Comparison of Anti-Aliasing Techniques for Real-Time Applications

F. Liberatore, J. Longazo, S. Pettinati, D. Weise
Department of Computer Science
USC Viterbi School of Engineering
Los Angeles, CA
{fliberat, longazo, pettinat, dweise}@usc.edu

Abstract

In this project, we compare three classes of anti-aliasing techniques for real-time applications: Screen Anti-Aliasing (FSAA), Image Post-Processing Anti-Aliasing (IAA), and Geometric Anti-Aliasing (GAA). For FSAA, we extend our previous work on Super-Sampling Anti-Aliasing (SSAA) to consider different combinations of sub-sample distributions and sizes. For IAA, we implement Morphological Anti-Aliasing (MLAA), while for GAA we implement Geometric Post-Processing Anti-Aliasing (GPAA). In addition, we will create a secondary application that will allow us to load and render custom models. The study focuses on analyzing attributes such as efficiency (e.g., memory, time), image quality, and any particular weaknesses the AA implementations have.

1 Introduction

There is a great number of anti-aliasing (AA) implementations/techniques in the literature, and they each have their own strengths and weaknesses inherent in their method or requirements. We studied a variety of AA implementations to look at their efficiency as well as their ability to reduce the appearance of artifacts. The following anti-aliasing techniques are covered:

1. Full Screen Anti-Aliasing (FSAA): we extend Super-Sampling Anti-Aliasing (SSAA) by considering different combinations of sub-sample distributions and sizes [1, 4].

2. Image Post-Processing Anti-Aliasing (IAA): we implement Morphological Anti-Aliasing (MLAA) [3, 6].

3. Geometric Anti-Aliasing (GAA): we implement Geometric Post-Processing Anti-Aliasing (GPAA) [2, 5].

These AA techniques were added to our previous rendering work, which read in a teapot and small plane from a text file and rendered them with a jittered sub-sampling version of SSAA. Our renderer used a scan-line rasterization algorithm to render a stream of triangles into a Z-buffer, whose results were placed into a PPM file.

In addition, we created a secondary application that allows us to load and render custom models. The models are exported from Maya in the FBX format and an FBX parser creates files for vertex positions, vertex normals and vertex texture coordinate (UV) values. Our secondary application reads in these files and produces a single file readable by our application.

The study focuses on analyzing attributes such as efficiency (e.g., memory, time), image quality, and any particular weaknesses the AA implementations have. Our goal is to present a comprehensive look at the strengths and weaknesses of each form of anti-aliasing listed above. Specifically, we discuss how much additional time it takes to render an image for each technique. We also display comparisons of rendered images to discuss final image quality and point out scenarios when particular methods may fail.

The remainder of the paper is organized as follows:

- Three subsections present the implementation of the anti-aliasing techniques addressed in the project.

- Then, the techniques are tested to ascertain their performances in terms of memory, computational time, and quality.

- Finally, conclusions are given in the last section.

2 Super-Sampling Anti-Aliasing (SSAA)

2.1 Concept

FSAA methods consist of sampling the whole scene at a higher frequency than what is needed for display. More specifically, in this project we focus on SSAA, a class of FSAA, where the number of input pixels is increased with respect to the number of output pixels and there is a sample for each input pixel. This means that all the attributes are evaluated and stored for each sample (e.g., depth, color, normal and texture coordinates). This approach gives the best image quality, at the cost of fully computing these attributes per sample.

2.2 Methodology Implementation

Depending on the parameters, different versions of SSAA are possible. The parameters considered in this work are illustrated in the following.

2.2.1 Subsample Pattern

This attribute concerns the position of the subsamples (offset) relative to that of the pixel.

- Ordered Grid Super Sampling (OGSS): In this technique, the subsamples are taken from a regular grid subpixel distribution. OGSS has problems solving the most visible aliasing cases: nearly-vertical and nearly-horizontal edges.

- Rotated Grid Super Sampling (RGSS): This technique consists in rotating the grid of subsamples inside each pixel to overcome the disadvantage of the OGSS. However, regular patterns that can be easily perceived by the human eye can still be obtained.

