Thesis no: BCS-2016-11

An Image and Processing Comparison Study of Antialiasing Methods

Alexander Grahn

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements of the degree of Bachelor of Game Programming. The thesis is equivalent of 10 weeks of full time studies.

Contact Information:
Author(s):
Alexander Grahn
E-mail: algb12@student.bth.se

University advisor:
Dr Prashant Goswami
Department of Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context. Aliasing is a long standing problem in computer graphics. It occurs as the graphics card is unable to sample with an infinite accuracy to render the scene which causes the application to lose colour information for the pixels. This gives the objects and the textures unwanted jagged edges. Post-processing antialiasing methods is one way to reduce or remove these issues for real-time applications.
Objectives. This study will compare two popular post-processing antialiasing methods that are used in modern games today, i.e., Fast approximate antialiasing (FXAA) and Submorphological antialiasing (SMAA). The main aim is to understand how both method work and how they perform compared to the other.
Methods. The two methods are implemented in a real-time application using DirectX 11.0. Images and processing data is collected, where the processing data consists of the updating frequency of the rendering of screen known as frames per second (FPS), and the elapsed time on the graphics processing unit(GPU).
Conclusions. FXAA shows difficulties in handling diagonal edges well but show only minor graphical artefacts in vertical and horizontal edges. The method can produce unwanted blur along edges. The edge pattern detection in SMAA makes it able to handle all directions well. The performance results conclude that FXAA do not lose a lot of FPS and is quick. FXAA is at least three times faster than SMAA on the GPU.

Keywords: post-processing, antialiasing, real-time

Contents

List of Figures

Chapter 1

Introduction

Aliasing is a long standing problem in computer graphics. When a scene is rendered by sampling pixel colours, it will render colours incorrectly when the sampled pixels contains more or less information than what was sampled. This will cause an image to have jagged or uneven edges on objects when they are rendered onto the screen. To illustrate an example, observe figure 1.1. Due to performance and hardware limitations the graphical processing unit (GPU) is unable sample with an infinite precision in order to render a perfect image. This means that certain pixels of the sphere do not take the background into account and vice versa. The sphere that should look round and smooth will instead have pixelated edges. This is aliasing and is clearly visible. It has therefore a very high and negative impact on the image quality [1].

Antialiasing methods is used to reduce or remove these issues in an image. Choosing the correct method can be difficult as there is a balance between the image quality and the performance. The end-user may sit on a variety of different hardware, so in order to please the target audience it is good to know where the trade-off is in-between the two.

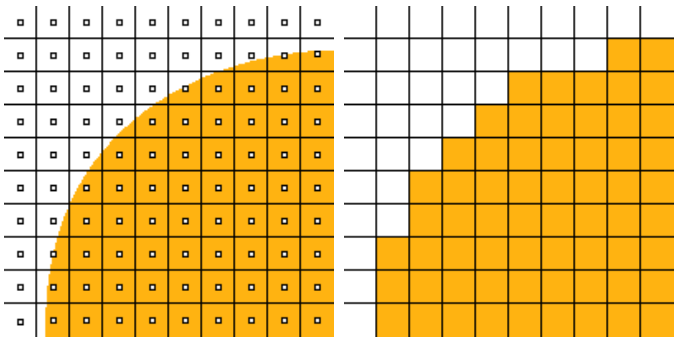


Figure 1.1: Illustrating aliasing with an edge of a sphere where each box represent a pixel space. Left: Shows a smooth edge of what we want to render if we had an infinite sample rate. The dots are the location of where we will sample. Right: Shows what will be sampled and what would be rendered.

1.1 Background and Related Work

SuperSampling AntiAliasing(SSAA) is a concept of producing antialiasing by calculating the scene in a higher resolution than normal, and then down-sampling it to the correct resolution. This will allow you to take more samples for each pixel. The method is, however, very inefficient as it perform these extra samples for all pixels in an image, even in those that do not need it [1].

MultiSample AntiAliasing(MSAA) follows in the footsteps of SSAA but introduced new techniques and optimizations. MSAA uses a two to eight times higher resolution than normal to find a limited amount of subsamples for the final pixel. By knowing the exposure of the subpixels it will perform supersampling of the edge of the polygon and perform colour calculations only once [2], [3].

SSAA and MSAA, with supersampling, will generate a very good image for photorealism but is very costly on the performance. Real-time applications may struggle as you have to calculate the scene in a higher resolution than what is displayed. Another form of antialiasing is called *post-processing antialiasing*(PPAA). PPAA takes the final image of a scene and essentially blurs it with various techniques to hide the aliasing. This method is significantly cheaper on performance, but may yield worse results in the final image. It is, however, important to note that supersampling can be combined with PPAA and that both of them are used in games and other real-time applications. The following methods are other related work that uses PPAA.

MorphoLogical AntiAliasing (MLAA) introduced new concepts and yielded good results which inspired new ideas and further development of methods such as *Fast approXimate AntiAliasing* (FXAA) and *SubMorphological AntiAliasing*(SMAA) that this thesis is about [4] - [6]. MLAA consists of three main steps.

1. Find discontinuities between pixels, i.e., finding edges of objects.
2. Identify objects in predefined patterns.
3. Blend the pixels with the help of the patterns.

Conservative Morphological AntiAliasing (CMAA) is an interesting method which is inspired by MLAA, FXAA and SMAA but with its own variation of, amongst other things, finding local dominant edges and handling of symmetrical long edge shapes [7].

Other methods may use different techniques where, e.g., *Topological reconstruction AntiAliasing* (TMLAA) uses topological reconstruction techniques to determine whether a pixel should recover missing geometry and subpixel features for the final image [8].

1.2 Aim and Objectives

The aim of this thesis is to understand the differences of removing aliasing between popular and modern PPAA methods that are used in games today. This will be achieved by implementing two popular and modern methods in order to understand what separates them from each other. It will determine the trade-offs each method have. The methods chosen for this paper are FXAA and SMAA [5], [6]. They were chosen by relevance which was determined through a direct approach by checking which methods was available in popular game titles. FXAA can be found in games such as *Dota 2*, *Counter-Strike: Global Offensive*, *XCOM 2*, *Grand Theft Auto V* and much more [9] - [12]. SMAA can be found in games such as *Hitman* and *Arma 3* and is also quite praised in the community [13], [14].

The objectives are:

1. Create a real-time application that will be able to test the methods with features such as deferred rendering, camera movement, time measurements, object handling and other logic.

2. Implement each method as provided by the authors.

3. Conduct tests in order to gather information of performance and images quality. The performance measurements will be frames per second (FPS) and elapsed GPU time. The image quality will be gathered by images from each test. Further details about this can be found in the test environment section.

4. Research each methods algorithm with the help of documentation and source code to determine the differences of implementation.

1.3 Research Questions

- What are the differences in performance, measured by FPS[1] and elapsed GPU[2] time, and the image quality between FXAA and SMAA?

- How does each methods algorithm differentiate in achieving antialiasing?

[1] frames per second
[2] graphical processing unit

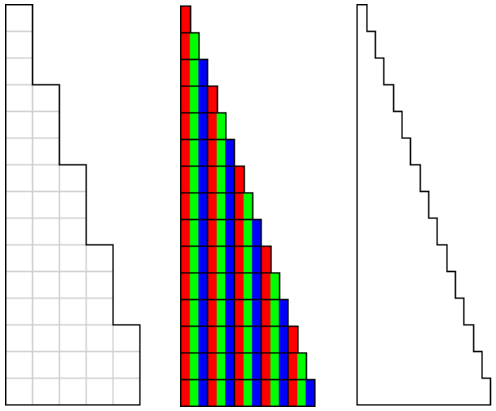


Figure 1.2: Subpixel technology. Left: No subpixel rendering is used and will have jagged edges as a result. Middle: With subpixels the edge is straighter. Right: The perceived end result [15].

1.4 Subpixel Rendering

A common feature in antialiasing is subpixel rendering. A single pixel on a LCD-monitor consists of three *subpixels*: red, green and blue (R,G,B). Together they generate the perceived colour of the pixel. However, subpixels can also be applied individually to improve the quality of the image. Observe the Figure 1.2, where the left figure displays the edge of an object without the use of subpixel rendering. This edge will look jagged but by applying subpixel rendering in the neighbouring pixels, like in the middle image, the edge looks much straighter. The right figure displays the end result and how it will be perceived [15].

The added subpixels will slightly change the colour of the image, which is why this technology bases on our knowledge of the human visual system and what humans perceive in an image. The human visual system is more sensitive to changes in luminance than changes in hue or saturation, i.e., humans are more capable of seeing nuance of brightness than seeing changes in colour. In other words, subpixel rendering works as long as the luminance is applied with care [16, p. 139].

Chapter 2

Methodology

To perform this comparison study a test environment was created to be able to test the two methods. The application use the source code as provided by the authors. The application gathered all the computation information and the images were manually gathered. More information about this can be found in the test environment section.

Information, references and articles were mostly found by conducting searches and reviewing references and citations in Google Scholar. The strings used was often broad and general, e.g., "Supersampling antialiasing", "post processing antialiasing", "morphological antialiasing", "antialiasing comparisons" and "FXAA SMAA comparison".

The relevant articles were reviewed and saved. The references of the articles were then examined and relevant articles were saved for later use. Any references that was not found through Google Scholar was searched on the normal Google search engine. These searches used strings such as "FXAA source code" and "Subpixel rendering".

Two conclusions were drawn by the information gathering. Despite the popular use of the methods, there is little work that have pointed out key differences between these two methods. The comparisons found have often been with other methods with unclear details.

The second conclusion is that much of the documentation and information about FXAA is undocumented or outdated. The version updates, specifically between version 3.8 and 3.11, is implemented very differently. This changed this paper slightly to include more source code segments to allow the reader to understand FXAA 3.11 easier. SMAA, on the other hand, have a detailed source code which is why code segments will be avoided for this paper. This will also be easier for the reader due to the sheer size of the method.

2.1 The Methods

2.1.1 Fast Approximate Antialiasing (FXAA)

Fast approximate antialiasing(FXAA) was created by Timothy Lottes from Nvidia 2009 [5]. The source code version for this paper is 3.11 which came out 2011. According to [17], FXAA never attempt to give the correct solution but instead attempts to create something that is fast and visually sufficient. FXAA 3.11 have a wide range of presets and graphics-engine compatibilities. It includes platform support for PC, XBOX-360 and Playstation 3.

FXAA can be implemented with a single render pass as suggested in [17], but it is done so with two for the PC-implementation in the version 3.11. The extra render pass is added to convert the image luminance of the scene before the algorithm begins in order to reduce the number of conversions. The luminance can then be stored on the alpha channel which would allow for a single input texture. This is good for the memory consumption on the GPU [17].

FXAA uses sRGB textures for its shader resources. sRGB textures change the colour space and uses a gamma 1/2.2 encoding in order to compensate for how monitors illuminate pixels. A gamma corrected image would redistribute tonal levels closer to how our eyes perceive them. This makes the image smaller and stored more efficiently. A linear texture, on the other hand, uses a high dynamic range and stores more colours. Rather than just applying sRGB textures for the backbuffer to be rendered on the screen, FXAA applies it also for its algorithm [18],[19].

