inserted into the pixel's surface list.

3.3 Discard and Second Merge Attempt

If we are unable to successfully merge our fragment with an existing surface and the per-pixel surface list is full, we cannot create a new surface. Kerzner and Salvi [2014] start discarding fragments in this situation, but this can lead to light leaking, especially when the discarded fragment lies on the geometry closest to the camera, as shown in Fig. 4.

Before discarding a surface, we instead perform a secondary merge step with relaxed merge rules. The idea is that instead of discarding information, we prefer to try to combine somewhat disjoint surfaces, resulting in approximate surface shading but with accurate coverage. Pixels with 5 or more surfaces generally contain aggregate geometry, like tree leaves, where shading with averaged G-buffer parameters makes sense [Crassin et al. 2015].

For our secondary merge attempt we select the surface with the *smallest visible coverage*, i.e., the surface whose contribution to pixel color would be smallest. This can be either the current fragment or any surface in the pixel's list. Since our surfaces do not contain resolved coverage, we *coarsely resolve depths* by ordering surfaces front to back and use bitwise operations to mask off samples covered by closer surfaces. The surface with the smallest coverage then becomes our candidate for merge (and possible discard).

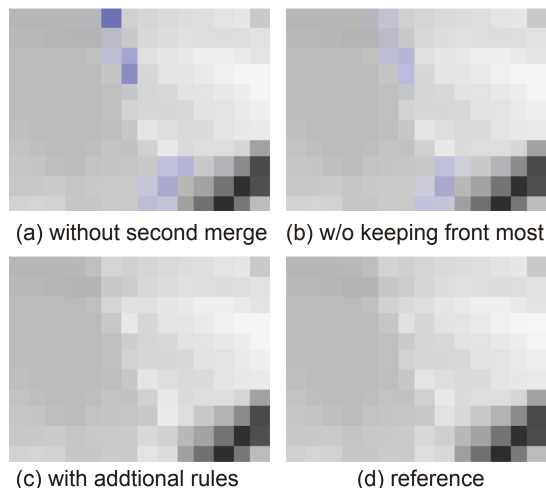

Figure 4: *Failure cases of original rules. (a) DCAA without second merge, some pixels leak to background color (blue). (b) Apply second merge attempt but without keeping front most surface, leaking is reduced, but some front most surfaces may be discarded and reveals the background. (c) DCAA with both additional rules, getting comparable result with (d) reference.*

We compare to other surfaces in the pixel using a single metric, the overlapping depth heuristic from Section 3.2. If the depth overlaps with any other surface, we merge them together (as before), otherwise the surface with smallest coverage is discarded (see Figure 5).

We also discovered that discarding the closest surface, even with low coverage, can cause visible popping and aliasing on fine details like hair, wires, and fences, as shown in Fig. 4(b). Therefore, we exclude the closest surface when identifying the one with the least coverage.

3.4 Final Depth Resolve

After processing all fragments we apply a depth resolve to determine coverage of each surface. Using the surface's depth and normal, we can approximate the depth of each coverage sample and mask off bits occluded by other surfaces. Specifically, for a screen

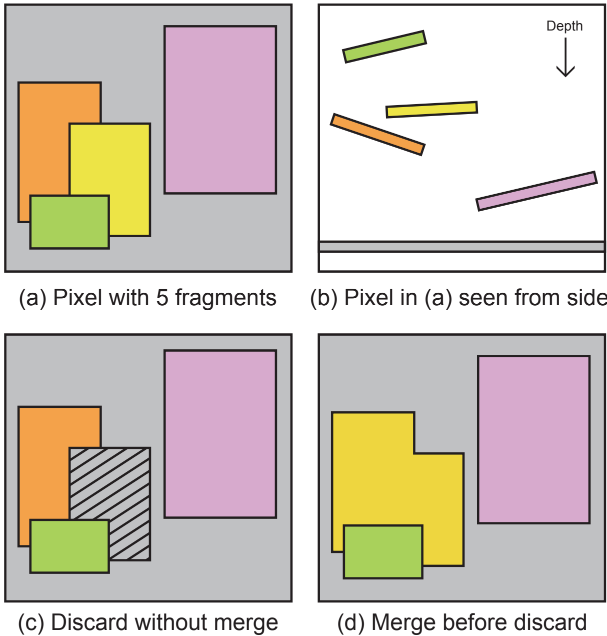

Figure 5: *Merge before discard. (a,b) When a new fragment comes, the pixel may have 5 surfaces, but can only contain 4 surfaces in buffer. (c) Discarding the smallest covered non-front-most one (yellow surface) may cause leak to the background. (d) Merge before discard keeps the coverage of the smallest surfaces, avoiding leak to the background.*

