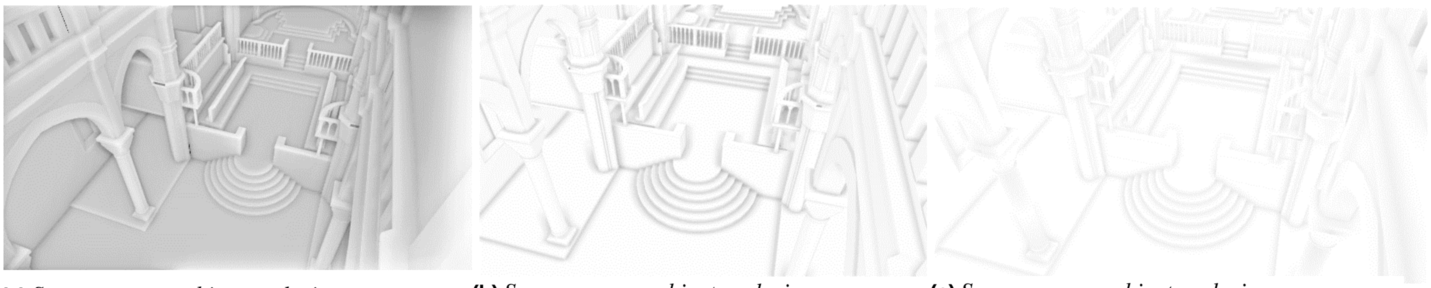
6COSC023W – Final Project Report

Examining the different Ambient Occlusion techniques used in Real-time Rendering

Student
Shahbaaz Hussain (w1726400)
University of Westminster

Supervisor
Anastasia Angelopoulou
University of Westminster



(a) *Screen-space ambient occlusion (Mittring, 2007)*

(b) *Screen-space ambient occlusion (Bavoil and Sainz, 2008)*

(c) *Screen-space ambient occlusion (McGuire et al., 2011b)*

Figure 1*: Sibenik Cathedral (Dabrovic, 2002) Demonstrating the three screen-space ambient occlusion approaches that will be discussed.*

This report is submitted in partial fulfilment of the requirements for the
BSc (Hons) Computer Science degree
BEng Software Engineering degree
at the University of Westminster.

School of Computer Science & Engineering

University of Westminster

Date: 09/10/21

Abstract

In this report we examine the different screens-space ambient occlusion (SSAO) methods most used in Real-time rendering applications. The aim is to compare the candidate methods used for ambient occlusion (AO) and evaluate their strengths and weaknesses to conclude which one among the most used methods is superior. The criterion for a superior method is one that offers both performance and visual fidelity where we consider a method performant if it computes ≤ 3ms. The candidate AO methods are tested under a quality and performance configuration with the aim of matching a ray-traced reference under both configurations by adjusting their relevant parameters. Performance metrics for each AO method for both configurations are captured at 1080p using Nvidia Nsight Graphics. We capture a wide shot of a scene for each method and note the performance metrics and visual-fidelity under both quality and performance configurations. This report further evaluates the performance configuration by measuring performance and visual-output using a close-up of scene objects as the methods in their performance configurations, due to lower sample count can produce visible artifacting and noise. This report uncovers that (Mittring, 2007) AO, while it computes faster than HBAO in both performance configuration tests, computing in 1.34ms compared to 3.16ms for HBAO in our wide-shot test and computing in 1.45ms for the close-up test compared to 3.16ms for HBAO; the method produces an unpleasing visual result with much self-occlusion, attributed to the methods use of a sphere for sampling, producing a result far from realistic AO thus failing to match the ray-traced reference. HBAO proved to be a very competitive candidate for the superior method, producing a visual result that came close to matching the ray traced reference. However, the visual fidelity produced came at a performance cost, which saw slower compute times in both the performance and quality configuration tests, computing in 3.16ms for both the performance configuration wide-shot and close-up test, and computing in 30.02ms for the quality configuration test. Alchemy AO produced a visual result that closely matched the output of HBAO, and the ray traced reference thus we compared the results of HBAO to Alchemy AO under the same test configurations and saw Alchemy AO compute in 1.10ms and 1.17ms for the performance configuration wide shot and close-up test respectively and computed in 5.88ms under the quality configuration test. The results we obtained from the testing uncovered that the Alchemy AO method could produce a visually pleasing result that closely matched our ray traced reference while maintaining acceptable compute times under both performance and quality configuration tests. The performance seen in our test for Alchemy AO under the quality configuration made it evident that the AO method scales well with higher samples and sampling radius to produce a high-quality output. The results we obtained from testing the candidate AO methods uncovered that the Alchemy AO method is superior among the other candidate methods, offering superior performance and visual fidelity in both test configurations.

Declaration

This report has been prepared based on my own work. Where other published and unpublished source materials have been used, these have been acknowledged in references.

Word Count: 11,797
Student Name: Shahbaaz Hussain
Date of Submission: 4/05/22

Table of Contents

Abstract 2
Table of Contents 4
List of Figures 6
List of Tables 8
1 Introduction 9
1.1 Motivation 9
1.2 Purpose 9
1.3 Aims and Objectives 10
1.4 The Project 10
1.4.1 Project requirements 12
1.5 Brief History 13
2 Background 14
2.1 Rendering Equation 14
2.2 Ambient Occlusion 14
2.3 Real-Time Rendering 15
2.3.1 Deferred Rendering 15
2.3.2 Depth and Positions 16
2.3.3 Normal 16
2.3.4 Colour 16
2.3.5 Shaders 17
3 Analysis 17
3.1 Raytracing and Rasterization 17
3.1.1 Rasterization 17
3.1.2 Raytracing 18
3.1.3 Monte Carlo Integration 18
3.2 Screen-Space Integration 18
3.3 Previous Work 19
3.4 Crytek Screen-Space Ambient Occlusion 19
3.5 Horizon-Based Ambient Occlusion 21
3.5.1 Screen-space Integration 22
3.6 Alchemy Ambient Occlusion 23
3.7 Overview 25
4 Implementation 25
4.1 Outline 25

4.2 Crytek CryEngine 2 SSAO 26
4.2.1 Retrieving data 26
4.2.2 Samples and Random vectors 26
.... 27
4.2.3 Determine Occlusion factor 27
4.3 Horizon-Based Ambient Occlusion 30
4.4 Alchemy Ambient Occlusion 35
4.5 Blur 38
5 Results & Analysis 39
5.1 Testing Methodology 39
5.1.1 Table Results 40
5.2 Performance Configurations 40
5.2.1 Close-up 41
5.3 Quality Configurations 41
5.4 Superior method 45
5.5 Future work 45
5.6 Conclusion 46
6 References 47
7 Bibliography 50
8 Appendix 54