The FXAA algorithm consists of five steps which I will go into more detail in the next section, Implementation of FXAA.

1. Acquire the luminance of the scene.
2. Determine irrelevant pixels for an early exit, i.e., edge detection.
3. Perform a subpixel aliasing test of surrounding neighbours.
4. Determine the direction of the edge by search vertically and horizontally.
5. Travel in the direction of the edge and determine the end of the edge.
6. Apply the correct colour after the information is gathered.

2.1.2 Implementation of FXAA

For this paper several iterations of FXAA was created in order to understand the complex nature of the source code. Many online sources and previous versions are very different from version 3.11. This section will go through algorithm and explain it. Apart from the luminance conversion, all of the code segments in this

section is rewritten from the source code of FXAA version 3.11 to more easily show how it works [17].

To implement FXAA the output texture of the deferred light pass should be written to the sRGB texture as mentioned in the previous section rather than the back buffer. The input shader resource view uses DXGI_FORMAT_R8G8B8A8_TYPELESS with a DXGI_FORMAT_R8G8B8A8_UNORM and the rest of the system uses DXGI_FORMAT_R8G8B8A8_UNORM_SRGB [5]. The render passes are then setup as usual.

Luminance Conversion

The first step, as to the five steps mentioned in the previous section, is to acquire the luminance. This is done by using a luminance constant which is dependent on the colour space. Observe the code segment below, which is the pixel shader of the first render pass. The dot product of the pixel colour and the luminance constant returns the scalar value of the luminance.

```
float4 PS_FXAALUMA(PSIn _input) : SV_TARGET
{
  float4 color = backBuffer.Sample(Sampler, _input.TexCoord);
  float3 luminanceConstant = float3(0.299, 0.587, 0.114);
  color.a = dot(color.xyz, luminanceConstant);
  return color;
}
```

Edge Detection

The pixel and its surrounding neighbour's luminance values are sampled. These values are then checked to see which has the highest and lowest value. The range of the two values are checked with a predefined limit to determine if they are part of an edge. If they are not part of an edge, the algorithm will make an early exit and return the colour unaltered [17].

```
float2 posM = _input.TexCoord;
float4 colour = lumaTexture.Sample(Sampler, posM);
float rangeMax = max(lumaM,max(max(lumaN,lumaW),max(lumaS,lumaE)));
float rangeMin = min(lumaM,min(min(lumaN,lumaW),min(lumaS,lumaE)));
float range = rangeMax - rangeMin;
if(range < max(PredefinedEdgeThresholdMin, rangeMax *
   PredefinedEdgeThresholdMax))
{
  return colour;
}
```

Subpixel Aliasing

The subpixel aliasing test is an algorithm that calculates the pixel offset in which to apply the color of the edge. Observe figure 2.1b for the targeted pixels. This offset is later multiplied with the texel size, the length of the pixel, in the direction towards the edge [17].

```
float subpixelPart1 = (2*(lumaN + lumaW + lumaE + lumaS) +
(lumaNW + lumaSW + lumaNE + lumaSE)) / 12.0f; //weights
float subpixelPart2 = abs(subpixelPart1 - lumaM); //range
float subpixelPart3 = saturate(subpixelPart2/range);
float subpixelPart4 = (-2.0*subpixelPart3)+ 3.0;
float subpixelPart5 = subpixelPart4 * subpixelPart3 * subpixelPart3;
float subpixelOffset =  subpixelPart5 * PredefinedQualitySubpix;
```

(a) FXAA (b) Subpixel aliasing test (c) Gradient

Figure 2.1: The modified pixels in the algorithm.

Edge Direction Test

In order for FXAA to hide an edge, it needs to know how it looks like. The next set of code determines if the edge is facing vertically or horizontally. By knowing that, the algorithm can after the code segment simply check which way the gradient is heading [17].

```
float edgeV1 = abs((-2.0 * lumaN) + lumaNW + lumaNE);
float edgeV2 = abs((-2.0 * lumaM) + lumaW  + lumaE);
float edgeV3 = abs((-2.0 * lumaS) + lumaSW + lumaSE);

float edgeH1 = abs((-2.0 * lumaW) + lumaNW + lumaSW);
float edgeH2 = abs((-2.0 * lumaM) + lumaN  + lumaS);
float edgeH3 = abs((-2.0 * lumaE) + lumaNE + lumaSE);

float edgeVert    = edgeV1 + (2.0 * edgeV2) + edgeV3;
float edgeHorz    = edgeH1 + (2.0 * edgeH2) + edgeH3;
bool horzSpan = edgeHorz >= edgeVert;
```

Edge distance for the gradient

To mask the edge the algorithm needs to know how long the edge is. This can be seen in figure 2.1c where the green marks out the pixels affected. This distance is determined by the number of iteration of the search loop. The search loop checks in both directions of an edge to smoothly mask. This is done by acquiring posN and posP which marks out the distance [17].

```
/* Luma N and S is set as the up and down of the direction of the
    gradient. offNP is a predefined irregular increment for the
    offset, e.g., 1.0, 1.5, 2.0, 2.0, ..., 4.0 etc. */
float lumaNN = lumaN + lumaM;
float lumaSS = lumaS + lumaM;
float gradient = (max(abs(lumaUpDir - lumaM), abs(lumaDownDir -
    lumaM)))/4;
for(int i=0; i< FXAA_SEARCH_STEPS; i++)
{
  if(doneNP)
  {
    if(!doneN) lumaEndN = lumaTexture.SampleLevel(Sampler,posN,0).w;
    if(!doneP) lumaEndP = lumaTexture.SampleLevel(Sampler,posP,0).w;
    if(!doneN) lumaEndN = lumaEndN - lumaNN * 0.5;
    if(!doneP) lumaEndP = lumaEndP - lumaNN * 0.5;
    doneN = abs(lumaEndN) >= gradient;
    doneP = abs(lumaEndP) >= gradient;
    if(!doneN) posN.x -= offNP[i].x;
    if(!doneN) posN.y -= offNP[i].y;
    doneNP = (!doneN) || (!doneP);
    if(!doneP) posP.x += offNP[i].x;
    if(!doneP) posP.y += offNP[i].y;
  }
}
```

Apply the colour

In the final step the algorithm comes together. The smallest distance to each point from the origin is determined. The greatest pixel offset is then applied to the original position of the pixel [17].

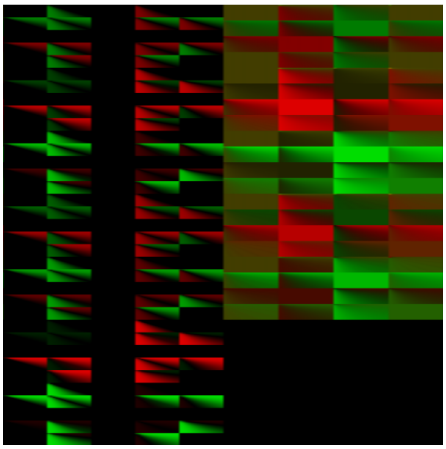
```
float pixelOffset = (distance * (-1/(distanceP + distanceN))) + 0.5;
FxaaFloat pixelOffsetSubpix = max(pixelOffset, subpixelOffset);
if(!horzSpan) posM.x += pixelOffsetSubpix * lengthSign;
if( horzSpan) posM.y += pixelOffsetSubpix * lengthSign;

return lumaTexture.SampleLevel(Sampler, posM, 0.0);
```

2.1.3 Submorphological Antialiasing (SMAA)

SubMorphological AntiAliasing(SMAA) was created 2011 and the source code version for this paper is 2.8, which was released 2013. SMAA is a big algorithm with optional features such as spatial and temporal multi- and supersampling. SMAA 1x is the PPAA feature which will be focused upon for this paper. SMAA is based on MLAA but have been reworked and expanded upon. The idea of MLAA is to determine what kind of edge the pixel is surrounded by. The different patterns surrounding a pixel are matched against a search texture which is given the corresponding colour of a precomputed area texture. An edge can take different shapes, e.g., it can look like a single long step as an L-shape, a Z-shape for short steps, or as a corner with a U-shape. The identified shape is translated through the area texture as seen in figure 2.2a [4], [6].

The algorithm has three different render passes:

1. Luminance edge detection: Determines the edges based on luminance and if they are vertical or horizontal.
2. Blending weight calculation: Identifies the predefined patterns.
3. Neighbourhood blending: Blends the pixels in the neighbourhood of the patterns and then proceed to render the final image.



(a) The area texture which is used to colour the pixel by the identified pattern.

(b) Left: Displays a graphical artefact of an uneven gradient of an edge. Right: SMAA corrects this with a smooth gradient [6, p. 4].

Figure 2.2

2.1.4 Implementation of SMAA

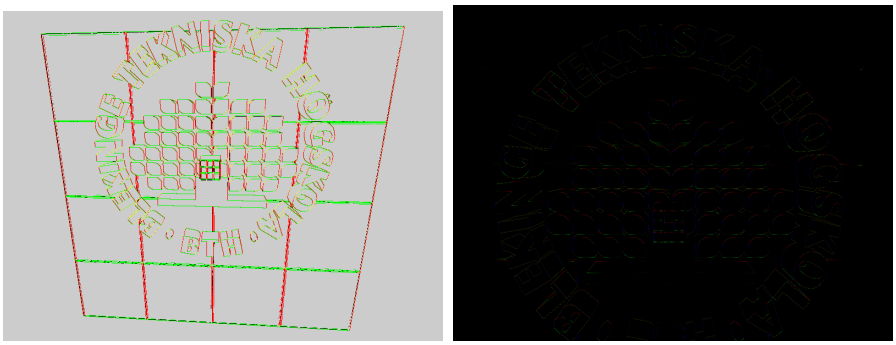
SMAA uses high dynamic range(HDR) textures for all the shader resources except the backbuffer and the textures. The source code comes with a search and an area texture which has to be loaded on the CPU from raw memory to shader resource views. This is easily done as values for the texture description is predefined inside the texture files.

The second render pass require both a linear and a point sampler. The point sampler is used to sample from the search texture.

Luminance Edge Detection

SMAA takes a different approach to detecting the edges. Instead of calculating or sampling the luminance of all surrounding pixels, the render pass only checks the top and left boundaries. If the delta value of the pixel and the boundaries are below the predefined threshold it is discarded. The discarded pixels will not be touched until the final render pass where they will be rendered for the final image. Unlike FXAA, due to the HDR textures the luminance constant is different and set as float3(0.2126, 0.7152, 0.0722).

The second half of the render pass is determining if the pixel is part of a horizontal or vertical edge. It does so by calculating the maximum deltas between the neighbours and delta of the "toptop" and the "leftleft" pixels. By performing a check by jumping a pixel, it is able to remove unwanted graphical artefacts that is displayed in figure 2.2b. As pointed out in the subpixel rendering section, these differences are well noticed by humans and avoided by this feature [16, pp. 139]. The final and maximum deltas are stored for the next render pass and can be viewed in figure 2.3a.