- Random Sampling: Subsamples are located at random locations around the pixel to avoid the appearance of regular patterns, a particularly distracting form of aliasing.

2.3 Filter Type

The filter type defines the weights by which the subsamples are scaled.

- Box Filter. This filter assigns the same weight to all the subsamples.

- Pyramid (triangle) Filter. The weights are assigned according to the following equation:

$$p(x,y) = h\left(1 - \frac{2\max(|x|,|y|)}{l}\right) \qquad (1)$$

  where (x,y) is the offset of the subsample with respect to the pixel, h is the height of the pyramid, and l is the length of the side of the pyramid. The weights of the subsamples need to be normalized after they have been computed. In this work, h is set to one and l is equal to the extent (see Section 2.4).

- Gaussian Filter. The weights are assigned according to the following equation:

$$g(x,y) = A\exp\left(-\left(\frac{x^2}{2\sigma_x^2} - \frac{y^2}{2\sigma_y^2}\right)\right) \qquad (2)$$

  where A is the amplitude, while σ_x^2 and σ_y^2 are the variances associated to the x and the y axis, respectively. Since the weights are normalized after their computation, the term A can be removed from the equation and, therefore, does not need to be defined. The variances σ_x^2 and σ_y^2 have been set to $\left(\frac{e}{6}\right)^2$, where e is the extent (see Section 2.4) so that nearly all the area underneath the Gaussian lies within the extent.

2.4 Other Parameters

- Number of subsamples per pixel (n). Suggested values are in the range $5 - 15$.

- Extent of the subsamples (e). Suggested values are in the range $1.0 - 1.7$.

3 Morphological Anti-Aliasing (MLAA)

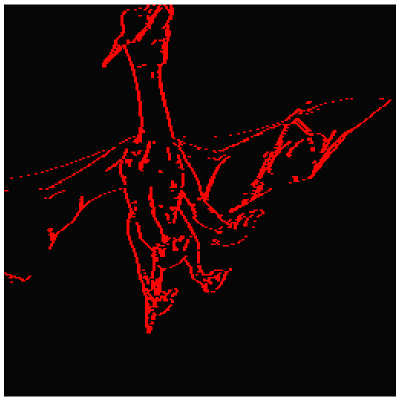
3.1 Concept

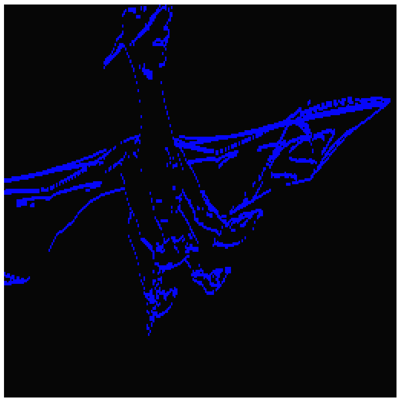
In Image Post-Processing Anti-Aliasing (IAA), anti-aliasing is performed after the initial image has been processed. The goal of IAA is to identify the aliased pixels in the image and smooth these pixels by blending them with their neighbors. We implemented Morphological Anti-Aliasing (MLAA), which first finds discontinuities between pixels by comparing each pixel's intensity to its neighbors, keeping track of the vertical and horizontal edges. It then identifies the shape of each edge and blends pixel color (of the pixels near the edge) accordingly. One advantage of MLAA is that it can quickly produce good results for nearly-vertical/horizontal edges. One of its weaknesses is that it can blur border pixels even when there is no aliasing.

3.2 Methodology Implementation

The first step we took in implementing MLAA was to identify the edges. We were able to accomplish it by looping through each pixel and comparing it to its neighbors. If the difference in color value exceeded the threshold, the horizontal flag (if one pixel on top of another) or vertical flag (if pixels beside each other) was set for that pixel in the corresponding array (see Figures 1a, 1b, and 1c).

The next step is to identify the shape of the edges (Z-shaped, L-shaped, or U-shaped), we identify these


(a) MLAA: Vertical detected edges.