space coverage sample (x_i, y_i) and a surface plane in normalized device coordinates $f(\mathbf{p}) : \mathbf{n} \cdot \mathbf{p} + w = 0$:

$$\text{depth}(x_i, y_i) = -(\mathbf{n}_x x_i + \mathbf{n}_y y_i + w)/\mathbf{n}_z.$$

The final pixel color C is a weighted average of surface colors:

$$C = \sum_{i=1}^{S} \frac{n_i}{N} c_i$$

where S is the number of aggregate surfaces ($S = 4$ in our implementation), n_i is the number of sub-pixel coverage samples for the ith surface still visible after depth resolve and c_i is the shaded color of that surface, and N is the total number of samples (bits) in the coverage mask.

4 Implementation and Optimization

Our prototype uses OpenGL 4.5 running on a GeForce GTX 980, which provides various hardware features to help accelerate our algorithm.

4.1 Primitive Ordering and Coverage

In a streaming process, we compress incoming fragments into a fixed number of surfaces per pixel. Since fragments in the same pixel (from different triangles) may be processed simultaneously, we use a critical section to ensure hazard-free surface merging. We use NV_fragment_shader_interlock to guarantee atomic processing.

We use 64 coverage samples per pixel, but current hardware provides at most 8 samples per pixel. Instead we compute sample coverage in software (via a lookup table). To avoid missed coverage samples, this requires a fragment to be generated anytime any part of the pixel is covered. We use NV_conservative_raster to achieve this and extrapolate triangle attributes outside the original triangle.

4.2 Avoiding Stalls Via Z-Prepass

Simultaneously using conservative raster and fragment interlock provides poor performance. Conservative raster generates overlapping fragments for triangles sharing mesh edges, and fragment interlock stalls to avoid read-write hazards on overlapping fragments.

Eliminating primitives known to have no impact on final image quality is thus vital. We use a z-prepass to cull triangles known to be occluded. Since the z-prepass and our per-surface coverage use different sampling rates, care is required to perform culling conservatively. Our prepass stores the maximum z within the pixel for all visible surfaces. This can be approximated with an $8\times$ MSAA z-pass with max resolve or implemented exactly in a fragment shader.

4.3 Compressed Surface Data Format

Memory accesses can quickly become the bottleneck on modern GPUs, especially when part of larger critical sections like our surface merge. We attempted to minimize per-surface memory to save bandwidth and maximize performance.

As discussed in Section 3.4, we need a facet normal and depth to merge surfaces. To shade surfaces, we need standard G-buffer data including albedo, specularity, emissiveness, and roughness in addition to the ability to extract shading normal and eye-space position. We use recent practical G-buffer formats (e.g., Crassin et al [2015], Tatarchuk et al. [2013], and Mittring [2011]) to guide our format.

We compress normals to two 16-bit values (see [Cigolle et al. 2014]) and store depth in a 24-bit format; an evaluation of the effect of depth and normal compression is presented in Section 5.2.

Finally, we store per-surface aggregate coverage. This requires a number of bits equal to the sampling rate; we use 64 bits. This sampling rate could vary, trading performance for sampling fidelity.

Figure 6 shows a reference G-buffer layout compared to our surface format. While our surfaces require more space than corresponding G-buffer samples, we use only 4 surfaces per pixel independent of the sample coverage rate. With 64 coverage bits, we achieve quality close to $64\times$ supersampling but require only 112 bytes (4×24) per pixel; a $64\times$ supersampled result requires 1024 bytes (64×16), and $8\times$ MSAA requires 128 bytes (8×16).