(a) The output of the first SMAA render pass, the luminance edge detection.

(b) The output of the second SMAA render pass, the blending weight calculation.

Figure 2.3

Blending Weight Calculation

In the second render pass the algorithm will try to establish which pattern the pixels have. It continues, by using the edge output texture in the previous render pass, to search diagonally. This is where the search and area textures comes into play. The algorithm searches the distance of the edge by traversing in each direction. It will then calculate the crossing edges inside the pixel and utilizes the search texture to determine exactly how the edge is behaved. With the search texture, it then sample with the corresponding pattern in the area texture for the best output. If the diagonal pattern fail, the algorithm check orthogonally. Lastly, it verifies the edge is not part of a corner. The output of the pass can be viewed in figure 2.3b.

Neighbourhood Blending

The third and last render pass is the neighbouring blending, the subpixel rendering. The algorithm checks the blend texture from the previous render pass and renders any unweighted pixels. These pixels are the discarded pixels from the first render pass.

The remaining pixels checks orthogonally which side has the more weight. Then, it performs a linear interpolation between the current pixel and pixel in that direction. You can see the final result in figure 2.4.

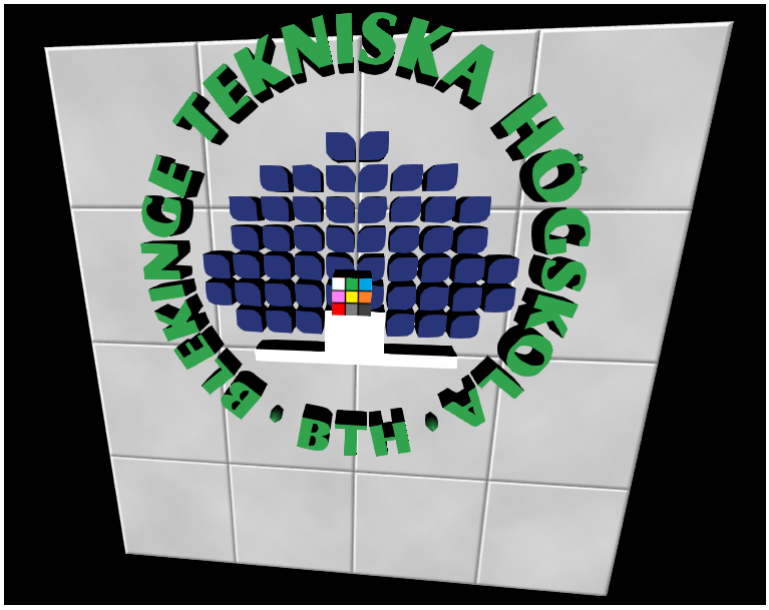


Figure 2.4: Final image after all the render passes.

2.2 The Test Environment

2.2.1 The Application

The program in this thesis uses C++, Microsoft Visual Studio 2015, DirectX 11.0 with shader model 5.0.

The scene that is set up uses three objects. One thin box to act as a background for the large 3-dimensional BTH-logo provided to me by Blekinge Institute of Technology. The BTH-logo is read from an OBJ-file. It acts as the primary source for the testing as it consists of many small shapes that have an aliasing problem. The scene also have one little Rubik's cube below it to verify colour integrity. You can view the scene in Figure 2.5.

Deferred rendering is used in this project. Games usually have a lot of lights which favours deferred rendering. The light in this program is a singular directional light that applies to all objects. The application will use HDR textures for the no antialiasing method.

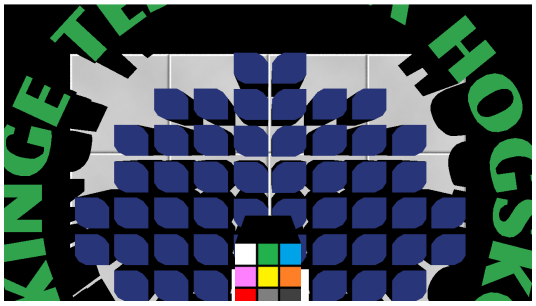


Figure 2.5: The scene

2.2.2 Measurements

Each method have many different presets and adjustable values. Changing these values could change the results greatly. This is why the values were set to the presets or the recommendations by the author. FXAA is measured with the quality preset 39. SMAA is measured with the quality preset of High. Code specifics can be found in the appendix.

Performance

Games and other real-time applications usually draw the scene in 30-60 *frames per second*(FPS) or more. This means that the scene has to be calculated and be ready before ~16.67milliseconds (60FPS). The FPS is measured by using QueryPerformanceCounter from the Windows library.

The elapsed GPU time is established by the DirectX 11 commands D3D11_QUERY_TIMESTAMP and D3D11_QUERY_TIMESTAMP_DISJOINT.

It takes double timestamps on the GPU. First at the beginning and the end of the scene and then between antialiasing call. This is done as the CPU works separated from the GPU as it waits until the GPU is finished on the swapchain present call.

The tests was conducted on three different computers. Each computer have very different components and will display the difference between low, medium and high specifications.

1. Low specifications computer
   (a) Nvidia GeForce 750M 4GB
   (b) Intel i5- 4200U @ 1.6GHz
   (c) 8GB RAM
2. Medium specifications computer
   (a) GeForce GTX 680 2GB
   (b) Intel i7-3770K CPU @ 3.5GHz
   (c) 16GB RAM
3. High specifications computer
   (a) Nvidia GeForce GTX 980 Ti
   (b) Intel i7-4770K @ 3.50 GHz
   (c) 16GB RAM

The results was collected by storing the averages in run-time and switching camera angles automatically. The results was after the data collecting printed to a text-file for easy accessibility. The performance measurements are averaged with a sample rate of 5000. 5000 is a high number and was as high as it could go without crashing the low specifications computer.

Image quality

It is difficult to measure image quality as the selected methods are not trying to be the correct image, but instead look a lot like it. Similar tests were performed by [21], that used a tool called peak-signal-to-noise that try to determine the differences in an image. FXAA and SMAA were included in the tests. The paper is inconclusive in-between the results of FXAA and SMAA and points out the problem with trying to measure an incorrect image, and that this is very subjective.

This is why this paper will show each method next to each other to allow the reader to draw their own conclusions, as well as read the conclusions by the author of this paper. Showing image comparisons is a very common method to help establish credibility and is conducted in almost all of the previously mentioned methods in this paper [4], [6] - [8], [21].

Each image have been collected by the use of print-screen and saved to the lossless portable network graphics(PNG) format with the help of the image software Paint.net with the version 4.0.9 [22]. The images have been compressed to the PDF-file in accordance to the graphics strategies of [23]. This is done by the use of pdflatex which directly supports PNG without a loss of image quality.

The tests was conducted in five scenes:

Scene A: This scene is meant to give an overview over the test environment. It shows aliasing at a distance.

Scene B: Horizontal and vertical edges are examined.

Scene C: Diagonal edges are examined.

Scene D: This scene was chosen as it is riddled with aliasing artifacts. It shows aliasing from the foreground and all the way back to the background.

Scene E: Alternative to the previous scene to add more FPS and time data.

The scenes are rendered with a resolution of 1280x720.

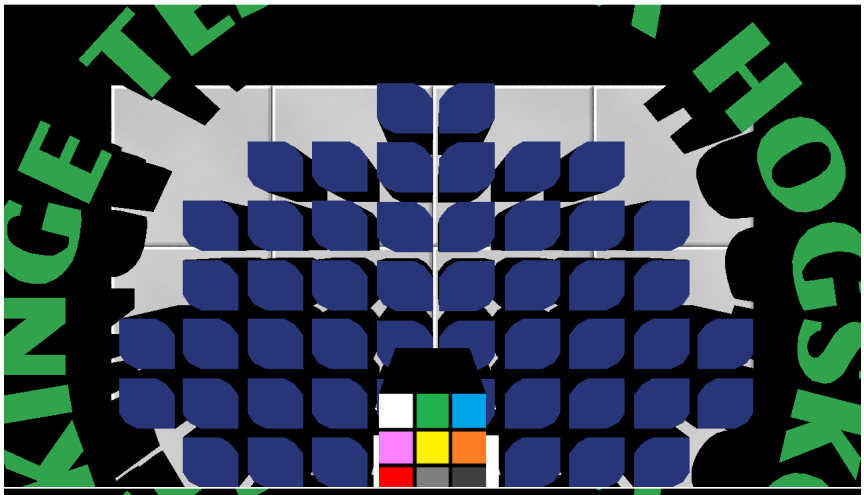
Chapter 3

Results

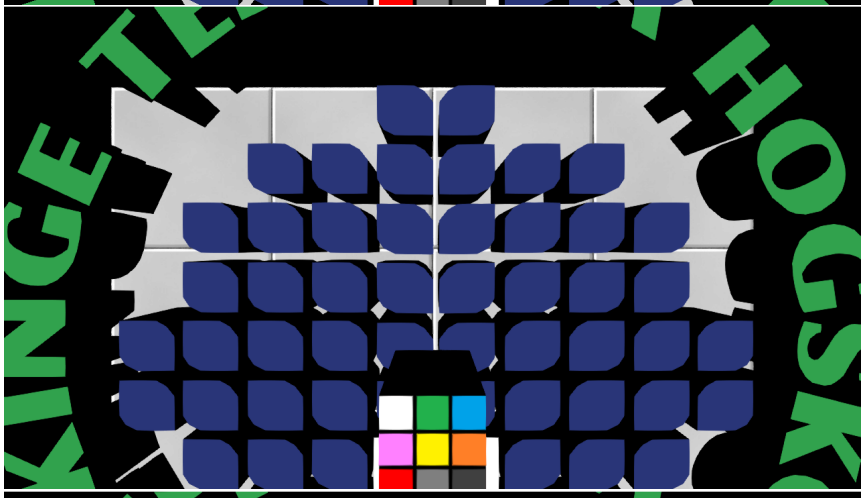
In this chapter the image results will be displayed. They are in five scenes and include highlights of certain areas of interest for easier observation. These highlights are scaled up without altering the pixel colours. After the images, the computation test results from each computer is displayed in graphs.

3.1 Images

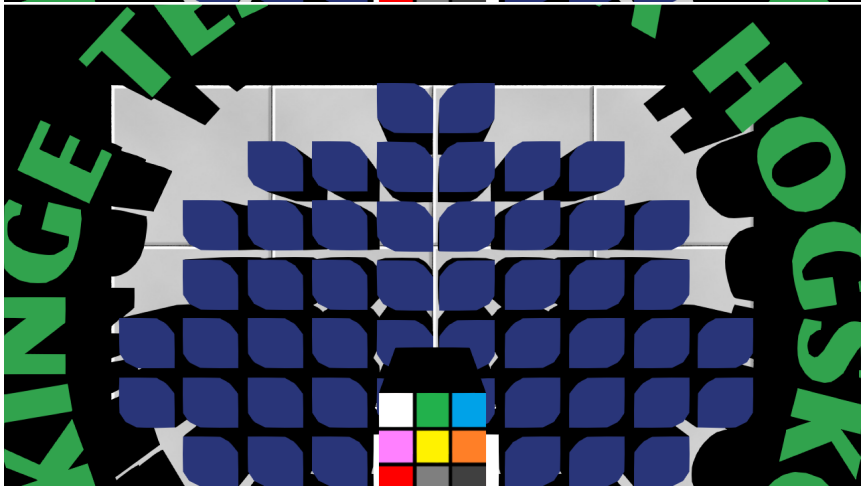
Figure 3.1: Scene A: The purpose of scene A is to capture the entire scene.