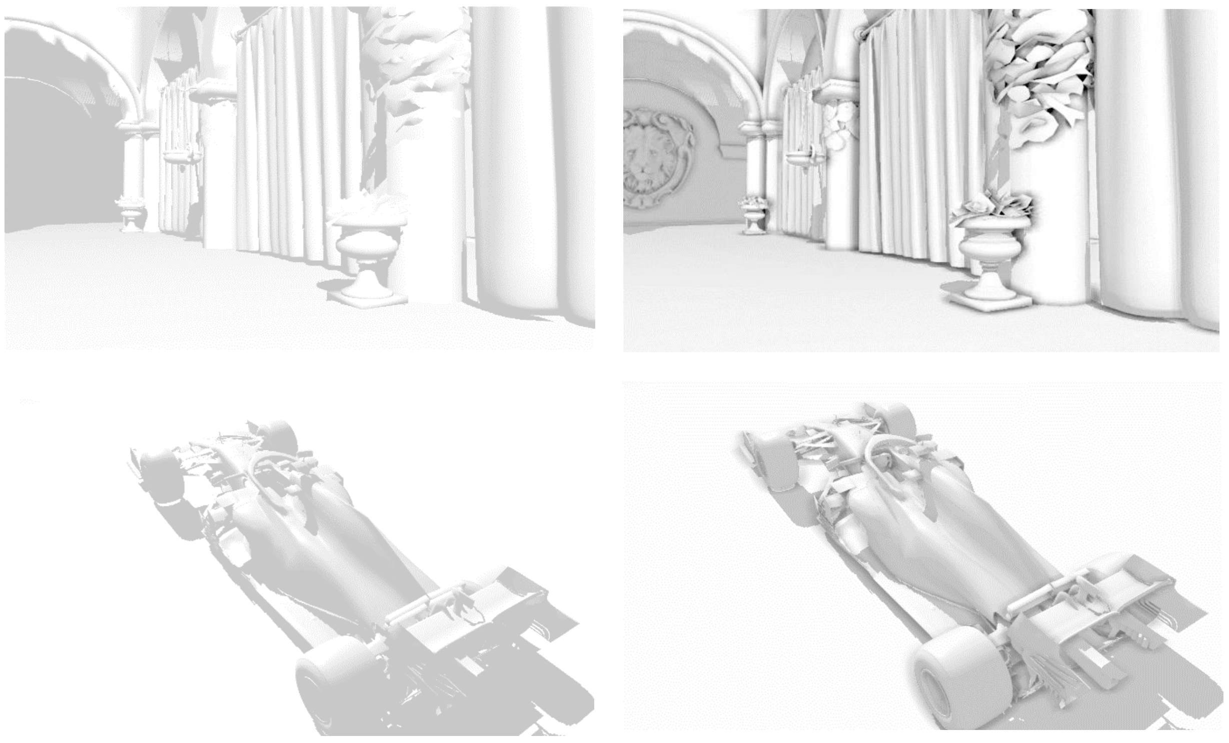
List of Figures

Figure 1*: Sibenik Cathedral (Dabrovic, 2002) Demonstrating the three screen-space ambient occlusion approaches that will be discussed.* 1
Figure 2*: Demonstrating the visual detail provided by an SSAO method (HBAO).* 9
Figure 3*: Rendering Equation visualised.* 14
Figure 4: *Reconstruct positions to view-space using the depth buffer (ZaOniRinku, 2021).* 16
Figure 5: *Buffers commonly used for deferred rendering.* 16
Figure 6: *Approximating Ambient Occlusion using offset points distributed around the surrounding sphere (Mittring, 2007). The green circles are the visible samples; the red are the occluded samples.* 20
Figure 7: *Ambient occlusion approximation using hemisphere to handle self-occlusion (Filion and McNaughton, 2008).* 21
Figure 8: *Visual overview of HBAO showcasing the required components and workings of the HBAO approach to approximate AO.* 23
Figure 9: *(McGuire et al., 2011b, p.27) Geometric construction to demonstrate the approximation of AO using Alchemy AO.* 25
Figure 10: *Generating random samples to use to determine the occlusion factor around a fragment (de Vries, 2020).* 26
Figure 11: *Generating random vectors to reduce the number of samples required (de Vries, 2020).* 27
Figure 12: *4 x 4 texture containing random rotation vectors to rotate and randomise sphere sample kernel (left). Noise texture containing random vectors in GPU (right).* 27
Figure 13: *SSAO false occlusion without range check [No Blur].* 29
Figure 14: *SSAO with range check [No Blur].* 29
Figure 15: *Rotating direction by angle where V is the vector of the direction angle and A is offset vector with a given angle which will be used to rotate the direction vector to create V' and B is the vector A rotated $\pi 2$ to construct a coordinate system.* 32
Figure 16: *Tangent bias 0.3.* 33
Figure 17: *No tangent bias.* 33
Figure 18: *Distribution of samples: Uniform sampling in disk (left) non-uniform sampling in disk (right).* 36
Figure 19: *(a) Without blur, random noise pattern is visible. (b) Blur filter mitigates the random texture noise.* 38
Figure 20: *Testing methodology visualized for a single AO method (HBAO). We test the AO method under a quality configuration setting (left) and gather the performance metrics using Nvidia Nsight Graphics. We also observe the visual output of the method under the quality configuration to see the level of detail produced and compare it to the ray traced reference. Similarly, we test the AO method under its performance configuration (right) and gather the performance metrics using Nvidia Nsight Graphics and observe the visual output produced by the AO method, also comparing it to the ray traced reference.* 40
Figure 21: *Performance of AO methods captured over 5-frames. Performance configuration (left) Quality configuration (right). All AO methods hold a relatively consistent framerate when running in real-time. We attribute the consistence frame rates largely to the fact that the observed scene is still, with no animation or movement to cause the AO method to significantly shift in compute time per-frame thus we leave measurement of framerates over multiple frames in real-time out of the testing methodology when evaluating the AO methods.* 42
Figure 22: Ray traced reference rendered using Arnold in Maya. SSAO methods will be compared against this reference render (wide shot). 43

Figure 23: Ray traced reference rendered with Arnold in Maya for close-up comparison for SSAO methods. 43

List of Tables

Table 1: Project requirements detailed before development of the accompanying software to ensure useability and meaningfulness....13
Table 2: *AO methods with their required input for calculation and some of their appearances in real-time graphics applications.* 25
Table 3: *Performance configuration [Match ray-traced reference]: Wide shot.*....44
Table 4: *Performance configuration [Match ray-traced reference]: Close-up.* 44
Table 5*: Quality configuration [Match ray-traced reference].*....45



(a) *Scene render using Blinn-Phong lighting and shadows without screen-space ambient occlusion.*

(b) *Scene render using Blin-Phong lighting and shadows with screen-space ambient occlusion used (Bavoil, Sainz and Dimitrov, 2008).*

Figure 2*: Demonstrating the visual detail provided by an SSAO method (HBAO).*

1 Introduction

Ambient Occlusion (AO) is a technique used in both real-time as well as offline rendering to visually improve the detail and lighting in a scene. It proves to be an important technique in computer graphics to enhance the perception and detail of scene topology for the human eye (Kajalin, 2009). In this section we discuss the motivation behind the examination of ambient occlusion (AO), the intent of this research and the project that will be produced. Then we will briefly discuss the history of ambient occlusion and the emergence of screen-space approaches for real-time rendering.

