# Evaluation of Antialiasing Techniques on Mobile Devices

**Tiago Manuel Videira Almeida**

Thesis to obtain the Master of Science Degree in

## Information Systems and Computer Engineering

Supervisors: Prof. João António Madeiras Pereira
Eng. David Bernardino Martins Sena

## Examination Committee

Chairperson: Prof. Mário Jorge Costa Gaspar da Silva
Supervisor: Prof. João António Madeiras Pereira
Member of the Committee: Prof. Fernando Pedro Reino da Silva Birra

**May 2018**

# Acknowledgments

I want to thank my parents, Emília Almeida and Manuel Almeida for their support during my school years, without them I would not reach this level of studies. In second place I wanna thank the supervisors Professor João Madeiras, David Sena and Nikolaos Papadopoulos, who help me greatly by providing feedback, by amicability reaching out to see how I was doing, they greatly helped making this work better. Samsung R&D UK helped a lot by providing equipment for benchmarks, bringing a lot of value to the work. Last but not least I wanna thank my family and friends, in special to André Fonseca, Daniel Rodrigues, Filipe Mendes, Isabel Costa, Ricardo Santos, Samuel Gomes for the fun times, their companionship helped to make this journey a lot better.

# Abstract

The adoption of mobile devices such as smartphones has been massive, people use their mobile devices to play video-games, view virtual worlds in VR and experience augmented reality. In the field of real-time rendering, antialiasing is still an open problem, there have been a lot of recent advances, especially in post-processing and temporal antialiasing. We found that mobile antialiasing research is very scarce, MSAA is the most used solution in the last decade. Our work brings the state of the art of antialiasing techniques from desktop to mobile. We have created an Android application to test four antialiasing techniques on two scenarios and analyzed their performance and image quality. The results show that MSAA and FXAA provide the fastest execution and SMAA obtains the best image quality, overall the most balanced technique is MSAA. The results on mobile are similar to desktop, except MSAA which obtained better performance due to this technique benefiting from tiled-based rendering architecture present on mobile GPUs.

# Keywords

antialiasing, image enhancement, computer graphics, real-time rendering, mobile computing

# Resumo

A adoção de dispositivos móveis como smartphones é massiva, as pessoas usam estes dispositivos para jogar, ver mundos virtuais em VR e experienciar realidade aumentada. No campo da renderização em tempo real, o *antialiasing* é um dos problemas em aberto, que recentemente tem tido grandes avanços, especialmente técnicas de pós-processamento e temporais de *antialiasing*. O *antialiasing* nas plataformas móveis não tem tido tanta pesquisa como deveria, MSAA é a técnica mais usada na última década. O nosso trabalho traz o estado de arte em algoritmos de *antialiasing* dos desktops para os dispositivos móveis. Criámos uma aplicação móvel para Android de modo a testar quatro técnicas de *antialiasing* em dois cenários e analizámos a sua performance e qualidade de imagem. Os resultados mostram que o MSAA e FXAA consegue obter a melhor performance geral e o SMAA obtém a melhor qualidade de imagem, tendo em conta todos os testes a técnica mais equilibrada é o MSAA. Os resultados em dispositivos móveis são semelhantes aos de desktop, excepto no MSAA que obteve melhor performance devido a esta técnica beneficiar da arquitectura tiled presente nas GPU dos dispositivos móveis.

# Palavras Chave

antialiasing, melhoramentos de imagem, computação gráfica, renderização em tempo real, computação móvel

# Contents

# List of Figures

# List of Tables

# Listings

# Acronyms

**AA** Antialiasing

**GPU** Graphics Processing Unit

**SSAA** Supersampling Antialiasing

**MSAA** Multisampling Antialiasing

**FXAA** Fast Approximate Antialiasing

**MLAA** Morphological Antialiasing

**SMAA** Subpixel Morphological Antialiasing

**TAA** Temporal Antialiasing

**OpenGL** Open Graphics Library

**OpenGL ES** OpenGL for Embedded Systems

**SSIM** Structural Similarity

# 1 Introduction

The following chapter will introduce the reader to the goals of this thesis. This includes the motivation that leads us to research antialiasing on mobile and a description of the objectives of this work.

## 1.1 Motivation

The adoption of mobile devices in the last years has been massive. The news indicate this growth, "1 billion smartphones shipped worldwide in 2013" [1]. In 2016 the mobile web usage was superior than desktop[2], also last year mobile game revenue has passed desktop and consoles [3].

The capabilities of mobile devices have improved over the past few years. The hardware is better, the displays have high pixel density allowing the visualization of high-quality images. The software has also been upgraded for better performance, better designed and secure. New technologies like virtual reality and augmented reality which are very hardware intensive are being run on mobile devices.

Aliasing is one of the major problems in graphics rendering. Pixel quality is important for real-time applications, noticeable artifacts can break immersion, especially in virtual reality. Pixel quality can be a quality differentiator that is noticed by the users.

In the last 7 years, there have been great advancements in graphics rendering for desktop, especially in the field of antialiasing. Unfortunately, some antialiasing solutions are not efficient on mobile devices, due to their architecture and constraints. Research on antialiasing for mobile is very scarce, the most used antialiasing technique on mobile is multisampling antialiasing (MSAA). On desktop and consoles, a variety of antialiasing techniques are being used in graphics applications such as games, these techniques so far have not been tested on mobile devices. There is no research of these new antialiasing algorithms on mobile devices.

[1] http://www.pcworld.com/article/2091940/global-smartphone-shipments-topped-1-billion-in-2013.html (as consulted on June 15, 2018)

[2] http://www.telegraph.co.uk/technology/2016/11/01/mobile-web-usage-overtakes-desktop-for-first-time (as consulted on June 15, 2018)

[3] http://www.cnbc.com/2016/04/22/mobile-game-revenue-to-pass-console-pc-for-first-time.html (as consulted on June 15, 2018)

## 1.2 Objectives

The main objective of this work is testing and comparing state of the art antialiasing solutions on mobile. This work consisted of creating a 3D Android application to analyze how each antialiasing technique behave. We had a partnership with Samsung R&D UK, who provided us support from their research team and a Samsung Galaxy S8 device to run the tests.

The techniques that we considered relevant to study were:

- Multisample Antialiasing (MSAA)
- Fast approximate Antialiasing (FXAA)
- Subpixel Morphological Antialiasing (SMAA)
- Temporal Antialiasing (TAA)

This study purpose is to answer the following questions:

- Which antialiasing techniques are faster?
- Which one obtains the best image quality?
- What else should be explored on mobile antialiasing?

Our tests covered multiple configurations such as 3D scenes and resolutions and collected behavior data for analysis.

## 1.3 Document structure

This document is structured in the following chapters: Related work which contains the research done on antialiasing techniques, a comparison of these techniques and also a section about tiled rendering; Methodology describes how we developed the benchmark application, the implementation of the antialiasing techniques and explains the test scenarios and metrics that we collected; Results and Discussion chapter presents and analyzes the results obtained; Conclusion chapter summarizes the achievements of this work and proposes future work on this topic.

# 2 Related work

In this chapter, we explain state of the art antialiasing solutions, techniques like supersampling and multisampling, also FXAA, SMAA and temporal antialiasing. We include a section about tiled rendering architecture, a key concept in mobile rendering.

## 2.1 SSAA and MSAA

Supersampling antialiasing (SSAA) and Multisample antialiasing (MSAA) are techniques that mitigate aliasing in the graphics pipeline. The rasterizer stage of the pipeline is where we sample a continuous function into a finite number of pixels on the screen, this discretization introduces aliasing artifacts. The rasterizer is a stage of the rendering pipeline, a simplified version of the rendering pipeline is presented in figure 2.1.

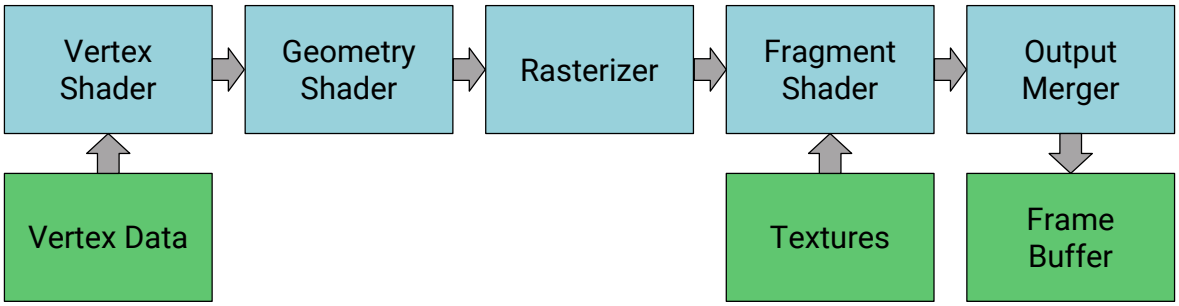


**Figure 2.1:** This block diagram is a simplified version of the rendering pipeline. The stages of the pipeline are in blue color and memory resources in green.

In simple terms, the rasterizer stage takes as input 3D vertices (primitives) to be rendered and outputs a set of fragments that indicate where the geometric primitive will be visible on the screen. The visibility of a primitive on a given pixel is determined by *coverage* and *occlusion* [1].

Coverage is a test that determines which pixels are being overlapped by a triangle or other primitive. Only the pixels that are covered in the center are considered as visible, this makes the geometry edges lose sharpness and instead display *jaggies*. See figure 2.2 for an example.

Even if a pixel is covered by geometry it might be not drawn because another triangle might be occluding it. The occlusion process is solved by the Z-buffer algorithm. The Z-buffer test is executed in the output merger stage after the fragment shader, as shown in figure 2.1. The Z-buffer is a two-

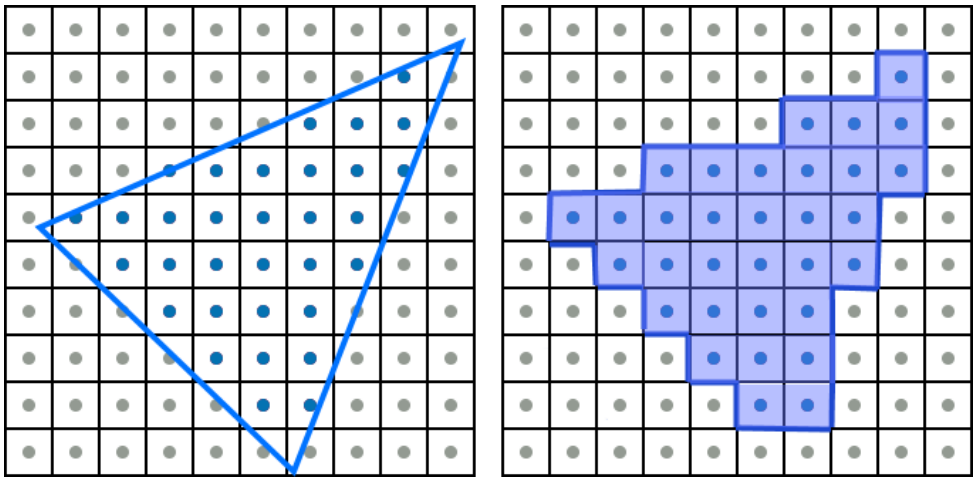

**Figure 2.2:** Coverage example for a triangle: the pixels whose center is inside the triangle passed the coverage test (marked in blue). Only those pixels advance to the fragment shader stage.

dimensional array with the capacity equal to the number of pixels on the screen. It stores the interpolated z-value of the primitive that is closer to the camera. Whenever a primitive is drawn, its depth values are stored in the z-buffer, in case that the next primitive covers the same pixel the z coordinate is tested, if it is closer to the camera than the current z-buffer value it is replaced, otherwise the z-buffer remains unchanged[1]. This process is done for all the geometry because the fragments closer to the camera are the ones that are drawn, it reproduces the realistic effect of occlusion [3]. In some cases, the Z-buffer test can be performed before the fragment shader. This early test provides an optimization that prevents computing the shade for occluded fragments [1].

### 2.1.1 Oversampling

Oversampling is the process of sampling a signal at a higher rate than the Nyquist rate [4]. The concept of oversampling in computer graphics is known as supersampling [1].

Supersampling is simple to implement in a rendering engine, it works by rendering to a higher resolution than the screen, normally a power of two ($2^x$) larger, then that image is downsampled into the intended output resolution [1].

In figure A.1 of the appendix you can view the results of supersampling with different sampling patterns, with more samples the less aliasing is visible. Supersampling obtains high-quality images, the disadvantage of this technique is its performance. Because the rendering resolution increases, the fragment shader sampling rate is also increased. The fragment shader normally does a lot of calculations such as lightning and texture accesses, with supersampling these operations are executed at a higher rate, consuming more resources (see figure 2.3). The memory consumption also increases, because

[1]Good explanation of the Z-Buffer algorithm and MSAA - https://mynameismjp.wordpress.com/2012/10/24/msaa-overview/ (as consulted on June 15, 2018)

the size of the color buffer and z-buffer are increased [1].

| | NO-AA | SSAA 2x | SSAA 4x |
|---|---|---|---|
| Cost | 1x | 4x | 16x |

**Figure 2.3:** Zoomed image results of a 3D scene rendered with supersampling using different number of samples. While the image quality increases, the rendering cost also increases.

### 2.1.2 Multisampling

As seen in the last chapter, the cost of supersampling is too high, real-time applications need to maintain a steady number of *frames per second* (FPS), normally above 30 FPS for a good viewing experience. One thing that we can observe is that aliasing is most visible on the edges of triangles, this is known as spatial or geometric aliasing, multisampling mitigates this form of aliasing.

> ”A multisampling algorithm takes more than one sample per pixel in a single pass, and shares some computations among the samples.” [3]

Multisampling is performed on the GPU, it is supported by most of the GPU manufacturers. Nvidia has a slight different technique called coverage sampling antialiasing (CSAA) which does some optimizations to the original technique.[2]

The main difference between multisampling and supersampling is that the fragment shader is executed only once per pixel. Coverage and occlusion tests are performed at a higher rate. The coverage

[2] http://www.nvidia.com/object/coverage-sampled-aa.html (as consulted on June 15, 2018)

test is executed with N sampling points inside the pixel, these are called coverage samples or subsamples, N is the multisampling factor. Figure 2.4 shows an example of the samples in 4x MSAA.

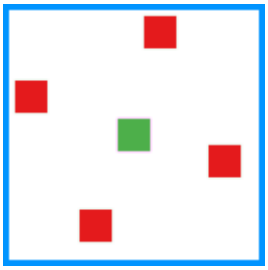

**Figure 2.4:** Example MSAA sampling pattern of a pixel, the red squares indicate the position of coverage and occlusion samples (also called subsamples), the green square is the fragment shader sample position.