(a) NO AA.

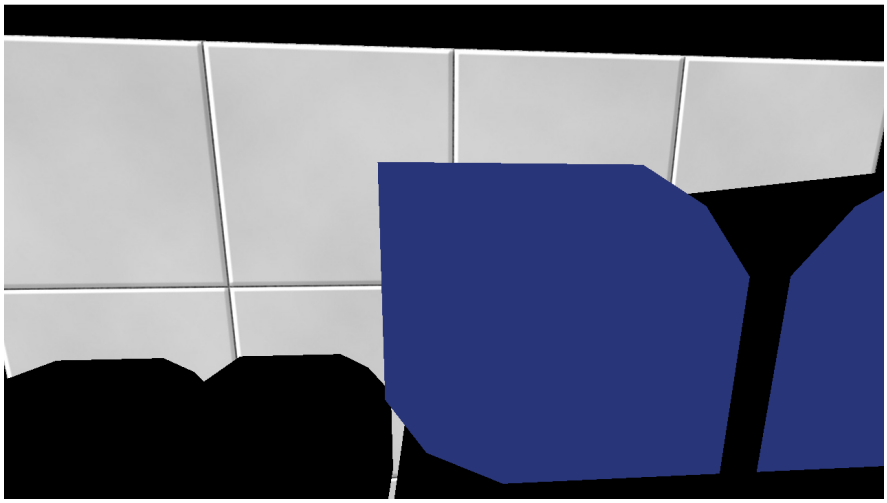


(b) FXAA.

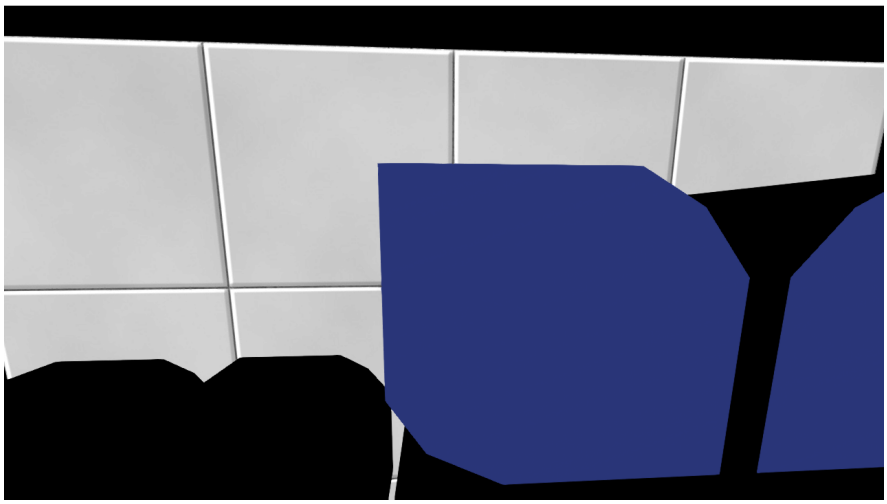


(c) SMAA.

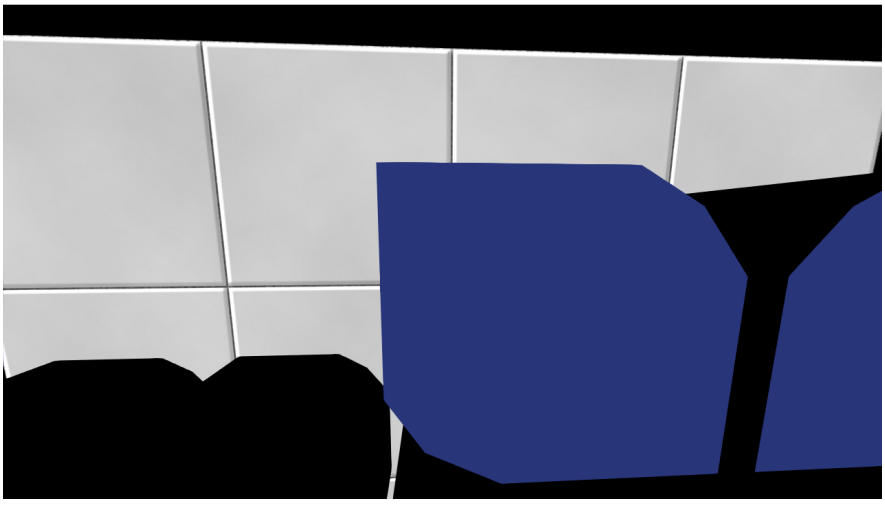
Figure 3.2: Scene B: Here we are looking on the vertical and horizontal edges of the blue object.



(a) NO AA



(b) FXAA



(c) SMAA

Figure 3.3: Scene B: A highlight of the scene to take a closer look at the edges.

(a) NO AA

(b) FXAA

(c) SMAA

Figure 3.4: Scene B: Highlight: Pixel pattern on the horizontal edge.

(a) NO AA

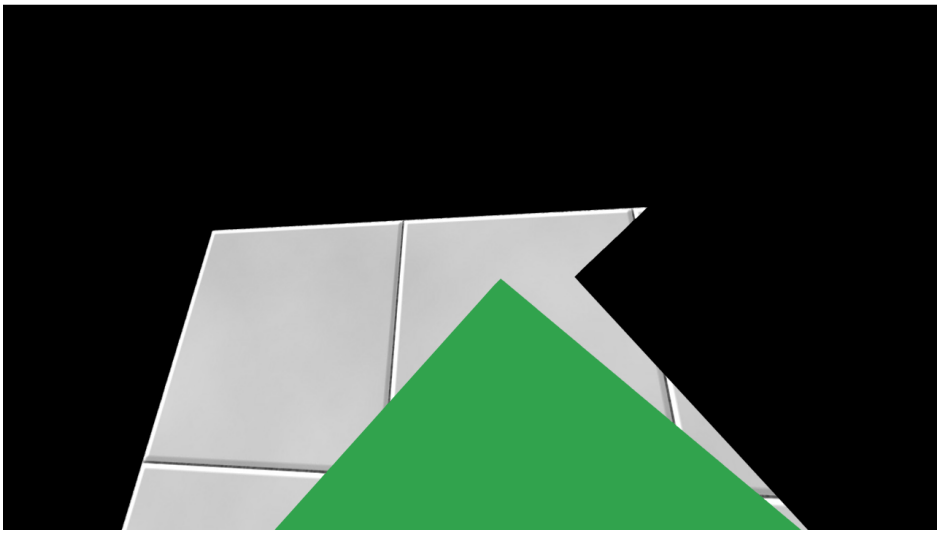
(b) FXAA

(c) SMAA

Figure 3.5: Scene C: In this scene we focus on the diagonal edges on the objects.

(a) NO AA

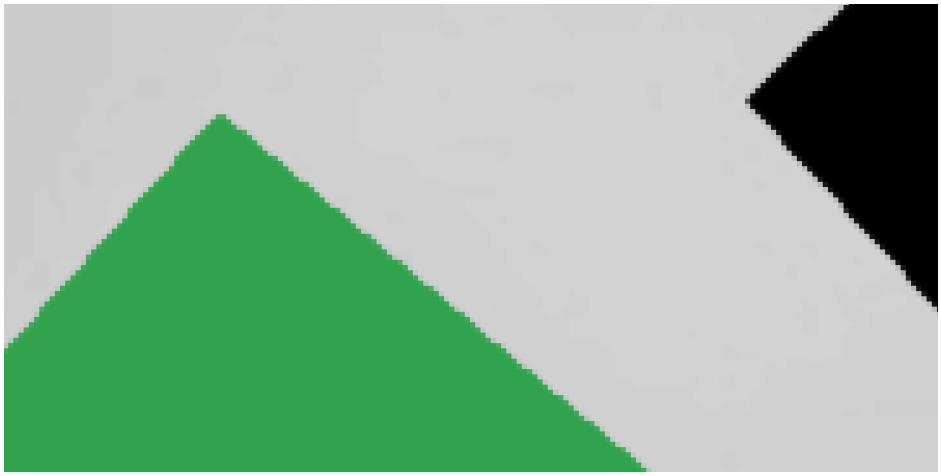
(b) FXAA



(c) SMAA

Figure 3.6: Scene C: Highlight: Close up of the diagonal edges.