5 Evaluation

We evaluate image quality, performance, and the surface compression on three scenes shown in Figures 1 and 7. CITADEL is a game-like scene with varying levels of geometric detail, ranging from terrain to tree branches. VILLA also contains fine tree branches, near-vertical fence poles, and distant window lattices. BAMBOO contains hundreds of bamboo stalks made up of tiny triangles, including sliver triangles and complex occlusion. TENTACLES is a 2.5 million triangle grass model. SPONZA is a scene with textures. SIBENIK is a scene with more complex material (Cook-Torrance model) and with textures. Table 1 shows our initial prototype's performance using DCAA.

All results were rendered on a GeForce GTX 980 at 1920×1080, using extended OpenGL 4.5. We compare, variously, with SBAA using "merge" strategy and 4 surfaces per pixel [Salvi and Vidimče 2012], $8\times$ MSAA, and a $512\times$ samples per pixel ground truth.

Table 1: *Performance for major steps of DCAA, and MSE value for comparison methods. Note that we merge fragments into surfaces in a single pass, so* Merge *includes all merge steps from in Sec. 3.*

Scene	Z-prepass	Merge	Resolve & Render	**Total**	MSE		
					4S SBAA	8× MSAA	DCAA
CITADEL	1.3 ms	23.2 ms	6.4 ms	**30.9 ms**	2.47×10^{-4}	1.32×10^{-4}	6.40×10^{-5}
BAMBOO	4.1 ms	39.3 ms	9.5 ms	**52.9 ms**	7.93×10^{-4}	4.65×10^{-4}	1.63×10^{-4}
VILLA	0.6 ms	8.7 ms	5.2 ms	**14.5 ms**	2.67×10^{-4}	2.20×10^{-4}	6.90×10^{-5}
TENTACLES	1.3 ms	574.5 ms	6.2 ms	**582.0ms**	2.28×10^{-3}	6.05×10^{-4}	5.65×10^{-4}
SPONZA	0.4 ms	12.8 ms	2.5 ms	**15.7ms**	1.00×10^{-4}	9.70×10^{-5}	9.60×10^{-5}
SIBENIK	0.3 ms	6.8 ms	2.7 ms	**9.8ms**	1.18×10^{-4}	1.03×10^{-4}	1.01×10^{-4}

R8	G8	B8	A8
	Depth		Stencil
Normal U		Normal V	
Diffuse Albedo RGB			Emissive
	Metal		Roughness

(a) A typical G-Buffer layout (per sample)

R8	G8	B8	A8
	Depth		Stencil
Normal U		Normal V	
Face Equ U		Face Equ V	
	Coverage Mask 1		
	Coverage Mask 2		
Diffuse Albedo RGB			Emissive
	Metal		Roughness

(b) DCAA G-Buffer layout (per surface)

Figure 6: (a) *A typical G-Buffer layout per sample for deferred rendering.* **(b)** *Our DCAA G-buffer layout per surface with same information as the reference. Note that though DCAA takes more space per surface, it has few surfaces (4 in our experiments), comparing to the number of samples for MSAA.*

5.1 Image quality

Figures 1 and 7 compare DCAA, 8× MSAA, and our ground truth. Along small details and near vertical and horizontal edges, our algorithm approaches our ground truth. In general, we generate higher quality weights, allowing small geometry to fade out more slowly as coverage decreases; 8× MSAA is limited to multiples of $\frac{1}{8}$ for weights, leading to higher magnitude flickering for fine details.

In general DCAA provides higher image quality than 8× MSAA and SBAA with 4 surfaces, preserving fine geometry almost as well as ground truth. DCAA has similar quality to the ground truth result, while using only 11% as much memory as 64x supersampling. Note that DCAA can process texture and non-diffuse material, as shown in SIBENIK scene in Fig. 7.

5.2 Compression

To improve performance we can compress geometry into fewer surfaces. Fig. 9 shows the number of surfaces need for CITADEL. Most of the scene only require 1 surface per pixel, but for regions such as the towers in CITADEL, we need more surfaces to correctly describe the scene with high quality coverage. We found 4 surfaces provided a sweet spot, capturing the most important layers in complex pixels and avoids discarding key geometry. Note that Crassin et al. [2015] use 2 aggregates; Kerzner and Salvi [2014] use 3 surfaces. Our method needs more surfaces because we have much more coverage samples in each pixel, which can capture more details and tremendously increase the complexity of clustering. The top row in Fig. 8 shows the effect of varying surface count in CITADEL.