The coverage test determines which coverage samples are overlapping a primitive, it can be represented in a bit mask with N bits, each bit represents if a subsample was covered or not. The occlusion test is also executed per subsample. The triangle depth is interpolated at each subsample location, resulting in N depth values. The Z-buffer size has to be larger to hold N depth values for each pixel. For MSAA 4x the Z-buffer is 4 times larger [1].

As mentioned before the difference between multisampling and supersampling is the frequency of execution of the fragment shader, table 2.1 shows a comparison between the techniques. On multisampling, the fragment shader is only executed when the coverage mask is non zero, which means that at least a subsample was covered by a triangle. If all the subsamples in a pixel are covered, the fragment shader location is at the center of the pixel. However, when the pixel is not totally covered, the fragment shader position can be shifted from the center to a position in the pixel which is covered. This process is called centroid sampling and is executed in hardware [3].

**Table 2.1:** Comparison between supersampling and multisampling in terms of samples and antialiasing.

| | Samples | | Antialiasing | |
|---|---|---|---|---|
| | Pixel | Coverage | Spatial | Shader |
| Supersampling 4x | 4 | 4 | ✔ | ✔ |
| Multisampling 4x | 1 | 4 | ✔ | ✘ |

In MSAA the fragment shader result is stored in each of the covered samples. The color buffer size must be larger to hold the shader result in each of the covered subsamples, the Z-Buffer size is also larger storing the depth per subsample. This is needed because a pixel might have multiple triangles overlapping it, and the subsamples can have different color values. Figure 2.5 shows how the subsample colors are stored.

Until this point, we have a color buffer with more samples than the output can display. The frame buffer has to be downsampled similarly to what is done in supersampling. In MSAA this process is called *resolve*, the most common method is to average the subsample colors to obtain the final pixel

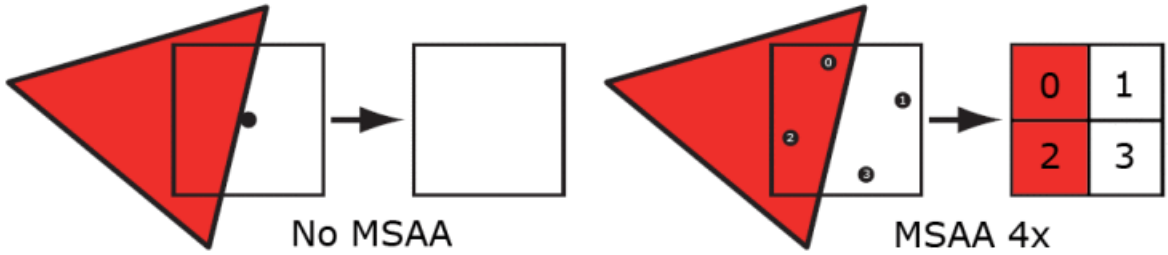


**Figure 2.5:** Example of the color buffer in no MSAA and MSAA 4x when a triangle doesn't fully cover a pixel. Image is taken from article [1]

color [1]. In this case, the subsamples have equal weights in the final color calculation. There are other methods of resolving the final color. Nvidia in the era of DirectX 8 and 9 used a sampling pattern called quincunx [1], this and other patterns can be seen in figure A.1. The pattern is composed of 4 samples on the edges of the pixel and 1 in the center, resembling a cross. The resolve was done by giving $\frac{1}{2}$ weight to the center sample and $\frac{1}{8}$ weight to the corners, this scheme alleviated aliasing but displayed too much blur and loss of details [1].

To summarize, multisampling provides a good balance between image quality and performance. It focuses on sampling the coverage at a higher rate and sharing the fragment shader result. MSAA is executed on the GPU and allows parametrization of the number of subsamples, OpenGL and Direct X have support for it.

## 2.2 Post-processing antialiasing

Post-processing is the process of applying effects to an image. Examples of such effects are ambient occlusion, depth of field and bloom.[3]

Antialiasing can also be performed as a post-processing step. This type of antialiasing has gained popularity since the appearance of Morphological Antialiasing (MLAA) developed by Intel [2]. Two post-processing antialiasing techniques are explained in this chapter: fast approximate antialiasing (FXAA) and sub-pixel morphological antialiasing (SMAA). One of the advantages of post-processing antialiasing algorithms is that the memory and processing cost on desktops is normally lower than the techniques described in the previous chapter. This is because post-processing techniques only use the information of the frame, unlike MSAA which increases the number of samples. The same cannot be said for mobile devices, due to their architecture MSAA 4x can be executed with minimal performance drop, this statement will be explained in more detail in section 2.4.1. Another advantage of post-process is that shader aliasing is also corrected.

[3] Additional examples of post-processing effects can be checked on the Unreal Engine website `https://docs.unrealengine.com/latest/INT/Engine/Rendering/PostProcessEffects/` (as consulted on June 15, 2018)

Nowadays rendering techniques are shifting towards deferred shading architectures. For example, graphical engines like Unreal Engine[4] or Unity[5] recently added support for such techniques. Post-processing techniques can be easily integrated alongside deferred shading. However, the same cannot be said for multisampling. Post-processing techniques, in general, apply a filter on an image to reduce aliasing artifacts. The drawback of filtering techniques is that the results might be more blurry, oversampling techniques perform better in this aspect [2]. The blur comes from the fact that filtering techniques only use the information from the color and depth buffer. The best these techniques can do is to smooth the artifacts present in the frame. In the next sections, we will see that FXAA and MLAA are effective to fight aliasing.

### 2.2.1 FXAA

Fast approximate antialiasing (FXAA) is an open-source technique developed by Nvidia [5], which consists on finding the edges in an image and applying a filter to smooth them. FXAA takes as input an image and returns an antialiased version of the image. It is designed to run on a single pass on the fragment shader. The shader takes as input a texture containing the image to perform antialiasing on, this texture must have bilinear filter access.

#### 2.2.1.1 Bilinear Filtering

Texture accesses are done with UV coordinates, when accessing a position with bilinear filtering enabled the result is a weighted blend between the adjacent texels [6], the weightings depend on the distance from the sample point to the center of each texel (figure 2.6). Bilinear filtering is a requirement for FXAA to work.

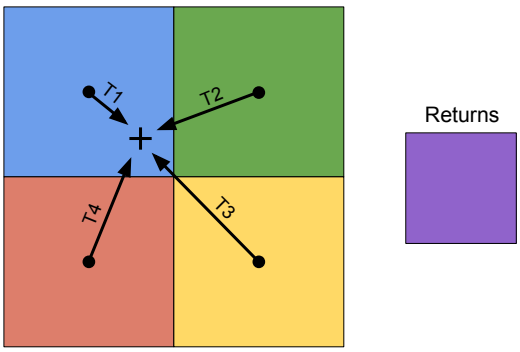


**Figure 2.6:** Example of texture fetching with bilinear filtering. The smaller the distance from the UV coordinates to the texel center the more it contributes to the resulting color.

[4] https://www.unrealengine.com (as consulted on June 15, 2018)
[5] https://unity3d.com (as consulted on June 15, 2018)

#### 2.2.1.2 Luminance formula

All the calculations values of FXAA are based on the relative luminance (luma) of the pixels. The luma formula is the following:

$$luma = 0.299 * R + 0.587 * G + 0.114 * B$$

This formula gives more weight to the green channel because the eye is most sensitive to green light and least sensitive to blue light[6].

The source code to calculate the luma value is shown on listing 1.

```
float rgb2luma(vec3 rgb){
    return sqrt(dot(rgb, vec3(0.299, 0.587, 0.114)));
}
```

**Listing 1:** FXAA code snippet to convert the RGB color value to luma.

We use square root in the code as an approximation to convert from linear RGB color to perceived color space(sRGB). This process is called gamma correction[7], it makes darker tones of color easier to differentiate on the screen. FXAA is intended to work with textures in sRGB color space [5].

The next steps are executed independently for each pixel. To explain how FXAA works we will use a black and white pixel grid as the example, see figure 2.7. The outlined pixel in red is the current one being post-processed.

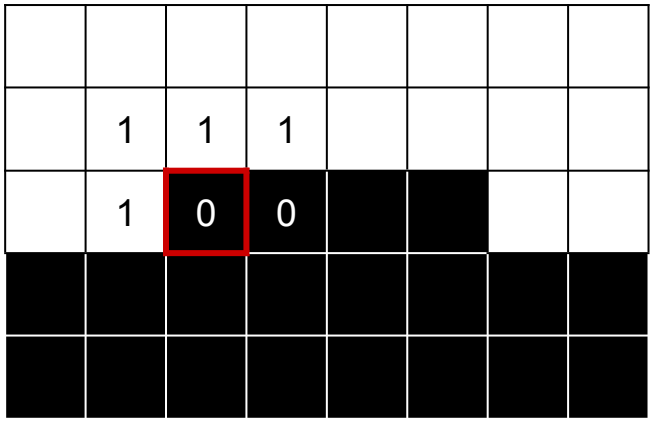


**Figure 2.7:** Example 8x5 pixel grid to demonstrate how FXAA works. The values present on the pixels corresponds to the luma at that pixel.

[6] http://blog.simonrodriguez.fr/articles/30-07-2016_implementing_fxaa.html (as consulted June 15, 2018)
[7] A more detailed explanation about gamma correction can be found here: https://www.cambridgeincolour.com/tutorials/gamma-correction.htm (as consulted on June 15, 2018)

#### 2.2.1.3 Local contrast check

First, we fetch the color of the current pixel from the texture and calculate the luma, this is also done for its four neighbors (up, down, left and right). Then we calculate a delta of those luma values, if the delta of the lumas is above a defined threshold it means that the contrast is too high between the pixels thus needs antialiasing. Otherwise, in low contrast, the aliasing is less visible and in this case, we don't perform anything and leave the color as it is [5]. The threshold values can be variable, changing thresholds can optimize the algorithm by exiting early, on the other hand, more permissive thresholds improve the resulting image quality [5].

In our example, the maximum luma of the center pixel and its four neighbors is 1 and the minimum is 0, thus the delta between lumas is 1. This delta is higher than the threshold for high-quality preset (0.125), so antialiasing will be performed for the current pixel.

#### 2.2.1.4 Edge test

The next step is to find the edge orientation, FXAA can filter horizontal and vertical edges. To determine the orientation all the 8 neighbors lumas are calculated. Then an estimated gradient is calculated vertically and horizontally, the higher value is the edge orientation.

The gradient formulas use the luma of the neighbor pixels, the formulas are the following: [5]

$$\begin{aligned} horizontal = &|(upleft - left) - (left - downleft)| + 2 \cdot |(up - center) - (center - down)| \\ &+ \ |(upright - right) - (right - downright)| \end{aligned}$$

$$\begin{aligned} vertical = &|(upright - up) - (up - upleft)| + 2 \cdot |(right - center) - (center - left)| \\ &+ \ |(downright - down) - (down - downleft)| \end{aligned}$$

In the example $horizontal = |(1-1)-(1-0)|+2\cdot|(1-0)-(0-0)|+|(1-0)-(0-0)| = 1+2+1 = 4$ and $vertical = |(1-1)-(1-1)|+2\cdot|(0-0)-(0-1)|+|(0-0)-(0-0)| = 0+2+0 = 2$, $horizontal > vertical$ so in this case the edge is horizontal.

#### 2.2.1.5 Edge direction

At this point we know the edge orientation, the next step is finding the edge direction it can be above, below, to the left or to the right of the current pixel. To figure the direction we just need to calculate a gradient between the center pixel and the top/down neighbor for horizontal edges or left/right for vertical edges [5]. The edge lies on the border between the pixels where the gradient has the higher value. In

the example the first gradient between the center and top pixel is 1, the second gradient between center and bottom is 0, the first has a higher absolute difference, so the edge is above the pixel. Now we offset the UV coordinates by half a pixel in the edge direction to land exactly on the edge, as depicted in figure 2.8.

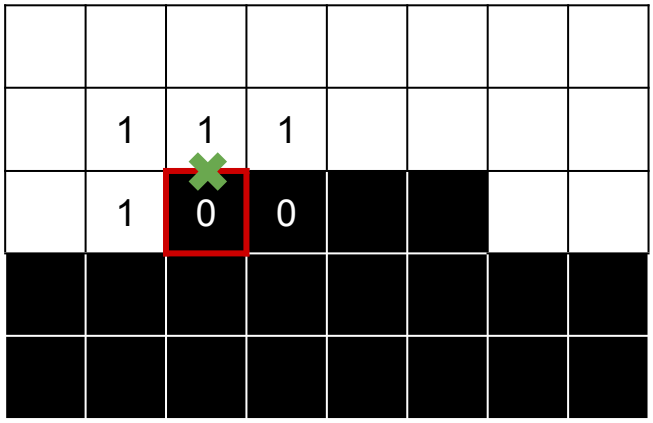


**Figure 2.8:** The found UV coordinate at the edge of the initial example.

#### 2.2.1.6 Iterating along the edge

Now we iterate along the edge to find the distance in both directions from the center of the edge to the extremities. The luma value is calculated at each iteration position, if the luma delta is below the edge gradient value then the iteration stops and the position is saved. The iteration step can be variable, the lower the step distance the higher the quality however it hurts the performance. A good strategy to speedup FXAA is to start increasing the step distance after some iterations, e.g., start with 1-pixel step and after 5 iterations increase the step to 1.5 pixels and so on. The number of iterations is fixed to accelerate the process. Another reason for a low number of iterations is that if both extremities are very far, the precision of the next calculations will not be impacted significantly. In the medium-quality preset of FXAA, the maximum number of iterations are 8.

The iteration step in the example is shown in figure 2.9.

#### 2.2.1.7 Calculating the offset

To calculate the offset we use the edge length and the length from the edge center to the closer edge extremity. The formula for the pixel offset is the following: [5]

$$offset = \frac{-min(edgeDistance1, edgeDistance2)}{edgeLength} + 0.5$$

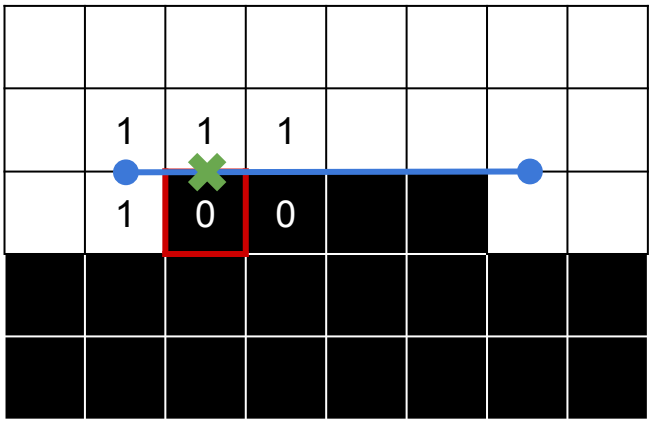


**Figure 2.9:** Iteration along the aliased edge.

In the example the edge length is 5 and the edge distances from the center are 1 and 4, it give us the following: $offset = -min(1,4)/5 + 0.5 = 0.3$

#### 2.2.1.8 Final read

Finally, we can offset the initial UV coordinate and with bilinear filtering obtain from the texture an antialiased color for the current pixel. Figure 2.10 depicts the location of the final UV coordinate.

