Directional Diffusion Anti-Aliasing (DDAA)

Almar Klein
Independent scholar

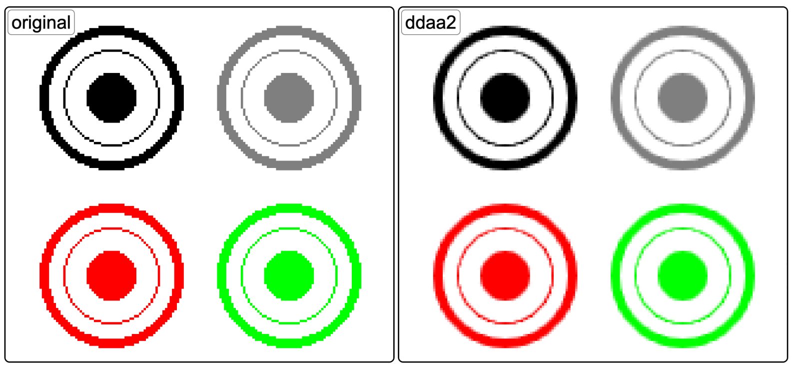


Figure 1. Result of applying DDAA to an example image.

Abstract

Post-processing Anti-Aliasing (PPAA) is a widely used technique for real time rendering. Its benefits include simplicity and performance, and because it applies anti-aliasing to the full rendered image, it is also useful in cases where MSAA cannot be used, such as for lines, markers, and text rendering.

PPAA algorithms can be categorized in a few categories. In one category, the direction of an edge is estimated and then smoothing is applied along the edge, thereby softening the edge's appearance without (in theory) blurring it. In another well known category, an edge-search walks horizontally or vertically along the gradient direction, computing a subpixel offset to produce a smooth gradient in long edges.

The proposed algorithm is a combination of the aforementioned categories, applying both diffusion and edge-search. It maintains small structures, and avoids artifacts by performing a complete edge search, making it suitable for scientific visualizations such as plots. Furthermore, it introduces optimizations that allow it to outperform FXAA.

 ISSN 2331-7418

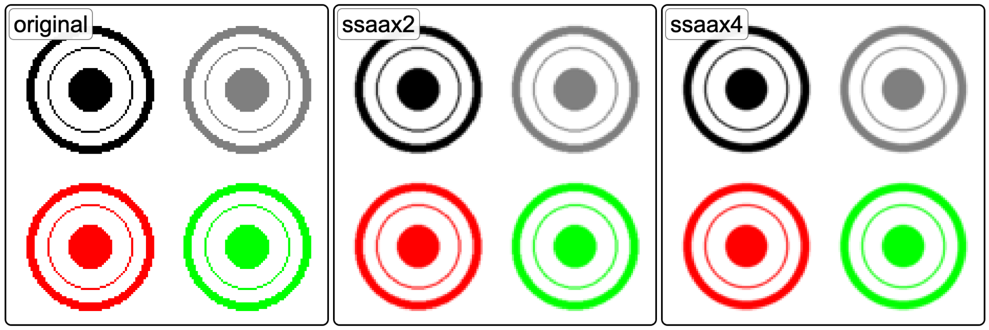


Figure 2. Comparison of SSAA with sampling factors 2 (4x pixels), and 4 (8x pixels). Higher sampling factors improve quality, but significantly increase computational cost due to the larger pixel count and the need for wide downsampling filters.

1. Introduction

1.1. Strategies for anti-aliasing

The most effective way to reduce aliasing effects in rendered images is Super-Sample Anti-Aliasing (SSAA), in which the image is rendered at a higher resolution than the final resolution, and then downsampled with an appropriate filter (see Figure 2 for an example). However, in real-time rendering, the larger number of pixels impacts fragment shader performance, as well as memory usage. This makes SSAA with large sampling factors impractical, while smaller factors (e.g. $<= 2$) are less effective at countering aliasing.

The Multi-Sample Anti-Aliasing (MSAA) method addresses the performance problem by storing multiple samples at mesh edges, without significantly increasing fragment shader cost. It is a widely used technique in video games, but because it does not address aliasing effects for fragments that are *not* on the edge of a mesh, it is not generally applicable. Examples for which MSAA does not help include rendering of thick lines, markers, grids, sdf-based text, etc. Therefore, MSAA is barely useful in scientific applications.

For the aforementioned lines, markers, and text it is common practice to implement anti-aliasing in the fragment shader by reducing the fragment's alpha value at the edge of the object, e.g. [Rougier 2014]. This can be an effective method, since the coverage of the current fragment can usually be calculated. However, if such an object was originally opaque, it will now result in a mix of opaque and semi-transparent fragments, which is problematic because renderers typically treat such fragments differently. E.g. transparent objects must typically be ordered from back to front to be rendered correctly. Therefore, this approach is prone to artifacts, especially in 3D scenes [Klein 2025].