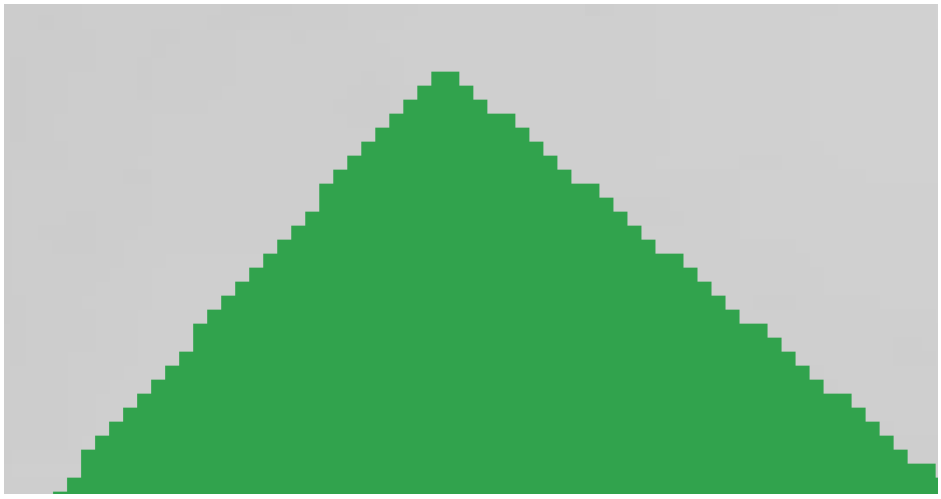
(a) NO AA



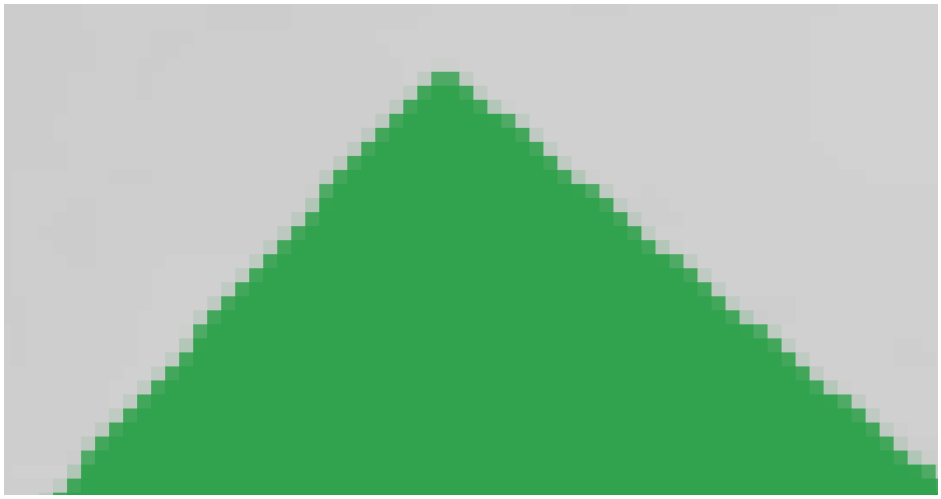
(b) FXAA

(c) SMAA

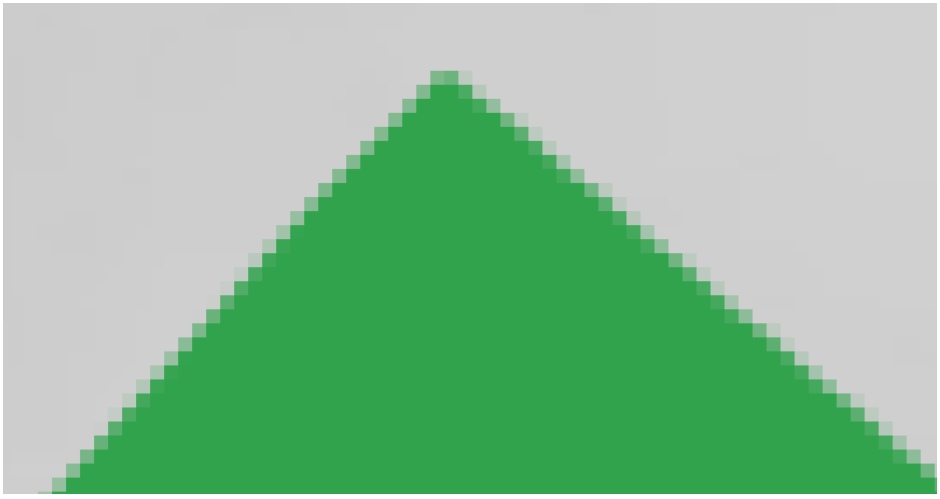
Figure 3.7: Scene C: Highlight: Pixel pattern of the diagonal edge.



(a) NO AA

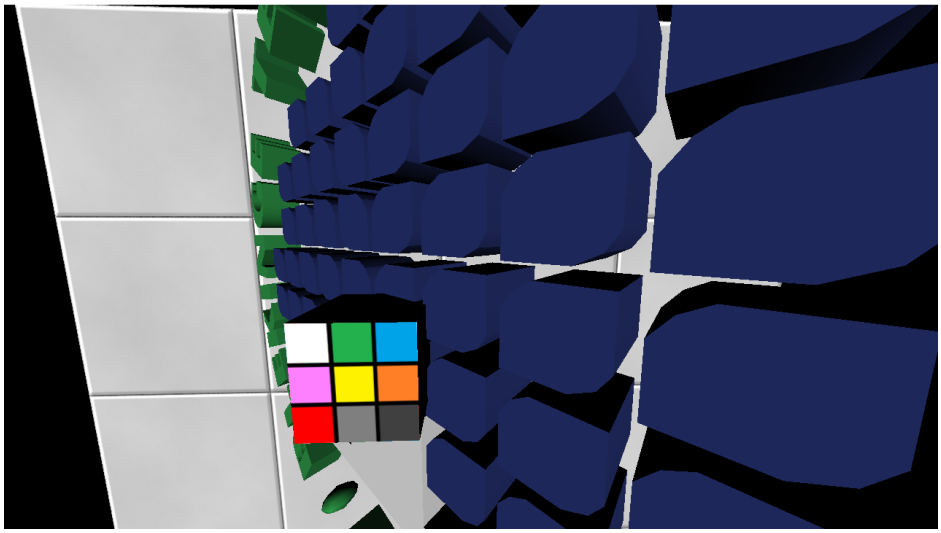


(b) FXAA

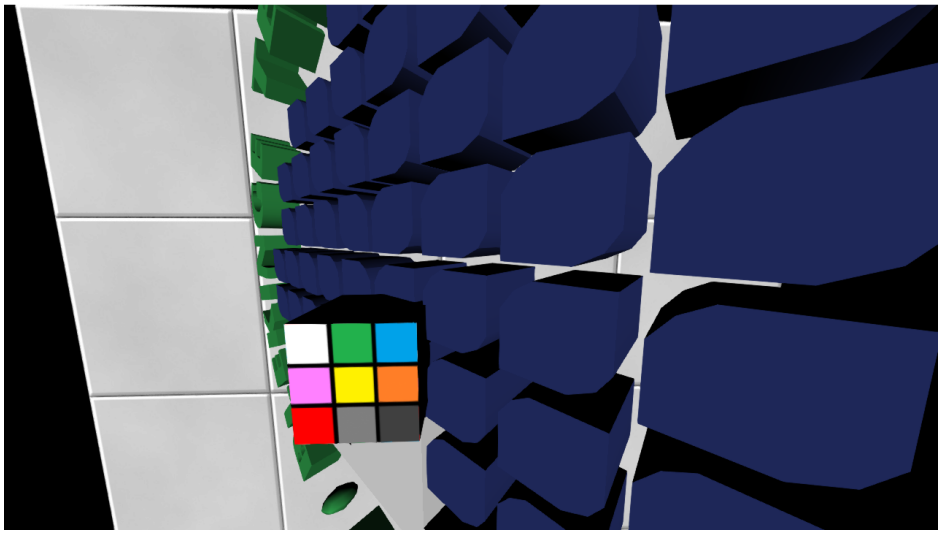


(c) SMAA

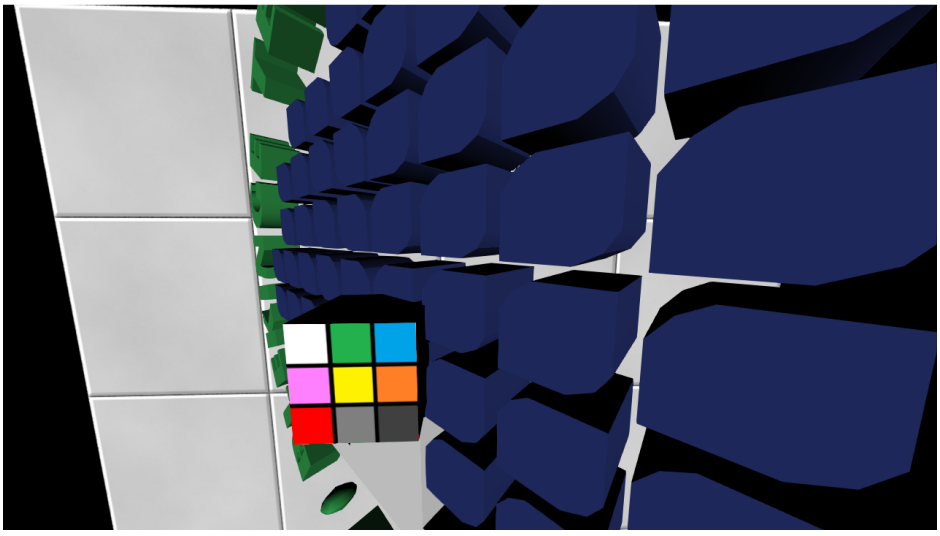
Figure 3.8: Scene D: This scene represents a scene that is riddled with aliasing



(a) NO AA

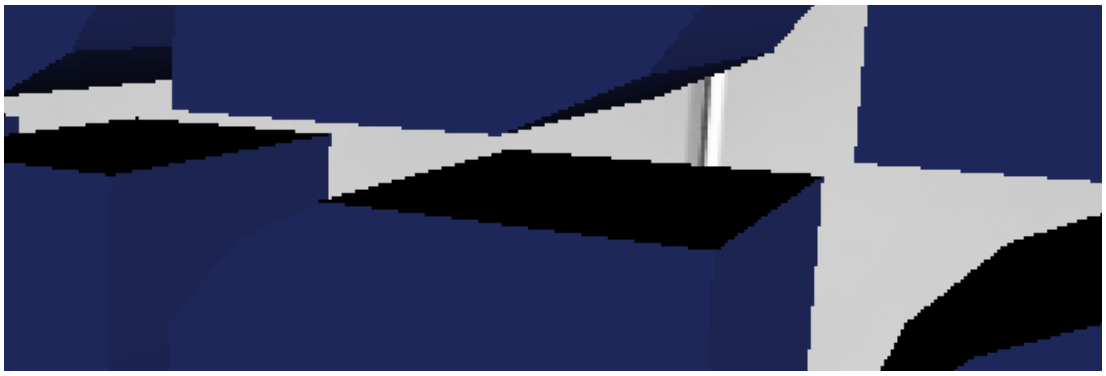


(b) FXAA

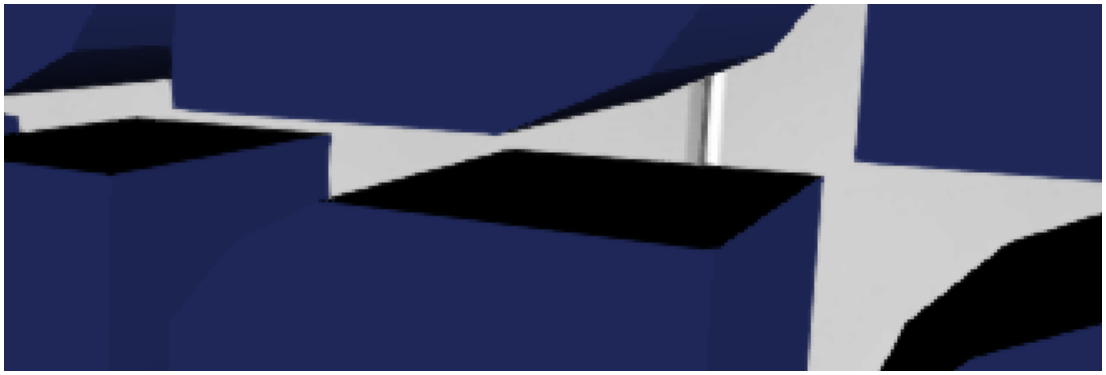


(c) SMAA

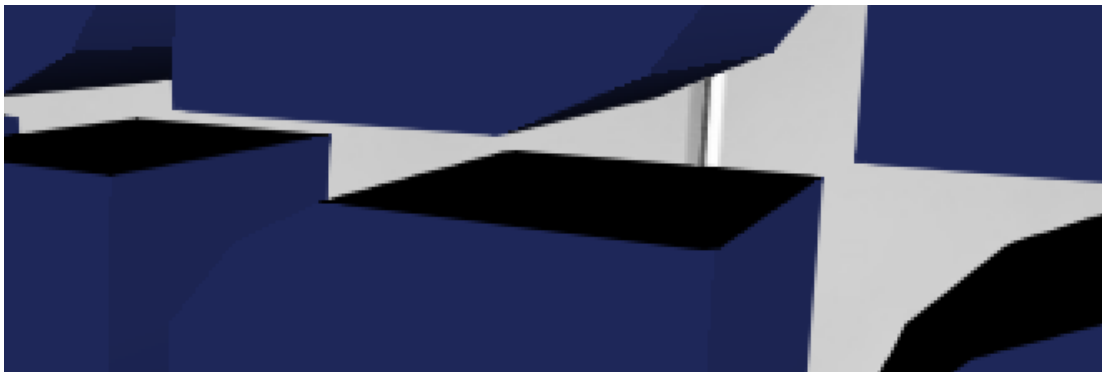
Figure 3.9: Scene D: Highlight: Various shapes and edges affected by aliasing