1.1 Motivation

One of the most sought-after goals in the field of computer graphics is to be able to generate realistic looking images in real-time frame rates where real-time framerates fall between an acceptable range of 30-60 frames per second (FPS). Ambient occlusion is an effect that adds a great amount of depth and realism to computer generated images, outlining the soft shadow detail, indicating contact between objects within the scene, increasing the visual fidelity and providing greater realism to the scene. The depth and detail created by AO proves to be crucial in enhancing the realism of a scene (**Figure 2**). The significance of AO in a computer-generated scene, motivated the exploration of the various approaches regarding real-time AO.

1.2 Purpose

Ambient Occlusion (AO) implementations in real-time graphics applications have existed in various forms over the years. The varying methods proposed for ambient occlusion in real-time rendering opens way for questions such as which ambient occlusion technique is most superior? In order to answer this question, tests need to be performed on the varying ambient occlusion algorithms in order to observe how the different approaches fare when utilised in a real-time graphics application, understanding and comparing their strengths and weaknesses by analysing their performance and visual output under various conditions, to ultimately conclude which method among the most commonly used ambient occlusion approaches is superior in real-time graphics applications, offering both great performance and visual fidelity. This is the purpose of this thesis; to uncover the strengths and weaknesses of the most commonly used screen-space ambient occlusion (SSAO) methods in real-time graphics applications and to uncover, which method among those is superior.

The results presented will benefit those who would like to get an insight into the performance characteristics and visual fidelity offered by the varying ambient occlusion methods. While the thesis aims to uncover the superior method, the testing results will provide insightful details regarding how other ambient occlusion approaches can be utilised for different scenarios. For example, a Rendering Engineer who is looking to implement ambient occlusion within their real-time graphics application would be able to make a more informed decision, based on this research about which method would be most suitable for their project and specification. Often a method offering both performance and visual fidelity is desirable. Examples of such projects include video games where performance would be of a higher priority than visual fidelity thus the search to find and uncover an AO approach that can offer both performance and visual quality would be of great value. While a Rendering Engineer may prioritise performance, film industries looking to utilise real-time graphics applications as a back-drop during production as seen in (Farris, 2020) may prioritise visual fidelity where the price paid in performance is less of a concern.

1.3 Aims and Objectives

The aim of this research is to display the different techniques most commonly used in real-time graphics applications that can be utilised to produce ambient occlusion in real-time rendering and compare their strengths and weaknesses through testing to present the most superior method, that can offer both quality and performance. The commonality of the methods was determined through research of implemented AO methods found in Open-Source graphics projects on Github, exploration of game engines and implemented AO methods found in AAA games. This research will prove useful for teams that are; small, low-budget, or have tight schedules as these teams might find themselves allocating resources and time implementing techniques which have little-to-no visual impact on their project or does not provide the desired result and instead, negatively impacts the performance of the application, worsening the experience, thus likely an implementation that is removed. This ultimately results in a waste of development time and crucially, a waste of the allocated budget for the project. The intent of this research is to provide a detailed analysis on the varying ambient occlusion techniques used in real-time graphics applications and present a superior method offering both performance and quality. While the research will present the superior method, suggesting the most optimum solution and method to use, the research results will provide further insights on the varying AO methods through the data captured during testing, which will allow teams to make a more informed decision on which ambient occlusion technique best suits the needs of their application.

1.4 The Project

The project intends to analyse and implement a set of commonly used ambient occlusion methods in real-time graphics applications with the intent to find a method which is superior all-round among those commonly used, that is, providing both performance via low compute time and quality. The AO methods that have been selected vary algorithmically in order to achieve the broadest possible overview. Since the project focuses on techniques most often employed in real-time graphics applications, we will be limited to exploring only a few screen-space ambient occlusion methods.

The quality of the ambient occlusion methods will be visually assessed, compared with a ray traced AO image and will be displayed and observed using the same scene as a reference for each technique, to highlight and make visual differences clear. The parameters for the method for each assessment will be mentioned alongside the visuals.

The screen-space methods that we will be comparing are designed to be efficient for real-time graphics hardware thus the ambient occlusion methods will be best implemented on a GPU using shaders. The project will use the cross-platform Vulkan API to implement the candidate methods and GLSL as the shading language for the shader implementation. The shaders will be compiled to Standard Portable Intermediate Representation (SPIR-V) format in order to use the shaders with the Vulkan application. The Vulkan API has been selected as it is a modern graphics API, designed for high-performance computer graphics applications, and provides cross-platform support. The cross-platform possibility was crucial to ensure the resulting application was as accessible as possible to all users.

An interactive application will be the resulting product of the project where the user will be able to interact by flying a virtual camera around the scene in real-time, accompanied by a UI that utilises the ImGUI library which provides the user controls over the AO methods parameters which can be adjusted in real-time to inspect the visual impact the various screen-space ambient occlusion methods parameters have on the scene. The ImGUI library has been chosen for the UI as it provides an easy-to-understand interface with various built-in tools such as sliders, radio buttons, and check box buttons which-both the library and the libraries tools easily integrate into a Vulkan application to provide UI functionality and controllability. The application serves as a tool for stakeholders of the applications such as Rendering Engineers, Environment Artists, Graphics Engineers and VFX artists at film or game studios to be able to observe and experiment with the AO method within the application to assist in making an informed decision regarding their AO choice in their respective work. The application will further serve to evaluate each of the ambient occlusion methods in this report.

The C++ programming language was chosen to build the interactive Vulkan application as it offers low-level control which is necessary and required to build a high-performance Vulkan application. C++ is referred to as a compiled language, meaning the code is compiled directly into machine code that can then be executed by the CPU. The compilation to machine code provides the speed and efficiency seen in C++, especially at runtime which is otherwise not seen in many other programming languages. The low-level control given to the programmer provides the opportunity for the programmer to optimise the application. While other programming languages might be simpler to write such as Python, C# or Java, their underlying architecture utilise processes such as garbage collection which performs automatic memory management and comes at the cost of a lack of ability to optimise the application. Furthermore, because of the details mentioned above, the language is used in a wide range of game engines such as Unreal Engine, Unity Engine, CryEngine and Godot and many more (Insight, 2022). Since this research aims to produce software that is like those mentioned albeit at a smaller scale, it was clear C++ would be the best choice and therefore has been used to develop the accompanying software.