Post-Processing Anti-Aliasing (PPAA) algorithms attempt to reduce the aliasing

Name	Algorithm summary
blur	Simple blurring using a 3x3 kernel with (truncated) Gaussian weights.
ssaax2	Super-Sample Anti-Aliasing with a factor 2 (4 times as many pixels).
ssaax4	Super-Sample Anti-Aliasing with a factor 4 (8 times as many pixels).
fxaa3c	Fast-Approximate Anti-Aliasing 3.11 console version (no edge-search).
fxaa3d	Fast-Approximate Anti-Aliasing 3.11 desktop version (with edge-search).
ddaa1	The proposed algorithm version 1.1 (no edge-search).
ddaa2	The proposed algorithm version 2.4 (with edge-search).

Table 1. Overview of the of the different algorithms compared in this paper. Unless stated otherwise, the term "FXAA" means `fxaa3d`, and "DDAA" means `ddaa2`.

effects on the image after it has been rendered, by subtly changing the values of individual pixels, mostly at edges/boundaries in the image. Because it is applied as a post-processing step, it is relatively easy to implement, without affecting the rendering pipelines or shaders, and is usually more performant than MSAA and SSAA. However, PPAA is applied once the aliasing has already occurred, so artifacts can not always be *removed*, but rather their appearance can be *softened*. Similarly, while individual frames can look good, a stream of images can have temporal artifacts as boundaries shift from one pixel to another.

To mitigate these downsides, PPAA can be combined with SSAA, by rendering at a slightly higher resolution (e.g. 1.5 or 2.0), then apply PPAA, and then downsample to the final resolution.

In the present work, a novel PPAA algorithm is presented, with several benefits compared to existing methods.

1.2. Post-processing anti-aliasing

The many known PPAA algorithms can be roughly divided into three categories.

In morphological anti-aliasing, structures in the image are identified and subpixel blending is applied depending on the detected shape. Examples are MLAA [Reshetov 2009], SMAA [Jimenez et al. 2012] and CMAA [Davies 2014].

Another common approach is to estimate the direction of the edge, and apply some kind of blurring along the edge, but not perpendicular to it, so that the edge remains sharp. Examples include early versions of FXAA and the console versions of FXAA 3.11 [Lottes 2009], as well as some lesser known algorithms such as Directionally-Localized Anti-Aliasing [ForserX 2022].

The "desktop" version of FXAA 3.11 [Lottes 2009], can be considered a different category, as it walks vertically or horizontally along the gradient direction, sampling outward in both directions until the local contrast falls below a threshold, then using the measured span to compute a subpixel offset so that the final sampled pixel corre-

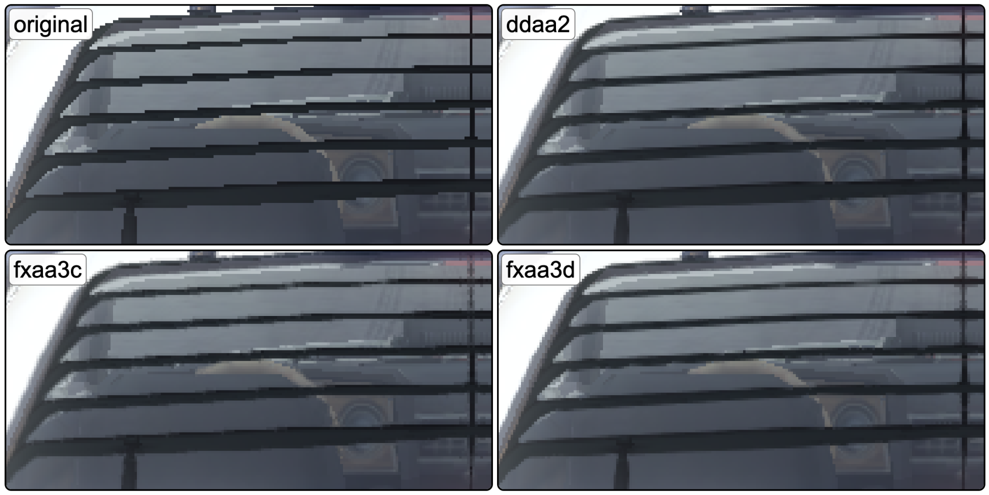


Figure 3. Comparison of the proposed algorithm (`ddaa2`), FXAA 3.11 console (`fxaa3c`), and FXAA 3.11 desktop (`fxaa3d`) on an image containing near-horizontal lines. The algorithms that include edge-search (`fxaa3d` and `ddaa2`) produce a smoother result.

sponds with the reconstructed edge. The variable number of texture lookups results in some loss of performance, but it results in smooth transitions for edges that are nearly horizontal/vertical, as shown in Figure 3. Improvements to this algorithm have also been proposed, e.g. adaptive approximate anti-aliasing (AXAA) [Nah et al. 2016].