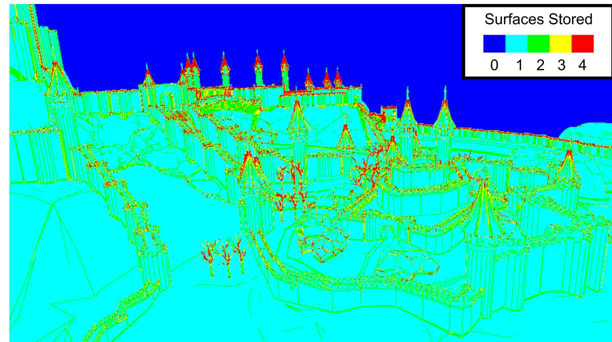

Figure 9: *Number of surfaces stored per pixel for* CITADEL.

The second and third rows in Fig. 8 compare various methods for compressing normal and depth information. We tried various bit representations for the normal using both 2- and 3-component representations, using two 16-bit components provided reduced storage without a significant reduction in image fidelity. We found a 32-bit depth plane is unnecessary, but further compression beyond a standard 24-bit depth plane introduces errors when merging smaller, nearby surfaces.

5.3 Merge and discard heuristics

Our merge and discard heuristics reduce the surfaces needed for most of the scene, while using more surfaces for geometrically complex regions. This allows high quality anti-aliasing while reducing memory usage. Fig. 7 shows our merge and discard heuristics are effective and robust for different shading situations, e.g., texturing and non-diffuse materials. Though our method is order dependent, we use fragment shader interlock to ensure the atomic operation when merge and discard fragments. It also guarantees that the interlocked fragments are processed in application call ordering, which is unlikely to change from frame to frame. Thus, our method has no temporal artifact in general cases.

However, compression based on heuristics is lossy and introduces errors that fall into two major categories: merge errors and discard errors. Fig. 10(a) shows the blurred texture artifact. Having a maximum of 4 surfaces, our method shades up to 4 times per pixel, which may not be enough for high frequency textures in some scenes. Additionally, our merge heuristic may blend textures from different surfaces together, further blurring the result. Note that 8×MSAA also suffers from the low shading sampling rates. Using

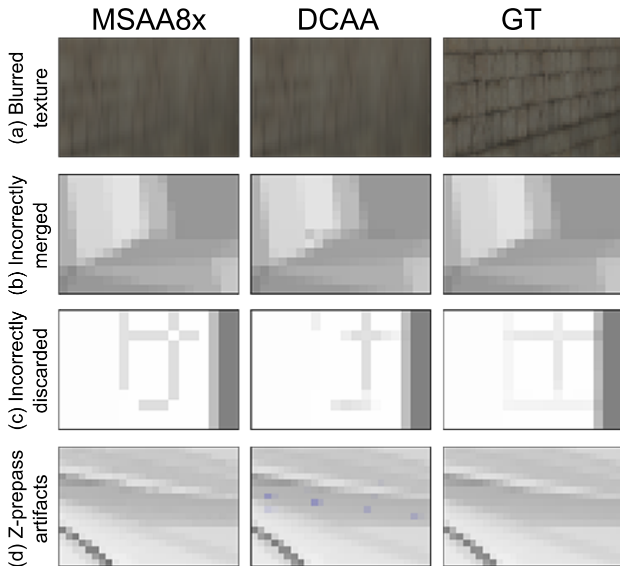

Figure 10: *Artifacts introduced by different phases of our method, comparing with MSAA and ground truth. In those cases, MSAA has more aliasing than the our method, but introduces less artifacts, e.g., wrongly merged surfaces and missing geometry, as discussed in Sec. 5.3.*

proper shading anti-aliasing technique will reduce such artifacts.