While I have some experience using C++, my knowledge of building larger applications with the language was limited. Throughout the development of my project, I built a stronger understanding of C++ and how to approach Object-Oriented Programming using this language. Each step of development lead to challenges I learnt to solve using C++ which continued to build my knowledge of the language. In addition, my knowledge of the Vulkan API was limited, only having succeeded in rendering a triangle before taking on the current complex project. The project allowed me to build a stronger understanding of the API the more I used it which was crucial for implementing vital features such as UI using ImGUI with Vulkan. The ImGUI library was not a library I had previously worked with and integrating would have been much more difficult if the foundational elements of the application utilising Vulkan did not allow me to build a strong understanding of the API. My previous experience writing shaders using GLSL was also limited. The projects development allowed me to further develop my skills in GLSL since all AO methods are implemented using GLSL. The skills developed using GLSL gave me the confidence to work towards building the Alchemy AO shader myself however, due to some issues, a similar shader was used as a reference to correct calculations. Nevertheless, the skills developed, learning from other shader implementations allowed me to apply the gained GLSL skills to almost succeed in a complex algorithm implementation.

The environment used to develop the software was Visual Studio Code (VSCode). This is because VSCode is a more accessible environment because it is a cross-platform editor. Cross-platform availability is a crucial aspect of the project as mentioned in the project's requirements (**Table 1**) to allow all users access and the ability to extent the projects abilities, thus Visual Studio Code was chosen as the programming environment for the development of the application.

1.4.1 Project requirements

To ensure the accompanying software met the aforementioned functionality and the software is meaningful to stakeholders; a list of functional and non-functional requirements for the software have been detailed prior to the development.

Requirement	Type	Category
The application must maintain an understandable code base to ensure software clarity and user extensibility.	Essential	Non-functional
The application must use the Vulkan API as the graphics API.	Essential	Functional
The application must be cross-platform.	Essential	Non-functional
User must be able to inspect geometry up close to build an understanding of the impact of the ambient occlusion technique.	Essential	Functional
Application must allow the user to use the keyboard keys to move a virtual camera around the scene.	Essential	Functional
Users should be able to adjust ambient occlusion shader input parameters in real-time using the application.	Essential	Functional

Users should be able to turn the blur filter on and off using the application.	Essential	Functional
Users should be able to lock and unlock camera movement to make UI accessibility easier.	Essential	Functional
The application must provide an easy-to-use UI	Essential	Non-Functional
The UI provided in the application must be moveable by the user to ensure the UI does not obstruct points of interest	Essential	Functional
Features provided by the UI should not cause problems for the end user	Essential	Functional
The provided UI should allow users to select different ambient occlusion methods to toggle and view in real-time.	Essential	Functional

Table 1: *Project requirements detailed before development of the accompanying software to ensure useability and meaningfulness.*

1.5 Brief History

Ambient occlusion implementations have existed over the years in one form or another; used today in both film and real-time graphics applications in offline and real-time renderers. The technique made its first appearance in the film industry, in the film Pearl Harbour in 2001 (Parker, 2014). The technique was inspired by reflection occlusion; achieved and implemented at Industrial Light & Magic (ILM) in their ray tracing renderer, RenderMan where a single ray-traced occlusion pass was generated independently of the light and a final rendering pass. The independent ambient occlusion pass was used as a pre-processing step to the final render (Seymour, 2011). It was not previously feasible to compute ambient occlusion on-the-fly in real-time rendering applications such as computer games, games that contain other effects alongside complex, dynamic environments (Kajalin, 2009). It was not until 2007 with the introduction of the so-called screen-space approach (Shanmugam and Arikan, 2007) which allowed ambient occlusion to be calculated in time-critical applications such as video games with detailed geometry, animated models, and complex physics.

While screen-space ambient occlusion approaches (SSAO) made ambient occlusion feasible in real-time graphics software, allowing ambient occlusion to be computed per frame for real-time applications, the screen-space approaches are not without their limits and drawbacks. For example, Ray traced ambient occlusion will produce images of a higher visual quality than screen-space methods as information will need to be discarded at the rasterization stage in order to make computing ambient occlusion in real-time fast and efficient thus feasible. This will inherently reduce the quality of the effect. Recent advancement in GPU hardware has made ray-tracing feasible in real-time, introducing Ray-traced ambient occlusion (RTAO) which uses hardware accelerated ray tracing to produce AO that is of a superior quality to that of screen-space approaches. However, while ray tracing maybe feasible in real-time, the computational cost for utilising RTAO remains high and use of the technique has provided little value because of its performance cost (Waldner, 2AD). Further advancement in GPU hardware could make screen-space approaches obsolete in the future.

2 Background

This section will briefly go over some key computer graphics concepts and then move onto discuss the theory behind ambient occlusion. This is to prepare the reader for the sections that follow regarding screen-space ambient occlusion.

2.1 Rendering Equation

The *rendering equation* (RE) presented in 1986 by James T. Kajiya subsumes the wide range of rendering algorithms and provides a unified context for viewing such computer graphics algorithms (Kajiya, 1986). The *rendering equation* states that the outgoing radiance L_o from one surface point $\boldsymbol{x}$ in direction $\boldsymbol{\omega}$ is the sum of the emitted radiance $\boldsymbol{L_e}$ and the reflected radiance $\boldsymbol{L_r}$ (Dutré, Kavita Bala and Philippe Bekaert, 2009).

$$L_o(x,\omega) = L_e(x,\omega) + L_r(x,\omega) \tag{1}$$

The reflectance properties of a surface affects the scattering of light. We include this in the equation to form the *rendering equation* (Kajiya, 1986).

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_\Omega f_r(x, w_i \to w_o) L_i(x, w_i)(w_i \cdot n)\, d\omega_i \tag{2}$$

Where $\boldsymbol{\Omega}$ is the upper hemisphere oriented around the surface normal $\boldsymbol{n}$ at the point $\boldsymbol{x}$. $\boldsymbol{fr(x,\omega_i \to w_o)}$ is the bi-directional reflectance distribution function (BRDF) and $(\boldsymbol{\omega_i} \cdot \boldsymbol{n})$ is a dot product between an incoming direction $\boldsymbol{\omega_i}$ and the surface normal $\boldsymbol{n}$, that is clamped to zero; a weakening factor incorporating Lambert's cosine law which states that the outward radiant intensity is dependent upon the $\boldsymbol{cos\theta_i}$ of the angle between the viewing direction and the surface normal.