Table 1 lists the different algorithms that are compared in the current work. For comparison, this work also includes a method that simply blurs the image with a truncated Gaussian kernel.

2. Proposed algorithm

The proposed algorithm, Directional-Diffusion Anti-Aliasing (DDAA) consists of multiple contributions: an improved method to estimate the angle of edges for consistent diffusion, the combination of directional diffusion and edge-search, a cleaner edge-search without artifacts, and batched sampling during edge-search. These contributions are discussed in detail in the following subsections.

For the proposed algorithm, we distinguish between a simpler version without edge-search (`ddaa1`), and the full version with edge-search (`ddaa2`).

2.1. Directional diffusion

The first contribution—after which the algorithm is named—is the way that the direction of the edge is measured and how the subsequent diffusion (i.e. smoothing) is performed.

To measure the direction of an edge, different kernels can be used. A commonly

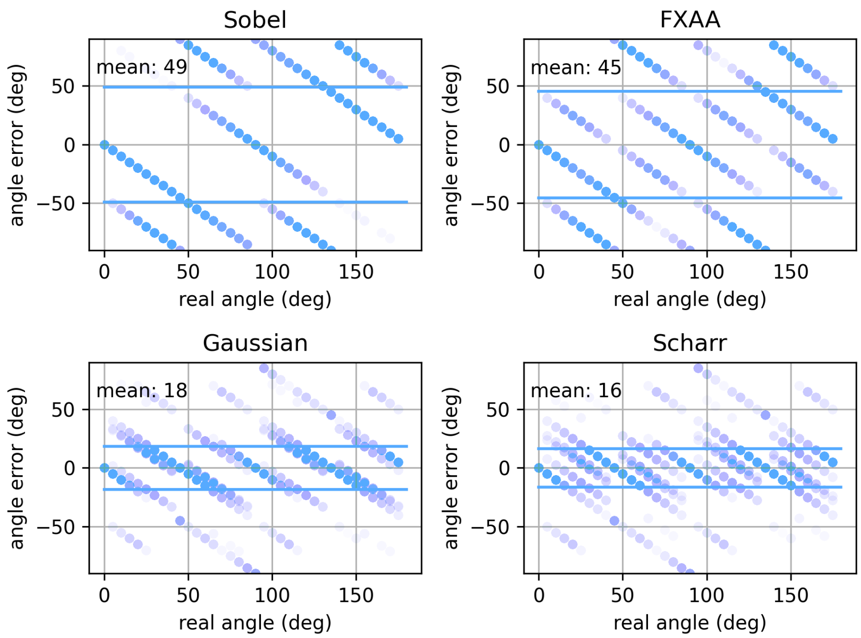


Figure 4. Comparison of angle estimation errors for different kernels. The mean error in degrees are shown in the plots using the horizontal lines and its value in text. The Scharr kernel is most capable of estimating an edge's angle.

used derivative kernel is the Sobel operator. However, it has been shown that this filter is not directionally invariant, and better kernels exist [Jähne et al. 1999; Kroon 2009].

An experiment was performed, applying different kernels to estimate the angle of an edge in an image containing a line at a known angle. The measured angles were compared with the reference angle to calculate the error. The results in Figure 4 illustrate that the Scharr kernel [Jähne et al. 1999] performs much better than the Sobel operator. Errors in the measured angles result in excessive and angle-dependent blurring, as shown in the *fxaa3c* panel in Figure 5. For this reason, the proposed algorithm uses the Scharr kernel, resulting in a consistent smoothing, see the *ddaa1* panel in Figure 5.