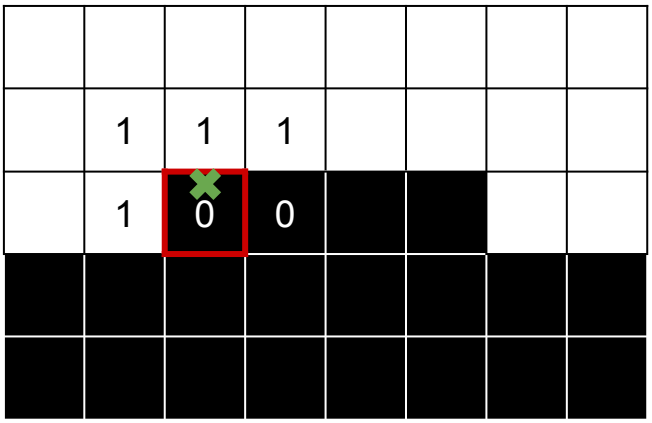


**Figure 2.10:** UV coordinate location with the added offset marked with the cross.

The snippet for this step can be viewed on listing 2.

Notice that bilinear filtering is crucial for FXAA to work, adjusting the UV coordinates change noticeably the read color from the texture.

In the example the pixel final color is $0.7 * 0 + 0.3 * 1 = 0.3$ or (76,76,76) in RGB. Figure 2.11 shows the result after the technique is applied to all the pixels. A more realistic use of FXAA can be seen in Figure 2.12.

```
vec2 finalUv = UV;
if(isHorizontal){
    finalUv.y += finalOffset * stepLength;
} else {
    finalUv.x += finalOffset * stepLength;
}

vec3 finalColor = texture(screenTexture,finalUv).rgb;
fragColor = finalColor;
```

**Listing 2:** FXAA snippet to read the resulting color from the offset position

**Figure 2.11:** FXAA visual result for the 8x5 grid example after all pixels are computed. In this resulting image the edge is smoothed and the colors vary less abruptly.

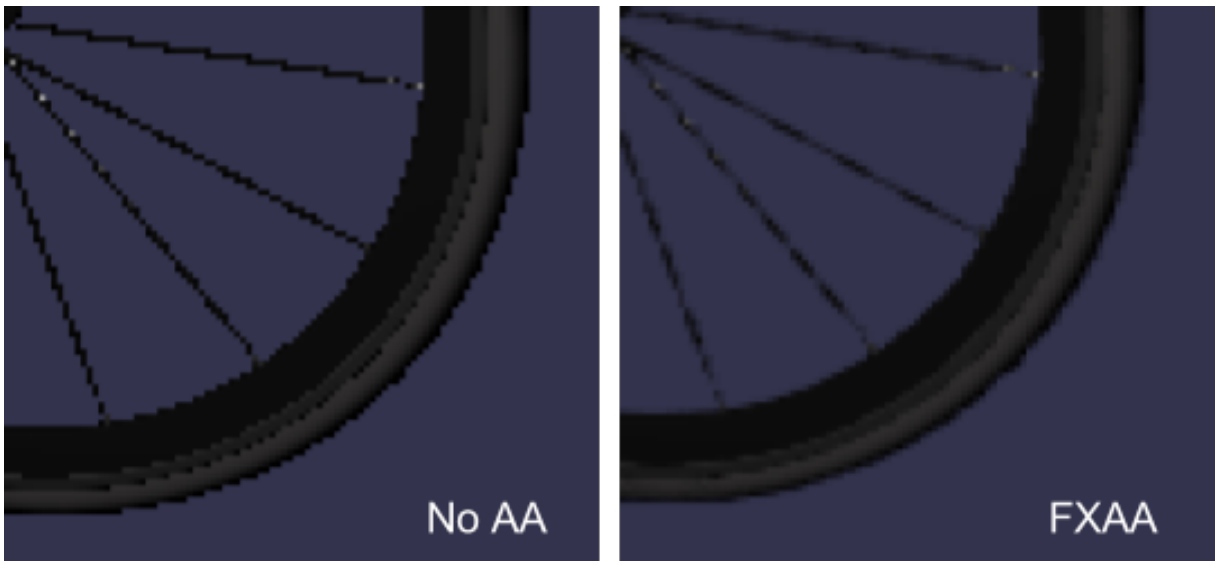


**Figure 2.12:** Close up of a rendered scene with no AA and FXAA. The first image has a lot of jaggies, the second has smoother diagonals and curves.

As a summary, FXAA is a fast antialiasing technique. It runs as a post-processing step and its calculations are based on the luminance of the pixels and the contrast between them. The parameters of FXAA are customizable, allowing the developer to choose between quality and performance. An unwanted side effect of FXAA is texture blur, the technique will also filter inside object textures making them lose some details.

### 2.2.2 SMAA

Enhanced Subpixel Morphological Antialiasing (SMAA) is a shader based antialiasing technique, it is a work made upon other technique called Morphological Antialiasing (MLAA) developed by Intel. SMAA improves some areas of MLAA and goes even further with new features like improved edge detection, sharper geometric features and diagonal edges handling [2]. Their extended list of pattern classifications allows for better reconstruction of silhouettes. SMAA is a complete antialiasing solution, it allows the use of multisampling for better results and allows the use of temporal reprojection for temporal antialiasing. On the most basic configuration of SMAA, the results are comparable to MSAA/SSAA, the execution time is around 1 ms on desktop, allowing its use in 3D applications without affecting the performance too much [2].

#### 2.2.2.1 MLAA

SMAA is based on MLAA, some steps of MLAA were improved or redefined in this new technique. MLAA processes the image in three steps: [7]

1. Find discontinuities between adjacent pixels.
2. Identify predefined patterns.
3. Blend colors in the neighborhood.

MLAA first step is finding the edges on the image, to find that the RGB value of the pixels is used, SMAA improved this step [2], we will explain it in the next section. The second step is classifying the discontinuities, there are three patterns considered: Z, U and L, this step is illustrated in figure 2.13.

On the final step, the Z and U shapes are the decomposed into simpler L shapes. We can reference an L shape as a set of three vertices. To calculate the blending weight we connect the center of the secondary edge with the remaining vertex of the primary edge, see the red line on figure 2.14.

The line connection splits the pixels into trapezoids, each trapezoid area defines the blending factor for that pixel. To compute the new color the following equation is used:

$$c_{new} = (1 - a)\, c_{old} + a\, c_{opposite}$$

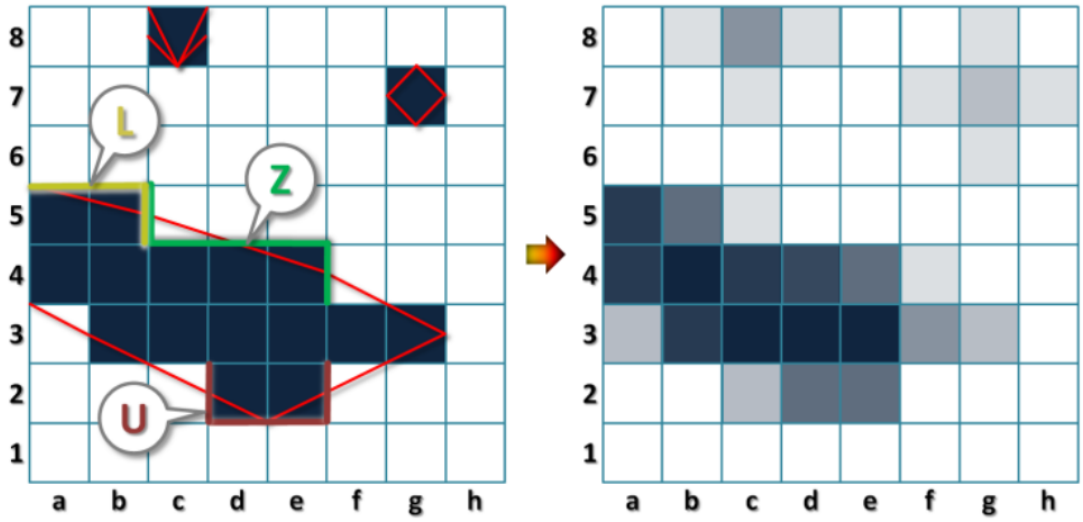


**Figure 2.13:** MLAA processing on a black-and-white image, the image on the left illustrates the second step where the patterns are shown. Image taken from the original paper Morphological Antialiasing.

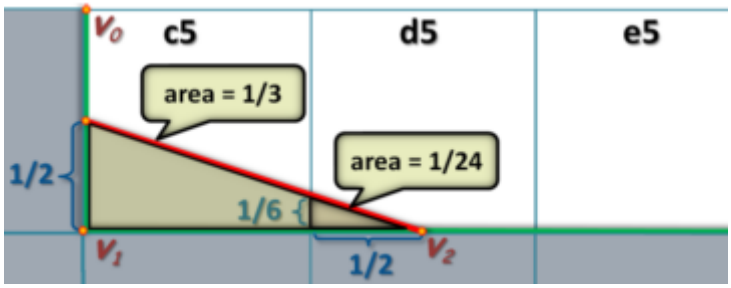


**Figure 2.14:** Weight calculation based on the area of the trapezoid intercepting the pixels. The L shape is defined by $v_0v_1$, the secondary edge and $v_1v_2$ the primary edge.

In the example of figure 2.14 the new color of pixel *c5* is calculated as $2/3 \cdot 0 + 1/3 \cdot 1$ (assuming the color below *c5* is black).

The MLAA technique was originally implemented on CPU, Jimenez ported it to be executed on GPU and did an important change, instead of using the three defined Z, U and L shapes they were changed to precomputed classification textures [2]. This change improved the general performance and increased the number of patterns handled.

#### 2.2.2.2 Edge detection

Edge detection was one of the areas improved in SMAA, it is important in antialiasing filtering techniques because undetected edges are not smoothed. SMAA allows the use of various strategies to search for edges: RGB color, luma, depth, normals, object ids or a combination of them. Luma is the recommended edge detection because the color information is always available, shader aliasing is also corrected and calculating luma deltas is faster than RGB color deltas [2].

The edge detection used in SMAA avoids soft edges that most humans can't distinguish, instead of just checking the direct neighbors like most filtering AA techniques, the edge searching area is larger. This way more edges are detected and in case there are multiple edge discontinuities, the most dominant is handled. Figure 2.15, left image shows the edges considered (marked in blue) for finding left edges,

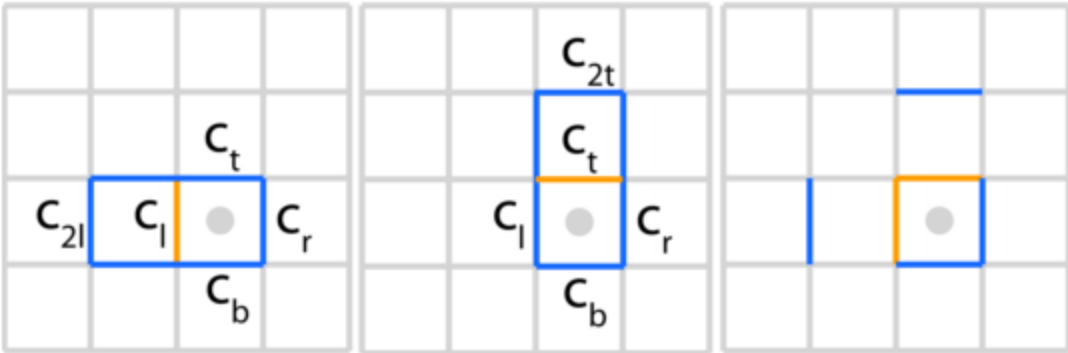


**Figure 2.15:** Edges considered for the edge detection of SMAA. The left image shows the case for finding left edge, the center for finding top edges and the right image shows all edges considered. Taken from the original SMAA paper [2]

for the given pixel (grey dot) notice that not only the neighbor edges are considered but also the edge $c_{2l}$ this helps in case there are two parallel edges, the most dominant is chosen. In the left case the maximum contrast $c_{max}$ is calculated for all the edges and is compared to the left edge contrast $c_l$, if the latter is above $0.5 \cdot c_{max}$ then it is considered an edge, otherwise it is not an edge. The threshold of 0.5 was chosen empirically by the authors as the tests with that value had good results [2]. To keep the memory accesses low the technique only retrieves a subset of the edges for both left and top cases, present on figure 2.15 right image. There is other condition to determine if a edge is active or not, $e_l$ is a boolean value for that condition, it is calculated as follows: $e_l = |L - L_l| > T$, $L$ is the luma value at the current pixel and $L_l$ is the luma at left neighbor pixel, T is the threshold that depends on the preset quality chosen (ranges from 0.05 and 0.2). The expression for finding edges on the left case can be put as:

$$c_{max} = max(c_t, c_r, c_b, c_l, c_{2l})$$
$$e_l^{'} = e_l \wedge c_l > 0.5 \cdot c_{max}$$

In the expression $c_t, c_r, c_b, c_l, c_{2l}$ are luma deltas calculated between adjacent pixels, $e_l^{'}$ is the final boolean value that represents if the edge is active or not. Top edges $e_t^{'}$ are calculated in the same manner.

#### 2.2.2.3 Pattern handling

Pattern detection was improved in comparison to MLAA, this new version preserves more geometry features like diagonals, corners and long edges. To classify correctly sharp corners SMAA uses a two-pixel-long edge scan to process aliasing but maintain the appearance close to original shape.

Diagonal edge detection and handling is a novelty of this technique, most post-processing solutions focus solely on horizontal and vertical edges. To detect diagonals a precomputed texture is used, using the distance on both ends of the diagonal and the ending pattern of the diagonal the texture returns the corresponding area for blending, see figure 2.16 (b).

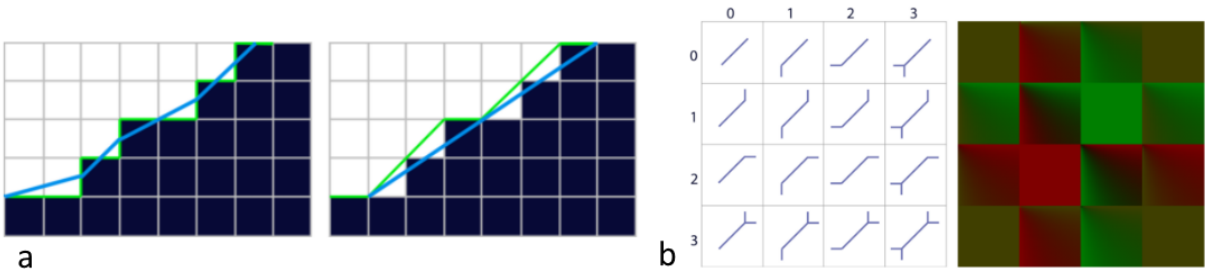


**Figure 2.16:** (a) On the left the original MLAA handling of diagonals and on the right the new technique with sharper diagonals (blue line) (b) Diagonal patterns handled by the technique on the left and the corresponding areas on the right. Taken from the SMAA paper [2]

The diagonal pattern detection occurs before horizontal and vertical (orthogonal) in the area calculation. The orthogonal test is only done in case the diagonal test fails. This way the blending step works independently of the orientation of the edges [2].