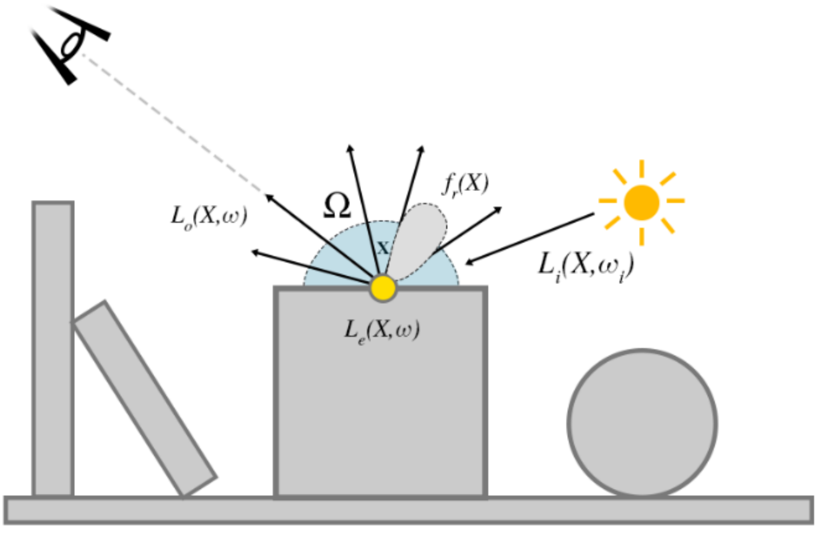


Figure 3: *Rendering Equation visualised.*

2.2 Ambient Occlusion

Ambient Occlusion (AO) is a global illumination effect that improves environment lighting by approximating indirect light-darkening corners, cracks, creases, and intersections of objects;

providing detail and visual cues about the shape of objects where the lighting has no direction; giving the scene a greater feel of depth (Möller et al., 2018). The use of ambient occlusion as a rendering pre-processing step was introduced by Landis in 2002 for film production.

Ambient light is described as light that is not directly from a specific light source but a combination of the light that has reflected on other surfaces; indirectly impacting the colour of objects in the scene (Cook and Torrance, 1981). The mathematical formulation of ambient occlusion was defined by Cook and Torrance 1982; based on the following assumptions regarding ambient light:

- Ambient lighting is uniformly incident; suggesting that the incoming radians is the same for *all* incoming directions.
- Ambient lighting is independent of the viewing direction. That is, light reflected is equal in all directions.

Based on these assumptions, (Cook and Torrance, 1981) defined ambient illumination as:

$$f = \frac{1}{\pi} \int (N \cdot L) d\omega_i \tag{3}$$

Where $\boldsymbol{f}$ is the part of the hemisphere that is not obstructed by surrounding geometry.

2.3 Real-Time Rendering

Real-time rendering is a subfield of computer graphics which looks to produce the best possible degree of photorealistic images at an acceptable rendering speed which is usually 24 frames per second (Technologies, 2021). The framerate of 24 frames is acceptable as this is the minimum number of frames per second needed to create the illusion of movement for the human eye. Real-time rendering is commonly used in video games and other interactive graphics applications. This section will unveil some rendering methods used in real-time rendering to assist in producing photorealistic images in real-time with acceptable performance.

2.3.1 Deferred Rendering

In modern real-time rendering applications, shading is the most computationally expensive stage of the image creation process. The purpose of Deferred rendering is to reduce the performance cost of this shading stage, in other words to reduce the overdraw (Thaler, 2011). Forward rendering is the default way to render images in real-time rendering. We render an object and light it relative to all the lights within the scene using the objects shader (Joey De Vries, 2020). This operation is performed individually for every object within the scene; for fragments that may or may not end up in the final image. The complexity of this approach can be written in "big O" notation as $O(Lights * Objects)$. The overdraw is inherently evident with the forward rendering approach. Deferred rendering aims to remedy the overdraw by deferring the shading stage of the geometries at the end of the pipeline. This is achieved by storing geometric information about the scene in a group of textures referred to as the Geometry buffer (G-buffer), storing information such as surface positions, normal, colour and/or specular values. The geometric information stored in the G-buffer is then later used to perform shading calculations. The key thing to note with deferred rendering is the decoupling of the geometry processing from the shading process (Thaler, 2011). This approach reduces the fragment count, performing shading calculations in screen-space, that is, only for pixels that will be visible on the screen, using the resolution of the screen instead of the total fragment count which greatly reduces the complexity of the shading process thereby boosting performance (Owens, 2013).

We will briefly look at some of the buffers that can be found in the G-buffer collection. The buffers we will discuss are commonly used buffers found in real-time graphics applications that utilize deferred rendering.

2.3.2 Depth and Positions

One of the most popular buffers found in a deferred renderer is the depth buffer in eye coordinates. This buffer contains the depth information of the scene where the colours black and white distinguish positions that are near and far from the viewer, respectively. Often the depth buffer is used in deferred rendering for more than it's intended purpose; it is also used to reconstruct geometrical information regarding the scene such as positions in eye or world coordinates which can later be used to apply shading to the scene's geometry. This will conserve memory bandwidth as we would only need to create the depth buffer once, often necessary in graphics applications and re-use it to reconstruct geometric information instead of creating multiple render targets (MRT).
The following demonstrates how position data can be reconstructed using the depth buffer.

```
float depth = texture(depthMap, uvCoords).x;

vec4 clipSpace = vec4(uvCoords * 2.0 - 1.0, depth, 1.0);

vec4 viewSpace = inverse(camera.projectionMatrix) * clipSpace;

viewSpace.xyz /= viewSpace.w;
```

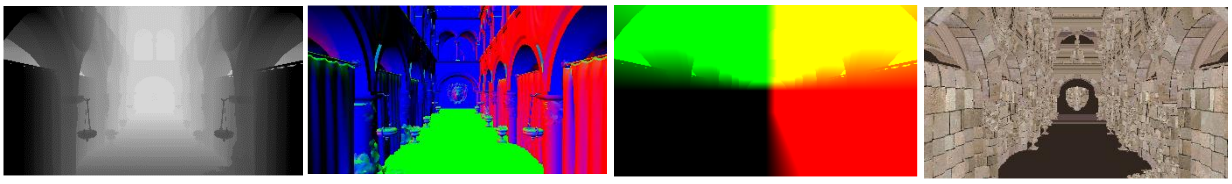
Figure 4: *Reconstruct positions to view-space using the depth buffer (ZaOniRinku, 2021).*

2.3.3 Normal

The collection of G-buffers may also contain a buffer which stores the normals of the scene's geometry. This can be stored as a texture or reconstructed from the depth buffer, like the position reconstruction mentioned previously. For some post-processing algorithms, the latter is the preferable choice as an MRT output texture often contains normals that have normal map perturbation applied, which some algorithms find difficult to work with (turanszkij, 2019). The reconstruction of the normal from the depth buffer follows a similar procedure shown in **Figure 4**.

2.3.4 Colour

The material of the scene objects, also referred to as the colour or albedo is stored as a buffer. This is because the material information is required for lighting calculations and cannot be reconstructed.