To perform diffusion (i.e. smoothing) in a particular direction, an elongated kernel should be used. In practice, a few (bilinearly interpolated) samples are taken that lie on a line. The proposed method uses just two samples, whose distance is larger if more smoothing is required. Their maximum distance from the center is such that the contribution of the center pixel is still the largest. For a horizontal line that distance would be 0.66 pixels. However, if the diagonal case is considered, an interesting effect occurs. As shown in Figure 6, if the sample is taken at the position where the 4 pixels meet, their contribution would be equal, which would cause a significant amount of blur. To reduce this effect, a maximum distance of 0.5 pixels is used instead (indicated

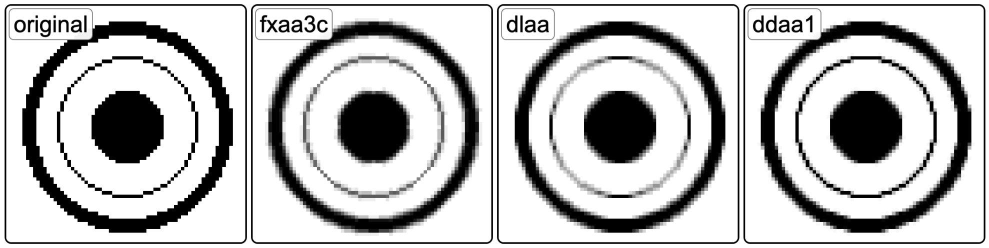


Figure 5. Comparison of PPAA algorithms that estimate the angle of an edge and then smooth in its direction. Errors in the estimate of the edge's angle result in an uneven and blurry appearance of the circles. Due to its improved angle estimate, DDAA produces the most uniform result.

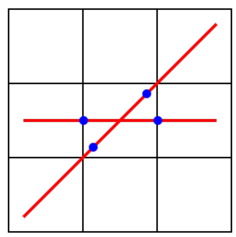


Figure 6. Illustration of the sample location for maximum diffusion for a diagonal and horizontal edge. The center square represents the pixel being evaluated. The blue dots represent a distance of 0.5 pixels from the center. A larger value, like $\sqrt{2}$ would for diagonal lines put the sample point at the point where 4 pixels meet, resulting in an overly blurry result.

by the dot).

2.2. Combining diffusion with edge-search

The diffusion discussed in the previous subsection provides good results, except for lines that are nearly horizontal or vertical. In such cases, the amount of diffusion required to provide a smooth result is enormous and prone to artifacts.

Handling such ”long edges” is a key contribution of FXAA (desktop version); the proposed algorithm adopts a similar edge-search strategy. The idea is to determine how far the current pixel is to the nearest end of the line segment, and then sample the current pixel with a small subpixel offset perpendicular to the edge, where the offset is determined by the distance to the end.

In effect, the proposed algorithm employs two types of blurring: the diffusion in the direction of the edge, and the subpixel offset from the edge-search. These are combined by a) applying the offset to both samples mentioned in section 2.1 and shown in Figure 6; b) diminishing the diffusion strength based on the magnitude of the subpixel offset. This results in a smooth transition as an edge goes from horizontal

```
diffuseStrength = calculateDiffuseStrength()
subpixelEdgeOffset = calculateSubpixelOffset()

edgeStrength = length(2.0 * subpixelEdgeOffset)
diffuseStep = diffuseDir * diffuseStrength * (1.0 - edgeStrength)

texCoord1 = texCoord - diffuseStep + subpixelEdgeOffset
texCoord2 = texCoord + diffuseStep + subpixelEdgeOffset
```

Listing 1. Pseudo-code to combine diffusion with edge-search. The real code can be found at https://github.com/almarklein/ppaa-experiments/blob/main/wgsl/ddaa2.wgsl.

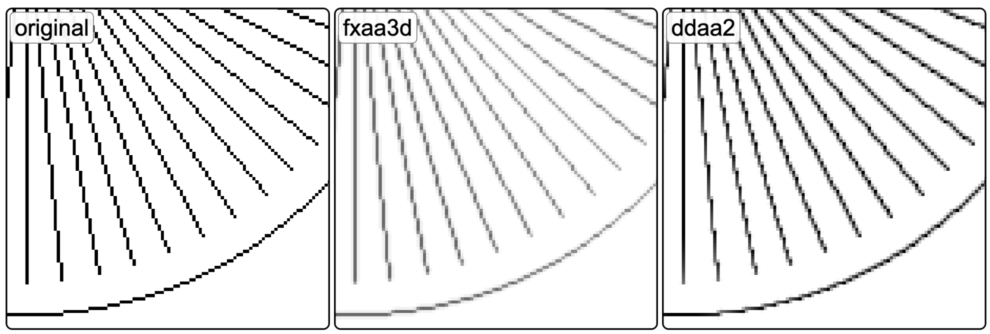


Figure 7. Comparison of FXAA and DDAA on an image containing thin lines. With FXAA the line is less visible, while DDAA maintains the line.

to diagonal, to vertical. See Listing 1 for details.

2.3. Preserving small structures

FXAA is known to diminish small structures. An approach to prevent this has been proposed as part of AXAA [Nah et al. 2016]. The idea is to detect when a pixel is on a ridge, and then disable any blurring. However, the implementation in AXAA only does this for positive ridges, not for valleys. In the proposed algorithm, both ridges and valleys are considered.

