High Performance Graphics (2012)
C. Dachsbacher, J. Munkberg, and J. Pantaleoni (Editors)

Reducing Aliasing Artifacts through Resampling

Alexander Reshetov

Intel Labs

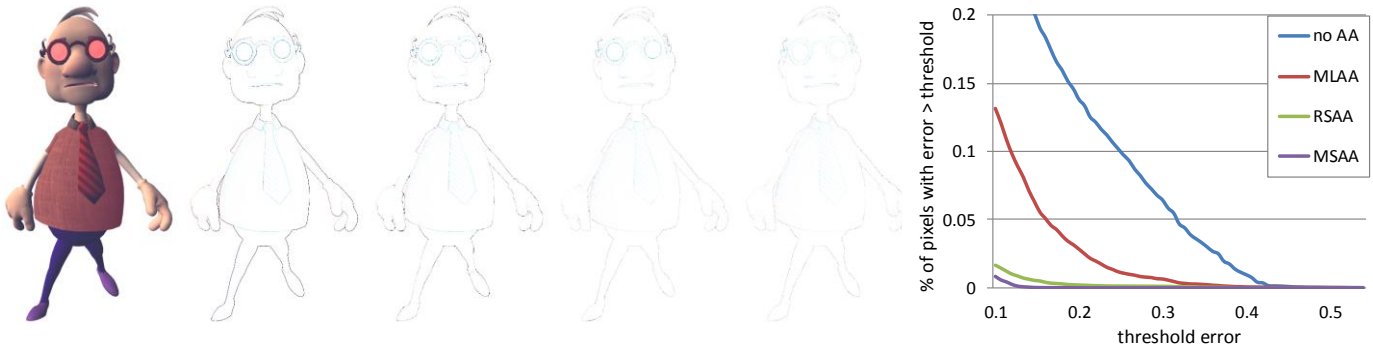

Figure 1. *Comparison of different antialiasing techniques for "Edgar" model. Left: absolute RGB errors with respect to 64x supersampling, multiplied by 4 (the bigger the error the darker the pixel). Right: percentage of pixels with luminosity error exceeding the threshold (i.e. for MLAA, 0.1% of pixels have errors exceeding 0.13). The maximum threshold error is 1.0.*

Abstract

Post-processing antialiasing methods are well suited for deferred shading because they decouple antialiasing from the rest of graphics pipeline. In morphological methods, the final image is filtered with a data-dependent filter. The filter coefficients are computed by analyzing the non-local neighborhood of each pixel. Though very simple and efficient, such methods have intrinsic quality limitations due to spatial undersampling and temporal aliasing. We explore an alternative formulation in which filter coefficients are computed locally for each pixel by supersampling geometry, while shading is still done only once per pixel.

During pre-processing, each geometric subsample is converted to a single bit indicating whether the subsample is different from the central one. The ensuing binary mask is then used in the post-processing step to retrieve filter coefficients, which were precomputed for all possible masks. For a typical 8 subsamples, it results in a sub-millisecond performance, while improving the image quality by about 10 dB.

To compare subsamples, we use a novel symmetric angular measure, which has a simple geometric interpretation. We propose to use this measure in a variety of applications that assess the difference between geometric samples (rendering, mesh simplification, geometry encoding, adaptive tessellation).

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation — Antialiasing
Keywords: antialiasing, post-processing effects, deferred shading, clustering

e-mail: alexander.reshetov@intel.com

1 Introduction

Rendering is about sampling. Modern graphics hardware efficiently antialiases texture samples, allowing shading parameters to be processed at a lower rate than geometric characteristics. This is rather fortunate, as shading is the most expensive part of the graphics pipeline.

Table 1 shows a high-level snapshot of existing antialiasing techniques. We used shades of gray to indicate typical sampling rates for a number of entities required for antialiasing computations (a white color for a single sample per pixel and scaling up to a black color for many subsamples). This table illustrates the fact that all of these values can be sampled at different rates. Bandwidth requirements are estimated for the deferred shading pipeline [GPB04]. This makes hardware-accelerated multisample antialiasing and coverage sampling antialiasing [Ake93, You06] less efficient, since all color subsamples have to be written to output buffers for later processing.

Morphological methods [IK99, Res09, BHD10, Per10, Bir11, JME*11, Lot11, JES*12] are the most economical in conserving bandwidth, as they require only a single color sample per pixel. These methods *hallucinate* silhouette edges from color (or depth) discontinuity data and then blend the colors around the found edges. This results in plausible single image antialiasing, comparable with more evolved image-space optimization techniques [DH72, Fat07]. However, this plausibility is broken in animation sequences, as the silhouettes might be reconstructed differently in subsequent frames.

On the contrary, geometric methods [BWG03, CD05, Mal10, GG12, Per12] antialias pixels intersected by precisely computed silhouette edges. Though very accurate, this limits the usability of such methods to moderately complex scenes.

Temporal aliasing artifacts are handled in high-quality variants of Subpixel Morphological Antialiasing [JES*12] and TXAA [Lot12] by processing additional color subsamples. TXAA uses hardware multi-sampling, an advanced sample-to-pixel filter, and temporal supersampling. SMAA addresses temporal aliasing from a software perspective by supplementing morphological antialiasing with additional multi/supersampling strategies. Naïve combination of supersampling and morphological methods results in excessive blurring in the spatial domain and ghosting in the temporal domain. SMAA resolves this by offsetting the reconstructed silhouettes to match the subpixel positions, which is computationally efficient.

DOI: 10.2312/EGGH/HPG12/077-086

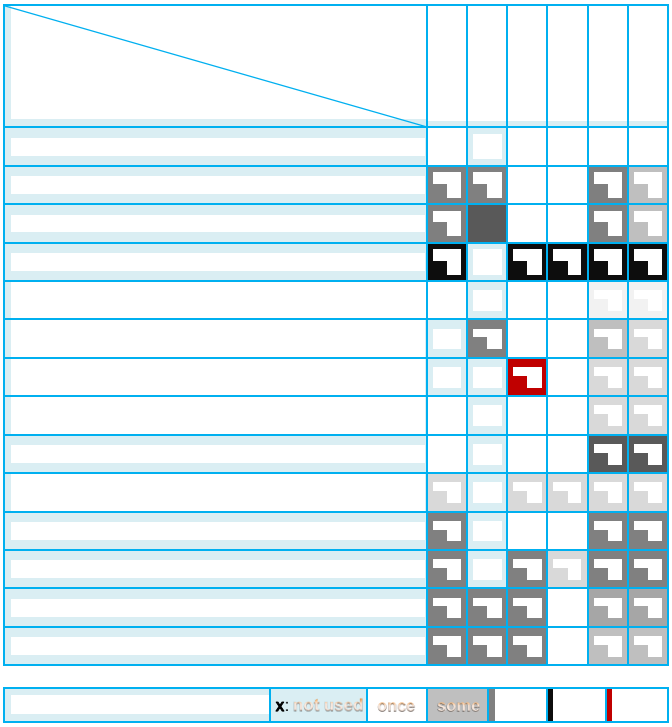

Table 1. *Guestimate of sampling rates for different antialiasing techniques. Bandwidth (last column) is estimated for the deferred shading pipeline.*

Unlike these two techniques, we restrict ourselves to a single color sample per pixel, but allow multiple geometric subsamples (positions and normals). Similar approaches were tried before. In sub-pixel reconstruction (SRAA), cross-bilateral filtering is applied to upsample per-pixel colors using multisampled geometry [YSL08, CML11]. Deferred MSAA [Pet10] averages the positions of those MSAA samples, whose depths are meaningfully different from that of the central one. The resulting coordinates are then used to bilinearly filter the final image.

Our approach requires a geometry pre-pass, at which a bitmask is computed for each pixel. At post-processing stage, this mask is used to fetch filter coefficients from a precomputed lookup table. We call our new algorithm Resampling Antialiasing (RSAA).

Each bit in the mask corresponds to an MSAA subsample indicating whether the subsample is different from the pixel center. The similarity is determined by analyzing geometric data alone, relying on the fact that color and geometry are closely correlated. The bitmask splits all subsamples in a pixel into two clusters, depending on whether they are similar or dissimilar to the pixel center.