(b) MLAA: Horizontal detected edges.

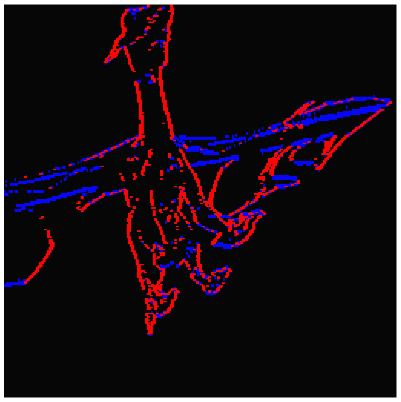

(c) MLAA: Overlapping vertical and horizontal edges.

Figure 1: MLAA: Detected edges.

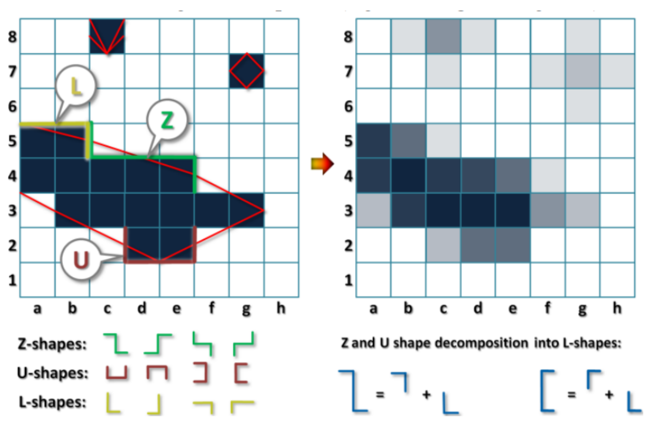


Figure 2: MLAA: Possible edge shapes.

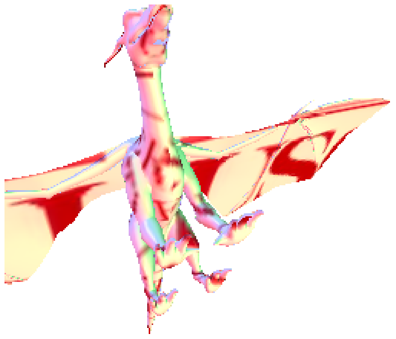


Figure 4: MLAA: Final anti-aliased image.

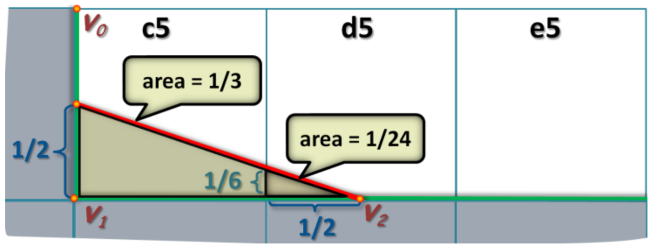
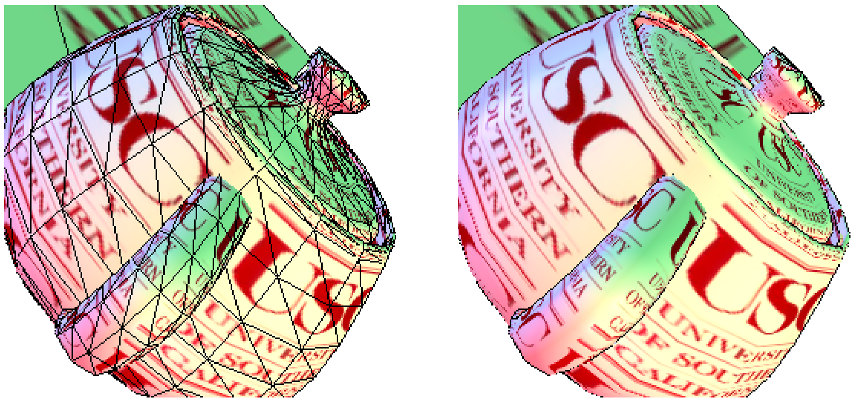