We target merging spatially aligned surfaces in close proximity. But this also averages merged normals, smoothing the normal changes. In Fig. 10(b), two walls intersects in a small angle; our method generates pixels in averaged color of the two walls at the intersection, which is caused by blurred normal. Using more strict normal comparisons reduces this artifact, but also generates more surfaces.

Discarding visible surfaces obviously can introduce artifacts. However, for highly complex pixels where a large number of fragments contribute, discarding barely visible surfaces saves a lot of memory with only small noticeable errors. Our second merge attempt is designed to reduce the impact of discards as much as possible. In Fig. 10(c) the vertical part of the window frame covers very small portion on the pixel, but visually contributes to the final result. Our heuristic tends to discard the frame and reveals the white wall, introducing mild artifacts compared to ground truth. Another similar possibility, though not observed in our experiments, is that small surfaces with high contribution may be discarded. More precise discard heuristics considering material information may alleviate this artifact.

5.4 Limitations and hardware implications

Our major limitation is performance, mainly due to simultaneous use of conservative rasterization and fragment interlock, which forces serialized processing of more fragments than common rendering techniques; disabling either introduces artifacts but improves performance by an order of magnitude in simple scenes. Performance degrades more in scenes with sub-pixel triangles, like TENTACLES, where partially-covered pixels that require synchronization with neighbors form a larger percent of fragments. If GPUs exposed finer grained synchronization at the fragment level or merging to reduce the additional synchronization from conservative rasterization, performance would dramatically improve by reducing the percentage of threads needing serialization. We use shaders to generate coverage maps and face plane equations; exposure of this information at a lower level would benefit our algorithm.

Due to the fragment synchronization issues noted above, overdraw in areas of high depth complexity introduces serialization that has a large performance impact. We minimize this with a z-prepass to cull triangles. Our prototype uses an 8× MSAA pass to find the maximum z per pixel, and we discard more distant fragments during our merge. Because of the difference in sampling rates, this is not fully conservative and can cause the background to leak along geometric cracks (see Figure 10(d)). A more conservative z-prepass would solve this issue; either higher hardware MSAA, analytically computing the maximal depth per-pixel, or matched sample rates would work.

6 Conclusion and Future Work

We introduce *decoupled coverage anti-aliasing*, a streaming compression algorithm for geometric anti-aliasing. Our method approaches the quality of our 512 sample per pixel ground truth while using less memory than 8× MSAA.

The key insight is a decoupling of visibility into depth and coverage samples, allowing much higher resolution coverage samples in a usable memory footprint. Our prototype demonstrates only a few surface shades per pixel can provide quality comparable to highly supersampled images.

Our implementation relies on early hardware conservative rasterization and fragment shader interlocks, limiting performance due to artificial serialization. Additional hardware support, such as native high-sample coverage masks of finer grained synchronization would significantly improve performance. Additionally, we hope to explore combinations with aggregate anti-aliasing [Crassin et al. 2015] to simultaneously address shader and geometric aliasing.

Acknowledgements

We thank NVIDIA for the hardware donation of a GeForce GTX 980, as well as the following artists for providing the scenes: Erik Sintorn (CITADEL, VILLA), Dylan Lacewell (BAMBOO), Crytek (SPONZA) and Marko Dabrovic (SIBENIK). We also thank Aaron Lefohn and Anjul Patney for helpful discussions, Zi Wang for making the illustrations, and Chieh-Chi Kao, Ekta Prashnani, and Abhishek Badki for helping to put together the results. This project was funded by NSF grants IIS-1342931 and IIS-1321168, and by an NVIDIA internship.

References

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

AMD. 2012. EQAA modes for AMD 6900 series graphics cards. Tech. rep., AMD.

CARPENTER, L. 1984. The A-buffer, an antialiased hidden surface method. *ACM Siggraph Computer Graphics 18*, 3, 103–108.

CIGOLLE, Z., DONOW, S., EVANGELAKOS, D., MARA, M., MCGUIRE, M., AND MEYER, Q. 2014. A survey of efficient representations for independent unit vectors. *Journal of Computer Graphics Techniques 3*, 2, 1–30.

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.

CRASSIN, C., MCGUIRE, M., FATAHALIAN, K., AND LEFOHN, A. 2015. Aggregate G-Buffer anti-aliasing. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*, 11.