Conceptually, RSAA

1. Sets the colors of all subsamples in the similar cluster to the color of the central subsample.
2. Assumes that all subsamples in the dissimilar cluster have the same color, and that this color can be acquired from the pixel neighborhood.
3. Computes the final pixel color by blending the colors of these two clusters with weights proportional to their coverage.

This is all executed in a single bilinear texture request. In effect, RSAA attempts to emulate supersampling. This would work for simple scenes consisting of polygons of only two colors. For such scenes, we could precompute the optimal filtering coefficients for all possible bitmasks by minimizing the average error. This is essentially how we compute the lookup table, which is then used for real scenes.

This approach works as long as assumptions 1 and 2 are accurate. This allows antialiasing quality closely matching the quality of MSAA, which is used for geometry sampling, except for scenes with a substantial amount of sub-pixel detail (see Figures 1 and 9). Even for such scenes, the quality of RSAA exceeds that of any morphological method with a single color sample per pixel. In comparison with hardware-accelerated MSAA, RSAA (and other post-processing methods) allows a significant bandwidth reduction in deferred shading applications.

RSAA depends on a considerable correlation between geometry and shading parameters. For most situations, this is a correct assumption. However, violation of this assumption can result in whimsical artifacts. We resolve this by adding an additional verification step, which mitigates spurious filtering decisions by verifying that assumption 2 is indeed true.

In all comparisons with MLAA in this paper, we have used the original implementation [Res09], as our goal was to ascertain the quality of different antialiasing techniques. The subsequent modifications [Per10, Bir11, JME*11, JES*12] were aiming at improving performance and content-specific optimizations. We observed that SMAA at 1x preset does not improve the peak signal-to-noise ratio, compared with the original MLAA implementation. The likely explanation is that most of the performance-driven versions of MLAA, unlike the original version, restrict the neighborhood size and use half-edge silhouette intersections only.

A detailed account of state-of-the-art filtering approaches for real-time antialiasing can be found in [JGY*11].

2 Antialiasing through Resampling

During the pre-processing pass, we compare each subsample in a pixel with the pixel center, setting up a bit mask ('1' for similar, '0' for different subsamples). This mask, in effect, allows retrieving precomputed filter coefficients. To fully utilize hardware texture unit, we restrict ourselves to the situations when filtering is done through bilinear interpolation by issuing a single texel request per pixel.

For each pixel, we fetch texture coordinate offsets from a pre-computed lookup table, using the mask as an index. At the post-processing step, the final color of each pixel is computed by bilinearly interpolating the original image at the position defined by the fetched offset vector. Figure 2 shows such offsets for a given triangle as red vectors.

The offsets are precomputed for all possible masks and stored in a lookup table. For any given mask, the 2D offset vector is chosen to minimize the average error for a realistic training data set. Automated learning of low-dimensional parametric models from training data is a standard paradigm in computer vision; we explain the details of our approach in section 2.3.

In principle, any similarity measure can be used for subsample comparison. Each choice has its own pros and cons. In the next section, we describe a novel approach that we call *sum of the absolute dot products* (SADP). It has a simple geometric interpretation, is reasonably fast, and robustly handles difficult cases. SADP can be used in various computer graphics problems that infer sample similarity from geometric data. In particular, most methods in Table 1 can benefit from this new measure.

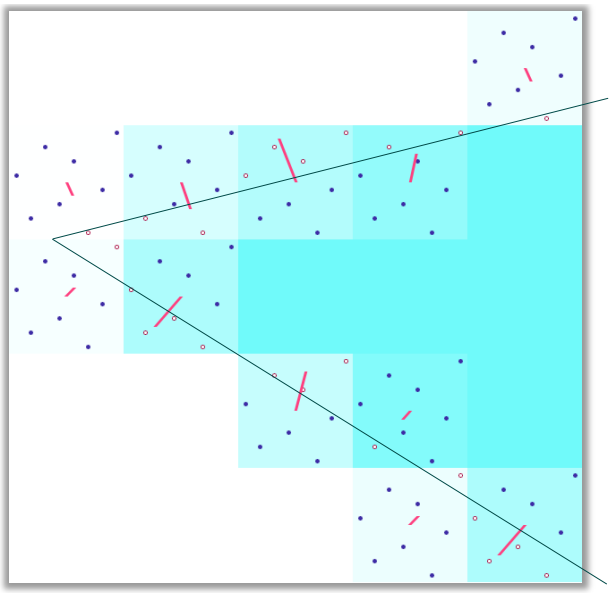

Figure 2. *RSAA offsets for a single triangle (shown as red vectors from pixel centers). Blue dots stand for subsamples that are similar to the pixel center, red for subsamples that differ.*

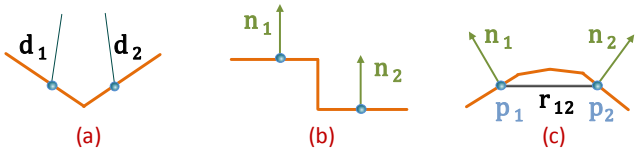

Figure 3. *Three ways to detect different samples: using (a) depth, (b) normals, (c) the new SADP test.*

2.1 Geometry → Similarity Measure

One of the most widely used approaches is to compare the depth of two subsamples, since it is readily available in the graphics pipeline. Deferred MSAA [Pet10] uses linearized depth to measure subsample similarity. Besides requiring calibration and linearization, this approach cannot reliably detect corners, as shown in Figure 3a ($d_1 == d_2$).

Corner detection could be resolved with a measure that uses subsample normals – for example, their dot product. However, this measure will not be able to distinguish spaced-out almost-parallel surfaces as in Figure 3b.

We can hope that combining these two approaches will somehow avoid the described pitfalls, and this combination is indeed used in practice. Surface based antialiasing [SV12] sorts out MSAA subsamples using the dot product of the subsample normals, in addition to their depth distribution. Nevertheless, assigning weights to the utilized components is non-trivial and scene-dependent. The situation in Figure 3b might still be problematic if the distance between two planes were small.

For two subsamples with positions p_1 and p_2 and (geometric) normals n_1 and n_2, we propose to use the following measure:

This is a sum of the absolute dot products of two normals and a normalized vector from one subsample to another. We call it SADP. HLSL implementation of this measure is given in Listing 1 (lines 36-38).

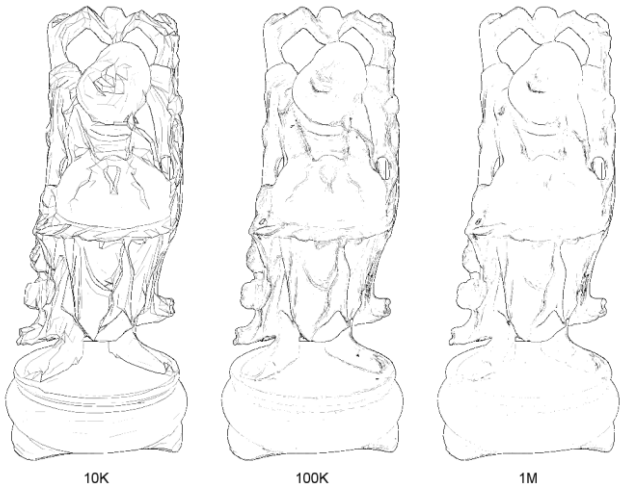

Figure 4. *Pixels with non-zero RSAA offsets for Happy Buddha at three different simplification levels (10K, 100K, and 1M triangles). Pixel colors are proportional to the maximum dissimilarity value.*

For two subsamples on the same polygon, SADP is zero, and it increases as angles between normals and r_{12} become bigger. When the dot products are relatively small, each dot product is close to $\pi/2 - \alpha$, where α is the angle between the normal and r_{12}.

To cluster subsamples, we will also need a similarity threshold. Two subsamples with SADP exceeding the threshold will be considered dissimilar. To choose the threshold, let's remember that interpolated normals are used to create a smoothly varying illumination of a coarse mesh [Pho75]. For a triangle, the maximum angle between all its per-vertex normals signifies a tolerance threshold of what is still considered to be "smooth". This value, averaged over all geometry, can be used as the SADP threshold as well.

The RSAA authentication phase (section 2.4) can remove the majority of "wrong" offsets, so, in a practical sense, exact choice of the threshold is not that important. In all examples in this paper, we have used the value 0.4, which corresponds to two angles of about 11°.