Figure 3: MLAA: Finding the area of the pixel covered by the triangle to blend.

by doing two passes. The first pass follows each horizontal line, constructed from the array of horizontal line flags previously discussed, and looks for vertical lines on either end of the line. If there is a vertical line in different directions at the start and end of the line, the current line is Z-shaped. If there are vertical lines in the same direction, it is U-shaped. If there is only one vertical line at the start or end then it is L-shaped. A similar pass is then done for the vertical lines. The Z and U-shaped edges are then broken down into L-shapes (Figure 2). Once we only have L-shapes, blending can then be performed using the midpoints of the vertical and horizontal lines and forming a triangle. The triangle area is used to weight the blending with the neighboring pixels, as demonstrated in Figure 3. The resulting image is shown in Figure 4.



(a) GPAA: Edges for all Rendered Triangles.

(b) GPAA: Silhouette edges, shared edges removed.

Figure 5: GPAA: Edge identification.

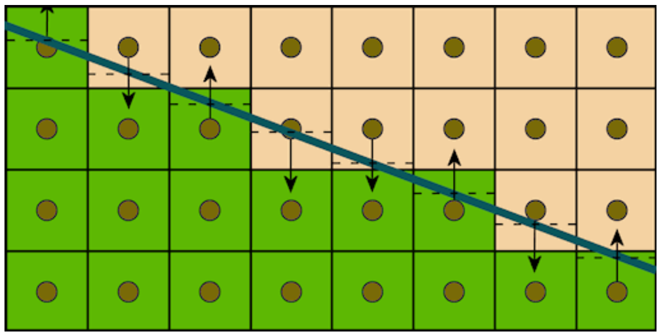


Figure 6: GPAA: Pixel blending/weighting depending on proximity to geometric edge.

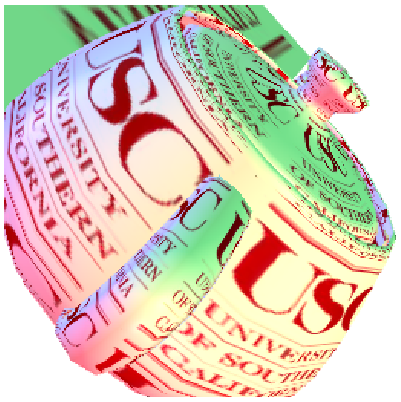


Figure 7: GPAA: Final anti-aliased image.

4 Geometric Post-Process Anti-Aliasing (GPAA)

4.1 Concept

Geometric Anti-Aliasing seeks to solve the problem of aliasing caused by geometry, as some of the most dramatic aliasing artifacts are caused by geometric edges. We focus on the technique of GPAA to demonstrate the merits of geometric anti-aliasing. GPAA works by taking the geometric silhouettes of the various objects in a scene and blending the pixels that make up the silhouette edges to produce a softer edge. This technique is implemented in the post processing stage and requires very little memory and computation in comparison with super-sampling methods.

4.2 Methodology Implementation

The implementation of GPAA in our renderer had two main steps: extracting silhouette edges from the triangle rasterization stage, and blending the pixels in the final frame buffer that make up these edges to remove aliasing artifacts.

To generate and save silhouette edges, several new structures and steps had to be added to our rasterizer. First, new structures were created to house crucial information about edges, namely their start point, end point, x/y slope, and their parent triangle's ID. Edges are saved in a Generic_Edge structure, and each triangle (3 Generic_Edges) is saved in a Triangle structure along with the triangle's ID and a Boolean for it's rendered status. Every time a triangle is rasterized, a new Triangle object and its edges is generated and saved into an array. The triangle ID serves as both the index of the Triangle in the triangle array and as a way to identify which triangle a pixel belongs to in the frame buffer. The existing pixel structure was modified to contain a triangle ID as to ensure only the IDs of visible triangles will exist in the final frame buffer. After all triangles are rasterized, the application steps through every pixel in the frame buffer, setting the rendered Boolean to true for every new triangle ID found and adding that triangles Generic_Edge to a hash map of Booleans (see Figure 5a). Since a silhouette edge is an edge that is unique to one triangle, every time an edge that already exists is added to the hash map, the Boolean in that position is set to false (every edge was set to true when first added to the hash map). After this process is complete, the hash map contains only the edges from triangles that are visible in the frame buffer, where each edge is not shared with any other triangle, producing the silhouette edges for every object in the scene (see Figure 5b).

