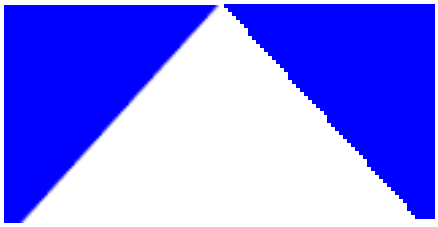
Comparative Analysis of Anti-Aliasing Algorithms

Christian Elie Abboud



Deviations from project proposal

There were no major deviations from the original project proposal. The project successfully implemented the targeted anti-aliasing algorithms (MSAA, FXAA, and SMAA) within the custom C/OpenGL rendering engine as scheduled,

Introduction to the problem

Anti-aliasing is a key process in real-time graphics rendering, balancing image fidelity and computational efficiency. It is a computer graphics technique used to smooth jagged edges, especially along object outlines, in order to create a more realistic and natural appearance.

Motivation and Significance

Understanding different algorithms trade-offs is relevant to visual simulation, in game engines or 3D modeling tools for example. Efficient anti-aliasing is critical for enabling high-fidelity rendering on constrained hardware, such as mobile devices and VR headsets, where maintaining high frame rates is essential for user comfort.

However, a significant gap exists in the available literature and online resources. Most comparative studies of these algorithms are conducted within commercial game engines. These environments introduce uncontrolled variables such as lighting passes, physics simulations, and proprietary optimizations that obscure the true computational cost of the anti-aliasing pass. This project is motivated by curiosity and the need for rigorous data: it aims to provide an exhaustive comparison of industry staples static Anti-Aliasing algorithms in a controlled environment.

Project Scope

A custom rendering pipeline serves as the foundation for this study, as this standard approach facilitates a direct comparison between hardware-based methods and software filters on equal footing. Specifically, the study compares Multi-Sampling Anti-Aliasing (MSAA) at 2x, 4x, and

8x quality levels against two post-processing techniques: Fast Approximate Anti-Aliasing (FXAA) testing both standard and high-quality version and Subpixel Morphological Anti-Aliasing (SMAA) from Low to Ultra settings. To accurately determine the computational cost of each effect, the primary metric for analysis is Frame Time measured in nanoseconds, which isolates the exact duration the Graphics Processing Unit (GPU) takes to execute the rendering commands, ensuring that performance differences are attributed solely to the algorithms computational cost. To see how the algorithms behave under different conditions, we test two specific scenarios:

The triangle scene: A very simple scene with almost no edges. This measures the base cost of the algorithm essentially, how much performance do we lose just by turning the effect on, with the strict minimum of geometry rendered?

The dartboard scene: A complex, high-contrast pattern designed to create significant amount of jagged edges. This serves as a stress test, forcing the algorithms to work as hard as possible to smooth out the image.

Two notable static AA algorithms were excluded due to their obsolescence in modern real-time rendering. Super-Sampling Anti-Aliasing, while offering the highest theoretical quality by rendering the image at a higher resolution and downscaling it, was excluded because its extreme computational cost, making it rarely used. Similarly, Morphological Anti-Aliasing was omitted; as the CPU-based predecessor to SMAA, it has been effectively rendered obsolete. SMAA represents the refined, GPU-accelerated evolution of that technique, offering superior performance and subpixel accuracy without the bandwidth bottlenecks of the original MLAA implementation.

Approach used

Experimental Framework

Our minimalist rendering engine was developed in C and uses the OpenGL 4.3 graphics API, along with shaders written in GLSL. In order to render images on the screen, the application first needs to supply the GPU with geometric data. This is achieved using Vertex Buffer Objects (VBOs), which store raw arrays of data such as the coordinates of the triangle's summits. To interpret this raw memory correctly, we utilize Vertex Array Objects (VAOs), which define the layout and attributes of the vertex data, effectively telling the GPU how to read the coordinate streams.

Once the geometry is defined, the rendering pipeline requires specific instructions on how to process it. These instructions are provided via Shaders, programs that run directly on the graphics card. The process begins with the Vertex Shader, which handles the transformation of individual vertices in 3D space. The output is then rasterized, where data is interpolated across the geometric shapes, before being passed to the Fragment Shader, which runs once. This stage determines the final color of each and every pixel of our screen, making it the primary location for the expensive calculations involved in our procedural dartboard scene. These shaders are compiled and

linked together into a Program object, which serves as the complete executable rendering state for a given draw call.