Further, the amount of diffusion is diminished depending on the steepness of the ridge/valley, instead of an on/off approach based on a threshold. The effect comparing FXAA and DDAA is illustrated in Figure 7.

2.4. Clean edge-search

During the edge-search, which is implemented in both FXAA ($fxaa3d$) and DDAA ($ddaa2$), the algorithm walks vertically or horizontally along the edge, sampling outward in both directions until the end of the edge is detected.

Since it is now known beforehand how many samples are required, the algorithm

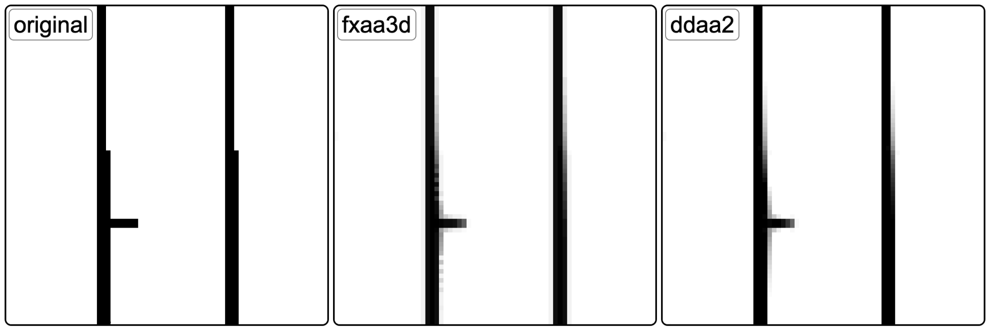


Figure 8. Comparison of FXAA and DDAA on an image that contains two near-vertical lines, one of which has an obstruction to ”end the edge”. FXAA introduces artifacts because it skips pixels (i.e. misses the obstruction), while DDAA produces a clean albeit slightly shorter gradient.

adopts a maximum distance to search in. Searching further produces smoother transitions in the end-result, at the cost of performance. Multiple strategies can be applied to improve the performance or distance of the edge-search.

The FXAA algorithm uses a strategy that increases the size of the steps it takes along the edge, effectively skipping pixels; it starts with steps of 1 pixel, but increases the step size to 2, 4 and even 8 pixels. The idea is that if the edge was already found to be quite long, there's a good chance it is even longer. By skipping pixels, the algorithm can search much further along the edge without losing too much performance. Many derived algorithms, including [Nah et al. 2016] use a similar strategy in their edge-search.

The downside is that the algorithm can miss the end of the ridge, which leads to stipply artifacts. In natural images such cases are rare and the artifacts usually not severe. However in artificial data, e.g. in scientific plots, such cases are much more likely to occur and the artifacts also more pronounced due to the high contrast.

In Figure 8 the effect can be observed. It also shows how DDAA does not suffer from these artifacts; it consistently takes steps of 1 pixel, without skipping any.

While this strategy fixes this particular group of artifacts, it is also more costly. To compensate for that, DDAA uses a trick to take samples more efficiently, which is explained in the next section. Further, DDAA's edge-search is less long than that of FXAA; With the default configuration FXAA takes 12 samples, reaching 26 pixels. The default configuration of $ddaa2$ reaches 15 pixels.

2.5. Batched sampling in edge-search

The proposed algorithm makes use of the WGSL function `textureSampleLevel()` with the integer offset parameter, which must be in the range $[-8..7]$. The edge-search takes multiple samples at each step, using the same texture coordinate, but with different integer offsets.

This *batched sampling* allows the GPU compiler to do multiple texture fetches in parallel, thereby saving time, and improving the overall performance of the algorithm.

One important parameter is the number of samples per batch. The number of fetches that can run in parallel is limited, and with larger numbers the algorithm is increasingly likely to take more samples than needed. A good value for the number of samples per batch was determined empirically, using the same hardware as in Section 3.2. The details of these experiments are out of scope for the current paper, but they suggested that a value of 3 is appropriate.

It is worth mentioning that FXAA does not include this optimization because sampling with an integer offset was not yet available at the time.

Another small optimization is possible by calculating the first edge-search sample from the luma values that are already known, thus saving two texture-fetches (one for each direction).

3. Results

The following section evaluates the proposed algorithm in terms of image quality and performance.

3.1. Quality

The goal of a PPAA algorithm is to reduce the effects of aliasing so the image is pleasing to the eye. To what degree this is successful (i.e. the quality) is hard to measure objectively. Instead, several negative effects that should be reduced and/or avoided are listed, and the visual results compared against this list.