Now that a list of silhouette edges can be produced, the next step involves taking the pixels on either side of each edge and blending them to create a smoother edge. Each edge is divided into one of two categories, horizontal (x/y slopes between 0 and 1) and vertical (x/y slopes between 1 and infinity). For horizontal edges, we begin at the edges left side (ceiling of start X value) and increment X at each step, checking the distance of the pixels below and above the edge to determine which pixel to blend and by how much (see Figure 6). Whichever pixels center is closest to the edge, that pixels color is blended with its neighboring pixel. The same process is used for vertical edges, swapping the Xs and Ys. The final product is an image where geometric edges are anti-aliased (see Figure 7).

5 Technical Analysis

The algorithms have been written in C and compiled on Visual Studio 2010 Professional, using the "Release" configuration. The experiments have been run on a PC equippped with an Intel Core i7-4500U CPU, using only one core per run. The algorithms have been tested on two separated scenes: (I) a scene presenting a 3D model of the Newell teapot (898 triangles) and a plane (2 triangles), and (II) a scene showing the 3D model of a dragon (11,692 triangles; seen in Figure 4).

5.1 Memory Analysis

Tables 1 and 2 show the memory occupation for each algorithm to render scene I and II, respectively. In the tables, the first column lists the technique; the number after SSAA indicates the number of subsamples computed. The second and third columns illustrate the memory occupation expressed in KB and as percentage of the "no anti-aliasing" (No AA) case, respectively. From the tables the following conclusions

AA Technique	Memory (KB)	Memory (%)
No AA	1,992	100%
SSAA 2	3,552	178.31%
SSAA 4	5,120	257.03%
SSAA 9	9,052	454.42%
MLAA	3,284	164.86%
GPAA	2,416	121.29%

Table 1: Memory occupation for scene I.

AA Technique	Memory (KB)	Memory (%)
No AA	1,992	100%
SSAA 2	3,552	178.31%
SSAA 4	5,116	256.83%
SSAA 9	9,052	454.42%
MLAA	3,284	164.86%
GPAA	3,788	190.16%

Table 2: Memory occupation for scene II.

can be drawn:

- The memory required by SSAA grows linearly with respect to the number of subsample per pixel. However, it is not affected by the number of triangles to be rendered.
- MLAA memory occupation does not depend on the number of triangles in the scene.
- The memory required by GPAA is affected by the number of triangles in the scene.

5.2 Computational Time Analysis

Tables 3 and 4 display the computational time required by each algorithm to render scenes I and II, respectively. To reduce the impact of noise, each renderer has been run 100 times. In the tables, the first column corresponds to the technique used. The second column shows the average computational time and the corresponding standard deviation. Finally, the third column illustrates the average CPU time, expressed as a percentage of the "No AA" case. By observing and comparing the values in the tables we can draw the following conclusions:

- The computational time for SSAA increases linearly with respect to the number of subsamples, although the slope is slower for scene II, that has a much higher number of triangles. Interestingly, the computational time for SSAA 9 to render scene II is much lower than that required to render scene I. Also, SSAA is the only technique that

AA Technique	CPU Time (s.)	CPU Time (%)
No AA	0.07609 ± 0.00758	100%
SSAA 2	0.13328 ± 0.00844	175.15%
SSAA 4	0.24391 ± 0.01015	320.53%
SSAA 9	0.52422 ± 0.01244	688.91%
MLAA	0.07938 ± 0.00654	107.17%
GPAA	0.07906 ± 0.00536	103.90%

Table 3: CPU time for scene I.