While the standard output of this pipeline is the computer monitor, the implementation of post-processing algorithms like FXAA and SMAA requires intermediate steps where the image is not immediately displayed onscreen. This is handled through Frame Buffer Objects (FBOs), which allow the engine to redirect the rendering output away from the screen and into a Texture. By rendering the scene to an off-screen texture first, we treat the entire rendered image as a new input, allowing a secondary "fullscreen quad" pass to sample that texture, analyze its edges, and apply anti-aliasing filters before the final result is finally displayed.

Multi-Sampling Anti-Aliasing (MSAA)

In a basic rendering pass, the hardware tests a single point at the exact center of the pixel grid; if this solitary point lies inside a triangle, the entire pixel is filled with that color. This binary "all-or-nothing" decision is what creates the harsh, staircase-like edges seen on diagonal lines. MSAA refines this process by introducing multiple sub-samples patial test points distributed within the pixel's footprint. By detecting exactly how many of these points are covered by the geometry, the GPU can mix the object's color with the background proportionally, producing a smooth, average gradient along the edge rather than a jagged step.

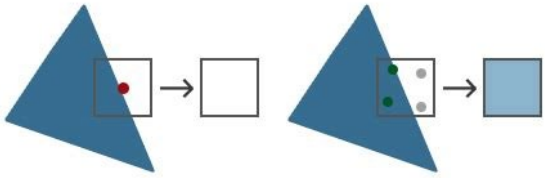


Figure 1:Multisampling[2]

Since a standard texture can only hold one color value per pixel, we must allocate a specialized multisampled texture. This resource is significantly larger than a standard screen buffer, as it reserves memory to store distinct color and depth data for every individual sub-sample (4, 8, or 16 times the data per pixel, depending on algorithm version needs).

However, a computer monitor cannot display these multiple samples directly; it expects a single, final color for every physical pixel. This necessitates a resolve step, which we handle via a "blit" operation. During this stage, the GPU takes the heavy multisampled texture, reads the valid sub-samples for each pixel, averages them together based on their coverage, and writes the final, resolved image into a standard FBO ready for presentation.

Fast Approximate Anti-Aliasing (FXAA)

FXAA is an image-based filter that operates entirely as a post-processing step. The algorithm does not know where the triangles are; it simply looks at the final rendered image (the texture produced by our FBO) and attempts to infer where edges are based on color contrast.

The core mechanism for both implemented versions of FXAA begins with Luminance Conversion. Since human eyes are more sensitive to brightness changes than color shifts, the shader first converts the RGB pixel data into a single grayscale "Luma" value. It then compares the center pixel's luma against its neighbors (North, South, East, West). If the difference between the maximum and minimum luma in a local neighborhood exceeds a specific threshold, the algorithm flags that pixel as part of an edge.

It is at this point how the algorithm decides to smooth that edge that our two implementations diverge.

The "Console" implementation: This version is designed for maximum speed. Once an edge is detected, this algorithm performs a single-pass analysis of the immediate 3x3 pixel neighborhood. It calculates a gradient direction (horizontal or vertical) and immediately determines a blending factor. It then samples the texture just slightly off-center along that gradient to blur the jagged line. This method is incredibly fast because it uses a fixed number of texture reads and no loops, but it can sometimes over-blur fine details because it assumes the edge is short and straight.

The "Quality Preset FXAA 3.11"[1] implementation (iterative): This one is a direct port of the original algorithm (industry-standard) authored by Timothy Lottes at NVIDIA, adapted for OpenGL 4.3. Unlike the console version, which guesses the edge length, this implementation performs an Iterative Edge Search. Once an edge is found, the shader enters a loop that "walks" along the edge (left/right or up/down) step-by-step to find exactly where the line ends. By knowing the precise start and end points of the edge, the algorithm can calculate a sub-pixel shift amount with much higher accuracy. This results in sharper textures and better reconstruction of long diagonal lines, albeit at the cost of higher GPU usage due to the branching logic and multiple texture fetches required for the search loop.

Figure 2: No AA vs "Console" FXAA vs FXAA 3.11

Subpixel Morphological Anti-Aliasing (SMAA)

SMAA is also a post processing effect, and represents a sophisticated attempt to reconstruct the geometric details usually lost during the initial rasterization. Instead of simply blurring high-contrast areas, SMAA treats the image as a collection of geometric shapes. It analyzes the jagged patterns on the screen and uses morphological pattern recognition to deduce the underlying sub-pixel geometry that likely produced them, and then tries to recreate them as precisely as possible.