#### 2.2.2.4 Subpixel details and temporal antialiasing

Filtering techniques like MLAA work with one color sample per pixel, subpixel features can't be recovered with just that information. SMAA allows the use of multi-sample strategies along with their technique. Combining both filtering and multi-sampling improves the image quality and show subpixel features, SMAA S2x on figure 2.17 is more detailed than MSAA 4x.

Temporal antialiasing is a feature also available in SMAA, by using techniques like per-frame jitter, and temporal reprojection, more subpixel details are obtained and frames are temporally stable. This theme will be explored in the next subsection.

To summarize, SMAA is a complete antialiasing solution. It was based on MLAA and improved in many areas: edge detection, diagonal handling and subpixel features. The multiple AA modes allow temporal reprojection or using multi-sampled buffers, the code is open source[8] and can be integrated with the popular APIs: OpenGL and DirectX.

[8] https://github.com/iryoku/smaa (as consulted on June 15, 2018)

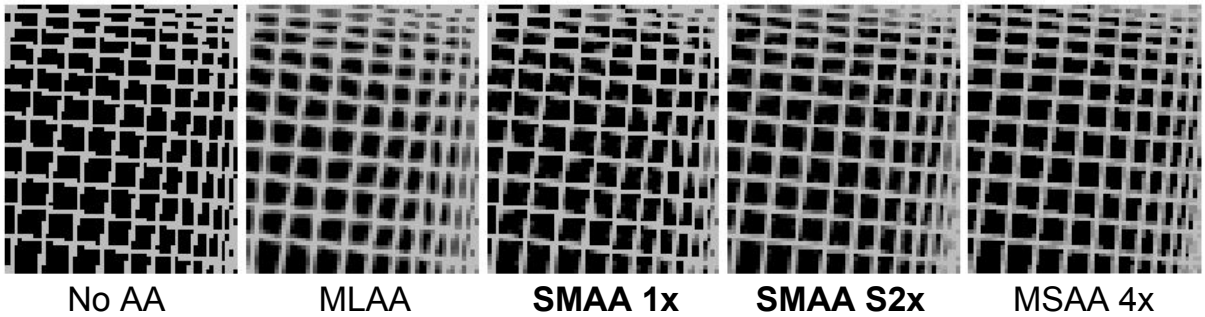


**Figure 2.17:** A difficult case with many artifacts and the results from SMAA and other techniques. Figure taken from SMAA paper [2]

### 2.2.3 Temporal Antialiasing

Temporal antialiasing (TAA) is a recent technique that reduces geometry and temporal artifacts. There are multiple implementations of this technique, the main idea of TAA is using the information of priorly rendered frames to perform visual smoothing on new frames. TAA displays a motion blur effect and prevents the artifact of flickering pixels. Lately, this technique has gained a lot of attention for its film-like features, it has been implemented in Unreal Engine and other game engines [8].

Unlike geometry antialiasing which increase samples spatially, temporal AA spreads gathers over time. Figure 2.18 shows the main idea of this technique.

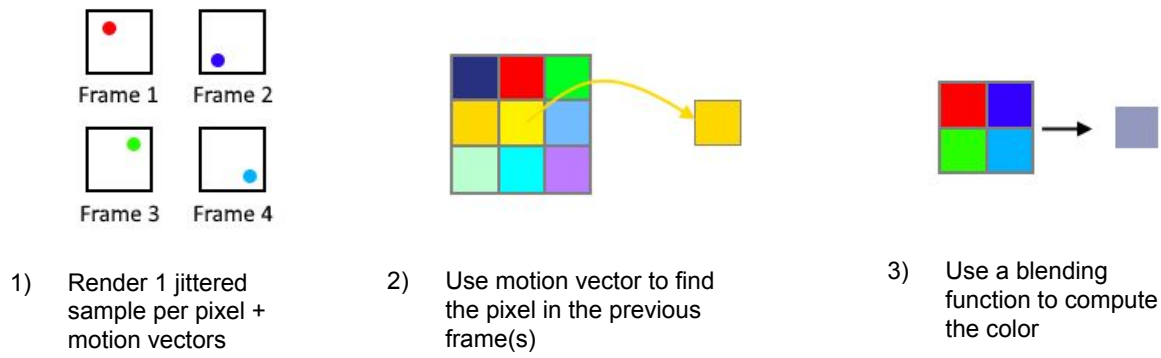


**Figure 2.18:** Temporal antialiasing idea in steps.

First, take one jittered sample per pixel and the motion vector for each pixel. The motion vector tells us where each pixel came from in the previous frame, for example, it can be the camera movement vector. Then whenever a frame is generated, all pixels must be matched with the pixels from the previous frame using the motion vectors. After a defined set of frames a blending function is used to calculate the final color, normally this function gives more weight to pixels from more recent frames.

This algorithm remains efficient because only 1 sample is taken per pixel. Static scenes benefit from

TAA because of the per-frame jitter, the accumulation of samples at different positions improves the pixel detail. For dynamic scenes there are some open problems, a naive approach would be to perform linear blending between frames but it does not work well, it introduces a trail behind objects known as ghosting, see figure 2.19.

**Figure 2.19:** Ghosting artifact when linearly blending at low frame-rates.

There are many problems with this technique, for example, a moving object might disappear between frames, a light source might be turned off or particles can be spawned. Tiago Sousa introduced velocity weighting to solve some problems with this technique [9]. The idea is that if the velocity changes abruptly between frames then the history for that pixel is probably invalid. Just velocity weighting is not enough to fix all artifacts, Unreal Engine TAA version uses depth comparison to determine the correctness of the previous frame. Brian Karis introduced a check called neighborhood clamping, the idea is that if the history RGB value is not within the min/max of the 3x3 neighborhood then the history is discarded [8]. Besides neighborhood clamping there are a lot more tweaks that are done in TAA for good results, this type of antialiasing needs a lot of perceived tuning which can be time-consuming.

There is not much available information about temporal antialiasing performance, however, its presence in game engines demonstrates that the performance and quality are as good or better than other solutions.

## 2.3 Comparison

In this section, we present a comparison between the algorithms described before. In our opinion each of the techniques has their own strengths, some excel better in some aspects than others. For example, MSAA obtains images with subpixel details but it is not easy to integrate on a deferred rendering engine, on the other hand, FXAA is easy to integrate with any rendering technique but the quality of the

images are not as good.

**Table 2.2:** Comparison between antialiasing algorithms. We consider algorithms to be post processing, in case that they are not involved in the rendering pipeline, these algorithms only act upon the generated image. Memory footprint row defines the number of samples in the color buffer, depth buffer and additional render targets. * Some SMAA modes can be used with multisampling so they can display subpixel details.

| | SSAA 4x | MSAA 4x | FXAA | SMAA | TAA |
|---|---|---|---|---|---|
| Subpixel details | yes | yes | no | no* | yes |
| Post-processing | no | no | yes | yes | yes |
| Rendering technique | any | forward | any | any | any |
| GPU acceleration | | yes | | | |
| Geometry AA | yes | yes | yes | yes | yes |
| Temporal AA | | | | optional | yes |
| Input (color buffer) | | | 1x | 1x (2x*) | 1x |
| Memory Footprint | 4x/4x/1x | 1x/4x/1x | 1x/1x/1x | 1x/1x/2x | 1x/1x/2x |
| Performance | | 2.3ms | 0.62ms | 1.02ms | |

Table 2.2 shows the characteristics, inputs and performance of the AA techniques. The performance results for MSAA 4x, FXAA and SMAA were obtained from the SMAA paper [2], the post-processing algorithm times do not include the frame render time, which was 1.57ms. The tests were executed on Nvidia Geforce GTX 470 for 1080p frames. According to Tiago Sousa research on TAA, it runs on PC at 1080p costing 0.2ms on Crysis 2 [9]. The performance time for TAA, however, can vary a lot depending on the quality. SMAA temporal antialiasing is optional, it only works in some modes of the technique, using temporal in conjunction with edge AA is common in games.

From the data of the table 2.2 FXAA is the fastest technique, SMAA comes in second and MSAA 4x in last. The image quality of SMAA is better than FXAA as it handles more aliasing cases, FXAA focuses on executing fast and approximating the image results.

## 2.4 Tiled-based rendering

The goal of this thesis is to evaluate the antialiasing techniques on the mobile environment, mobile architecture differs from the desktop as there are some constraints that desktops do not have.

In desktops, GPUs use an architecture called immediate mode, in this architecture the fragment shader is computed on each geometric primitive sequentially. Each geometric primitive is fully rendered before starting a new one. The working set data for this architecture takes a lot of memory, needed by the color, depth and stencil buffers. It is necessary to access RAM to store these buffers. While this approach works on desktops, the data transfers between GPU and RAM are very expensive, tiled rendering architecture solves this problem on mobile devices.

### 2.4.1 Tiled Rendering

Tiled rendering architecture is designed to reduce the amount of data being transferred from GPU to external memory, it works by dividing the frame into subframes or tiles and each tile is rendered independently after all tiles are rendered they are joined together to make the full frame [10].

Modern tiled architectures are divided into two steps: geometry processing and fragment processing. First, all the geometry processing is done for the scene, producing the information for the pixel shading phase. Also in this step, the screen is divided into small tiles. Tiles can have different sizes, the Mali GPU uses 16x16 pixel tiles[9]. In Qualcomm mobile GPUs the number of tiles is calculated by dividing the final framebuffer size by GMEM (specialized graphics memory). PowerVR GPUs have a more efficient tiling algorithm, they generate their tile list only where the geometry cover area of the tile, see figure 2.20.

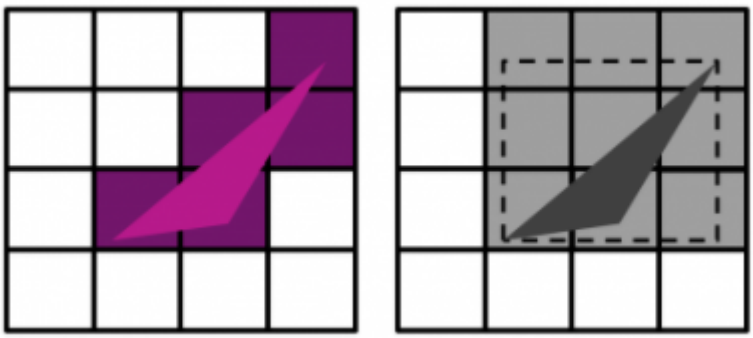

**Figure 2.20:** PowerVR perfect tiling algorithm on the left and bounding box tiling on the right.

In the fragment processing phase, each GPU core processes one tile at a time, until all pixel colors are calculated, then the tile is saved in RAM, this is done until all tiles are complete.

See figure 2.21 for a diagram and data flow between GPU and RAM.

The tiled rendering architecture is transparent to software developers, as it is done on the GPU, there are many advantages. Because tiles are small, the working set memory of the fragment shader fits on local GPU memory, without needing to access RAM until the tile is completed. A good improvement over immediate rendering because accessing RAM is slow and uses a lot of battery on mobile devices. Another advantage is that since tiles are small, it allows storing extra samples for multisample antialiasing up to 16x, gaining image quality at a very low cost, the ARM Mali GPU contains this technology.

[9] https://community.arm.com/graphics/b/blog/posts/the-mali-gpu-an-abstract-machine-part-2---tile-based-rendering (as consulted on June 15, 2018)

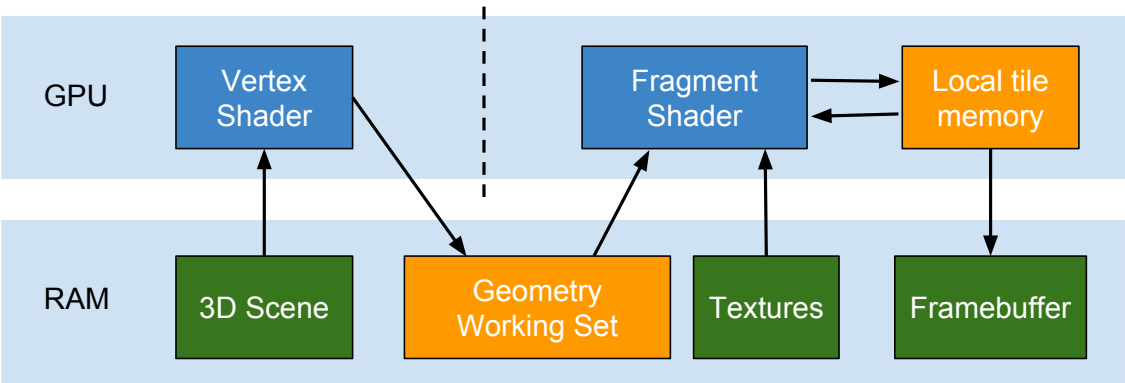


**Figure 2.21:** Tiled renderer phases and data flow. The dashed line separates between the geometry processing and fragment shading processing steps.