FUCHS, H., GOLDFEATHER, J., HULTQUIST, J. P., SPACH, S., AUSTIN, J., BROOKS, JR., F. P., EYLES, J., AND POULTON, J. 1986. Fast spheres, shadows, textures, transparencies, and image enhancements in pixel-planes. In *Advances in Computer Graphics I (Tutorials from Eurographics'84 and Eurographics'85 Conf.)*, Springer-Verlag, London, UK, UK, 169–187.

HAEBERLI, P., AND AKELEY, K. 1990. The accumulation buffer: Hardware support for high-quality rendering. *SIGGRAPH Comput. Graph. 24*, 4 (Sept.), 309–318.

JIMENEZ, J., GUTIERREZ, D., YANG, J., RESHETOV, A., DEMOREUILLE, P., BERGHOFF, T., PERTHUIS, C., YU, H., MCGUIRE, M., LOTTES, T., MALAN, H., PERSSON, E., ANDREEV, D., AND SOUSA, T. 2011. Filtering approaches for real-time anti-aliasing. In *ACM SIGGRAPH Courses*.

JOUPPI, N. P., AND CHANG, C.-F. 1999. Z^3: an economical hardware technique for high-quality antialiasing and transparency. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, ACM, 85–93.

KERZNER, E., AND SALVI, M. 2014. Streaming G-buffer compression for multi-sample anti-aliasing. In *Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics*, The Eurographics Association, 159–164.

LAURITZEN, A. 2010. Deferred rendering for current and future rendering pipelines. *SIGGRAPH Course: Beyond Programmable Shading*, 1–34.

LIKTOR, G., AND DACHSBACHER, C. 2012. Decoupled deferred shading for hardware rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '12, 143–150.

LOTTES, T., 2009. Fast approximate anti-aliasing. `http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf`.

MAMMEN, A. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *Computer Graphics and Applications, IEEE 9*, 4 (July), 43–55.

MITTRING, M. 2011. The technology behind the Unreal Engine 4 Elemental demo. *SIGGRAPH Course: Advances in Real-Time Rendering in 3D Graphics and Games*.

NVIDIA, 2010. Coverage sampling antialiasing. `http://www.nvidia.com/object/coverage-sampled-aa.html`.

NVIDIA, 2014. NVIDIA OpenGL extensions specifications. `https://developer.nvidia.com/nvidia-opengl-specs`.

NVIDIA, 2014. TXAA technology documentation. `http://www.geforce.com/hardware/technology/txaa/technology`.

OLANO, M., KUEHNE, B., AND SIMMONS, M. 2003. Automatic shader level of detail. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, HWWS '03, 7–14.

RAGAN-KELLEY, J., KILPATRICK, C., SMITH, B. W., EPPS, D., GREEN, P., HERY, C., AND DURAND, F. 2007. The lightspeed automatic interactive lighting preview system. *ACM Transactions on Graphics (Proc. SIGGRAPH) 26*, 3 (July).

RESHETOV, A. 2009. Morphological antialiasing. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, HPG '09, 109–116.

RESHETOV, A. 2012. Reducing aliasing artifacts through resampling. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGGH-HPG'12, 77–86.

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

SALVI, M., AND VIDIMČE, K. 2012. Surface based anti-aliasing. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, 159–164.

SALVI, M. 2013. Pixel synchronization: Solving old graphics problems with new data structures. *SIGGRAPH Course: Advances in Real-Time Rendering in Games*.

SHANNON, C. E. 1949. Communication in the presence of noise. *Proceedings of the IRE 37*, 1, 10–21.

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*.

WALLER, M., EWINS, J., WHITE, M., AND LISTER, P. 2000. Efficient coverage mask generation for antialiasing. *Computer Graphics and Applications, IEEE 20*, 6 (Nov), 86–93.

WILLIAMS, L. 1983. Pyramidal parametrics. *SIGGRAPH Comput. Graph. 17*, 3 (July), 1–11.

WYMAN, C., HOETZLEIN, R., AND LEFOHN, A. 2015. Frustum-traced raster shadows: Revisiting irregular z-buffers. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*.