Among other desirable SADP properties, it is symmetric and does not depend on camera position. Figure 4 shows a Happy Buddha model at three different simplification levels. All pixels in which subsamples differ from the pixel center by more than 0.4 are marked with a gray color. The color intensity is proportional to the SADP value. SADP generally allows detecting sharp mesh creases. The total number of these creases decreases as the mesh becomes smoother.

For specific applications, SADP clustering can be supplemented by other factors, including light visibility, material ids, etc. (see discontinuity buffer discussion in [Kel98]).

2.2 Similarity Measure → Two Clusters

We want to split all subsamples in a pixel into two clusters based on their similarity. This is one of the most common problems in science and there are multiple approaches to its solution. The simplest approach is putting all subsamples with similarity measures below the given threshold in one cluster, and all others in another. This is not advisable, as it fails to maximize the dissimilarity between the clusters. Indeed, in the case with the similarity = {0.4, 0.4, 0.4, 0.5, 1.1, 1.2, 1.2, 1.3}, all eight subsamples will be put into the "dissimilar" cluster. This is clearly not desirable and, if anything, we would like to err on the side of caution and put the questionable subsamples into the "similar" cluster.

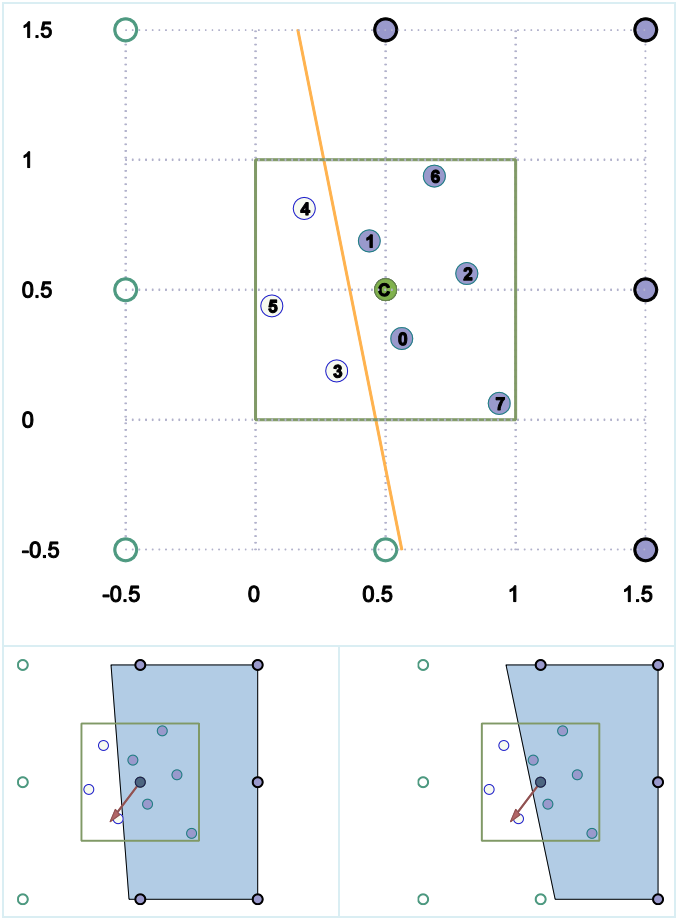

Figure 5. *Subsamples similar to the pixel center are shown as blue, dissimilar – as white. Bottom: the best offset is shown as red vector (and it is not orthogonal to the yellow SVM classifier on top image).*

This problem can be solved with *k-means* [Mac67], but this algorithm is iterative and generally time-consuming. We have opted for a single iteration, consisting of two steps:

1. Maximum of all similarity values is found (**maxsim**).
2. If **maxsim** exceeds a given **threshold**, all subsamples are classified into two groups by comparing their similarity values with **maxsim/2 + threshold/2**. Otherwise, no actions are taken (pixel's color will not be modified).

This classifier would split the group above into {0.4, 0.4, 0.4, 0.5} and {1.1, 1.2, 1.2, 1.3}, which is the desired result. The bias **threshold/2** is introduced to favor the center of the pixel for which the color is known a priori.

Listing 2 provides HLSL implementation of this classifier: step 1 in lines 195–202 and step 2 in lines 276–288.

The subpixel reconstruction algorithm [CML11] does not explicitly classify the subsamples, instead using their similarity values as an input to cross-bilateral upsampling filter. It is a very interesting approach, allowing 1 ms execution on GeForce 560 at 1080p. It is not void of artifacts and its combination with morphological methods is advised [JGY*11]. Some of these artifacts are due to the differences between bilateral filtering and *bona fide* supersampling. We believe that SADP measure (1) can reduce SRAA artifacts.

Deferred MSAA [Pet10] utilizes non-adaptive (linearized) depth thresholding. It might be interesting to see if adaptive clustering, introduced in this section, will improve deferred MSAA quality.

2.3 Two Clusters → Resampling Offsets

We want to compute resampling offsets based on a bit mask that characterizes two clusters. We will first describe the prior art solutions followed by our approach.

In simple cases, the found clusters could be separated by a line. This intuitively corresponds to a situation of two distinct surfaces overlapping the pixel. If these surfaces were simple polygons and we knew the neighborhood, this would allow very accurate reconstruction, perhaps even using the Hough transformation [DH72]. Indeed, looking at Figure 2, the reconstruction error is necessarily bounded by a small value.

Geometric antialiasing methods [BWG03, CD05, Mal10, GG12, Per12] use scene data to improve image quality. In our case, we would like to restrict ourselves only to sampled data. Moreover, we do not want to infer costs associated with accessing subsamples in the neighboring pixels. If we restrict ourselves only to the current pixel data, the accuracy of the silhouette reconstruction (either explicit or assumed) will suffer. Notably different cases may result in the same clustering, as shown in the bottom of Figure 5.

Deferred MSAA [Pet10] avoids explicit silhouette reconstruction by directly computing the resampling offsets. This is done by averaging directions to all dissimilar subsamples. Considering the situation at the top of Figure 5, the resampling offset will be computed by averaging vectors `C3`, `C4`, and `C5`. Though very simple, it appears that this approach does not scale well with the increased number of samples. If only subsamples `2`, `3`, `5`, and `6` were different, the resulting offset would be close to zero and the final pixel color would not change meaningfully. It would differ significantly from the supersampled solution, in which only subsamples `0`, `1`, `4`, and `7` will have color `C`.

Support Vector Machine (SVM) methods allow classification of both linearly separable (as in Figure 5) and inseparable data sets [BB00]. This might seem like a good idea; Figure 5 shows such classification as a yellow line. This line maximizes the minimum distance to the two clusters. We could just consider the orthogonal vector as a direction of the resampling offset, but this approach has its drawbacks as well. It is derived for a continuous case, while in computer graphics sampling positions are predefined, essentially making the problem discrete.

The red vector on the two images at the bottom of Figure 5 deviates from the vector orthogonal to the SVM line by tilting down. Consequently, when the final image is bilinearly resampled, the pixel with coordinates `[0.5,-0.5]` will contribute more. Since this pixel is blue on the left image and white on the right, the resampled color for the pixel `C` will be closer to blue for the situation on the left. This is exactly the result we want to achieve since it corresponds more closely to the supersampled solution, which basically uses area of the trapezoids overlapping the pixel `C` to blend between white and blue colors, and this area is bigger on the left. Indirectly, we will be able to take into account samples outside the current pixel, even though we are using only the current pixel data.

To actually find the best values for the resampling offsets in each of 255 cases (for 8 subsamples), we will approach this problem from a computer vision perspective. In CV problems, it is customary to come up with a generic model and then train it on experimental data. Since our model has 2 parameters (resampling coordinates) for each of 255 cases, we just treat it as an optimization problem: find the values of these parameters which minimize average error for the given training set.