AA Technique	CPU Time (s.)	CPU Time (%)
No AA	0.16902 ± 0.00952	100%
SSAA 2	0.20047 ± 0.015555	118.58%
SSAA 4	0.25531 ± 0.02014	151.02%
SSAA 9	0.38343 ± 0.02933	226.80%
MLAA	0.21531 ± 0.02006	127.36%
GPAA	0.36563 ± 0.03226	216.27%

Table 4: CPU time for scene II.

has better percentages of the CPU time in scene II than in scene I.

- The performance of MLAA is slightly worse in scene II than scene I.

- GPAA has the best performance in scene I, though it is the second worst in scene II. This suggests that GPAA performances is strongly affected by the number of triangles in the scene.

To ensure that the differences reported by the average values are really significant, we performed statistical analyses. The Shapiro-Wilk test shows that the observations do not follow a normal distribution. Therefore, we use the Kruskal-Wallis rank sum test, obtaining a p-value lower than 2.2E-16, indicating that at least one of the groups is different from at least one of the others. To determine which groups are different with pairwise comparison we use a multiple comparison test after Kruskal-Wallis. The results are shown in Tables 5 and 6, corresponding to the data relative to scene I and II, respectively. From the tables we can infer that, for scene I, GPAA and MLAA have a computational time that is statistically identical to not applying any anti-aliasing technique. On the other hand, for scene II only MLAA and SSAA 2 have statistically similar performances. In conclusion, we can state that the best techniques in terms of computational performance are GPAA and MLAA for scene I, and MLAA and SSAA 2 for scene II.

	No AA	SSAA 2	SSAA 4	SSAA 9	MLAA
GPAA	False	True	True	True	False
MLAA	False	True	True	True	
SSAA 9	True	True	True		
SSAA 4	True	True			
SSAA 2	True				

Table 5: Results of a multiple comparison test after Kruskal-Wallis on the CPU time data for scene I. A “true” indicates significant statistical difference between the two groups.

	No AA	SSAA 2	SSAA 4	SSAA 9	MLAA
GPAA	True	True	True	True	False
MLAA	True	False	True	True	
SSAA 9	True	True	True		
SSAA 4	True	True			
SSAA 2	True				

Table 6: Results of a multiple comparison test after Kruskal-Wallis on the CPU time data for scene II. A “true” indicates significant statistical difference between the two groups.

5.3 Picture Quality Evaluation

Figure 8 provides a visual comparison of the top-right corner of scene I, rendered using all the techniques. For the SSAA, the following parameters are used:

- Filter type: Gaussian. This filter has been chosen over the box and the pyramid as it possesses the best theoretical properties.

- Subsample pattern: We test with random, ordered grid, and 20-degree rotated grid subsamples.

- Number of subsamples: 2. The higher the number of subsamples, the better the final quality of the picture. For the comparison to be fair, we chose to test the case with 2 subsamples per pixel as the computational time for this case is the closest to MLAA and GPAA when rendering scene I.

- Extent: 1.0.

The analysis of all the possible combinations of parameters for SSAA is beyond the scope of this paper. By observing the figures the following conclusions can be drawn:

- When using MLAA (Fig. 8c), a certain degree of anti-aliasing can be observed, although we can still perceive artifacts due to regular patterns (edge of the plane behind the teapot). However, we can also observe that the texture has been deformed (the “C” in the USC logo presents a gap in the middle).

- GPAA produces good results (Fig. 8e). In fact, the staircase artifact in the edge of the plane behind the teapot is greatly reduced and it is almost invisible when observed with the naked eye.

- No anti-aliasing can be observed in Figure 8b. This is due to the fact that the efficacy of SSAA strongly depends on the position on the subsamples. Using a completely random distribution of the subsamples might defeat the purpose of anti-aliasing when the subsamples are located too close to the pixel.

- In the picture generated combining SSAA and OGSS (Fig. 8d) there is an observable anti-aliasing effect. However, we can still perceive a staircase in the edge of the plane behind the teapot.

- The best results for SSAA are obtained when using RGSS (Fig. 8f). In fact, the staircase artifact in the edge of the plane behind the teapot can be observed only by zooming the picture.