Our implementation integrates the industry-standard *SMAA.hlsl* developed by Jorge Jimenez et al., and executes this logic through a synchronized pipeline of three distinct rendering passes. The process begins with the Edge Detection pass, which scans the source image to locate high-contrast boundaries, but unlike simpler methods, it is strictly focused on topology. It identifies pixels that represent a transition between objects and marks them in a specialized texture. This creates a clean, binary "wireframe" representation of the scene.

We then move to Blending Weight Calculation pass**.** This one acts as a pattern matcher, scanning the edge texture from the previous pass to identify specific geometric shapes. SMAA accelerates the process using two pre-computed helper textures. The first, SearchTex, is a lookup table that optimizes the search for edges start and endpoints, allowing the shader to quickly determine the shape and length of an edge without checking every single pixel. he maximum distance the shader tries to follow the edge is restricted by quality presets (low to ultra). AreaTex then provides the pre-calculated coverage values for that specific pattern [3]. This allows the shader to instantly retrieve the precise fractional coverage of the pixel based on the line's length and orientation, avoiding expensive runtime calculations.

The pipeline concludes with the Neighborhood Blending pass. It takes the original image and the blending weights calculated in the previous step to mix the pixel colors with their neighbors.

Results

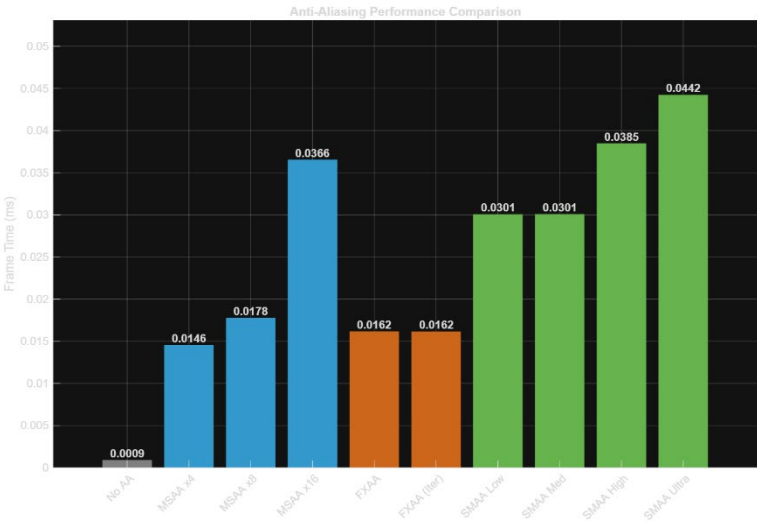


Figure 3:Performance comparison on triangle scene

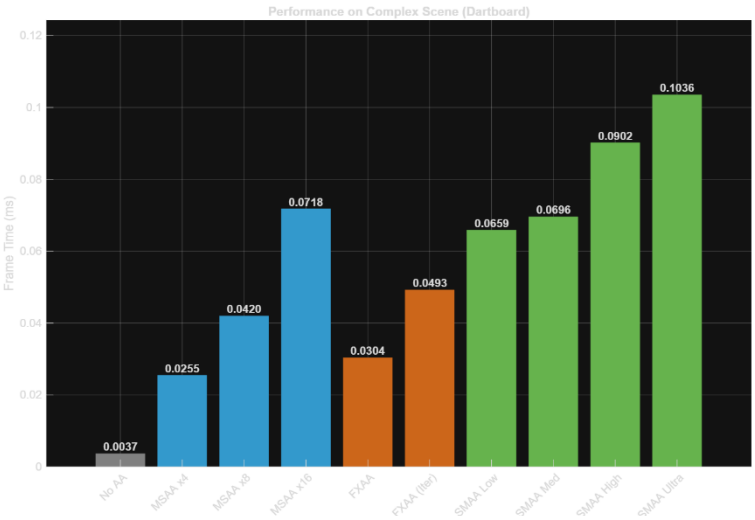


Figure 4: Performance comparison on dartboard scene

All tests were conducted on an NVIDIA GeForce RTX 4060. While the absolute frame times converted to milliseconds will naturally vary depending on the specific hardware the relative performance ratios between the algorithms offer a consistent, transferable assessment of their computational cost.