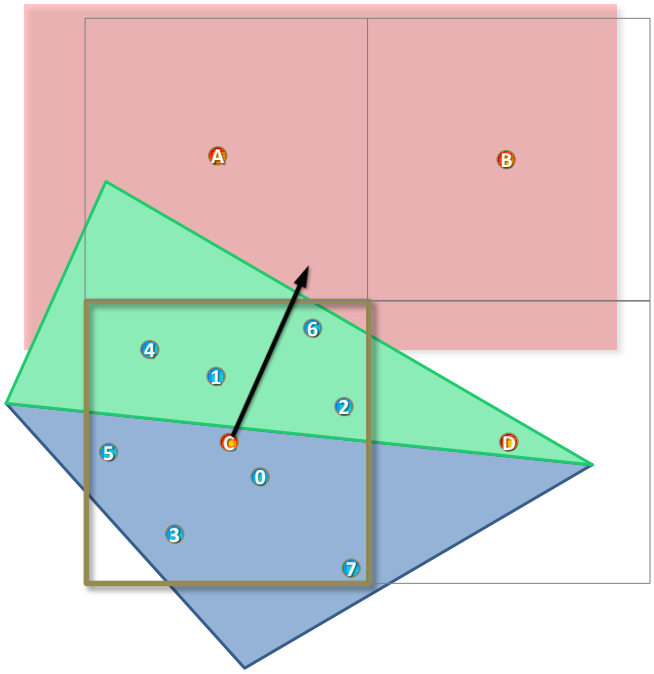

Figure 6. *Why authentication is needed: without it, the color of the chosen pixel C will have significant contributions from pixels A and B.*

It is possible to have a domain-specific training set (for example, for each game), and it is quite likely that it will result in the most accurate solution for a particular domain. We tried to come up with a broader approach that will work in all cases.

We created a training data set with 10^6 cases when one or two randomly generated lines intersected a pixel. These lines split the plane into polygons. Subsamples in the same polygon as the pixel center were assigned the color white, while others were set to black.

For each bit mask (generating subsample clustering), we looked on all cases from the training data that would create such a mask. If a particular mask could be created by a single line crossing the pixel, we used only those cases from the training set; otherwise two lines were used. This was done in spirit of *Ockham's razor,* preferring the simplest possible explanation.

If we use DirectX 8x MSAA subsamples, 40 possible configurations can be explained by just a single line, while 151 can be explained by two lines. For the remaining cases, direction to the most distant (from the center of the pixel) dissimilar subsample was used as the direction of the resampling vector and its length was chosen to approximate the coverage area.

A float array `uva[512]` in Listing 3 contains 256 resampling offsets for 8x MSAA subsamples.

2.4 Authentication of Resampling Offsets

This section describes some RSAA problems and suggests ways to mitigate them. Essentially, RSAA works by blending colors assigned to two clusters. One of these clusters contains a pixel center, and its color is always available. This is not the case for the second cluster, which has to get its color by resampling the pixel neighborhood. This may create problems in certain situations.

In Figure 6, the pixel `C` is intersected by two small triangles, blue and green, which are detected by SADP. The final color of pixel `C` will be computed by bilinearly resampling the image at the position indicated by the black arrow. Accordingly, it will be greatly affected by the colors of pixels `A` and `B`, belonging to the red background object. This is not the desired result, as we would prefer to blend colors `C` and `D`.

This situation could not only occur with small geometric details, but also near sharply curved silhouette edges.

One way to fight this would be to verify that dissimilar subsamples are more comparable with the nearest outside pixel than with the current pixel center, and move them to the similar cluster otherwise. This process would reclassify subsamples `1`, `4`, and `6`, while using only subsample `2` to find the resampling offset. HLSL implementation of this approach (executed at the geometry pre-processing step) is given in Listing 2, starting at line 136.

The disadvantage of this method is that it requires processing of all subsamples in a pixel without first deciding whether we need the resampling offset for the pixel at all.

An alternative way to address this problem is to reduce the length of the resampling vector in problematic situations once we decide that resampling is indeed necessary. To detect such situations in the post-processing step, we first choose one dissimilar subsample, which is the furthest away from the pixel center, as a representative of the dissimilar cluster (it will be subsample `6` in Figure 6). Then, we compare two SADP values. One is the difference between the pixel center and the representative dissimilar subsample. Another is the difference between the pixel center and the outside pixel that is closest to the representative subsample (pixel `A`). This will result in halving the length of the black resampling vector, thus reducing the artifact. Ideally, we would just compare subsample `6` and pixel `A`, but we do not want to keep g-buffer data for all subsamples at the post-processing stage. The rationale for this is that we want the dissimilar subsamples to be closer to the outside pixels than to the similar subsamples. This implementation is given in Listing 3 starting at line 294.

The most accurate approach is to assign a color to the dissimilar cluster by checking the representative subsample against all neighboring pixels and then blending the colors of the two clusters with weights proportional to their coverage areas. With a suitable neighborhood size, this helps fight severe undersampling effects at the price of increased execution time.

In short, we can always be sure of the color of the central subsample and, by proxy, the color of the similar cluster. The accuracy of the color assigned to the dissimilar cluster can be increased with an additional processing.

3 Implementation Details

3.1 Data Layout Optimization

RSAA similarity masks are used to fetch resampling offsets for each pixel (by indexing array `uva[]` in Listing 3). These masks can be stored as *per-pixel* indices during the geometry pre-pass. However, since individual bits are not addressable and locks are too expensive, it is more expedient to output *per-subsample* SADP values during the pre-pass. These values will then be retrieved at the post-processing step and converted to a bona fide binary index for each pixel. Per-subsample SADP values are also required for the post-processing authentication mode (section 2.4). Since SADP values are used only in comparison operations, it is possible to compress them. In the accompanying listings, compression/decompression to and from 8-bit values is executed in lines 174 and 199.

In addition to the compressed SADP values, we use the DXGI_FORMAT_R16G16_UINT buffer to store/retrieve compressed normals at pixel centers (see EncodeNormal/DecodeNormal functions in Listing 1). These surface normals can be supplied by an application, but we opted for a more generic solution, computing the normals in the geometry shader (preGS shader in Listing 2). Typically, normals will be a part of a deferred shading pipeline, so this may be unnecessary. Rough estimation of the accuracy of the normals, compressed to 32 bits, is $2\pi / 2^{16-2} = 3.8 \cdot 10^{-4}$, which is quite sufficient for our needs (this is significantly smaller than threshold = 0.4).

3.2 Geometry Pre-pass Optimization

RSAA requires processing of multiple subsamples per pixel. For fully covered pixels, this is unnecessary (since all SADP values will be zero anyway). DirectX 11 allows the recognition of these situations by looking at the coverage mask (SV_Coverage semantics). This optimization reduces the processing time by about 0.2 ms on the NVIDIA GeForce GTX 580, and we provide such implementation in the preGS shader (line 104). The disadvantage of this method is that it requires a DirectX 11 capable card. It is also possible to optimize this pass in DirectX 9, for example, by comparing primitive ids. This approach will be application-specific, however, particularly if hardware tessellation is used.

Another DirectX 11 specific optimization is to use geometry multisampling. Ordinarily, all values except depth are sampled at pixel centers. Yet, subsample values can be explicitly accessed in pull-mode by using the EvaluateAttributeAtSample function. However, this is not allowed for SV_Position variables. We remedy this by replicating positions (lines 86 and 126). Since we are using geometric normals, pull-mode evaluation is unnecessary for normals (which are computed for each triangle in the preGS shader).

4 Discussion

4.1 Performance *vs* Quality

Performance can be measured precisely and leaves little room for interpretation. This is not the case with image quality. The image processing community has adopted peak-signal-to-noise ratio (PSNR) as an objective gauge, while no generally accepted or widely used metric exists for rendering problems. This is quite understandable, considering the differing goals of these communities: accurate reconstruction of a known image compared with creating a believable illusion.

Even so, PSNR is a useful tool, particularly for comparison of similar methods applied to the same scene [HTG08]. We used it to compare different variants of RSAA. Figure 7 shows performance/quality tradeoffs, considering the following factors:

- 4 or 8 MSAA geometric subsamples per pixel;
- either the depth or SADP test is used for subsample differentiation;
- three authentication modes (no authentication, post-processing authentication, pre-pass authentication) defined by **authentication_mode** in Listing 1.

By choosing a single factor from each row above, we will get 12 possible combinations. Figure 7 shows the obtained quality improvement and the incurred performance penalty for all of them.