- Jagged structures (mostly apparent in near-diagonal lines).
- Steps in straight lines (mostly apparent in near-horizontal or vertical lines).
- Excessive smoothing (can be introduced by the algorithm).
- Other artifacts (can be introduced by the algorithm).

3.1.1. Results

In Figure 9 the proposed algorithm (`ddaa2`) is compared against other algorithms. It can be observed that the diagonal lines are less jagged than they are for `fxaa3d`, but also not blurry like `fxaa3c`.

It can also be seen that near-vertical and near-horizontal lines are smooth, albeit the length of the 'gradient' is slightly shorter than that of `fxaa3d`. In Figure 8 the gradients can clearly be seen for a vertical line with an obstruction and one line without an obstruction. It shows that the proposed algorithm has a smooth gradient

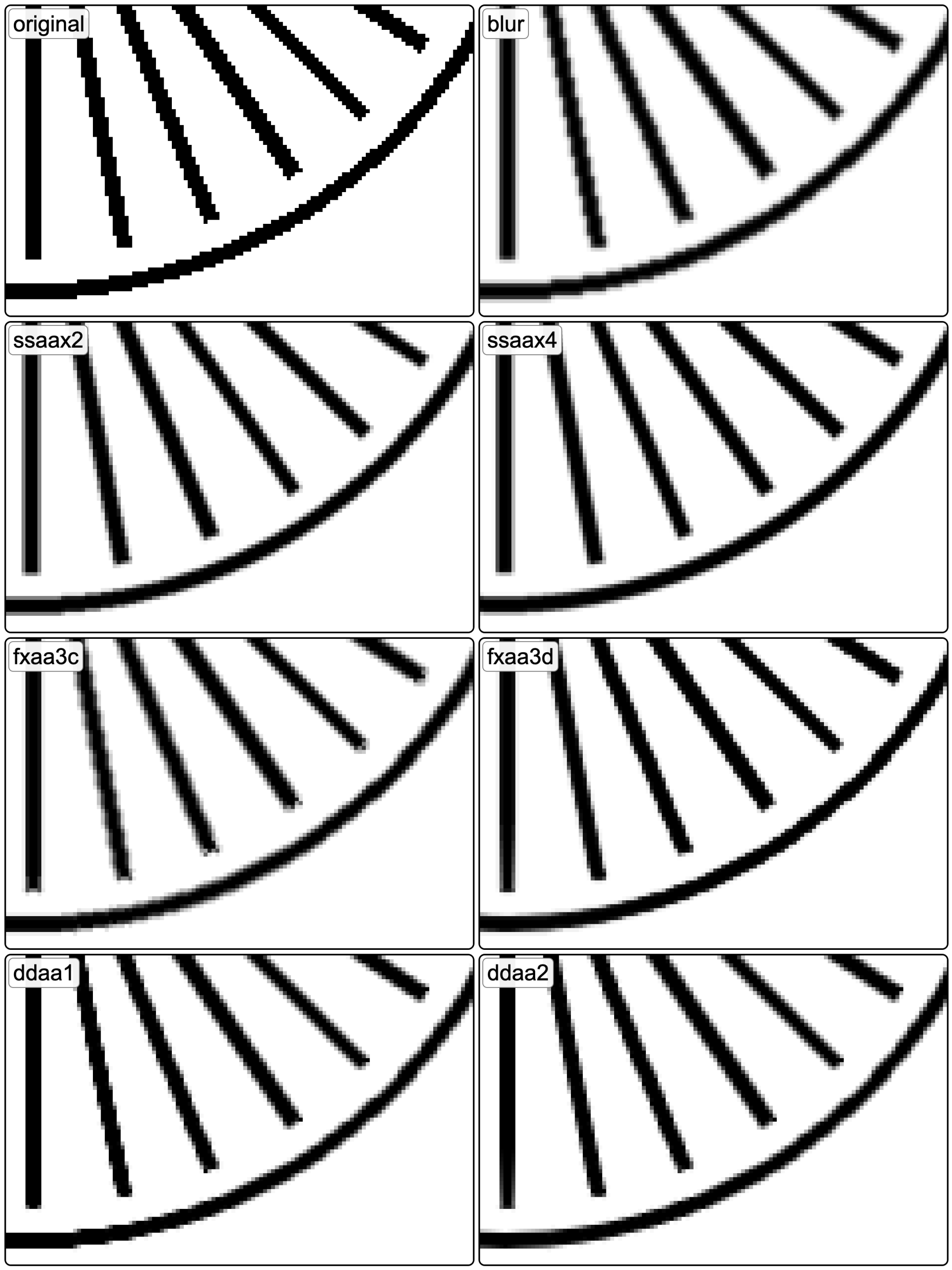


Figure 9. Comparison of all considered PPAA algorithms on an image containing lines at different angles.