5.4 Final Observations

Overall, MLAA has good performances and low memory requirements. However, the anti-aliasing effect is very limited and deformation of the textures can be observed. According to our experiments, GPAA is the best technique to render scenes with a low number of triangles, as it has a low memory occupation, very good performances in terms of computational time (statistically identical to not applying any anti-aliasing technique), and effectively reduces the aliasing in the picture. However, the performance of GPAA greatly worsens when rendering scenes with a high number of triangles. SSAA is the technique having the greatest memory requirements and its computational performance depends on the number of subsamples. However, we observed that SSAA is relatively better than the other techniques when rendering scenes with a high number of triangles. Also, the best results in terms of anti-aliasing have been obtained when using RGSS.

6 Conclusions

In this work we implemented and compared three anti-aliasing techniques for real-time applications: SSAA, MLAA, and GPAA. The memory and computational performance of the techniques have been tested on two scenes having vastly different numbers of triangles. Also, a visual comparison of the anti-aliasing effect of each technique has been given. The experiments show that GPAA is the best technique to render scenes with a low number of triangles, while SSAA coupled with RGSS provides the best results when rendering scenes with many triangles. Finally, although MLAA has good performances, it has limited anti-aliasing capabilities and can alter the textures on the models.

References

[1] K. Beets,"Super-sampling anti-aliasing analyzed," Beyond 3D, Technical Report, 2000.

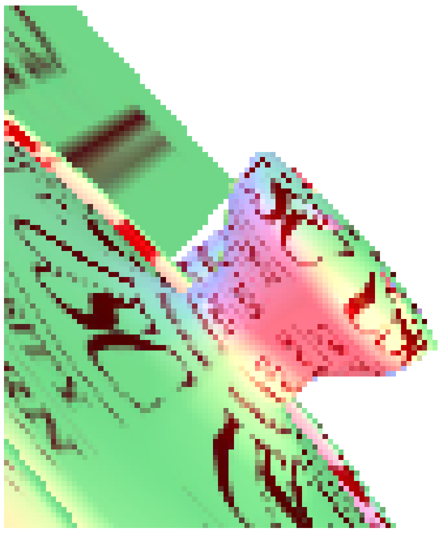
[2] J. Jimenez, D. Gutierrez, J. Yang, A. Reshetov, P. Demoreuille, T. Berghoff, C. Perthuis, H. Yu, M. McGuire, T. Lottes, H. Malan, E. Persson, D. Andreev, and T. Sousa, "Filtering approaches for real-time anti-aliasing," in *ACM SIGGRAPH Courses, 2011.*

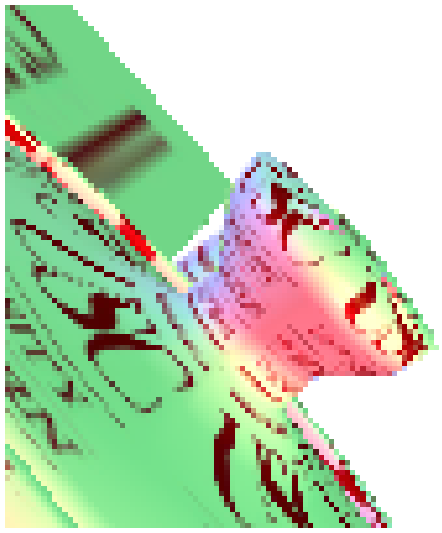
[3] J. Jimenez, J. I. Echevarria, T. Sousa, and D. Gutierrez, "SMAA: Enhanced morphological antialiasing," *Computer Graphics Forum (Proc. EUROGRAPHICS 2012)*, 2012.