SADP, in comparison with the depth-only test, improves quality by up to 3 dB while incurring a penalty of about 0.1 ms on the NVIDIA GeForce GTX 580. The difference between 4 and 8 subsamples is more significant: about 4 dB and a 0.4 ms penalty.

The performance data in this chart excludes the time required for computing and storing per-pixel normals, which we assume could be a part of the deferred shading pipeline anyway. Our implementation of EncodeNormal/DecodeNormal functions in Listing 1 incurs a 0.25 ms penalty.

PSNR measurements are also rather convenient when analyzing the temporal behavior of different methods. Figure 9 shows PSNR values (with respect to 64x supersampling) for two animation sequences. Though absolute PSNR values vary from frame to frame, distance between different antialiasing techniques remains almost constant, with one important exception. When the UNC exploding dragon finally disintegrates into thousands of pieces, RSAA quality goes below 4x MSAA, while still exceeding one for MLAA. MSAA is generally more robust to such calamity by using more color samples for problematic pixels.

PSNR may not say much about what artifacts are actually noticeable. In Figure 1, we used a different approach by counting the number of pixels for which the luminosity error exceeds a given threshold. Though aliasing artifacts affect only a fraction of pixels, they are quite noticeable due to the hyperacuity of human vision.

For more fine-tuned methods to compare images, a calibrated visual metric [MKR*11] or other methods may be considered. We have opted for PSNR, as it is generally well known, and provides important yardsticks for visual quality (45 dB considered to be 'very good').

4.2 Limitations

RSAA's problems are most noticeable when its assumptions are broken, i.e., when there are more than two distinct surfaces overlapping a pixel or when there are not enough valid samples in the neighborhood (see Figure 8). The first situation typically corresponds to the *intersection* of two silhouettes when the background is also visible. This occurs less frequently than typical MLAA artifacts caused by mispredicting a single silhouette line.

It also seems that humans are wired to pay particular attention to silhouettes, while deciphering triple-overlapping regions is more difficult.

There is no cure for the significant undersampling, except more samples at run-time or prefiltering at off-time.

5 Summary

RSAA translates knowledge about the geometry, sampled at a higher rate than the shading parameters, into better images. This knowledge, obtained in the pre-processing step, is converted to the resampling coordinates, which are applied in the post-processing step, leaving the rest of the pipeline intact.

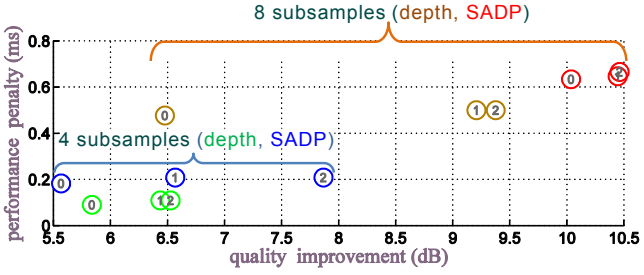

Figure 7. *Performance-quality tradeoffs for the different RSAA variants.* **0**, **1**, and **2** *stand for {no, post, and pre}-processing authentication respectively. The data is gathered on an NVIDIA GeForce GTX 580 at 1280x720 resolution.*

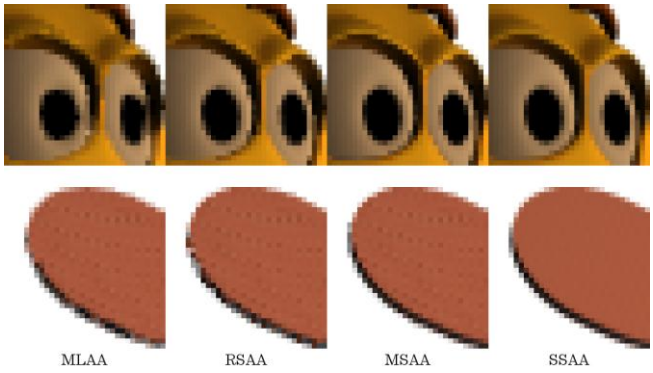

Figure 8. *Aliasing artifacts for the different techniques. MLAA predicts silhouette edges that may be different from the real ones (top of the racket, the right pupil, space between eyes). This results in high-frequency temporal aliasing. RSAA artifacts are mostly restricted to situations in which three or more distinct surfaces overlap a pixel (left edge of the racket) or when there are no valid color samples in the vicinity of a pixel (the space between the eyebrow and the head). SSAA allows effective implicit texture mipmapping (see the center of the racket). For other techniques, this has to be done explicitly.*

It appears that an 8X increase in the geometry sampling rate still allows noticeable improvements in quality. Beyond that, there is a point of diminishing returns, mostly reducing errors, which are already rather small, and not really addressing undersampling artifacts. The intriguing question is whether undersampling can be mitigated by using the third dimension, i.e. computing 3D resampling offsets for mipmaped images. We believe that this is possible, given that the SADP measure would allow the detection of undersampling by measuring the standard deviation (std) among subsamples. This value can be used to choose a proper mipmaping level (0 for small std and then increasing with std).

RSAA allows for about a 10 dB improvement in image quality. To improve it even further, there is always the possibility of increasing the sampling rate for shading parameters and then downsampling for the final image. This is true for most anti-aliasing methods from Table 1 (and was actually done for FXAA and SMAA).

We have provided a set of quality/performance tradeoffs that may be useful in practical applications. Among the contributions that might be interesting for researchers and practitioners are:

- Novel geometric similarity measure (section 2.1).
- Fast adaptive clustering (2.2).
- Computer vision approach to precomputing resampling offsets (2.3).
- Authentication methods (2.4).
- Subsample processing through pull-mode attribute evaluation, consistent with DirectX 11 (section 3).

In a typical scene, geometry and colors are highly correlated. We propose a new way to learn and exploit this correlation by applying computer vision principles.

Acknowledgements

We would like to thank the anonymous reviewers for their detailed and helpful comments and suggestions. We gratefully acknowledge invaluable discussions with and assistance from Bill Mark.

The "Emoticon" model is a part of digital content of the DAZ Studio. The "Exploding Dragon" is courtesy of UNC. The "Edgar" model belongs to digital content of the Poser editor. The Happy Buddha model is courtesy of Brian Curless and Marc Levoy (Stanford University).

References

[Ake93] AKELEY, K. Reality Engine graphics. In *Proc. of SIGGRAPH 1993, ACM Press / ACM SIGGRAPH*, New York, K. Akeley, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 109–116.

[And11] ANDREEV, D. Anti-Aliasing From a Different Perspective. In *Game Developers Conference*, San Francisco, 2011.

[BB00] BENNETT, K., AND BREDENSTEINER, E. Duality and Geometry in SVM Classifiers. In *Proc. of the Seventeenth International Conference on Machine Learning*. Ed. Pat Langley, Morgan Kaufmann, San Francisco, 2000, 57-64.

[BHD10] BIRI, V., HERUBEL, A., AND DEVERLY, S. Practical Morphological Antialiasing on the GPU. In *SIGGRAPH 2010 Talks*, ACM, New York, NY, USA, No. 45.

[Bir11] BIRI, V. Morpological antialiasing and topological reconstruction. In *Proc. of* GRAPP 2011.

[BWG03] BALA, K., WALTER, B., AND GREENBERG, D. Combining Edges and Points for Interactive High-Quality Rendering. In *ACM Trans. Graph. (Proceedings of ACM SIGGRAPH 2003)*, 22(3), 631-640.

[CD05] CHAN, E. AND DURAND, F. Fast Prefiltered Lines. In *GPU Gems 2*. Ed. Matt Pharr. Addison-Wesley, 2005.

[CML11] CHAJDAS, M., MCGUIRE, M., AND LUEBKE, D. Subpixel Reconstruction Antialiasing for Deferred Shading. In *Proc. of 2011 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 15–21.

[DH72] DUDA, R. O. AND HART, P. E. Use of the Hough Transformation to Detect Lines and Curves in Pictures. In *Comm. ACM*, 1972, v. 15, 11–15.

[Fat07] FATTAL, R. Image Upsampling via Imposed Edge Statistics. In *ACM Trans. on Graphics,* 2007, 26(3), Article No.: 95.

[GG12] GJOL, MIKKEL AND GJOL, MARK. Inexpensive Anti-Aliasing of Simple Object. In *GPU Pro 3*, ed. Engel W., A.K. Peters Ltd, 2012, 169–178.