without artifacts. The gradient for `fxaa3d` is longer, but shows steps, and has gaps around the obstruction. This illustrates how the proposed algorithm exchanges a bit of edge-search distance for correctness. Figure 3 shows that the longer gradient of

Device	blur	ssaax2	fxaa3c	ddaa1	fxaa3d	ddaa2
Intel UHD 630	100	74	94	98	230	175
Intel UHD 730	100	72	84	125	352	247
AMD Radeon 780M	100	87	81	94	186	184
MacBook M1 Pro	100	94	95	104	219	197
Nvidia RTX 2070	100	71	105	139	282	255
Nvidia RTX 3050	100	80	98	125	160	180
Nvidia RTX 5060 Ti	100	82	102	150	202	203

Table 2. The performance of the considered PPAA algorithms. The values represent relative performance (% of `blur` baseline). Lower values indicate faster performance.

`fxaa3d` has a negligible effect.

Finally, Figure 7 shows that the proposed algorithm avoids blurring away small-/thin structures.

3.2. Performance

3.2.1. Method

To measure the performance in terms of speed, benchmarks were performed on a range of different hardware, including integrated GPUs, Apple Silicon GPUs, and dedicated GPUs.

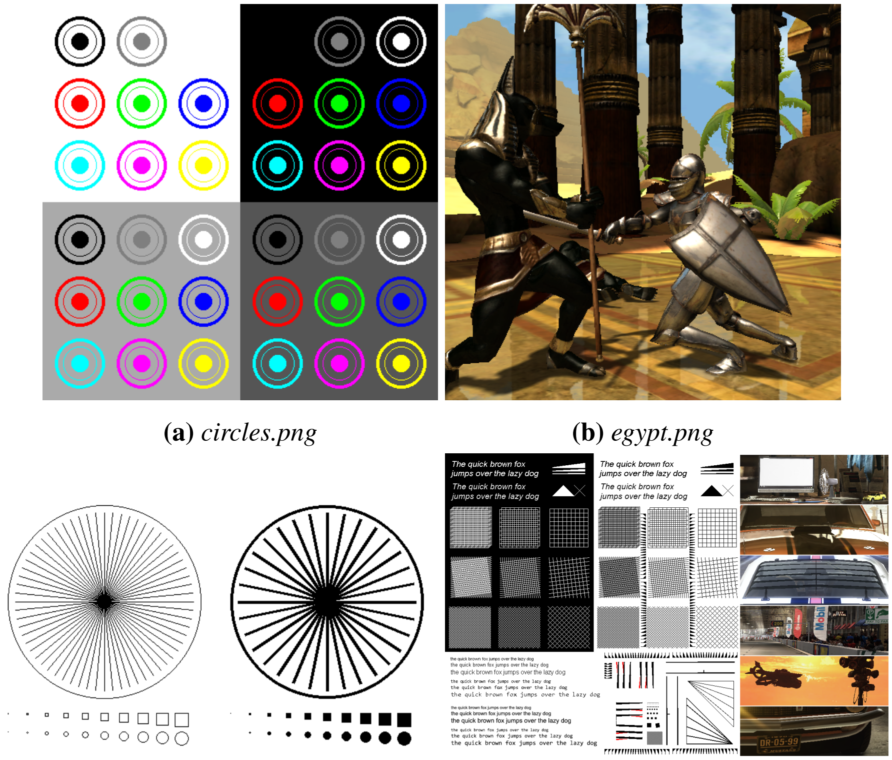
The render-pass for each algorithm was run 100 times, while the time spent by the GPU was measured using a timestamp query. The mean and standard deviations were displayed. A high standard deviation indicates that the GPU was likely also doing other work, so the benchmark could be repeated.

To make it easier to compare the numbers across devices, the measured times were normalized by using the time for the ”blur” algorithm as a reference. (Since its speed is in the same order of magnitude as the PPAA methods, using the blur shader as a reference yields more stable numbers than a shader that only copies the image.) The final result was calculated by taking the mean of the four test images shown in Figure 10.

3.2.2. Results

The final results are shown in Table 2. The values for `blur` are all 100%, since it is used as the baseline.

The `ssaax2` shader is not a PPAA algorithm, but included for comparison. This shader is a cubic Mitchel filter, optimized to take just 12 texture samples. It performs consistently faster than the blur filter, which is surprising because the latter takes only 9 texture samples. This illustrates that SSAA with a factor 2 is a feasible method. Note that the costs for rendering the frame at a higher resolution are not included in



(a) *circles.png* **(b)** *egypt.png*

(c) *lines.png* **(d)** *synthetic.png*

Figure 10. The four test images used for visual comparison and benchmarking.

the benchmarks, but these costs would be the same as when a HiDPI screen would have been used.