The results from the triangle scene establish mainly the base overhead for each technique. Here, with almost zero geometric complexity, the "No AA" render time is negligible (0.0009 ms). The most immediate observation is the behavior of the post-processing filters. Both the "Console" and 3.11 versions of FXAA clock in at an identical 0.0162 ms. This equality is reasonable and

expected: because the scene contains almost no edges, the "3.11" version's expensive iterative search loops are never triggered. Both shaders effectively exit early after the initial contrast check, revealing that 0.0162 ms is the constant "floor" cost of binding the FBO and running the fullscreen pass on this hardware. Similarly, SMAA demonstrates a higher baseline overhead (0.0301 ms for Low/Medium) compared to FXAA. This confirms the cost of its multi-pass architecture; even when doing very little smoothing.

The dartboard scene successfully stresses the algorithms, revealing the cost of complex edge processing. Unlike the baseline test, the two FXAA variants now diverge significantly. The Iterative 3.11 version jumps to 0.0493 ms—nearly a 60% increase over the "Console" version (0.0304 ms). This confirms that the conditional loops described in the methodology are now active, actively "walking" along the thousands of jagged edges generated by the dartboard pattern.

We also observe an interesting plateau between the SMAA Low and Medium presets, which yielded nearly identical results (approx. 0.066 ms - 0.069 ms). This suggests that for this specific resolution and pattern complexity, the lower search-step limit of the Low preset was rarely hit, or that the overhead of the three passes dominates the slight difference in shader logic. However, as we move to "Ultra," the cost spikes to 0.1036 ms, making it the most expensive algorithm tested. This reflects the "Ultra" preset's exhaustive search for long diagonal patterns and its more expensive edge detection thresholds.

Perhaps the most striking result is the efficiency of MSAA. At 4x quality on the dartboard, MSAA (0.0255 ms) outperforms most of our post-processing method, including the simplest FXAA. MSAA does not run on the general-purpose shader cores (ALUs) that FXAA and SMAA must consume; instead, it relies on the GPU's ROPs (Raster Operations Pipeline), specialized fixed-function hardware designed specifically for this task. The RTX 4060, being a powerful modern card, has ample ROP throughput to handle MSAA 4x with reduced penalty relative to the shading cost. This confirms that while post-processing methods are theoretically lighter [1, 3], on powerful dedicated hardware, the MSAA can actually be faster because it runs in parallel with the fragment shader rather than after it.

Overall, the data demonstrates coherency with the theoretical models. We observe a clear performance hierarchy where the cost of anti-aliasing correlates directly with settings as well as algorithmic complexity: the single-pass logic of FXAA is consistently faster than the multi-pass morphological reconstruction of SMAA.

Conclusion and outlook

The data provides a clear answer to the initial problem statement: there is no single "superior" algorithm, but rather a spectrum of solutions where performance is traded for geometric accuracy and memory bandwidth.

While our results painted MSAA as a highly efficient solution on modern hardware, this conclusion comes with a significant real-world caveat. Our test environment utilized a Forward Rendering pipeline, which naturally complements the hardware-based approach of MSAA. However, a large part of modern rendering engines utilize Deferred Shading (a technique where geometry is rendered into intermediate buffers before color is calculated), which is usually

incompatible or very expensive with MSAA therefore requiring post processing effects.

Limitations

A valuable extension of this work would be to deploy the testing framework across a broader spectrum of hardware configurations, ranging from low-power integrated solutions to varying generations of discrete cards. It would be particularly interesting to observe how the performance hierarchy shifts on less capable hardware; we anticipate that bandwidth-heavy methods like MSAA may degrade disproportionately less adapted devices.

Final Verdict

Ultimately, the best algorithm depends entirely on the deployment scenario. MSAA remains a very solid option for Forward Rendering contexts where clarity is critical and deferred shading is too slow. FXAA reigns supreme for constrained hardware or mobile devices where every millisecond of shader budget counts. Finally, SMAA occupies the vital middle ground, offering a high-fidelity, sharp image for modern deferred engines without the prohibitive memory cost of hardware sampling.

Authorship statement

This project was completed individually. All code implementation, data analysis, and report writing were performed by Christian Elie Abboud.References

References

[1] T. Lottes, "FXAA 3.11 Quality FxaaPixelShader," NVIDIA, 2011.

[2] J. de Vries, "Anti-Aliasing," *LearnOpenGL*. (https://learnopengl.com/Advanced-OpenGL/Anti-Aliasing)

[3] J. Jimenez, J. I. Echevarria, T. Sousa, and D. Gutierrez, "SMAA: Enhanced Subpixel Morphological Antialiasing," in *Proceedings of Eurographics 2012*, vol. 31, no. 2, 2012.