# 3 Methodology

In this chapter, we explain how we implemented the application to run the test cases and what metrics we collected to analyze antialiasing algorithms on mobile devices. A screenshot of the application is shown in figure 3.1

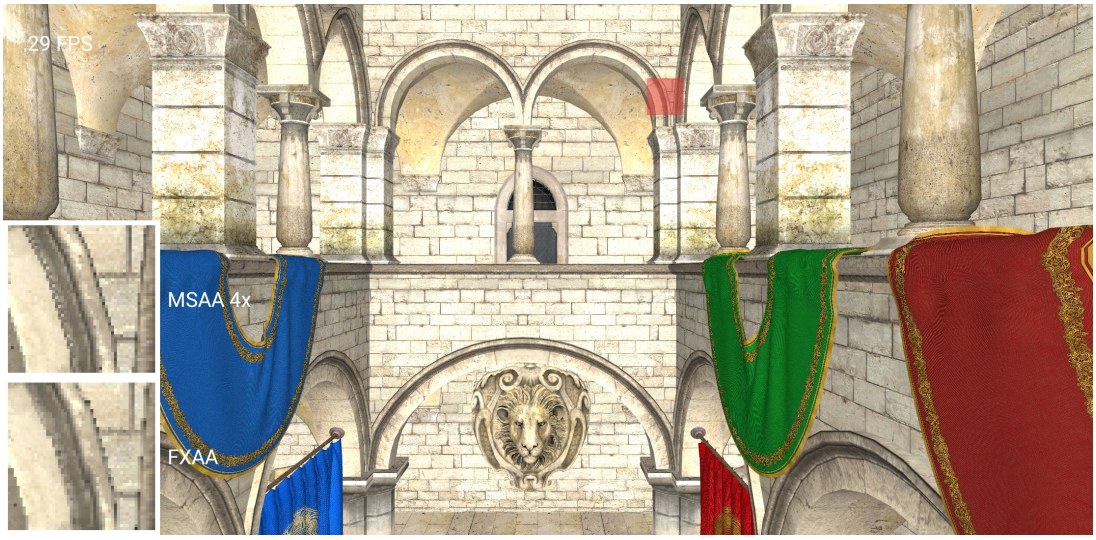


**Figure 3.1:** Screenshot of the application running on Samsung Galaxy S8.

## 3.1 Technologies

There are two main mobile operating system in today's market, Android and iOS from Apple. We decided to create the test application on Android because it is open-source, it is more popular than iOS and it has fewer requirements to develop applications. For example, iOS applications can only be developed on MacBooks while Android applications can be created in Linux, Windows and macOS. In regards to technology, both platforms support OpenGL ES and Vulkan APIS for high-performance graphics rendering.

We chose to go with OpenGL ES because we already had some experience with it, it is very popular for 3D applications and has great support in Android devices. OpenGL ES or GLES is a subset of OpenGL designed for embedded systems like smartphones, tablets and other mobile devices. Its API allows the use of shaders, textures, framebuffers, among other features. Shaders are small programs

created by the developer to run on the GPU since GLES version 3.0 the vertex and fragment shader stages are required in OpenGL applications.

Vulkan is a new technology also offered by Khronos Group[1], that offers low-level API to create 3D applications. The higher performance comes with a cost of complexity for developers, OpenGL comes with more functionality already implemented. The time of learning this new technology would make it difficult to finish the application on the expected date. Also because Vulkan is a recent API some bugs might appear while developing and the support and information available about Vulkan is not as good as OpenGL at the moment.

Instead of OpenGL ES, we could have chosen a rendering engine, like Unity 3D or Unreal Engine, both can run on Android. It would give us some benefits as there are some functionalities already implemented like 3D mesh loading, texture loading, math functions and scene creation. The integrations of the antialiasing techniques on these engines would be easy because most of the AA techniques are ported to these engines and are simple to set up. The abstractions these engines provide have a performance cost, in our experiment we wanted to get the best performance we could get so we chose a more low-level technology.

## 3.2 Application architecture

The technologies that were part of the solution were Android Java API, Java Native Interface (JNI), OpenGL ES version 3.2, OpenGL Mathematics (GLM) and the smartphone that ran our tests was a Samsung Galaxy S8. In the diagram present in figure 3.2, the Android application contains the JNI and native library because everything is packaged in an APK file that is installed on Android devices.

Android Java API provides a set of classes and methods that can be used to build Android applications, we used this API to create the settings screen. Java Native Interface (JNI) enables communication between Java code and native code. We used JNI to call code of our native library (written in C++), this library can render 3D models and run the different antialiasing modes. The antialiasing modes are listed in the next section. This library can load 3D meshes via obj files, mathematical operations such as vectors and matrices are provided by GLM library, the rendering is achieved by using OpenGL ES APIs. The Android phone used in the tests was the Samsung Galaxy S8, a very powerful smartphone with Qualcomm Snapdragon 835 processor and GPU Adreno 540.

## 3.3 Algorithm implementation

The antialiasing modes implemented in the application are the following:

[1] https://www.khronos.org/ (as consulted on June 15, 2018)

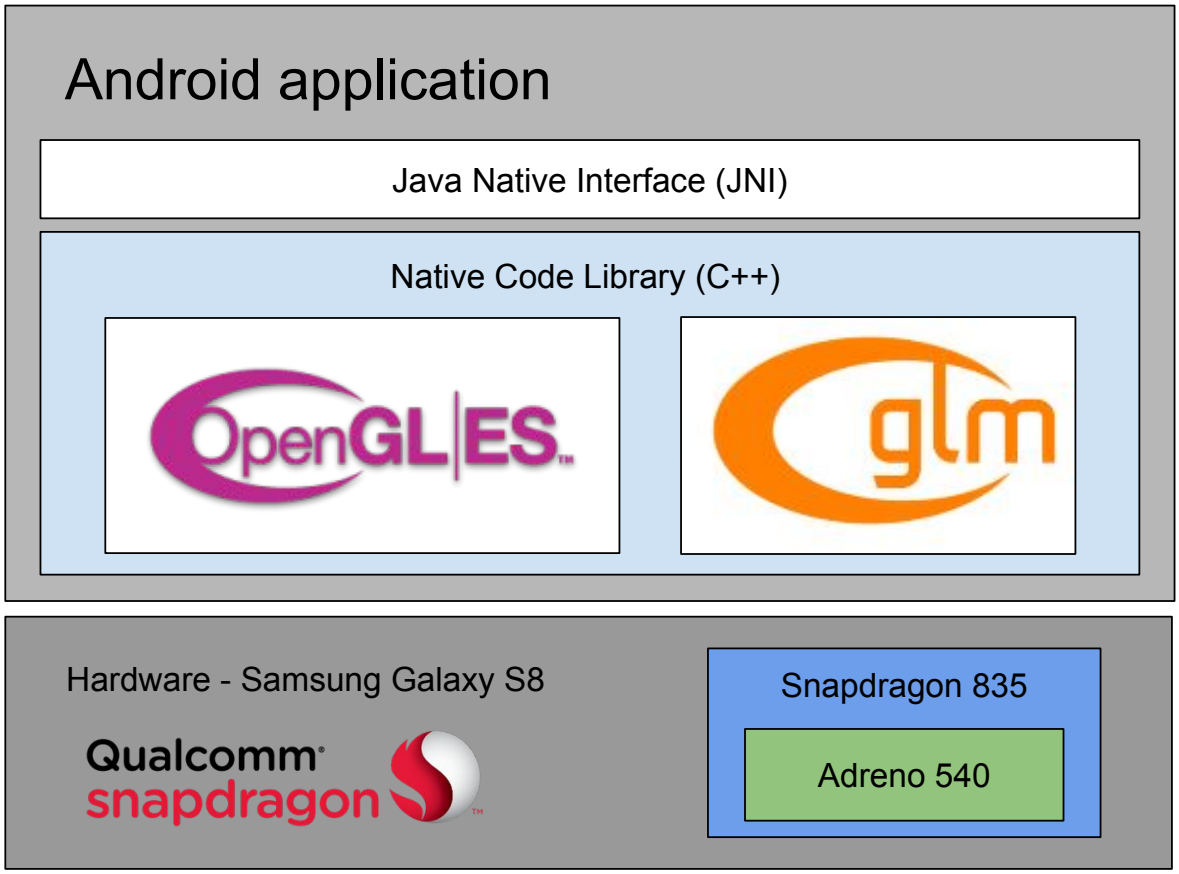


**Figure 3.2:** Test application architecture diagram.

- No AA
- SSAA 4x
- MSAA 4x
- FXAA
- SMAA 1x
- SMAA T2x
- Temporal AA

The implementation of the antialiasing techniques was built with OpenGL ES APIs and open source code from the algorithm authors. We used two rendering passes, first pass for rendering the scene and a second pass that executes different code depending on the antialiasing mode. Each pass has a vertex shader and fragment shader, on the first pass the vertex shader transforms the geometry with the model-view-projection matrix, the geometry on model space is transformed to clip space. The fragment shader

performs Phong lightning with one light source and textures the materials of the scene. The second pass the geometry is a full-size quad that fills the window and the fragment shader code changes accordingly to the antialiasing technique selected.

Supersampling (SSAA 4x) was implemented by rendering the scene with a viewport 4x larger and using a 4x larger framebuffer and then in the fragment shader downscaling the buffer to native resolution. For MSAA we used multisampled framebuffer, a feature that is part of the Open GL ES API and allows to perform multisampling. The API has a parameter to select the number of samples, in our case we chose four samples, check listing 3 for the snippet to activate multisampling on a framebuffer object.

```
glGenTextures(1, &tex);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, tex);
glTexImage2DMultisample(GL_TEXTURE_2D_MULTISAMPLE, num_samples,
                    GL_RGBA8, width, height, false);

glGenFramebuffers(1, &fbo);
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                    GL_TEXTURE_2D_MULTISAMPLE, tex, 0);

GLenum status = glCheckFramebufferStatus(target);
```

**Listing 3:** Code snippet in C++ OpenGL to create a multisampled framebuffer object. Source : https://www.khronos.org/opengl/wiki/Multisampling (as consulted June 15, 2018)

FXAA fragment shader code was available from the author, it receives the color buffer of the previous pass as a texture and the window dimensions, the fragment shader outputs an antialiased image. SMAA also works as a post-processing step, the code is freely available. SMAA has three passes chained together, each one receives as input the result from the previous pass. The first pass is *edge detection*, it receives the scene color buffer and produces a texture with the edges, the result of this step is shown in figure A.6. The second pass is *blending weight*, it returns the blending values of each edge. The third and final pass, called *neighborhood blending*, performs the edge smoothing, the result is an antialiased image. There were some changes that we did to the SMAA shader code file because it was designed for DirectX, for example, in OpenGL the UV coordinates are Y-inverted. In temporal AA, our fragment shader receives as input the current and previous frame color textures and also the velocity texture. In the shader we blend the current frame with the previous frame, we look up each pixel on the previous frame with the velocity texture and blend them together. Each frame has a jitter offset on the camera frustum, [(0.25, -0.25), (-0.25, 0.25)] different each frame, using jitter helps to remove antialiasing on static scenes. Our implementation of temporal AA is not ghosting-free and is far from being usable in a real application, still, the main concepts, of per frame jittering and velocity buffers are implemented.

## 3.4 Benchmark scenes

The 3D scenes that we use for our tests are important to find the strengths and weaknesses of the antialiasing techniques. The scenes should have a lot of thin geometry and diagonals, high vertex count and textures, these features are prone to aliasing. Figure 3.3 shows the two scenes that were used in the application.

**Figure 3.3:** Example render of the 3D scenes used, on the left Crytek Sponza and Hairball on the right.

Crytek Sponza is a 3D scene with very detailed textures, high triangle count and curved geometry. Post-processing antialiasing is known to use blurring as a way to mitigate aliasing with the downside of also filtering texture details. Sponza model is a good scene to analyze how the techniques perform on textured models. Hairball mesh has a huge number of polygons with very fine geometry, perfect to analyze spatial aliasing. Both scenes are really demanding on mobile hardware, the number of vertices is 153635 for Crytek Sponza and 1032654 on Hairball. The meshes were obtained on the McGuire Computer Graphics website[2]. Crytek Sponza average FPS is 60, that is the limit FPS value on the Android platform to minimize battery usage, the Hairball model runs at average 35 FPS.

## 3.5 Test variables and metrics

To compare the antialiasing techniques we used various metrics extracted from the application or from the smartphone. The test cases vary on three aspects: 3D scene used, resolution and antialiasing technique used. We used different resolutions: Native (2220x1080) and 720P (1280x720). We tried different resolutions because smartphones have high pixel density, so applying antialiasing at lower resolutions might show good image quality with lower performance requirements.

The AA techniques that are most important for our benchmarks are MSAA, FXAA, SMAA and Temporal AA. The no AA and SSAA 4x modes were also tested to serve as the reference for the other tech-

[2] http://casual-effects.com/data/ (as consulted on June 15, 2018)

niques, for example, the no AA FPS results are higher than the others. Supersampling mode (SSAA) obtains the best image quality possible and serves as a reference of the quality for the other AA techniques.

In each test case, we collected performance and image quality metrics. We implemented a camera rotation animation, around the Y-axis, in order to show various parts of the scene, during the performance tests this animation was active. The performance metrics were collected directly on the application and by using the Snapdragon Profiler. Frames per second (FPS) metric was calculated in the application and sent to output by logs. FPS is an important metric because it indicates the overall performance of the running application. Applications that have low FPS are perceived as slow and most of the times unusable.

The other metrics were collected from the Snapdragon Profiler, this PC application connects to an Android device with Snapdragon processor. The application allows viewing in real-time various metrics from the phone's CPU, GPU and Memory usage. The metrics collected from the profiler can be divided into two categories: GPU shader processing and GPU memory stats. The GPU shader processing metrics considered are:

- % Shaders busy
- % Time ALU working
- % Time EFU working
- % Time shading fragments
- Number of fragment instructions

The % Shaders busy indicate the percentage of time that all the shader cores are busy. A technique that uses all the shaders might be an indicator that it is performing well. The other metrics are important to measure ALU operations, EFU which are operations like square root, exponentiation, sine, cosine, among others. The percentage of time shading fragments is a good way to know if the technique is spending more time in the fragment shader or in the vertex shader.

The GPU memory stats metrics are also important to take note because in tiled architectures memory can be a bottleneck that can affect performance. The GPU memory stats that we collected are:

- Average Bytes / Fragment
- Read Total (Bytes/Second)
- Texture Memory Read BW (Bytes/Second)

These metrics provide a good overview of the memory that is transferred from external memory and textures. GPU temperature was recorded during the tests to know if there are relevant temperature differences between the techniques. High GPU temperatures are bad because it can lead to throttling and degradation of performance. In terms of usability, the temperature is important in mobile devices, high temperature can be noticed by the users of the device.

To measure image quality we captured images of the application running on the various antialiasing modes. We used the Structural Similarity Index (SSIM) to measure the quality of the images produced. The SSIM method evaluates two images in groups of pixels, each group is quantified by the luma, contrast and structure, all groups accumulate towards the resulting value. SSIM gives us a number between -1 and 1, based on the similarity of two images, it approaches the value 1 if both images are the same. We calculated the SSIM between a reference image, supersampled 4x, and each of the images produced by the antialiasing techniques. To calculate this metric we used ImageMagick a software that allows performing a variety of image operations.

# 4

# Results and Discussion

This chapter presents and discusses the results obtained during the execution of the various tests with the objective of studying each antialiasing technique's performance and image quality.

The goal of this evaluation was to study the behavior of antialiasing techniques on mobile devices. There were two aspects that we considered important to analyze and compare among the antialiasing techniques:

- Checking the performance impact of the techniques, discover which ones execute faster and behave well on mobile.
- Comparing image quality, use quality metrics to compare the visual result of each technique.

Figure 4.1 and 4.2 display the scenes that were used in the tests, Crytek Sponza and Hairball.