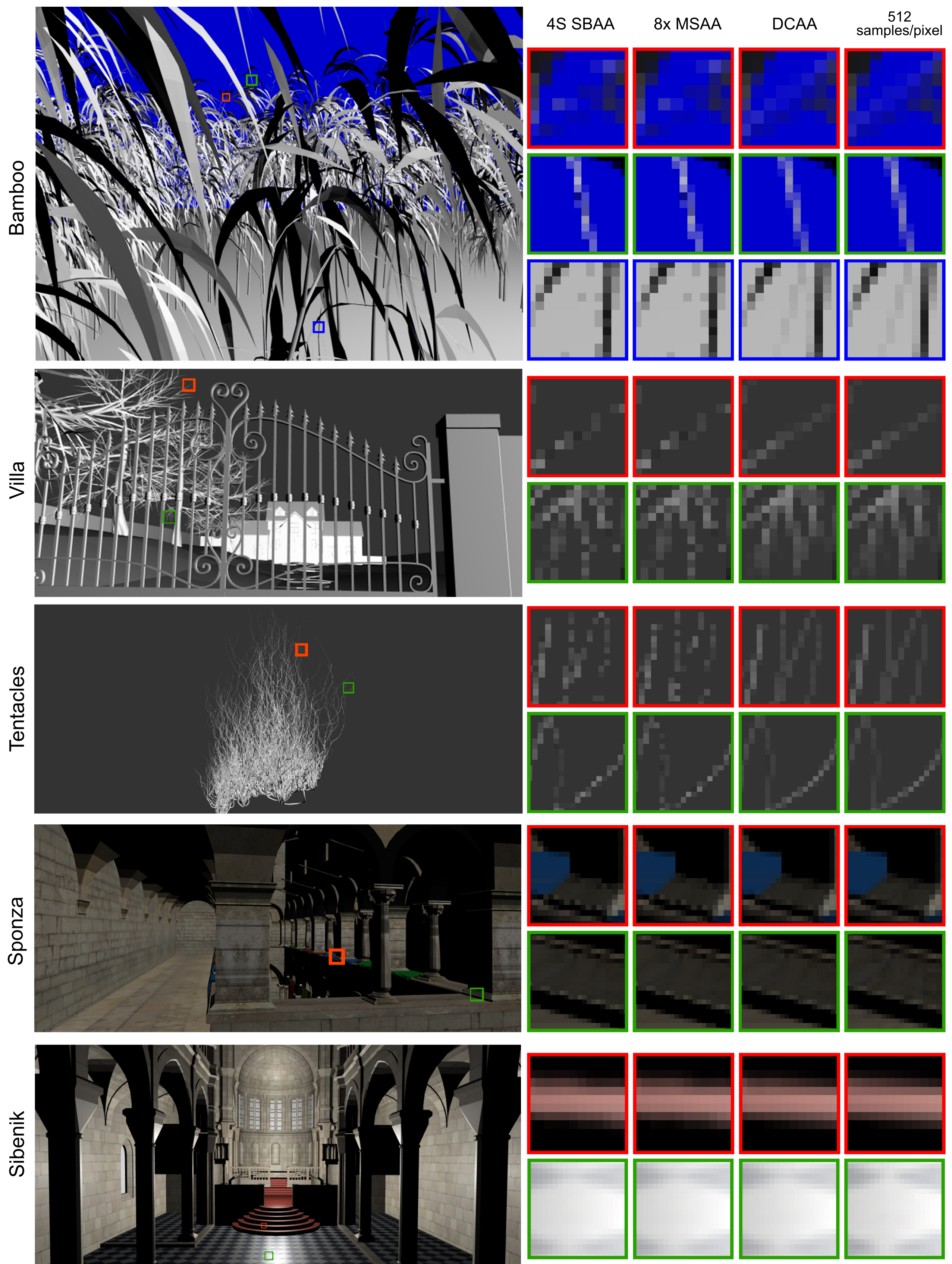

Figure 7: *Image quality comparison between Decoupled Coverage Anti-Aliasing (DCAA), Surface Based Anti-Aliasing (SBAA), $8\times$ MSAA and the reference. DCAA provides results close to $512\times$ supersampling, especially for fine-scale geometry.*