(a) NO AA

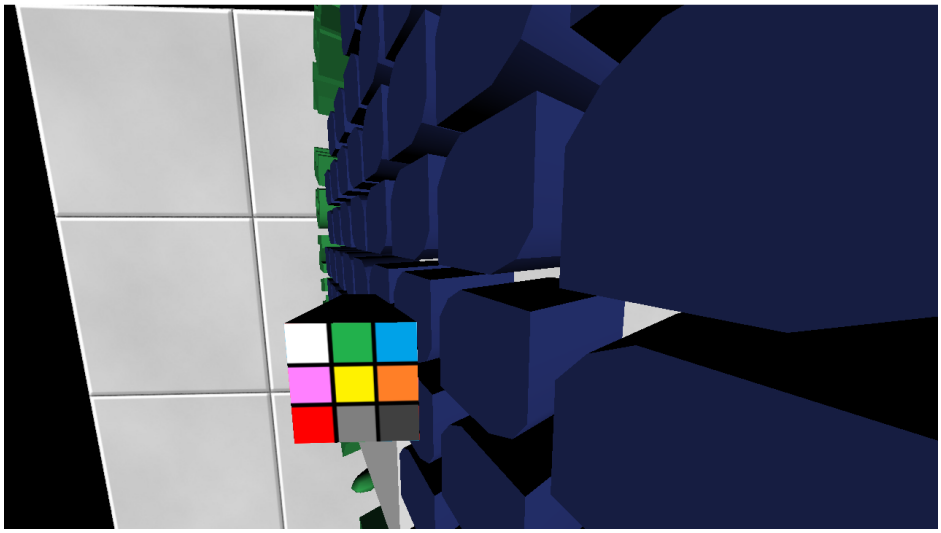


(b) FXAA

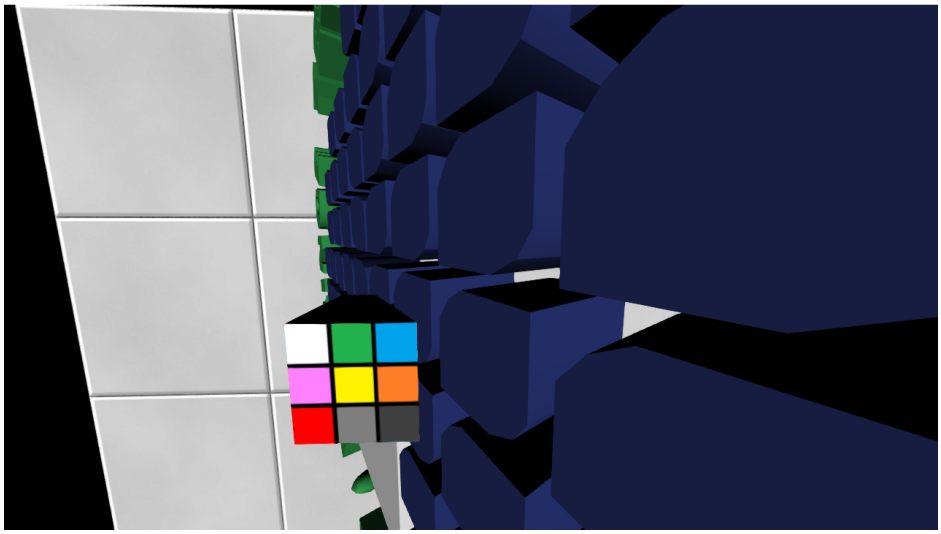


(c) SMAA

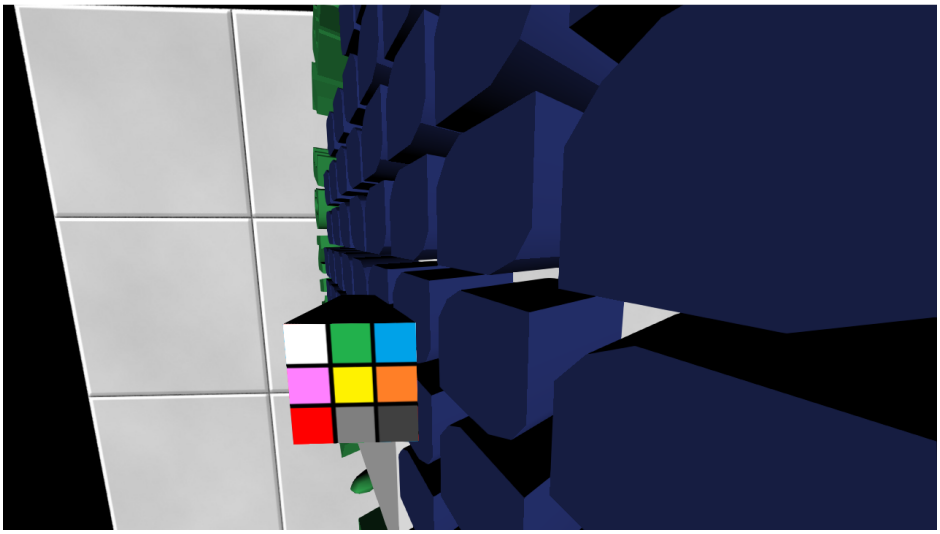
Figure 3.10: Scene E: Another example of a scene affected by aliasing.



(a) NO AA



(b) FXAA



(c) SMAA

3.2 Performance

3.2.1 Low specifications

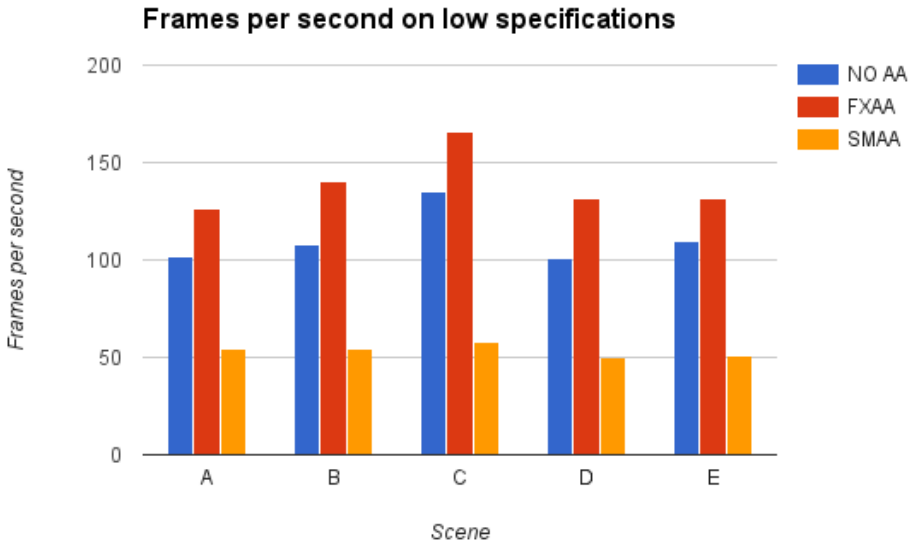


Figure 3.11: FPS measurements on the low specifications.

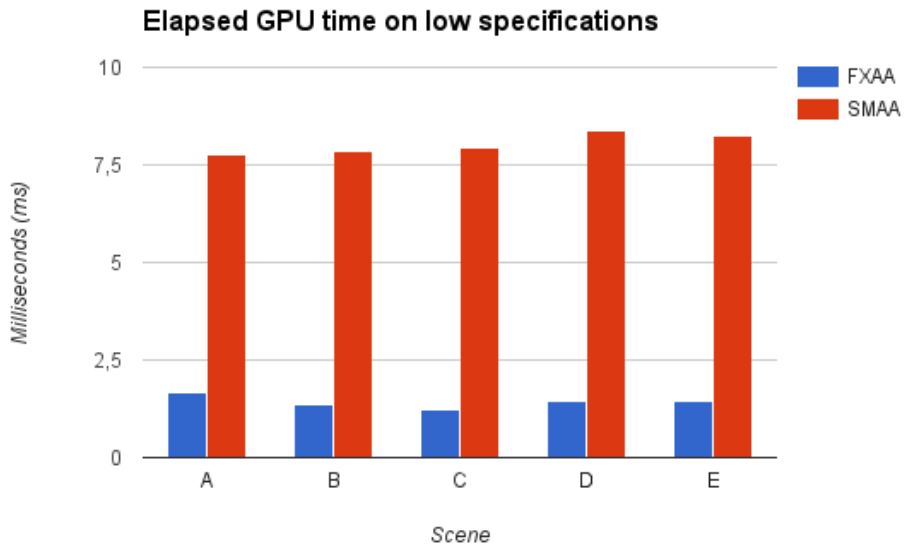


Figure 3.12: GPU measurements on the low specifications.

3.2.2 Medium specifications

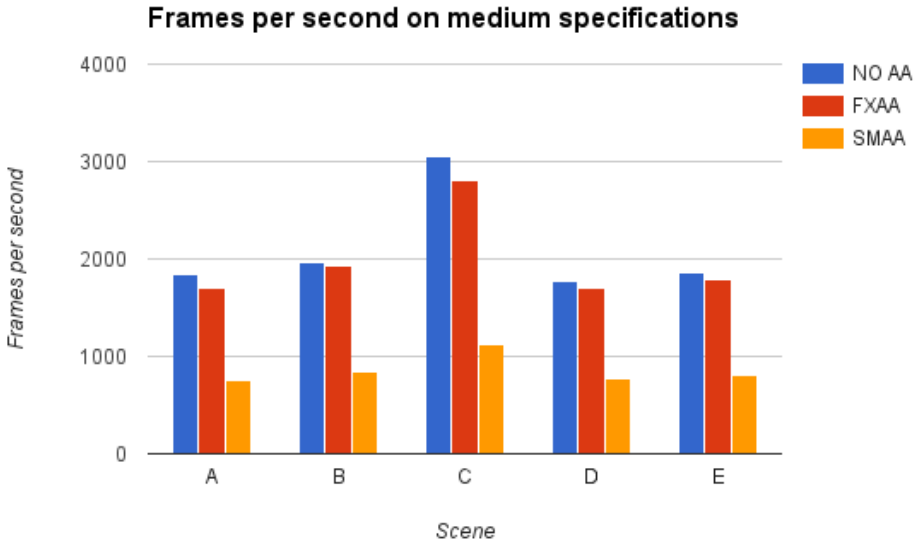


Figure 3.13: FPS measurements on the medium specifications.