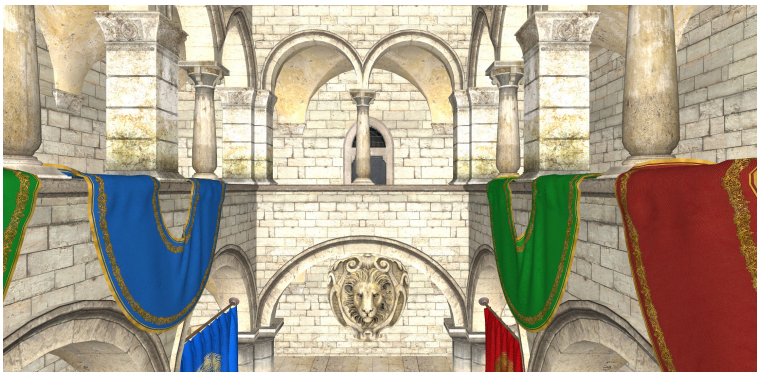

**Figure 4.1:** Screenshot of the Crytek Sponza scene rendered on the smartphone.

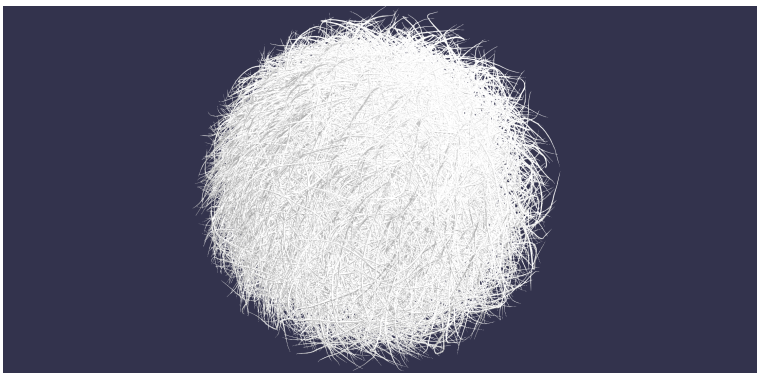

**Figure 4.2:** Screenshot of the Hairball scene rendered on the smartphone.

The test application aimed to simulate a 3D environment of a real application. Crytek Sponza performs well without antialiasing, Hairball, on the other hand, puts a lot more pressure on the GPU. One thing that has to be taken into consideration is that the test application only function is rendering, a real 3D application has lot more functionality, for example, physics or game logic. The number of vertices on Hairball is much higher than what is being used on high-quality games, Unreal Engine has a limit of 65k vertices per mesh on mobile[1], our Hairball model has one million vertices.

## 4.1 Studying the performance

To study the performance of the antialiasing techniques we collected various metrics from the application: frame rate, GPU shader processing metrics, GPU memory stats and GPU temperature.

### 4.1.1 Frame rate

One important metric when analyzing overall performance is the frame rate or frames per second (FPS), the number of frames an application can generate per second. The mean FPS of each technique was collected from the application in both scenes, the Crytek Sponza FPS results are displayed in figure 4.3.

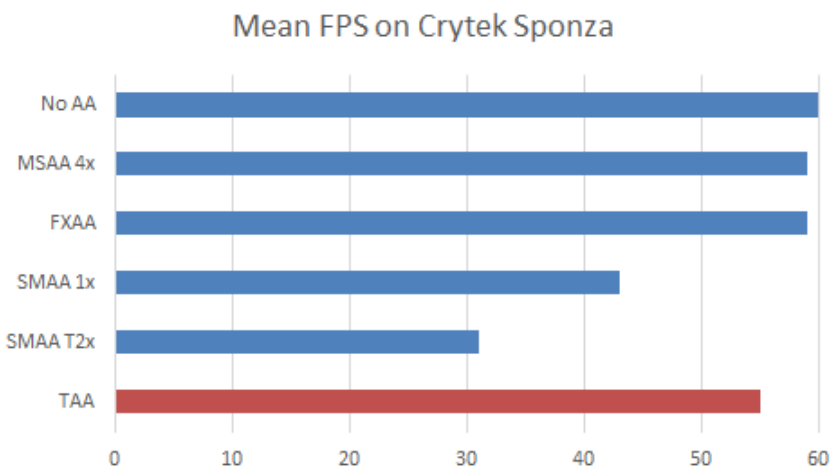


**Figure 4.3:** Chart comparing the mean FPS of the antialiasing techniques on the Crytek Sponza scene.

On the Crytek Sponza scene, the mean FPS value without antialiasing was 60, MSAA 4x and FXAA obtained the best FPS results with the mean value of 59. SMAA variants obtained lower values, 43 FPS for 1x and 31 FPS on the T2x. MSAA performed slightly better than FXAA with the minimum FPS

[1]Mesh limitations on Unreal Engine - https://docs.unrealengine.com/en-us/Platforms/Mobile/Content (as consulted on June 15, 2018)

of 51 while FXAA minimum was 47 FPS. Temporal AA is marked in red because the implementation is not ghosting-free, some changes were needed to improve its quality. These changes on TAA would affect its performance and decrease FPS. Our TAA version obtained an average of 55 FPS, even if the implementation is incomplete, we included the results as it can be meaningful for comparison.

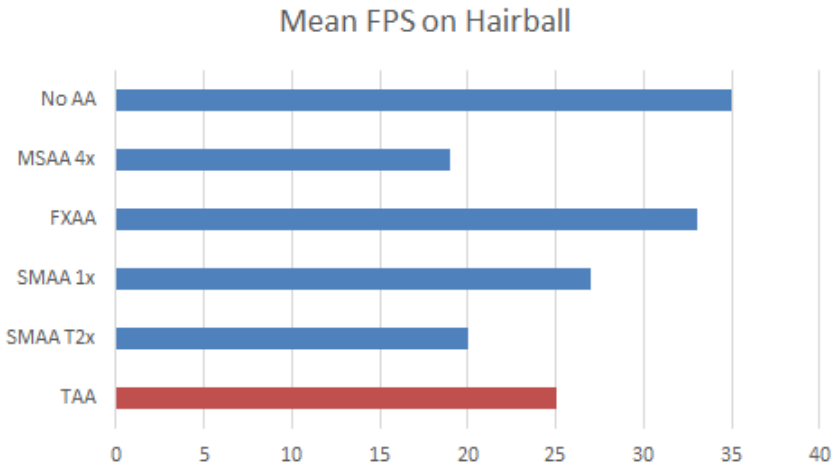


**Figure 4.4:** Chart comparing the mean FPS of the antialiasing techniques on the Hairball scene.

The mean FPS results on the Hairball scene can be seen in figure 4.4. On this test case, FXAA performed better than MSAA 4x, 33 FPS while MSAA 4x got 19 FPS, a huge difference considering that without applying AA the frame rate was 35. SMAA 1x and T2x executed at 27 and 20 FPS respectively, finally TAA achieved an average of 25 FPS. FXAA overhead was the lowest in both scenes, with the cost of 1 FPS on Crytek Sponza and 2 FPS on Hairball. MSAA performed well on Sponza but on Hairball the FPS was much lower, only 19 FPS while FXAA got 33 FPS. The low frame-rate on Hairball can be explained by the way MSAA works, the rasterization is executed with more samples, four in this case, this 3D model has high triangle count which also makes the number of operations on the rasterizer increase, which diminishes the technique performance. We can also observe that in the hairball scene the post-processing AA techniques performed better than MSAA, this happens because post-processing is not affected as much by scene complexity, unlike oversampling techniques like MSAA and SSAA.

### 4.1.2 GPU Shader processing

We collected GPU shader processing metrics using the Snapdragon profiler for both scenes. Table 4.1 contains the results of the Crytek Sponza scene.

The percentage of shaders busy metric indicates the percentage of time in which all the shader cores were active during the execution of the tests. A lower percentage indicates a lower usage of GPU processing resources, for example, if we get a value of 75%, it means that 25% of the time the shader

**Table 4.1:** Mean values of GPU Shader processing metrics collected on the Crytek Sponza scene.

| | No AA | MSAA 4x | FXAA | SMAA 1x | SMAA T2x | TAA |
|---|---|---|---|---|---|---|
| % Shaders Busy | 76.1 | 54.9 | 83.2 | 81.7 | 82.0 | 73.8 |
| % Time ALUs Working | 37.7 | 37.6 | 37.2 | 32.3 | 29.3 | 22.8 |
| % Time EFUs Working | 5.0 | 5.1 | 3.5 | 2.3 | 3.2 | 5.1 |
| % Time Shading Fragments | 71.7 | 70.4 | 80.1 | 81.7 | 88.5 | 81.5 |
| Fragment Instructions (x$10^9$) / Second | 21.0 | 20.3 | 33.6 | 33.3 | 30.0 | 23.8 |

cores were idle waiting for more work.

MSAA 4x performed better in this metric, with an average of 54.9%, this happens because MSAA does not have a fragment shader pass thus less work is given to the shaders cores in this technique, this is also confirmed by the metric % Time Shading Fragments. The Shaders Busy results of FXAA, SMAA and TAA appear to be dependent on the fragment shader complexity of each technique, however, between FXAA and SMAA, it was expected that FXAA would perform better in this metric because its shader has fewer operations and is more optimized than SMAA.

In the metric % Time ALUs Working, MSAA and FXAA obtained higher values, similar to the previous metric we believe that there are other factors influencing the results, time-based metrics are difficult to compare when the number of FPS between techniques has a huge disparity.

The table 4.2 contains the GPU shader processing results for the Hairball scene.

**Table 4.2:** Mean values of GPU Shader processing metrics collected on the Hairball scene.

| | No AA | MSAA 4x | FXAA | SMAA 1x | SMAA T2x | TAA |
|---|---|---|---|---|---|---|
| % Shaders Busy | 91.4 | 79.3 | 92.1 | 92.3 | 87.1 | 85.6 |
| % Time ALUs Working | 53.6 | 43.6 | 53.6 | 50.7 | 49.5 | 52.3 |
| % Time EFUs Working | 3.0 | 3.3 | 2.9 | 2.6 | 3.8 | 4.4 |
| % Time Shading Fragments | 37.4 | 38.7 | 41.0 | 48.8 | 53.2 | 44.8 |
| Fragment Instructions (x$10^9$) / Second | 20.7 | 21.0 | 23.6 | 27.1 | 23.9 | 18.9 |

When comparing the results from both scenes, we can notice higher on the % of Shaders Busy metric and % Time ALU working in all techniques, this happens because of the geometry complexity of the Hairball scene. The % Time Shading Fragments is much lower because most of the time the GPU is shading vertices instead of fragments.

The Fragment Instructions is not easily comparable since the techniques have different frame rates, for this reason, we calculated the number of instructions per frame (see figure 4.5). The results obtained of fragment instructions per frame make more sense, the more intensive techniques have a higher number of instructions.

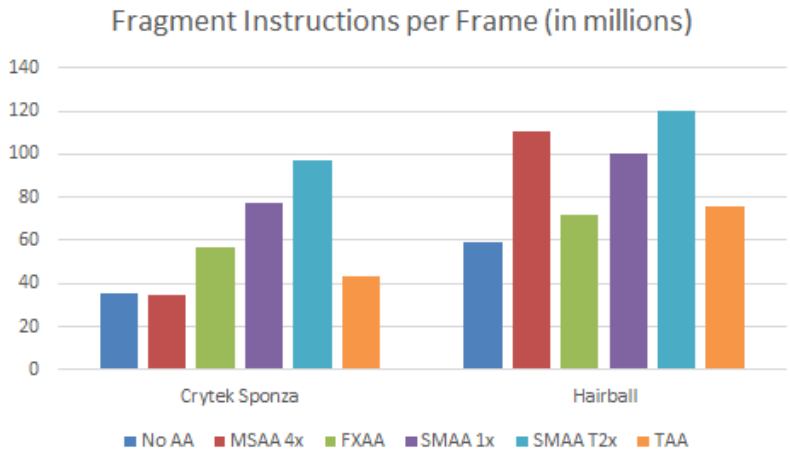


**Figure 4.5:** Chart comparing the fragment instructions per frame.

### 4.1.3 GPU Memory Stats

During the execution of the tests, we collected data related to the GPU memory, we considered three metrics: average bytes per fragment, total bytes read by the GPU from memory per second and texture memory read in bytes. Tiled architecture aims to minimize the memory transfers between GPU and RAM, the memory stats collected are important to analyze how each antialiasing technique performs.

The results for the scene Crytek Sponza are present on table 4.3.

**Table 4.3:** Mean values of GPU memory stats collected on the Crytek Sponza scene.

| | No AA | MSAA 4x | FXAA | SMAA 1x | SMAA T2x | TAA |
|---|---|---|---|---|---|---|
| Average Bytes / Fragment | 14.4 | 9.4 | 14.1 | 7.6 | 7.2 | 11.4 |
| Read Total (GB/s) | 8.2 | 9.4 | 7.8 | 6.0 | 7.8 | 11.6 |
| Read Total per Frame (MB) | 137.8 | 159.7 | 131.8 | 138.4 | 251.4 | 210.2 |
| Texture Memory Read BW (GB/s) | 5.5 | 4.7 | 5.0 | 3.2 | 3.8 | 6.4 |

From the memory stats we can see that both SMAA variants have less memory transferred, at first we could see this information as good behavior from these techniques, however, we think that this is not the case. Tiled architectures are known for batching the work on the GPU, for example, when executing two draw calls and using the result from the first draw, the GPU tries to execute everything together to gain performance and avoid memory transfers. While MSAA, FXAA, TAA have a higher value of memory being used, this might indicate that the GPU is batching all the work, on the other hand, SMAA looks like cannot execute all at once, and needs to separate the work in more GPU passes. We believe this is the case because the implementation of FXAA uses a lot less texture memory than SMAA T2x.

The table 4.4 contains the GPU memory stats on the Hairball scene.

The results obtained on the Hairball scene differ a lot from the previous scene because that this 3D

**Table 4.4:** Mean values of GPU memory stats collected on the Hairball scene.

| | No AA | MSAA 4x | FXAA | SMAA 1x | SMAA T2x | TAA |
|---|---|---|---|---|---|---|
| Average Bytes / Fragment | 0.0 | 0.0 | 0.0 | 0.6 | 2.1 | 3.5 |
| Read Total (GB/s) | 13.4 | 8.9 | 12.8 | 11.1 | 10.3 | 12.0 |
| Read Total per Frame (MB) | 383.7 | 466.8 | 386.9 | 412.5 | 516.9 | 481.8 |
| Texture Memory Read BW (MB/s) | 0.2 | 0.1 | 0.1 | 23.1 | 88.7 | 121.1 |

model does not have textures. In this case, we can verify that FXAA performed best in all the metrics, the difference on the read total memory among the techniques is not so significant however as we discussed before these metrics are difficult to analyze due to the techniques having different frame rates. The Read Total per Frame is a more reliable metric because it takes into account the FPS obtained, SMAA T2x and TAA use a higher amount of memory because the previous frame is saved in memory. SMAA and TAA obtain much higher values on the Average Bytes / Fragment and on Texture Memory Read BW because these techniques implementations need lookup textures to perform antialiasing.