(a) Depth *(b) Normal* *(c) Positions* *(d) Albedo*

Figure 5: *Buffers commonly used for deferred rendering.*

2.3.5 Shaders

The buffers within the G-buffer (**Figure 5**) and the final shading pass are implemented using *shaders*. A shader is a smaller user-defined program that runs on the GPU at some stage of the graphics pipeline that takes in some data to process. While the graphics pipeline contains several shader stages such as the *Geometry* shader and the *Tessellation* shader we will focus and utilise primarily the essential *vertex* and *fragment* shaders. The vertex shader is invoked after the *Input assembler* stage of the graphics pipeline. A vertex shader has many responsibilities, most importantly to perform setup work for later shader stages such as computing the position for a given vertex and evaluating the vertex output data such as texture coordinates and normal data (Möller et al., 2018). For every vertex that needs to be processed, the vertex shader is executed. The fragment shader is invoked after the *rasterization* stage of the graphics pipeline. A fragment shader is responsible for processing pixels, executing on the Graphics Processing Unit (GPU). When we create a fragment shader, we create a fragment processing function which will manipulate fragment data. A fragment shader is often driven by the vertex shader as the fragment shaders input data is often the output data from the vertex shader which the fragment shader then operates on. For example, in order for the fragment shader to process and calculate per-pixel lighting, the fragment shader requires data regarding the orientation of the triangle, the orientation of the light and also the orientation of the view vector, depending on the lighting technique being implemented. It is at the fragment shader stage that we able to read buffers that have been output to, containing information processed by the vertex shader. The fragment shader will read the buffers to perform shading calculations such as lighting to produce the final output image.

Shaders implement the Single Instruction, Multiple Data (SIMD), a term originating from a class of parallel computers in Flynn's taxonomy. SIMD is a parallel processing technique that makes use of data-level parallelism by performing the same operation on multiple data simultaneously (Rakos, 2022). This allows GPUs to execute the shaders in parallel which achieves great compute speeds. The discussion regarding shaders and GPUs quickly becomes more involved. For further reference the reader is advised to consult a literature dedicated to shaders.

We have now developed some understanding of AO and build some knowledge regarding real-time rendering theory and concepts which will now enable us to explore and discuss screen-space ambient occlusion methods.

3 Analysis

This section will uncover and introduce crucial topics regarding screen-space ambient occlusion methods and then move onto discuss the previous work that has been done regarding real-time ambient occlusion, focusing particularly on the most commonly used SSAO methods which make real-time AO feasible.

3.1 Raytracing and Rasterization

3.1.1 Rasterization

Real-time graphics applications have long used a technique known as rasterization to render three-dimensional objects onto a two-dimensional screen. This is because rasterization is fast and can produce great results although the results are not as realistic compared to ray tracing (Caulfield, 2019). Rasterization is part of the graphics pipeline which determines the pixels that are covered by a primitive. The aim of this stage is to convert the set of vertices into a raster image.

Objects rendered using computer graphics with rasterization are created from meshes, built up from triangles. The triangles are converted into pixels on a two-dimensional screen and for each pixel, the rasterization algorithm checks: if the object covers the centre of the pixel and if the object is in front of another object which is determined by the depth buffer. If the centre of the pixel is covered and the object is in front of another or any object, the pixels the object covers are coloured, and those pixels will be rendered however, if the pixel centre is not covered or another object is in front, those pixels will be discarded. This process is repeated for all pixels the triangles of the mesh cover. Further processing of the pixel such as shading which effects the colour of the pixel depending on the lighting in the scene or applying textures to pixel occur in the fragment shader at the fragment stage. This method is successful in real-time graphics applications because it is fast. The rasterization algorithm has a complexity of O(n), the complexity of rasterization is proportional to the number of primitives (Abi-Chahla, 2009).

3.1.2 Raytracing

The raytracing approach to computer graphics is much different from rasterization. Raytracing is an eye-oriented process that processes the scene per pixel and not per primitive as seen in rasterization, to determine what object should be rendered at that pixel thus the complexity is correlated to the resolution of the output image. Historically, raytracing has been reserved for offline rendering as real-time raytracing was not feasible due to hardware limitations. However, recent innovation has made real-time raytracing possible in real-time. Emerging graphics applications such as video games have implemented raytracing in a reserved manor, implementing a specific or sub-set of techniques in their applications such as Ray-traced Ambient Occlusion (RTAO) seen in *Digital-Illusions Creative Entertainments* (DICE) 2021 title *Battlefield 2042*; one of few titles utilising this technique. While raytracing has become available and feasible in real-time, it remains computationally more expensive than rasterization and seemingly as a result has yet see wider adoption and implementation in graphics applications.

3.1.3 Monte Carlo Integration

The screen space ambient occlusion methods that will be discussed in this thesis are implemented using a Monte Carlo approach. The Monte Carlo method allows us to approximate by integrating over an area and sampling random points between the constraints defined by the integral. The values obtained by the random points we then add up and average to compute an estimate of the expected value. Continuously, evaluating the function at different random points and adding up the values and averaging them, will get us closer and closer to the actual result of the integral (Scratchapixel, 2016). Monte Carlo is able to evaluate functions many times and average the result, these are tasks a computer can handle very well and perform significantly faster than humans. The Monte Carlo integration technique is often used in computer graphics as it allows us to approximate certain values to produce certain effects in graphics such as ambient occlusion. The technique is widely used in computer graphics as it extends to higher dimensions well, whereas other methods that calculate integrals such as quadrature, become more and more expensive to use the higher we go in dimensions.

3.2 Screen-Space Integration

The Z-buffer also known as the depth buffer has been used beyond it's intended purpose, primarily used to resolve the visibility of objects (Möller et al., 2018). In screen-space approaches the depth buffer is used for scene approximation. Screen-space techniques are applied to a 2D quad which covers the width and height of the screen; the effect of the screen-space technique is calculated on each fragment on the 2D quad (Joey De Vries, 2020). Since the technique is calculated on the quad, we will have no geometrical information regarding the scene. To obtain the scenes geometrical information, we can either use the depth buffer to obtain the information or the geometric information per-fragment can be rendered to screen-space textures; the combination of these textures is referred to as a G-buffer which can later be sent and sampled from in other shaders to obtain the scenes geometrical information. This process is parallel to how we go about performing deferred rendering, discussed in **Section 2.3.1**. SSAO makes ambient occlusion feasible in real-time rendering because the calculations are done in screen-space; the ambient occlusion calculation is only performed for fragments that are present in view space.

3.3 Previous Work

The following section will discuss some of the ambient occlusion techniques used in real-time rendering which vary in their approach from which, other real time ambient occlusion techniques follow. Note that the techniques we will be focusing on are the so-called screen-space approaches for ambient occlusion. Other methods maybe discussed where relevant.