[GPB04] GELDREICH, R., PRITCHARD, M., BROOKS, J. Deferred lighting and shading. In *Game Developers Conference,* 2004.

[HEM*10] HERZOG, R., EISEMANN, E., MYSZKOWSKI, K., AND SEIDEL, H. Spatio-temporal Upsampling on the GPU. In *Proc. of 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 91–98.

[HTG08] HUYNH-THU, Q., GHANBARI, M. Scope of validity of PSNR in image/video quality assessment. In *Electronics Letters*, 2008, 44(13), 800–801.

[IK99] ISSHIKI, T. AND KUNIEDA, H. Efficient Anti-Aliasing Algorithm for Computer Generated Images. In *Proc. of IEEE International Symposium on Circuits and Systems,* 1999, v. 4, 532–535.

[IYP09] IOURCHA, K., YANG, J. C., AND POMIANOWSKI, A. A Directionally Adaptive Edge Anti-Aliasing Filter. In *High Performance Graphics.* 2009, ACM, New York, NY, USA, 127–133.

[JES*12] JIMENEZ, J., B., ECHEVARRIA, J., SOUSA, T., AND GUTIERREZ, D. SMAA: Enhanced Subpixel Morphological Antialiasing. In *Proc. of Eurographics.* 2012, 31(2).

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

[JME*11] JIMENEZ, J., MASIA, B., ECHEVARRIA, J., NAVARRO, F., AND GUTIERREZ, D. Practical Morphological Anti-Aliasing. In *GPU Pro 2*, ed. Engel W., A.K. Peters Ltd, 2011, 95–13.

[Joh12] JOHNSON, M. Implementing a Directionally Adaptive Edge AA Filter using DirectX 11. In *GPU Pro 3*, ed. Engel W., A.K. Peters Ltd, 2012, 275–290.

[Kap10] KAPLANYAN, A. Hybrid Anti-Aliasing. In *SIGGRAPH 2010 talks. CryENGINE3: reaching the speed of light*.

[Kel98] KELLER, A. Quasi-Monte Carlo Methods for Photorealistic Image Synthesis. Ph.D. thesis, Shaker Verlag Aachen, 1998.

[Lel80] LELER, W. J. Human Vision, Anti-aliasing, and the Cheap 4000 Line Display. In *Proc. of SIGGRAPH 1980, ACM Press / ACM SIGGRAPH*, New York, K. Akeley, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 14 (3), 308–313.

[Lot11] LOTTES, T. FXAA. *NVIDIA white paper*, 2011.

[Lot12] LOTTES, T. Unofficial TXAA Info. In *Blogspot blog*, http://timothylottes.blogspot.com/2012/03/unofficial-txaa-info.html, 2012.

[Mac67] MACQUEEN, J. B. Some Methods for Classification and Analysis of Multivariate Observations. In *Proc. of 5-th Berkeley Symposium on Mathematical Statistics and Probability*, Berkeley, University of California Press, 1967, 281-297.

[Mal10] MALAN, H. Edge Anti-Aliasing by Post-Processing. In *GPU Pro*, ed. Engel W., A.K. Peters Ltd, 2010, 265–289.

[MKR*11] MANTIUK, R., KIM, K. J., REMPEL, A. G., AND HEIDRICH, W. HDR-VDP-2: a calibrated visual metric for visibility and quality predictions in all luminance conditions. In *ACM Trans. Graph.*, 2011, 40:1–40:14.

[NSL*07] NEHAB, D., SANDER, P. V., LAWRENCE, J., TATARCHUK, N., ISIDORO, J. R. Accelerating real-time shading with reverse reprojection caching. In *Proc. of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware,* 2007, 25–35.

[Per10] PERTHUIS, C. MLAA in God of War 3. In *Sony Computer Entertainment America, PS3 Devcon*, Santa Clara, 2010.

[Per12] PERSSON, E., Geometric Anti-Aliasing Methods. In *GPU Pro 3*, ed. Engel W., A.K. Peters Ltd, 2012, 71–88.

[Pet10] PETTINEO, M. Deferred MSAA. In *WordPress blog*. http://mynameismjp.wordpress.com/2010/08/16/deferred-msaa, 2010.

[Pho75] PHONG, B.-T. Illumination for Computer Generated Pictures. In *Communications of the ACM,* 1975, vol. 18 (6), 311–317.

[Res09] RESHETOV, A. Morphological antialiasing. In *High Performance Graphics, 2009*, ACM, New York, NY, USA, 109–116.

[SV12] SALVI, M. AND VIDIMCE, K. Surface Based Anti-Aliasing. In Proc. of *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2012, 159–164.

[Uni11] Unity Technologies. Unity Reference Manual. http://unity3d.com/support/documentation/Components, 2011

[YSL08] YANG, L., SANDER, P. V., LAWRENCE, J. Geometry-Aware Framebuffer Level of Detail. In *Comput. Graph. Forum*, 27(4), 1183–1188, 2008

[YNS*09] YANG, L., NEHAB, D., SANDER, P. V., SITTHIAMORN, P., LAWRENCE J., HOPPE H. Amortized supersampling. In *ACM Trans. Graph.* 28 (2009), 135:1–135:12.

[You06] YOUNG, P. Coverage Sampled Antialiasing. In *Tech. rep., NVIDIA*, 2006.

Listing 1. common.hlsl

```
#define authentication_mode 1 // 0 (no), 1/2 for post/pre pass)
#define dbits               8 // 8, 16, or 32 (for differencesMS)

// dbits == 8 requires DXGI_FORMAT_R8_UINT, etc.
// dscale is used to encode/decode sadot() values as integers.
#define dscale float(1 << (dbits - (dbits == 32? 2:1)))

uint2 EncodeNormal(float3 n) {
  // We do not care about normal's orientation,
  // since it will be used as abs(dot(n,r)).
  const float scale = (1<<(16-2)) - 1;
  return uint2(floor(((n.z < 0? -1:1)*n.xy + 2) * scale));
}

float3 DecodeNormal(uint2 c) {
  const float scale = (1<<(16-2)) - 1;
  float3 n;
  n.xy = (c - 2) / scale;
  n.z = sqrt(1 - n.x*n.x - n.y*n.y);
  return n;
}

float3 ScreenToWorld(float3 s) {
  // http://www.gamedev.net/topic/
     506573-reconstructing-position-from-depth-data/

  float x = +((s.x / halfwidth ) - 1);
  float y = -((s.y / halfheight) - 1);

  float4 world_position_4d =
    mul(float4(x, y, s.z, 1), WorldViewProjectionInverse);

  return world_position_4d.xyz / world_position_4d.w;
}

float sadot(float3 r12, float3 n1, float3 n2) {
  return (abs(dot(n1,r12)) + abs(dot(n2,r12))) * rcp(length(r12));
}
```