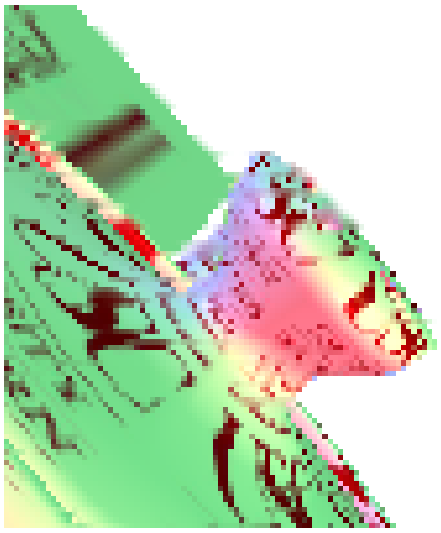
[4] M. Maule, J.L.D. Comba, R. Tochelsten, and R. Bastos, "Transparency and Anti-Aliasing Techniques for Real-Time Rendering," *Proceedings of the 2012 25th SIBGRAPI Conference on Graphics, Patterns and Images Tutorials (SIBGRAPI-T)*, pp. 50-59.

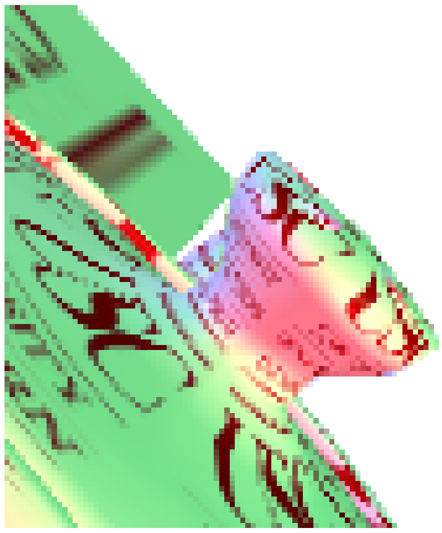
[5] E. Persson, "Geometry buffer antialiasing," *ACM SIGGRAPH Courses Presentation*, 2011.

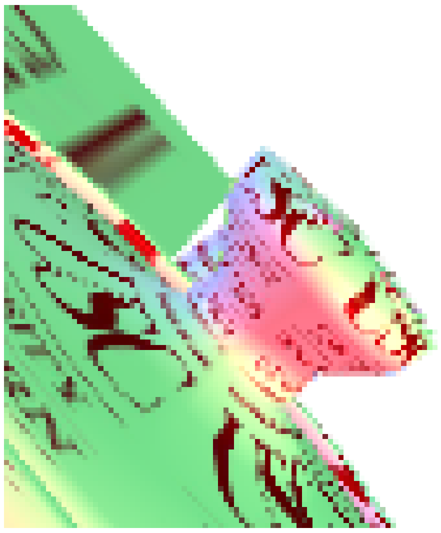
[6] A. Reshetov, "Morphological antialiasing," Intel Labs, Technical Report, 2009.


(a) Scene I rendered without any anti-aliasing.


(b) Scene I rendered using SSAA, Gaussian filter, random pattern, 2 subsamples, and 1.0 extention.


(c) Scene I rendered using MLAA.


(d) Scene I rendered using SSAA, Gaussian filter, OGSS pattern, 2 subsamples, and 1.0 extention.


(e) Scene I rendered using GPAA.

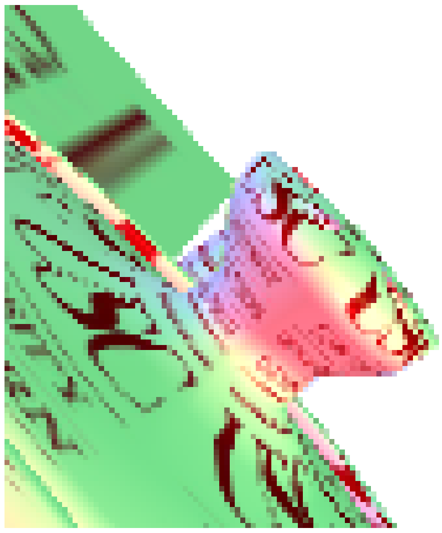

(f) Scene I rendered using SSAA, Gaussian filter, RGSS pattern, 2 subsamples, and 1.0 extention.

Figure 8: Picture comparison: Top-right corner of scene I.