3.4 Crytek Screen-Space Ambient Occlusion

The Screen-space ambient occlusion technique by (Mittring, 2007) calculates the occlusion factor for every fragment. The technique works by generating samples in a sphere kernel around a point of interest, approximating the ratio between the amount of geometry and empty space around the point (Kajalin, 2009).

$$AO \approx \frac{1}{N}\sum_{n=1}^{N} \mathrm{V}'(s_n) \tag{4}$$

In order to check the space around a point $\boldsymbol{P}$ to determine if the sample point is outside or inside the geometry, we can compare the 3D depth value $\boldsymbol{s_z}$ of the sample to its surface depth value $\boldsymbol{s_d}$ in the depth buffer. The point of interest $\boldsymbol{P}$ is in screen coordinates just like the sample points $\boldsymbol{S}$ and so they have an associated $\boldsymbol{s_z}$ coordinate which can be used to obtain the 3D depth value. All that remains is to determine the visibly $\boldsymbol{V}'$ by comparing the two values. If the depth value stored in the depth buffer of the sample $\boldsymbol{S}$ is less than the 3D depth value of the sample point $\boldsymbol{S}$, then the sample point lies within geometry.

$$V'(s) = \begin{cases} 1 & s_d > s_z \\ 0 & Otherwise \end{cases} \tag{5}$$

Figure 6 displays this computation; it is intuitive to notice that samples that lie below the surface are occluded.

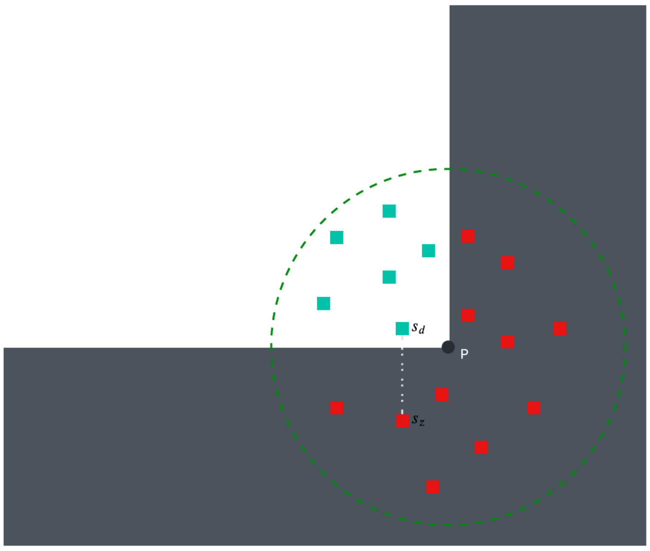


Figure 6: *Approximating Ambient Occlusion using offset points distributed around the surrounding sphere (Mittring, 2007). The green circles are the visible samples; the red are the occluded samples.*

3.4.1 StarCraft II (Filion and McNaughton, 2008)

There exists a variation of the (Mittring, 2007) technique mentioned above for ambient occlusion. In 2008 Filion and McNaughton while working at Blizzard for Starcraft II proposed a technique of their own, albeit like the ambient occlusion method used at Crytek (Mittring, 2007), the authors mention having arrived at this solution independently. (Filion and McNaughton, 2008) AO method mainly addressed the self-occlusion problem (Mittring, 2007) suffered from where a set of samples would be occluding by default (see **Figure 6**). The technique proposed differs by introducing an attenuation function and offsetting the sample points in world coordinates and then later projecting those samples back to screen space which has some implications in relation to self-occlusion which we will return to later. The attenuation function replaces the simple Boolean comparison performed by the ***V'*** function; the author explains that a surface that is close to a pixel that is being occluded, would have a greater impact on the occlusion of that pixel than if the surface was further away. The attenuation function simulates this drop off, determining the amount of occlusion that occurs; where the amount of occlusion is weakened past a certain threshold (Filion and McNaughton, 2008). It was previously mentioned that the proposed technique (Filion and McNaughton, 2008) addressed the self-occlusion issues from (Mittring, 2007) SSAO implementation. In **Figure 6** we see that samples are distributed inside the surrounding sphere, around the point ***P*** producing samples which are occluding by default. The alternative approach (Filon and McNaughton, 2008) proposed a possible solution to handle self-occlusion. Their approach was to perform a dot product between the surface normal ***N*** and the offset vector which penetrates the surface, negating the result to flip the vector. See **Figure 7** for differences. This technique produced a hemisphere oriented along the surface's normal vector which removed the default occluding samples found in (Mittring, 2007).

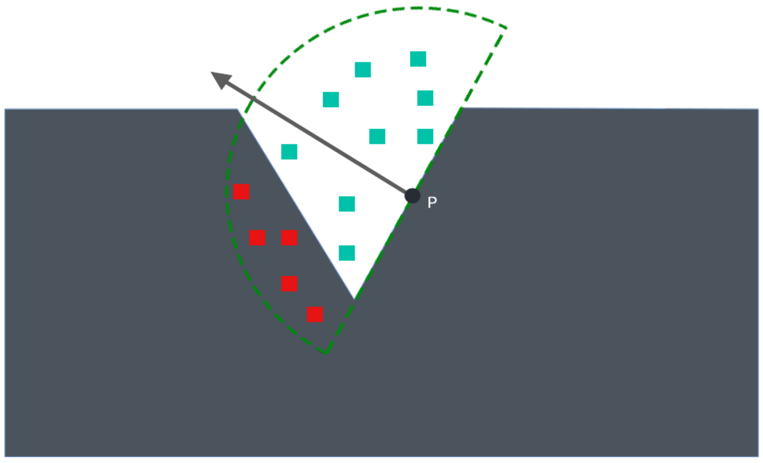


Figure 7: *Ambient occlusion approximation using hemisphere to handle self-occlusion (Filion and McNaughton, 2008).*

3.5 Horizon-Based Ambient Occlusion

In 2008 another technique to approximate AO was introduced as Image-Space Horizon-Based Ambient Occlusion, commonly known as Horizon-based ambient occlusion (HBAO) (Bavoil, Sainz and Dimitrov, 2008). (Bavoil, Sainz and Dimitrov, 2008) starts with the following formulation of ambient occlusion.

$$A = 1 - \frac{1}{2\pi}\int_{\Omega} V(\vec{\omega})W(\vec{\omega})d\omega \tag{6}$$

Where V is the visibility function over the normal oriented unit hemisphere Ω returning values between 0 and 1 representing the occlusion of a ray that begins at point P in direction $\vec{\omega}$ and W is a linear attenuation function, simulating 'falloff'.