Listing 2. prepassShaders.hlsl

```

struct VSInput {
  float4 PositionOS : POSITION;
  // ...
};

struct preVSOutput {
  float4 PositionCS : SV_Position;
  float3 PositionWS : POSITIONWS;
};

struct prePSInput {
  float4 PositionSS : SV_Position;
  float3 PositionWS : POSITIONWS;
  float3 NormalWS   : NORMALWS;
};

struct prePSOutput {
  uint SampleDifference  : SV_Target0;
};

preVSOutput preVS(in VSInput input) {
  preVSOutput output;
    // The world-space position
    output.PositionWS = mul(input.PositionOS, World).xyz;
    // The clip-space position
    output.PositionCS =
      mul(input.PositionOS, WorldViewProjection);
    return output;
}

[maxvertexcount(3)]
void preGS(triangle preVSOutput input[3],
           inout TriangleStream<prePSInput> TriStream,
           uint id : SV_PrimitiveID) {

    prePSInput output;
```

```

    // Compute triangle's geometric normal
    float3 n3 = normalize(
      cross(input[2].PositionWS - input[0].PositionWS,
            input[0].PositionWS - input[1].PositionWS));

    [unroll] for (int i = 0; i < 3; i++) {
      // The pull-model evaluation of SV_Position
      // (using EvaluateAttributeAtSample) is not allowed,
      // so we circumvent it by replicating the data.
      output.PositionSS = input[i].PositionCS;
      output.PositionWS = input[i].PositionWS;
      output.NormalWS   = n3;
      TriStream.Append(output);
    }

    TriStream.RestartStrip();
}

Texture2D<uint2> normal_data : register(t0);
// depth_data is used only for authentication_mode == 2
Texture2D<float> depth_data  : register(t1);

prePSOutput prePS(in prePSInput input,
                  in uint si: SV_SampleIndex,
                  in uint coverage: SV_Coverage) {
  prePSOutput ret;

  // A shortcut for fully covered pixels.
  if (coverage == (1 << 8) - 1) {
    ret.SampleDifference = 1; // decoded to 0
    return ret;
  }

  float2  positionSS = input.PositionSS.xy;
  uint2 sampleIndex2 = uint2(positionSS);
  uint3 sampleIndex3 = uint3(sampleIndex2, 0);

  uint2 nci = normal_data.Load(sampleIndex3);

  // A shortcut for background subsamples. Those subsamples
  // could only be compared with foreground subsamples
  // (since the relevant shaders are called only for
  // the foreground geometry).
  // In this case, we assume that they are always "different".
  if (nci.x == 0) {
    ret.SampleDifference = uint(dscale) + 1;
    return ret;
  }

  // subsample's position and normal
  float3 ps = EvaluateAttributeAtSample(input.PositionWS, si);
  float3 ns = input.NormalWS;
  // center's position and normal
  float3 pc = input.PositionWS;
  float3 nc = DecodeNormal(nci);
  // the difference
  float  sc = sadot(ps - pc, ns, nc);

  const float threshold = 0.2f; // half

  #if authentication_mode == 2

  if (sc > threshold) {
    static const float2 sampleOffset[] = {
     float2( 0, -1), // _________________
     float2( 0,  1), // |               7 |
     float2( 1,  0), // |      3          |
     float2(-1, -1), // |         0       |
     float2(-1,  1), // |  5     .    2   |
     float2(-1,  0), // |       1         | 0------> x-axis
     float2( 0,  1), // |    4            | |
     float2( 1, -1)  // |           6     | |
    };               // |_________________| V y-axis

    // Consider the pixel in the neighborhood
    // which is the closest to the current subsample.
    sampleIndex2 += sampleOffset[si];
    uint2 noi = normal_data.Load(int3(sampleIndex2, 0));
    if (noi.x) { // is the candidate pixel in foreground?
      float  zo = depth_data.Load(int3(sampleIndex2, 0));
      float3 no = DecodeNormal(noi);
      float3 po = ScreenToWorld(
        float3(positionSS + sampleOffset[si], zo));
      float  so = sadot(ps - po, ns, no);
      if (so > sc) {
        // There are bigger differences between the candidate
        // pixel and the current subsample than between the
        // current subsample and the pixel center,
        // so we will ignore this subsample completely.
        sc = 0;
      }
    }
  }
  #endif

  // For foreground subsamples, it will be always >= 1.
  // For background subsamples, ret.SampleDifference == 0
  // (since RT is cleared every frame).
  ret.SampleDifference = uint(dscale * sc) + 1;
  return ret;
}
```

Listing 3. postprocessingShader.hlsl

```
Texture2D<float4>    color_data : register(t0);
Texture2D<uint2>    normal_data : register(t1);
Texture2DMS<uint> differencesMS : register(t2);
// depth_data is used only for authentication_mode == 1
Texture2D<float>     depth_data : register(t3);

float4 postPS(in PSInput input) : SV_Target {

  float2  positionSS = input.PositionSS.xy;
  uint2 sampleIndex2 = uint2(positionSS);
  uint3 sampleIndex3 = uint3(sampleIndex2, 0);

 // Encoded normal @ pixel's center.
  uint2 nci = normal_data.Load(sampleIndex3);

  float maxsim = 0; // maximum   similarity
  float d[8];       // subsample similarity

  [unroll] for (uint i = 0; i < 8; i++) {
    uint sci = differencesMS.Load(sampleIndex2, i);
    // fg - bg tramps all other differences.
    // sc = sample in fg? decode it : big_number;
    float sc = sci? (sci - 1) / dscale : nci.x;
    d[i] = sc;
    maxsim = max(maxsim, sc);
  }

  const float threshold = 0.2f; // half

  float4 color;

  static const float uva[] = {
     0.889, -0.889,-0.799,  0.000, 0.839,  0.176,-0.755, -0.250,
     0.084,  0.778, 0.070,  0.668, 0.044,  0.661, 0.145, -0.536,
    -0.803, -0.066,-0.735, -0.096, 0.683,  0.107, 0.562,  0.164,
     0.025,  0.703,-0.509, -0.058,-0.552, -0.011,-0.356, -0.042,
     0.682,  0.283,-0.757,  0.025, 0.707,  0.204, 0.587,  0.175,
     0.667, -0.667, 0.556, -0.556, 0.073, -0.571, 0.051,  0.430,
    -0.621,  0.023,-0.019,  0.616, 0.589,  0.088, 0.468,  0.144,
     0.556, -0.556, 0.444, -0.444, 0.444, -0.444, 0.333, -0.333,
    -0.057, -0.792,-0.098,  0.694, 0.426, -0.426,-0.045,  0.565,
     0.065, -0.729,-0.033, -0.609, 0.065, -0.591, 0.090, -0.477,
     0.636, -0.140,-0.114,  0.561, 0.538,  0.144, 0.389,  0.199,
    -0.017, -0.658, 0.033,  0.453, 0.312, -0.312, 0.041,  0.315,
     0.700,  0.000, 0.589, -0.244,-0.074, -0.605, 0.506,  0.107,
     0.556, -0.556, 0.444, -0.444, 0.067, -0.480, 0.049,  0.367,
     0.369,  0.230, 0.357,  0.237, 0.447,  0.120, 0.332,  0.140,
     0.000,  0.318, 0.000,  0.314,-0.071,  0.438,-0.103,  0.247,
    -0.767,  0.120, 0.706, -0.176, 0.737,  0.081,-0.606, -0.230,
    -0.598,  0.203,-0.537, -0.095,-0.296, -0.347,-0.435, -0.100,
     0.101,  0.761,-0.631, -0.127, 0.617,  0.099, 0.484, -0.090,
    -0.620, -0.128,-0.455, -0.065,-0.511,  0.051,-0.319, -0.042,
     0.667, -0.667, 0.556, -0.556, 0.634,  0.136,-0.540,  0.074,
     0.556, -0.556, 0.444, -0.444,-0.148, -0.376,-0.139, -0.214,
     0.556, -0.556, 0.444, -0.444, 0.534,  0.148, 0.376, -0.061,
     0.444, -0.444, 0.333, -0.333,-0.415,  0.068,-0.210, -0.029,
     0.667, -0.667, 0.556, -0.556, 0.564,  0.031, 0.444, -0.444,
     0.556, -0.556, 0.444, -0.444, 0.039, -0.484, 0.039, -0.371,
     0.556, -0.556, 0.386, -0.909, 0.446,  0.145, 0.371, -0.074,
     0.444, -0.444,-0.633,  0.377, 0.236, -0.236,-0.074,  0.170,
     0.556, -0.556, 0.444, -0.444, 0.241, -0.283, 0.432,  0.097,
     0.444, -0.444, 0.333, -0.333, 0.088, -0.339, 0.051, -0.246,
     0.395, -0.896, 0.368, -0.936, 0.284, -0.171, 0.243, -0.087,
     0.488, -0.488, 0.474, -0.474, 0.159, -0.174, 0.068, -0.073,
    -0.720,  0.416, 0.145,  0.699, 0.676,  0.263,-0.616, -0.210,
     0.042,  0.688, 0.122,  0.604, 0.038,  0.638, 0.118,  0.505,
     0.286,  0.667, 0.152,  0.518, 0.238,  0.556,-0.494, -0.075,
     0.238,  0.556,-0.227,  0.255, 0.190,  0.444, 0.094,  0.320,
     0.684,  0.684, 0.730, -0.015,-0.759,  0.030,-0.500,  0.085,
     0.238,  0.556,-0.392, -0.928, 0.081,  0.596,-0.110,  0.323,
```

```
     0.238,  0.556, 0.366,  0.366, 0.190,  0.444,-0.347,  0.081,
     0.190,  0.444, 0.646,  0.452, 0.143,  0.333, 0.044,  0.243,
     0.286,  0.667, 0.042, -0.612, 0.238,  0.556, 0.447, -0.070,
     0.238,  0.556, 0.039, -0.531, 0.190,  0.444, 0.098, -0.400,
     0.238,  0.556, 0.169,  0.369, 0.190,  0.444, 0.190,  0.206,
     0.190,  0.444,-0.066,  0.343, 0.143,  0.333, 0.078,  0.229,
     0.238,  0.556, 0.623, -0.046, 0.190,  0.444, 0.328,  0.151,
     0.190,  0.444, 0.000,  0.266, 0.143,  0.333, 0.047, -0.280,
     0.039,  0.404, 0.340,  0.200, 0.259,  0.349, 0.162,  0.140,
     0.119,  0.270, 0.437,  0.267, 0.240,  0.307, 0.050,  0.112,
    -0.000, -0.682,-0.640, -0.111,-0.000, -0.679, 0.540, -0.177,
    -0.320, -0.283,-0.441, -0.129,-0.319, -0.281,-0.344, -0.158,
    -0.556, -0.079,-0.565, -0.144,-0.444, -0.063, 0.405,  0.034,
    -0.347,  0.960,-0.307,  0.118,-0.313,  1.000,-0.212,  0.055,
    -0.556, -0.079,-0.444, -0.063, 0.232, -0.560,-0.454,  0.072,
    -0.034, -0.385,-0.311, -0.366,-0.263, -0.307,-0.130, -0.151,
    -0.444, -0.063,-0.252,  0.000,-0.333, -0.048,-0.312,  0.042,
    -0.333, -0.048,-0.327, -0.481,-0.331,  0.260,-0.122, -0.020,
    -0.556,  0.556,-0.444,  0.444,-0.444,  0.444,-0.333,  0.333,
    -0.000, -0.372,-0.000, -0.369,-0.000, -0.371, 0.105, -0.290,
    -0.444,  0.444,-0.544,  0.604,-0.333,  0.333, 0.271,  0.047,
    -0.518,  0.518,-0.280,  0.289,-0.505,  0.505,-0.062,  0.077,
    -0.267, -0.444,-0.200, -0.333,-0.674, -0.448, 0.189, -0.189,
    -0.141, -0.314,-0.338, -0.380,-0.286, -0.291,-0.028, -0.146,
     0.333,  0.067, 0.314, -0.150, 0.339,  0.445, 0.192,  0.290,
     0.000,  0.162,-0.170,  0.150, 0.177, -0.169, 0.000,  0.000
  };
  static const float2 uvo[] = uva;

  if (maxsim/2 > threshold) {
    maxsim = maxsim/2 + threshold;
    uint m = 1;     // bit index
    uint mask = 0; // bits: 0 for different, 1 for the same
    uint fs;       // 'furthest sample'

    [unroll] for (uint i = 0; i < 8; i++, m <<= 1) {
      if (d[i] <= maxsim) {
        mask |= m; // similar
      } else {
        fs = i;    // different
      }
    }
```

```
    positionSS += uvo[mask]; // add precomputed offset
    positionSS /= InputSize; // convert to [0,1]
    color = color_data.Sample(LinearSampler, positionSS);

    #if authentication_mode == 1
    static const float2 sampleOffset[] = {
     float2( 0, -1), // _________________
     float2( 0,  1), // |               7 |
     float2( 1,  0), // |      3          |
     float2(-1, -1), // |         0       |
     float2(-1,  1), // |  5     .    2   |
     float2(-1,  0), // |       1         | 0------> x-axis
     float2( 0,  1), // |    4            | |
     float2( 1, -1)  // |            6    | |
    };               // |_________________| V y-axis
    // Consider the pixel in the neighborhood
    // which is the closest to the different subsample fs.
    sampleIndex2 += sampleOffset[fs];
    uint2 nco = normal_data.Load(int3(sampleIndex2, 0));
    if (nci.x && nco.x) { // see if both normals are valid
      // The current pixel: depth, normal, position
      float  zc = depth_data.Load(sampleIndex3);
      float3 nc = DecodeNormal(nci);
      float3 pc = ScreenToWorld(float3(positionSS, zc));
      // The closest neighborhood pixel (used for blending).
      float  zo = depth_data.Load(int3(sampleIndex2, 0));
      float3 no = DecodeNormal(nco);
      float3 po =
        ScreenToWorld(float3(positionSS + sampleOffset[fs], zo));
      float  co = sadot(pc - po, nc, no);
      if (co - d[fs] > d[fs]) { // so ~ co - sc
        // Use 1/2 resampling offset.
        positionSS  = input.PositionSS.xy + 0.5*uvo[mask];
        positionSS /= InputSize;
        color = color_data.Sample(LinearSampler, positionSS);
      }
    }
    #endif

  } else {
    color = color_data.Load(sampleIndex3); // no changes
  }
  return color;
}
```