The `ssaax4` shader is not included in the benchmarks because it is not as highly optimized in the implementation, and requires 64 texture samples.

As expected, the simpler algorithms `fxaa3c` and `ddaa1` are faster than the algorithms that apply an edge-search. They can be faster than the blur filter because of the early exit strategy for low-contrast regions. The `ddaa1` algorithm is consistently slightly slower than `fxaa3c`.

In Intel integrated graphics and Apple Silicon, DDAA performs consistently better than FXAA. This suggests that the optimization of batching samples is helping the hardware run the algorithm faster. For AMD integrated graphics, DDAA and FXAA have similar performance.

For the dedicated graphics cards (all Nvidia), DDAA performs either better or similar to FXAA, depending on the model.

4. Conclusion

The proposed Directional-Diffusion Anti-Aliasing (DDAA) algorithm has multiple advantages over existing methods such as FXAA. These include fewer jagged artifacts at diagonal edges, no excessive blurring, and maintaining small features. Its edge-search has less reach but does not skip pixels, thereby avoiding artifacts.

The performance of DDAA is better or similar to FXAA, depending on the hardware.

Together, these features make DDAA a promising option for real-time anti-aliasing, especially for scientific visualization and other rendering domains where precision and clarity are important.

References

DAVIES, L. Conservative morphological anti-aliasing (cmaa) - march 2014 update. Technical report, Intel, 2014. URL: https://software.intel.com/en-us/articles/conservative-morphological-anti-aliasing-cmaa-update. 3

FORSERX. Dlaa: Directionally localized antialiasing. https://github.com/ForserX/DLAA, 2022. 3

JIMENEZ, J., ECHEVARRIA, J. I., SOUSA, T., AND GUTIERREZ, D. SMAA: Enhanced subpixel morphological antialiasing. *Computer Graphics Forum*, 31(2pt1):355–364, 2012. EUROGRAPHICS 2012. 3

JÄHNE, B., SCHARR, H., AND KÖRKEL, S. Principles of filter design. In *Handbook of Computer Vision and Applications*. Academic Press, 1999. 5

KLEIN, A. The advantage of the ordered2 blend mode and how to go without it. Issue on the Pygfx repository on Github, 2025. URL: https://github.com/pygfx/pygfx/issues/1003. 2

KROON, D.-J. Numerical optimization of kernel based image derivatives. White paper, University of Twente, 2009. URL: http://www.k-zone.nl/Kroon_DerivativePaper.pdf. 5

LOTTES, T. Fxaa. White paper, NVIDIA, 2009. URL: http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAAWhitePaper.pdf. 3

NAH, J.-H., KI, S., LIM, Y., PARK, J., AND SHIN, C. Axaa: adaptive approximate anti-aliasing. In *ACM SIGGRAPH 2016 Posters*, SIGGRAPH '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343718. URL: https://doi.org/10.1145/2945078.2945129. 4, 7, 8

RESHETOV, A. Morphological antialiasing. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 109–116. ACM, 2009. 3

ROUGIER, N. P. Antialiased 2d grid, marker, and arrow shaders. *Journal of Computer Graphics Techniques (JCGT)*, 3(4):1–52, November 2014. ISSN 2331-7418. URL: http://jcgt.org/published/0003/04/01/. 2

Index of Supplemental Materials

- The DDAA algorithm is developed at https://github.com/almarklein/ppaa-experiments. This repository contains the source code, test images, and a small Python framework to run the PAA shaders.
- The shader code is available at https://github.com/almarklein/ppaa-experiments/blob/main/wgsl/ddaa2.wgsl.
- A web viewer to compare the outputs of multiple PPAA algorithms is available at https://almarklein.github.io/ppaa-experiments/viewer.html.

Author Contact Information

Almar Klein
Almar Klein Scientific Computing
De Steiger 24
1351 AB Almere, The Netherlands
(https://almarklein.org)
almar@almarklein.org

Almar Klein, Directional Diffusion Anti-Aliasing (DDAA), *Journal of Computer Graphics Techniques (JCGT)*, vol. volume, no. issue, 1–6, yyyy
http://jcgt.org/published/vol/issue/num/

Received:	yyyy-mm-dd		
Recommended:	yyyy-mm-dd	Corresponding Editor:	Editor Name
Published:	yyyy-mm-dd	Editor-in-Chief:	Eric Haines