### 4.1.4 GPU Temperature

In terms of GPU temperature, the results are similar to previous results among the AA techniques, MSAA obtains lower temperatures followed by FXAA, TAA and SMAA. The results are displayed in figure 4.6. Comparing the temperature on both scenes, Hairball obtains higher temperatures, 4 to 8 degrees higher. In terms of temperature within the same scene, MSAA performed better with 47.3 °C on Crytek Sponza and 53.2 °C on Hairball. SMAA 1x got higher temperatures, 53.3 °C on Crytek Sponza and 58.8 °C on Hairball. In the tests, we noticed that when the camera rotated to more difficult areas of the scene, the frame-rate lowered and the temperature rose.

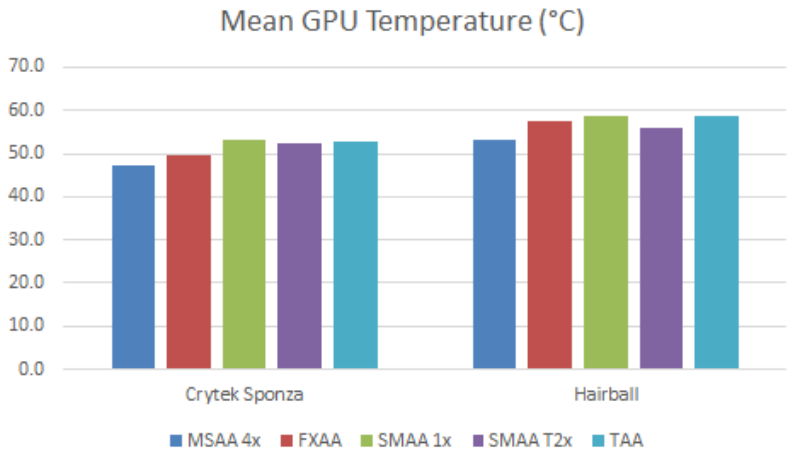


**Figure 4.6:** Chart comparing the mean GPU temperature obtained during the execution of antialiasing techniques on both scenes.

## 4.2 Studying the image quality

To study the quality of the image produced by the antialiasing techniques we used two methods: a quantitative method using SSIM and a more subjective method by visually analyzing how each technique performed in certain cases.

### 4.2.1 SSIM comparison

We applied the SSIM method between a supersampled 4x image (16 samples per pixel) and the resulting image from each technique. The techniques that obtained higher values on the structural similarity index have visual results that are approximated to the supersampled version, thus have better image quality. We executed this test on two resolutions: the native resolution of Samsung S8, 2220x1080 and a reduced resolution 1280x720. The objective of testing a lower resolution is to study the variation of the image quality.

Figure 4.7 contains the SSIM results on Crytek Sponza. We can observe that SMAA T2x obtained the best result with 0.94, followed by TAA, SMAA, MSAA and FXAA. In both resolutions the results were similar, however, on 720p the results have a lower order of magnitude, with exception of FXAA which obtained similar values, 0.831 in 1080p and 0.836 on 720p. The result on 720p being higher can be explained because FXAA default preset only calculates edge distances up to a threshold for performance reasons, in 720p the edge distances are more accurate since there are fewer pixels to process.

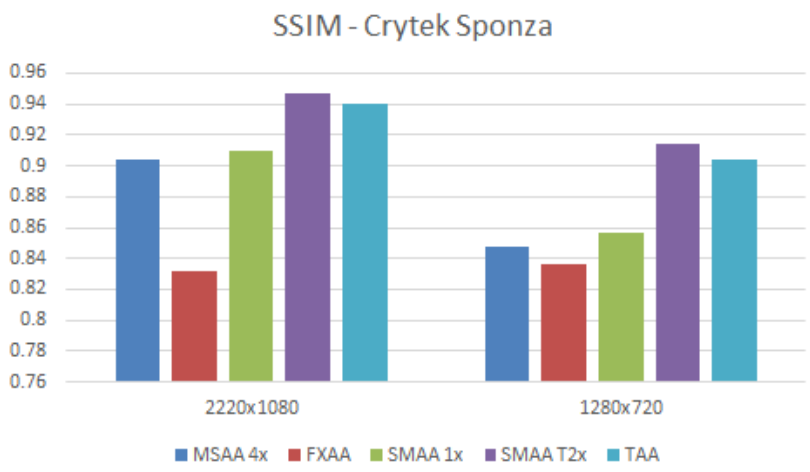


**Figure 4.7:** Chart comparing the SSIM results of Crytek Sponza on two resolutions.

The results on the Hairball can be seen in figure 4.8. On this scene, MSAA 4x obtains the best result with 0.977 of similarity to the supersampled reference. It performs better than on Sponza because

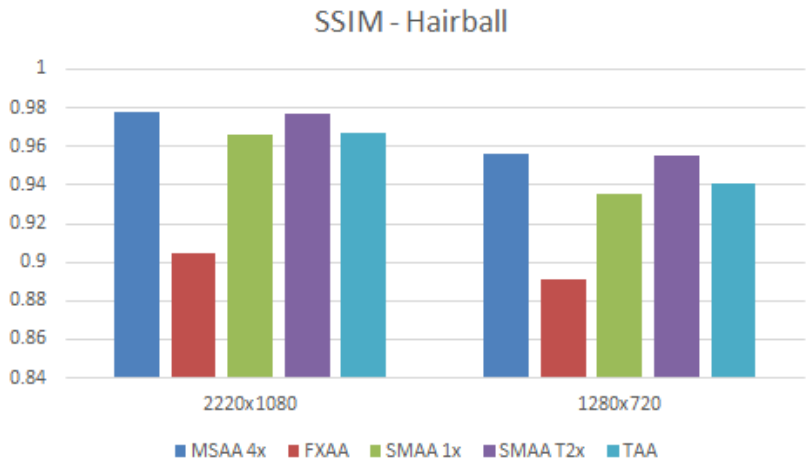


**Figure 4.8:** Chart comparing the SSIM results of Hairball on two resolutions.

MSAA cannot improve texture details, unlike temporal SMAA. SMAA T2x again performs well with the SSIM result of 0.976, followed by TAA, and SMAA 1x both with 0.966 and FXAA with 0.9. On 720p resolution, all the techniques lower the SSIM results but not as much compared to the Sponza 720p resolution test.

Considering all test cases, we can say that SMAA T2x obtains the best SSIM results, MSAA comes in second place, FXAA performs worst in this metric.

### 4.2.2 Image comparison

In this section, we present visual results of how the techniques performed in difficult cases.

We extracted a zoomed region on the images produced by each technique, to compare how they performed in different cases, see figure 4.9.

The following aspects were observed in the results of the antialiasing techniques: Shape preservation, Diagonals, Textures, Multiple edges and Fine geometry. SSAA and no AA are present on the figure for reference purposes.

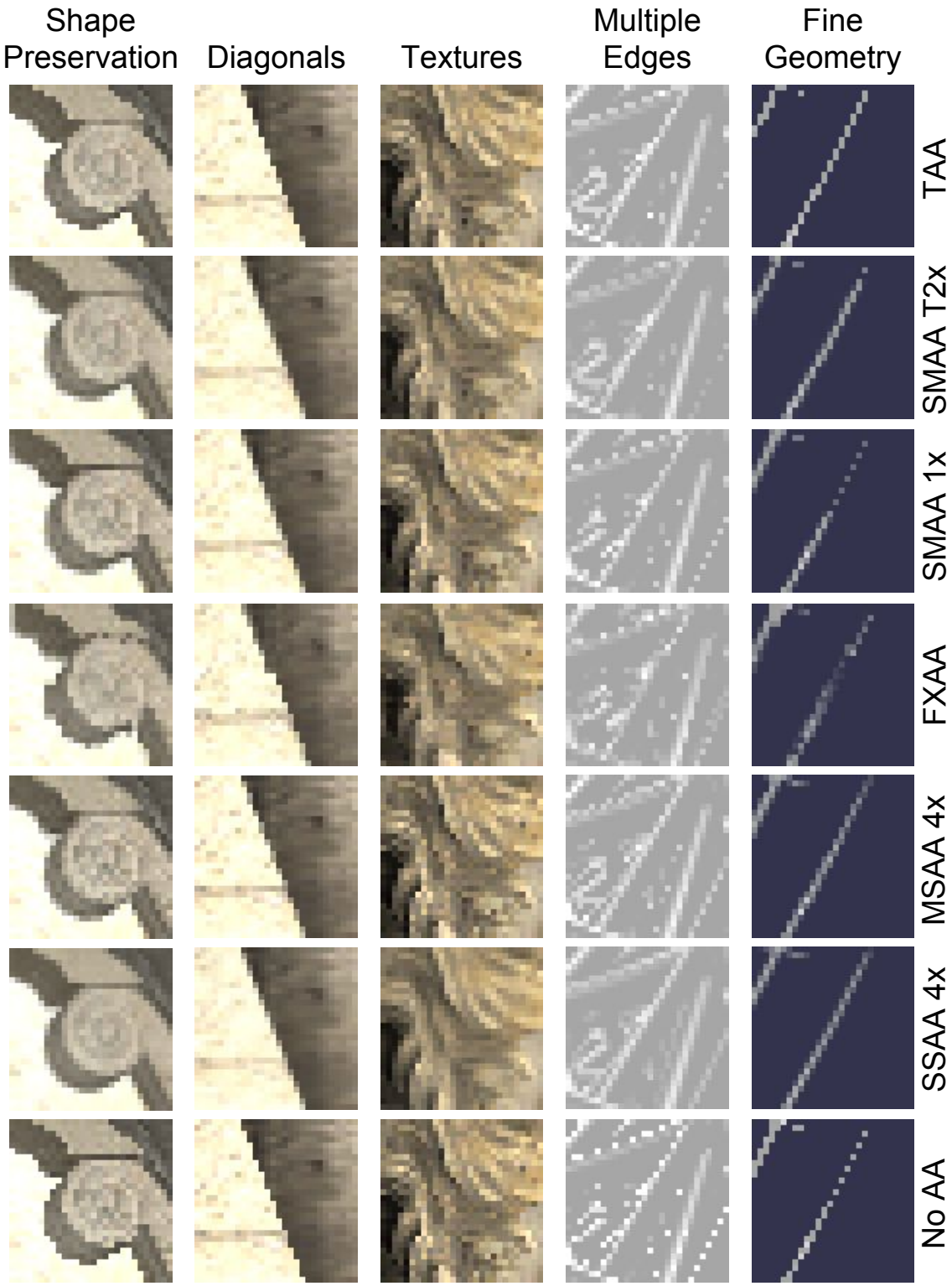


**Figure 4.9:** Zoomed details of the antialiasing techniques.

#### 4.2.2.1 Shape preservation

The case presented in the figure 4.9 is difficult to be handled by antialiasing techniques because of the round geometry and texture on the center. Between SSAA 4x and No AA, we can observe that SSAA has more intermediate detail on the edges, less abrupt changes and the texture is more detailed. Comparing each AA technique, SMAA T2x has the most similar result to supersampling. The other techniques also improve the result, FXAA on this case has the least appealing result, especially on the horizontal edge above the texture which loses its original shape.

#### 4.2.2.2 Diagonals

Diagonals are prone to aliasing in 3D applications, when the results show abrupt changes between pixels they are known as *jaggies*. SMAA diagonal handling really shines in this particular case, obtaining edge gradients similar to the supersampled version. MSAA result is better than FXAA. Our TAA version does not improve the diagonal significantly, one thing that would help improve this case is a more distributed sampling sequence, we only use two different sample locations. Unreal Engine TAA version uses the Halton Sequence[2] with 8 samples for jittering which is very well distributed.

#### 4.2.2.3 Textures

We can observe that MSAA does not improve the texture quality at all because this technique works at the geometry level. FXAA reduced the overall contrast from the original image, making it look less pixelated. When comparing FXAA with SMAA and TAA it can be said that FXAA results are more blurred, some texture details are lost. The temporal techniques improve the texture slightly without much blur, again in this case a better sampling distribution spread across frames could improve the quality.

#### 4.2.2.4 Multiple edges

In this category we can verify that there are high frequencies on the no AA version, looking more pixelated than usual. All the AA techniques smooth the highlights and improve the image in general. SMAA result is the most similar to the supersampled version.

#### 4.2.2.5 Fine geometry

In this last comparison, its a problem of geometry aliasing, the geometry is so thin that rasterizer did not cover some pixels. MSAA obtained similar results to the supersampled reference, oversampling

[2]Kocis, Ladislav, and William J. Whiten. “Computational Investigations of Low-Discrepancy Sequences.” ACM Transactions on Mathematical Software, vol. 23, no. 2, 1997, pp. 266–294.

works well in this case. FXAA reconstruction of the geometry gets poor quality results. The temporal techniques manage to reconstruct the shape.

## 4.3 Overall analysis

In terms of performance, the techniques that obtained the best results were MSAA 4x and FXAA, they performed well in every metric. MSAA cost increases linearly when the scene complexity increases, which was proved in the Hairball scene.

Regarding the image quality, the temporal variant of SMAA obtained the best results. It can be said that temporal techniques work very well to combat aliasing, especially using jittering improves the texture quality. MSAA also performed well, in the Hairball scene the SSIM results between MSAA and SMAA were very close.

We believe that the most balanced option between performance and image quality for antialiasing is MSAA.

# 5 Conclusions and Future Work

The objectives of this work are testing and comparing antialiasing techniques in a mobile environment. We chose four state-of-the-art antialiasing techniques to be compared in terms of performance and image quality. The techniques subject to tests were MSAA 4x, FXAA, SMAA and TAA.

The test results show that MSAA and FXAA have the best performance in the evaluated metrics, with a higher number of FPS, use less GPU processing power and memory. In terms of image quality, SMAA obtains the best results in SSIM, meaning that it produces results very similar to the supersampled reference. In the visual analysis, we found that SMAA results are better than the other techniques. MSAA gets the second best results in terms of image quality. Considering both aspects performance and quality, MSAA is the most balanced technique.

The post-processing antialiasing solutions, FXAA and SMAA, have good results, SMAA is more recent and obtains results very similar to supersampling.

In our work, the temporal antialiasing implementation was not perfect, it is a complicated technique that needed some improvements, especially to remove ghosting. Even so, it obtained good visual results and performance was decent.

## 5.1 Future work

We believe that there are some optimizations that could have been done in the test application. When using Framebuffer objects they should be invalidated in case they are not needed anymore in that render cycle. Compression of textures could also have been an optimization, using compression improves memory bandwidth which improves performance.