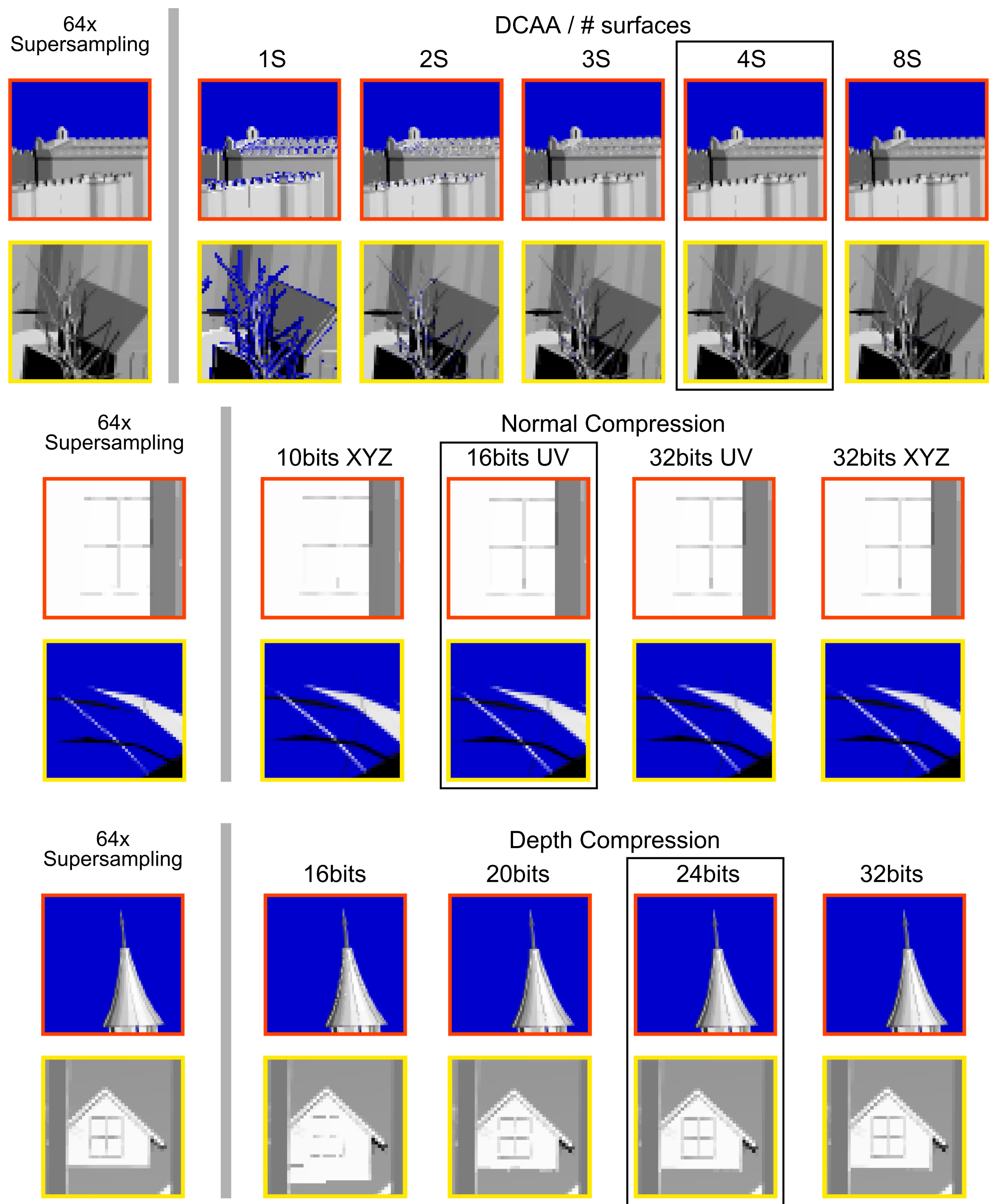

Figure 8: *Comparison of different compression settings. The insets are cropped from the corresponding scenes shown in Figs. 1 and 7. The settings with a box are the ones that gave us the best compromise of quality and performance, and were the ones we used for our experiments.*