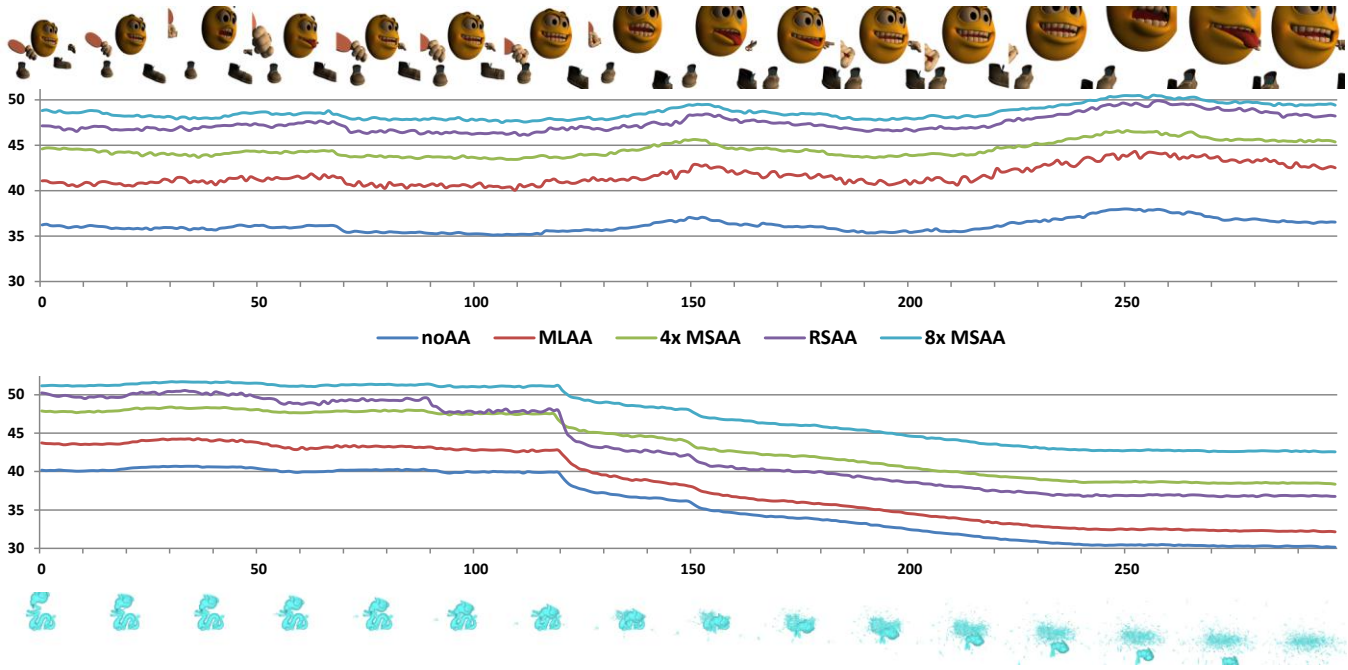

Figure 9. *Comparison of the five different antialiasing techniques using two animations: emoticon model (top) and the UNC exploding dragon (bottom).The peak signal-to-noise ratio is measured against 64x supersampling (the bigger the better). Note that RSAA goes below 4x MSAA when the dragon starts disintegrating, but it is still better than MLAA. The RSAA version with 8 geometric subsamples and post-processing authentication is used.*