In the techniques FXAA and SMAA, the default presets were used, tweaking the parameters of these techniques might have improved their performance. One of the parameters is edge search distances which iterate a sample point along a texture, in the tiled architecture this type of texture access might occur outside of the current tile, causing a halt until the neighbor tile is rendered, which is a huge performance drop. This behavior is visible on the results of average bytes per fragment on the Crytek Sponza scene (figure 4.3), SMAA should have had higher number of bytes per fragment but did not, probably because the technique accesses texture positions outside of the tile.

SMAA had the best image quality, it would be interesting improving the performance of this technique in tiled rendering.

# Bibliography


[1] Peter Young. Coverage sampled antialiasing. Technical report, NVIDIA, 2006.

[2] Jorge Jimenez, Jose I. Echevarria, Tiago Sousa, and Diego Gutierrez. SMAA: Enhanced subpixel morphological antialiasing. *Computer Graphics Forum*, 2012.

[3] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[4] Emmanuel C Ifeachor and Barrie W Jervis. *Digital signal processing: a practical approach*. Pearson Education, 2002.

[5] Timothy Lottes. FXAA. Technical report, NVIDIA. 2011.

[6] Dave Shreiner, Graham Sellers, John Kessenich, and Bill Licea-Kane. *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley, 2013.

[7] Alexander Reshetov. Morphological antialiasing. *Proceedings of the 1st ACM conference on High Performance Graphics*, 2009.

[8] Brian Karis. High-quality temporal supersampling. *Advances in Real-Time Rendering in Games, SIGGRAPH Courses*, 2014.

[9] Jorge Jimenez, Diego Gutierrez, Jason Yang, Alexander Reshetov, Pete Demoreuille, Tobias Berghoff, Cedric Perthuis, Henry Yu, Morgan McGuire, Timothy Lottes, et al. Filtering approaches for real-time anti-aliasing. *ACM SIGGRAPH Courses*, 2011.

[10] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE computer graphics and applications*, 1994.

# A
# Appendix

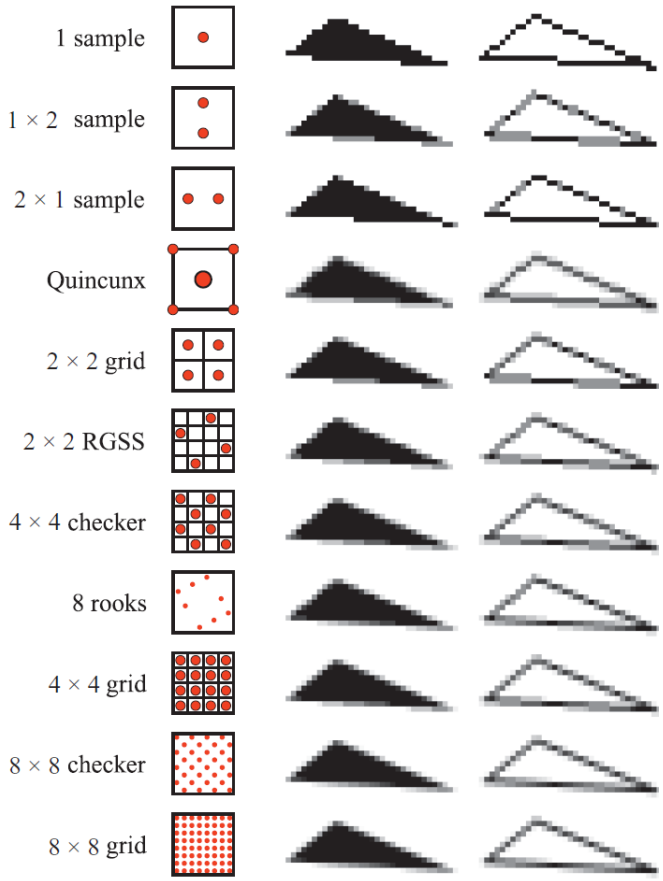


**Figure A.1:** Supersampling applied to a triangle, with different sampling patterns. Notice how the aliasing is less visible when more samples are used, also some patterns perform better than others with the same number of samples. Image taken from the book Real-Time Rendering, 3rd Edition, 2008 [3]

## Bonus Content

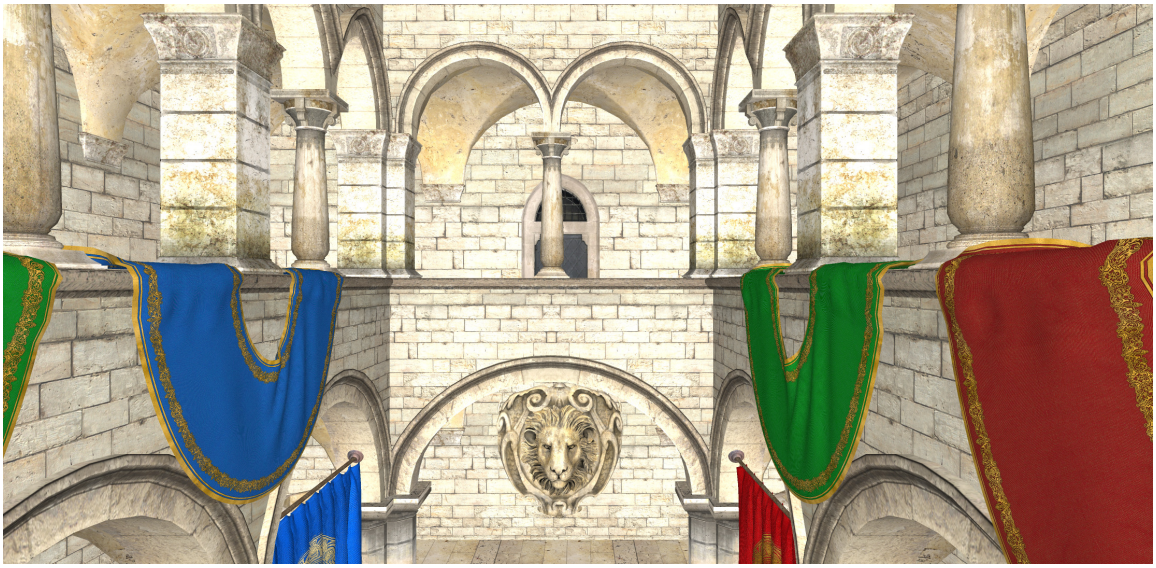

**Figure A.2:** Crytek Sponza scene rendered by the application.

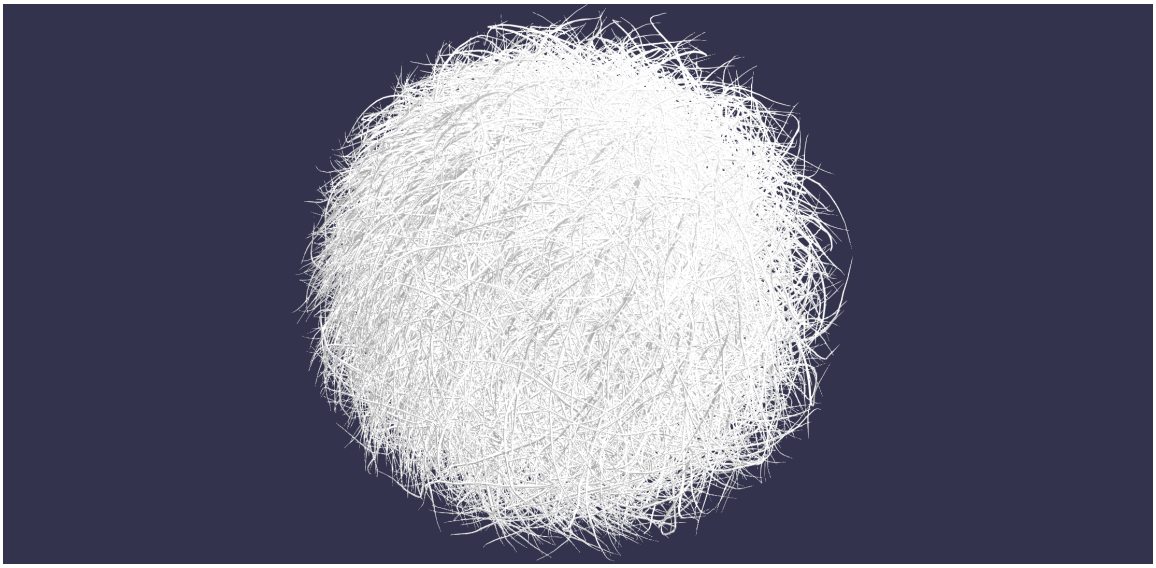

**Figure A.3:** Hairball scene rendered by the application.

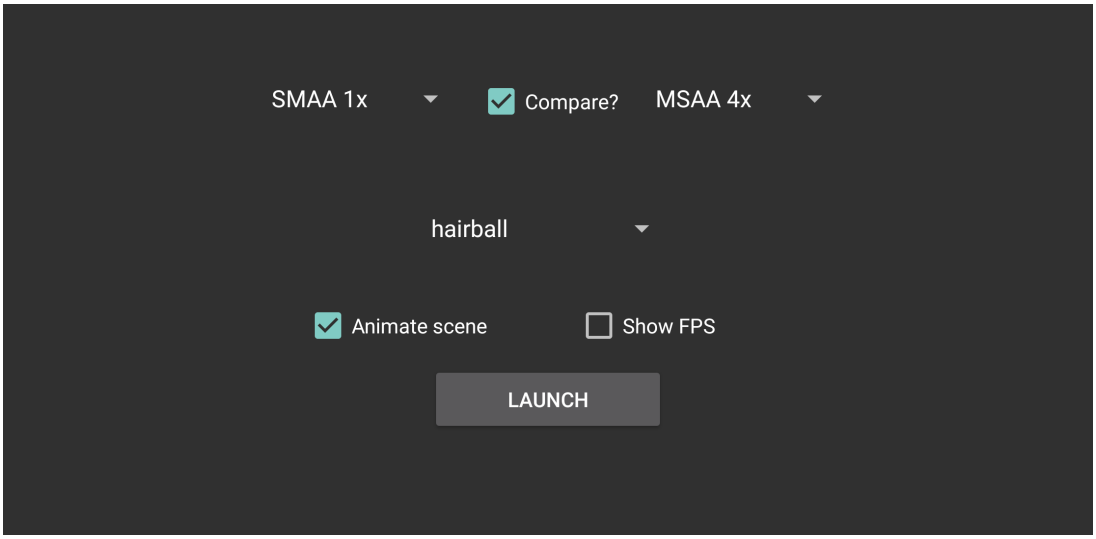


**Figure A.4:** Screenshot of the options screen of the application.


**Figure A.5:** Screenshot of the application showing the comparison feature of zoomed areas.

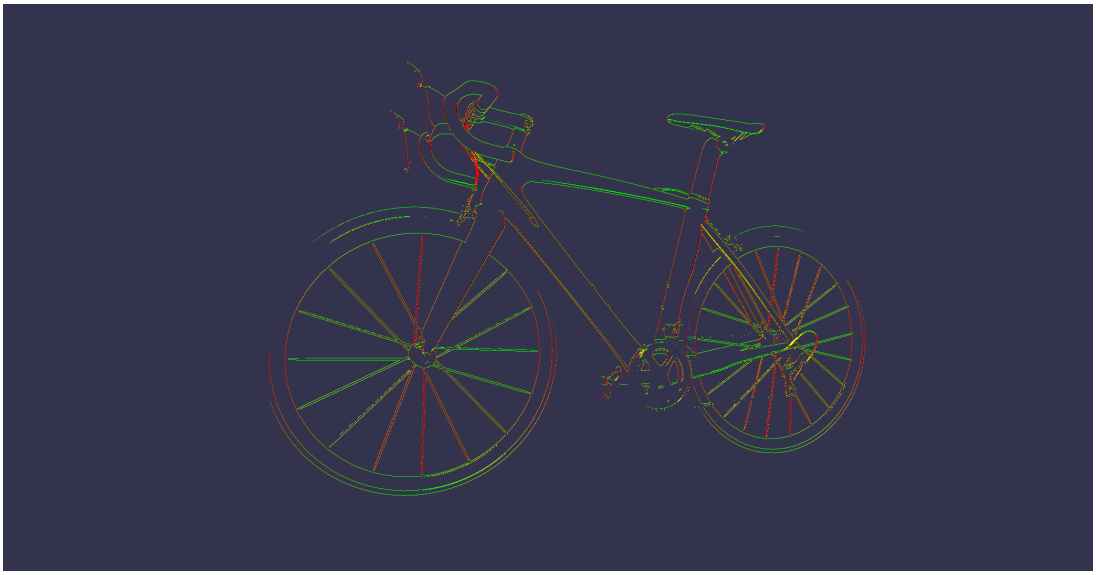

**Figure A.6:** SMAA edge detection pass.

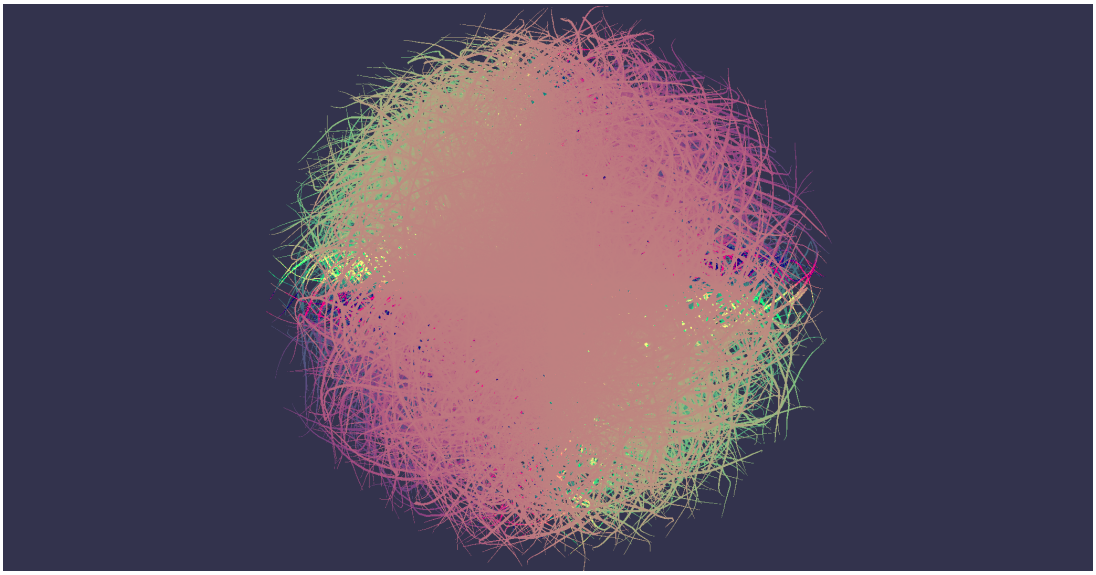

**Figure A.7:** Velocity buffer visualization of the Hairball model rotating.