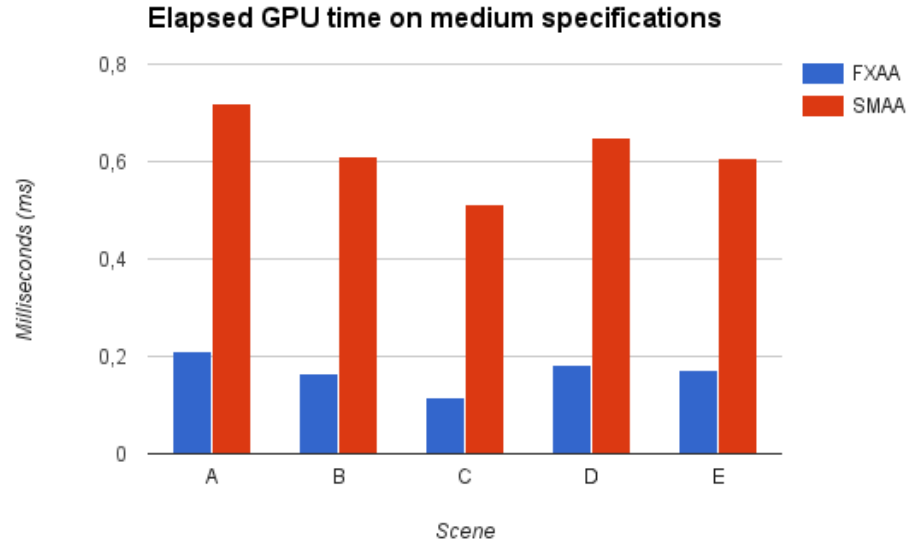


Figure 3.14: GPU measurements on the medium specifications.

3.2.3 High specifications

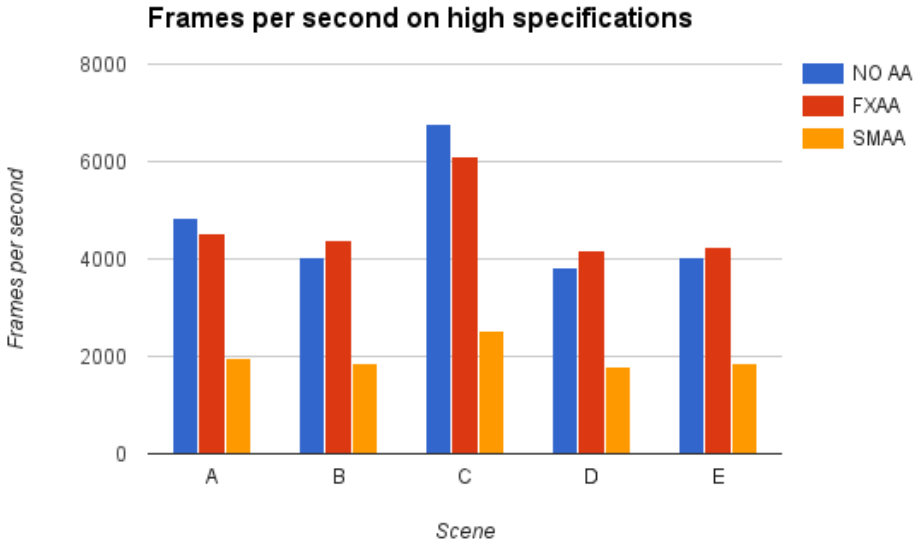


Figure 3.15: FPS measurements on the high specifications.

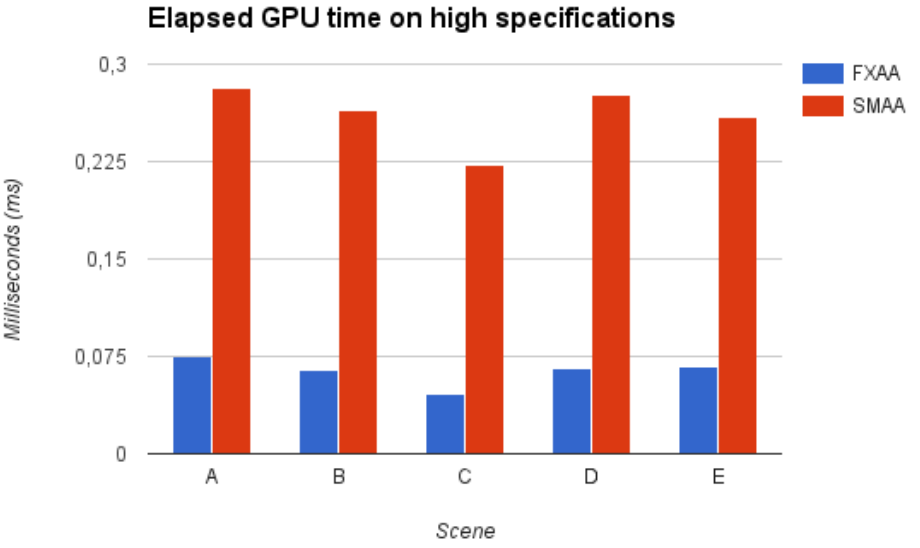


Figure 3.16: GPU measurements on the high specifications.

Chapter 4

Analysis and Discussion

This chapter will analyse the results in the previous chapter and attempt to answer the research questions in section 1.3.

4.1 Performance

SMAA use three render passes which use two textures to determine how the edge behave and how should look. FXAA use a single render pass and attempts to calculate and estimate the behaviour of the edge.

The results of the performance comparisons have a few interesting points. The biggest surprise is the increase of FPS of FXAA on the low specifications computer. The reason this occurs is that the sRGB textures are used in the entire rendering process of the scene which increases the FPS greatly. The NOAA FPS is just barely higher than FXAA on medium specifications and can vary on high specifications.

SMAA use HDR textures, the same as NOAA, and does not have the same FPS benefits. Instead, the FPS cuts in half compared to running the application without antialiasing. The thing to note, however, is that it does this regardless of computer setup.

The elapsed GPU time show similarities between FXAA and SMAA across all specifications. FXAA is three times or more fast than SMAA. The largest scale in-between the two is the low specifications, as the SMAA hits above 7.5ms to render. Other than that, the results are expected. The added render passes of SMAA, along with the extensive searches in the second render pass, lines up very similar to the scale against FXAA.

4.2 Image Quality

The NOAA images clearly show severe aliasing issues which supports the need of antialiasing methods. In all of the zoomed out figures, 3.2, 3.9 and 3.11, both of the method improve the quality greatly. It can be difficult to even see the differences between FXAA and SMAA without the zoomed in figures.

In the scene B, figure 3.3-5, shows that FXAA handle vertical and horizontal edges well but have a few graphical artefacts remaining. The gradient is not smooth all the way through like SMAA. The most interesting point shows in figure 3.5 with the difference of the grey vertical line blur. FXAA adds a line of blue blur that should not be blue.

Diagonally, figures 3.5-3.7, FXAA show clear outlines of the jagged edges. SMAA is able to handle it better with its diagonal search to detect the pattern. It is smooth all the way through.

These issues becomes clear when reviewing the algorithms by each method. The luminance edge detection in FXAA checks from both sides of an edge and try to blur it from each side. This is necessary as FXAA will otherwise be unable to perform its own pattern detection. This contributes to the unwanted extra blur to the image. SMAA can calculate the single pixel border as it follows up with the advanced pattern detection which identifies the edge behaviour. That helps to maintain the sharpness in the image.

Each method use similar methods of traversing vertically and horizontally. SMAA traverses diagonally as well which helps to adjust the values which is instead calculated and estimated in FXAA.

4.3 Validity

Each method have a number of customizable variables. These variables can be set by the programmer or by the available presets in the source code. The results can therefore change due to these settings. Each variable can improve a certain aspect of an image but may worsen another in the process. In order to avoid these issues or taking a stance of what is right or wrong, any variable that was not within a preset was set to a predefined default value by the author.

The difference between the presets is primarily the number of search steps orthogonally. SMAA use 16 steps on high quality and FXAA will do 12 on extreme quality. Deceptively, these values do not correlate well as it is only the number of iterations of the search. Each method use different increments for the reviewed pixels.

Chapter 5

Conclusions

In this paper I show the differences between FXAA and SMAA. FXAA is more optimised towards low range specifications where it can lead to increased FPS due to sRGB textures. The algorithm does not try to make a correct solution to aliasing but attempts at something that is fast and visually good. FXAA have problem in correcting aliasing in diagonal edges but perform well on vertical and horizontal lines. As the algorithm has check from both sides of an edge, it might cause some unwanted blur effects. The algorithm makes estimations with the help of predefined constants in order to guess the colour of the pixel.

SMAA attempts give the best image as it can. It does so by having three render passes and compare the edges to predefined patterns to determine the behaviour. It creates a very good image that can handle horizontal, vertical and diagonal edges well, but the performance evaluation shows that it costs more. It is three to four times slower than FXAA. The FPS is cut if half compared to running without antialiasing. This suggests it might be more suitable for medium to high specification computers.

Chapter 6

Future Work

It would be interesting to expand and compare other antialiasing methods that is not as established as the methods in this paper. That could also include the techniques used to determine if one is better than the other in hopes to find optimizations.

This paper could be expanded in other ways by exploring and comparing the techniques against super- or multisampling methods. There is also the possibility of investigating temporal and spatial coherence between frames. That is, if you move the camera or objects quickly, it is important that the scene do not become blurry or display graphical artefacts.

Another aspect for future work is expanding on the testing with more features, e.g., advanced patterns, partially see through objects, text, smoke or fog.

The evaluation of PPAA methods can be difficult as the methods may not attempt to give a correct solution but rather something that is visually sufficient. User studies and other perceptual evaluations studies can be performed in order to determine which method is better or worse.

Acknowledgements

I like to acknowledge Oscar Thaung in helping me with a few programming problems and providing a computer suitable for the high specification computer.

References

[1] B.Kristof, D. Barron, (2000, Apr. 28), "Super-sampling anti-aliasing analyzed.", [Online]. Available: https://www.beyond3d.com/ [Accessed: 2016, 05, 16]

[2] K. Akeley, "Reality engine graphics," in *SIGGRAPH '93: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, Anaheim, CA, 1993, pp. 109-116

[3] M. Pettineo. (2012, Oct. 24). *A QUICK OVERVIEW OF MSAA*[Online]. Available: https://mynameismjp.wordpress.com/2012/10/24/msaa-overview/ [Accessed: 2016, 06, 05]

[4] RESHETOV A., "Morphological antialiasing," in *Proceedings of the Conference on High Performance Graphics 2009*, New Orleans, LA, 2009, pp. 109–116

[5] T. Lottes. (2011, Jan. 25) *FXAA* (Version 1.0) [Online]. Available: http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf

[6] Jimenes, J., Echevarria J.I., Sousa, T., Gutierrez, D.: "SMAA: Enhanced Subpixel Morphological Antialiasing" *Comput. Graph. Forum* 31(2), 2012, pp. 355-364

[7] L. Davies. (2014, Mar. 18) *Conservative Morphological Anti-Aliasing(CMAA)* [Online]. Available: https://software.intel.com/en-us/articles/conservative-morphological-anti-aliasing-cmaa-update [Accessed: 2016, 05, 05]

[8] A. Herubel, V. Biri, S. Deverly, "Morphological Antialiasing and Topological Reconstruction" in *International Conference on Computer Graphics Theory and Applications (GRAPP'11)*, Vilamoura, Portugal, 2011, pp. 187-192

[9] *Dota2*[Online]. USA: Valve Corporation, 2012. [Accessed: 2016, 05, 06]

[10] *Counter-strike: Global Offensive*[Online]. USA: Valve Corporation, 2012. [Accessed: 2016, 05, 06]

[11] *XCOM 2*[Online]. USA: 2K Games, 2016. [Accessed: 2016, 05, 06]

[12] *Grand Theft Auto V*[Online]. USA: Rockstar Games, 2013. [Accessed: 2016, 05, 06]

[13] *Hitman*[Online]. Japan: Square Enix, 2016. [Accessed: 2016, 05, 06]

[14] *Arma 3*[Online]. Czech Republic: Bohemia Interactive, 2013. [Accessed: 2016, 05, 06]

[15] S. Gibson, (2016), *Sub-Pixel Font Rendering Technology*, [Online] Available: https://www.grc.com/ctwhat.htm [Accessed: 2016, 05, 15]

[16] J. Sheedy, Y-C. Tai, M. Subbaram, et al. "ClearType sub-pixel text rendering: Preference, legibility and reading performance." *Displays* 29.2, 2008, pp.138-151.

[17] J. Jimenez, D. Gutierrez, J. Yang, et al. "Filtering Approaches for Real-Time Anti-Aliasing", in *ACM SIGGRAPH Courses*, 2011

[18] J. Hable, (2010, Jun. 21), *Linear-Space Lighting(i.e. Gamma)*, [Online] Available: http://filmicgames.com/archives/299 [Accessed: 2016, 06, 07]

[19] M. Stokes, M. Anderson, S. Chandrasekar, R. Motta. (1996 Nov. 5), *A Standard Default Color Space for the Internet - sRGB*, [Online] Available: http://www.color.org/sRGB.xalter [Accessed: 2016, 06, 07]

[20] JeGX, (2011, Apr. 5) *(Tested) Fast Approximate Anti-Aliasing (FXAA) Demo (GLSL)* [Online]. Available: http://www.geeks3d.com/20110405/fxaa-fast-approximate-anti-aliasing-demo-glsl-opengl-test-radeon-geforce/ [Accessed: 2016, 05, 06]

[21] A. Reshetov. "Reducing aliasing artifacts through resampling." *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics.* Aire-la-Ville, Switzerland, 2012. pp. 77-86.

[22] R. Brewster *Paint.net*[Online] Available: http://www.getpaint.net/ [Accessed: 2016, 04, 14]

[23] K. Höppner, "Strategies for including graphics in LATEX documents.", *Practical TEX 2005 Conference Proceedings*, Chapel Hill, North Carolina, 2005, pp. 59-62.

Appendices

Appendix A

FXAA Code

A.1 Luminance conversion pass

```
Texture2D backBuffer : register(t0);
SamplerState Sampler : register(s0);

struct PSIn
{
    float4 Position : SV_POSITION;
    float2 TexCoord : TEXCOORD;
};

float4 PS_FXAALUMA(PSIn _input) : SV_TARGET
{
  float4 color = backBuffer.Sample(Sampler, _input.TexCoord);
  color.a = dot(color.xyz, float3(0.299, 0.587, 0.114));
  return color;
}
```

A.2 Apply FXAA 3.11

```
Texture2D sceneTexture : register(t0);
SamplerState Sampler : register(s0);

#define FXAA_PC 1
#define FXAA_HLSL_5 1
#define FXAA_QUALITY__PRESET 39

#include "Assets/Shaders/Fxaa3_11.h"

cbuffer FXAABuffer {
  float2 rcpFrame; // (1.0f/width, 1.0f/height)
  float4 rcpFrameOpt; // (2.0f/width, 2.0f/height, 0.5f/width, 0.5/
    height)
};

struct PSIn {
  float4 Position : SV_POSITION;
  float2 TexCoord : TEXCOORD;
};

float4 PS_FXAA(PSIn _input) : SV_TARGET {
  int width = 0;
  int height = 0;
  sceneTexture.GetDimensions(width, height);

  float qualitySubpixels = 0.75f; //default
  float qualityEdgeThreshold = 0.125f; //high quality
  float qualityedgeThresholdMin = 0.0625f; //high quality

  FxaaTex tex;
  tex.smpl = Sampler;
  tex.tex  = sceneTexture;

  float2 screenPos = (_input.TexCoord.xy);
  return FxaaPixelShader( screenPos, float4( 0, 0, 0, 0 ),
  tex, tex, tex,
  rcpFrame,
  rcpFrameOpt, float4( 0, 0, 0, 0), float4( 0, 0, 0, 0),
  qualitySubpixels, qualityEdgeThreshold, qualityedgeThresholdMin,
  0, 0, 0,
  float4( 0, 0, 0, 0));
}
```

Appendix B

SMAA Code

Common code amongst all SMAA shaders:

```
#define SMAA_HLSL_4 1
#define PS_VERSION ps_4_1
#define SMAA_PRESET_HIGH 1

cbuffer SMAABuffer {
  float2 TexelSize; // is (1.0f/width, 1.0f/height)
  float2 padding;
};

#define SMAA_PIXEL_SIZE TexelSize
#include "Assets/Shaders/SMAA.h"
```

B.1 Edge Detection Pass

B.1.1 Vertex Shader

```
struct VSIn {
  float3 Position : POSITION;
  float2 TexCoord : TEXCOORD;
};

struct VSOut {
  float4 Position  : SV_POSITION;
  float2 TexCoord  : TEXCOORD;
  float4 Offset[3] : TEXCOORD2;
};

VSOut VS_SMAAEdgeDetection(VSIn _input) {
  VSOut output;

  output.Position = float4(_input.Position, 1.f);
  output.TexCoord = _input.TexCoord;
  SMAAEdgeDetectionVS(
```

```
    output.Position,
    output.Position,
    output.TexCoord,
    output.Offset);

    return output;
}
```

B.1.2 Pixel Shader

```
Texture2D sceneTexture : register(t0);

struct PSIn {
    float4 Position  : POSITION;
    float2 TexCoord  : TEXCOORD;
    float4 Offset[3] : TEXCOORD2;
};

float4 PS_SMAAEdgeDetection(PSIn _input)  : SV_TARGET {
    return SMAALumaEdgeDetectionPS(
    _input.TexCoord,
    _input.Offset,
    sceneTexture);
}
```

B.2 Blending Weight Calculations Pass

B.2.1 Vertex Shader

```
struct VSIn {
    float3 Position : POSITION;
    float2 TexCoord : TEXCOORD;
};

struct VSOut {
    float4 Position : SV_POSITION;
    float2 TexCoord : TEXCOORD;
    float2 Texel    : TEXCOORD2;
    float2 PixCoord : TEXCOORD3;
    float4 Offset[3]: TEXCOORD4;
};


```

```
VSOut VS_SMAABlendingWeight(VSIn _input) {
  VSOut output;

  output.Position = float4(_input.Position, 1.f);
  output.TexCoord = _input.TexCoord;
  output.Texel = TexelSize;

  SMAABlendingWeightCalculationVS(
  float4(_input.Position, 1.f),
  output.Position,
  output.TexCoord,
  output.PixCoord,
  output.Offset);

  return output;
}
```

B.2.2 Pixel Shader

```
Texture2D edgesTexture  : register(t0);
Texture2D areaTexture   : register(t1);
Texture2D searchTexture : register(t2);

//Constants for the area texture that is given by SMAA
#define SMAA_AREATEX_MAX_DISTANCE 16
#define SMAA_AREATEX_PIXEL_SIZE (1.0 / float2(160.0, 560.0))
#define SMAA_AREATEX_SUBTEX_SIZE (1.0 / 7.0)
#define SMAA_AREATEX_MAX_DISTANCE_DIAG 20

struct PSIn {
  float3 Position : POSITION;
  float2 TexCoord : TEXCOORD;
  float2 Texel    : TEXCOORD2;
  float2 PixCoord : TEXCOORD3;
  float4 Offset[3]: TEXCOORD4;
};

float4 PS_SMAABlendingWeight(PSIn _input)  : SV_TARGET {
  return SMAABlendingWeightCalculationPS(_
  input.TexCoord,
  _input.PixCoord,
  _input.Offset,
  edgesTexture,
  areaTexture,
  searchTexture, 0);
}
```

B.3 Blending Weight Calculations Pass

B.3.1 Vertex Shader

```
struct VSIn {
  float3 Position : POSITION;
  float2 TexCoord : TEXCOORD;
};

struct VSOut {
  float4 Position  : SV_POSITION;
  float2 TexCoord  : TEXCOORD;
  float4 Offset[2] : TEXCOORD2;
};

VSOut VS_SMAANeighbourhoodBlending(VSIn _input) {
  VSOut output;

  output.Position = float4(_input.Position, 1.f);
  output.TexCoord = _input.TexCoord;

  SMAANeighborhoodBlendingVS(
  float4(_input.Position, 1.f),
  output.Position,
  output.TexCoord,
  output.Offset);

    return output;
}
```

B.3.2 Pixel Shader

```
Texture2D sceneTexture : register(t0);
Texture2D blendTexture : register(t1);
struct PSIn {
  float3 Position  : POSITION;
  float2 TexCoord  : TEXCOORD;
  float4 Offset[2] : TEXCOORD2;
};

float4 PS_SMAANeighbourhoodBlending(PSIn _input)  : SV_TARGET {
  return SMAANeighborhoodBlendingPS(
  _input.TexCoord,
  _input.Offset,
  sceneTexture,
  blendTexture);}
```

B.3.3 Pixel Shader

```
Texture2D sceneTexture : register(t0);
Texture2D blendTexture : register(t1);
struct PSIn {
  float3 Position  : POSITION;
  float2 TexCoord  : TEXCOORD;
  float4 Offset[2] : TEXCOORD2;
};

float4 PS_SMAANeighbourhoodBlending(PSIn _input)  : SV_TARGET {
  return SMAANeighborhoodBlendingPS(
  _input.TexCoord,
  _input.Offset,
  sceneTexture,
  blendTexture);
}
```