HBAO works by ray marching the heightfield, obtained from the depth buffer in screen-space to compute the visible horizon angle $\boldsymbol{h(\theta)}$ for all directions that are perpendicular to the surface normal $\boldsymbol{n}$; in the hemisphere Ω with a radius of $\boldsymbol{R}$, surrounding the point $\boldsymbol{P}$ where $\boldsymbol{P}$ is a position in eye-space (Bavoil, Sainz and Dimitrov, 2008). See **Figure 8** for reference. The authors present the following equation to calculate the horizon angle.

$$A = 1 - \frac{1}{2\pi}\int_{\theta=-\pi}^{\pi} AO_{horizon}(\theta)\, d\theta \tag{7}$$

The presented integral computes the result for a small set of $\boldsymbol{N_d}$ uniform directions, taking $\boldsymbol{N_s}$ samples in each direction from the depth buffer to compute the maximum horizon angle (**Figure 8**) (Vermeer, Scandolo and Eisemann, 2021). The horizon angle is then used to evaluate the $AO_{horizon}(\theta)$. We can trace rays in all directions from the tangent to the surface normal to compute the occlusion factor based on the surrounding geometry however, the method assumes that the heightfield surrounding the point $\boldsymbol{P}$ is continuous; this is mostly true when considering points only within the radius of influence $\boldsymbol{R}$, removing the need to test rays below the horizon, as all rays below the $\boldsymbol{h(\theta)}$ will intersect and therefore will be occluded; symmetrically rays above the $\boldsymbol{h(\theta)}$ will not intersect the heightfield and therefore will be visible. Under the premise of a continuous heightfield the visibility function from **Eq.6** can be evaluated as $V(w) = 1$ for all angles between $\boldsymbol{t(\theta)}$ and $\boldsymbol{h(\theta)}$, that is, the ambient occlusion contribution can be found as the integral between the tangent angle $\boldsymbol{t(\theta)}$ and horizon angle $\boldsymbol{h(\theta)}$**.** The authors present the Equation as:

$$AO_{horizon}(\theta) = \int_{\alpha=t(\theta)}^{h(\theta)} W(\vec{\omega}) \cos(\alpha) d\alpha \tag{8}$$

Where α is the elevation angle defined by the tangent angle which is calculated by creating a view ray that is perpendicular to the view direction which intersects the tangent plane defined by the surface normal $\boldsymbol{n}$ and the point $\boldsymbol{P}$ (See **Figure 8**). The $\boldsymbol{W}$ is a linear attenuation function which the authors define as a function of the distance to the horizon point, evaluated for every sample. The purpose of the attenuation function is to soften sharper occlusions from samples lying closer to the $\boldsymbol{R}$ (Bavoil and Sainz, 2009). We are now able to insert **Eq.8** into **Eq.7** (Bavoil and Sainz, 2008).

$$A = 1 - \frac{1}{2\pi} \int_{\theta=-\pi}^{\pi} \int_{\alpha=t(\theta)}^{h(\theta)} W(\vec{\omega}) \cos(\alpha) d\alpha \, d\theta \tag{9}$$

The authors further evaluate **Eq.9** to the following (Bavoil and Sainz, 2008):

$$A = 1 - \frac{1}{2\pi} \int_{\theta=-\pi}^{\pi} (sin\,(h(\theta)) - sin\,(t(\theta)))W(\theta)\, d\theta \tag{10}$$

3.5.1 Screen-space Integration

The **Eq.10** can be computed via a Monte Carlo approximation, computing the result for a small set of $\boldsymbol{N_d}$ uniform directions. We ray march the depth buffer in screen space to find the $\boldsymbol{h(\theta)}$. For each point $\boldsymbol{P}$, a ray is created in direction $\boldsymbol{\theta}$ in image space to sample from the heightfield, stepping in the direction given by the $\boldsymbol{\theta}$. In each direction $\boldsymbol{N_s}$ samples are taken, producing a sample $\boldsymbol{S_i}$. The samples position is reconstructed into eye-space positions $\boldsymbol{S_i}$ and is used to calculate the horizon vector **H** as $\boldsymbol{H} = \boldsymbol{S_i} - \boldsymbol{P}$. The elevation angle of sample $\boldsymbol{S_i}$ can be computed as: (Bavoil and Sainz, 2009):

$$tan\phi(S_i) = \frac{(P - S_i).z}{||(P - S_i).xy||} \tag{11}$$

In the process of sampling, samples that are outside the radius of influence $\boldsymbol{R}$ are ignored, that is, $||\boldsymbol{S_i} - \boldsymbol{P}|| > \boldsymbol{R}$. We keep track of the elevation angles obtained from the samples $\boldsymbol{S_i}$ to determine the maximum elevation angle. If the elevation angle of the current sample $\boldsymbol{S_i}$ is less than $\boldsymbol{S_{i-1}}$ then we ignore the current sample $\boldsymbol{S_i}$ by not updating the elevation angle and proceed to the next sample.

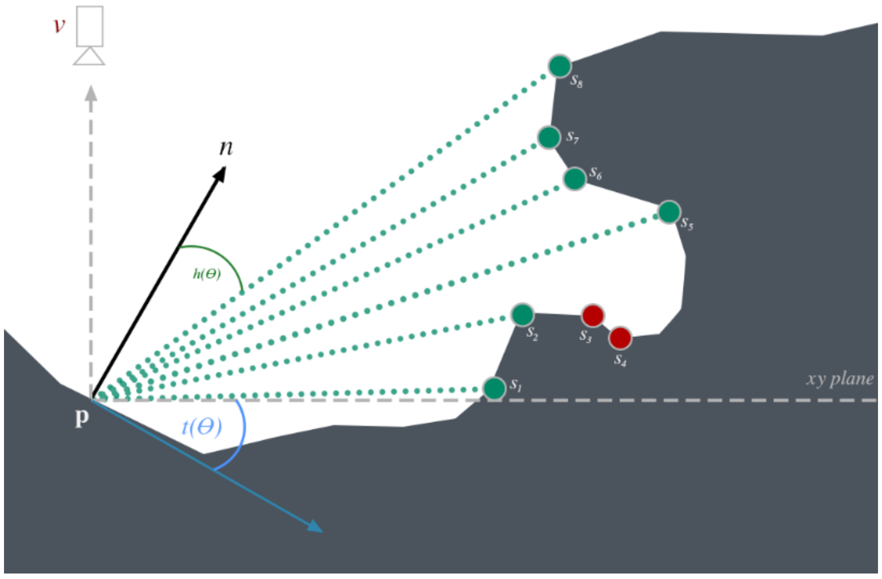


Figure 8: *Visual overview of HBAO showcasing the required components and workings of the HBAO approach to approximate AO.*

The authors recommend jittering the step size per pixel and to randomly rotate the $\boldsymbol{N_d}$ uniform directions per pixel in order to avoid banding. The introduction of the randomisation through the recommended jittering of the step size, exchanges banding for noise which the authors counter with the use of a cross bilateral blur filter. Once the maximum horizon angle is calculated, the inner integral seen in **Eq.9** can be computed as $\sin(h_\theta)$ and we can calculate the average AO contribution by computing $\sin(h_\theta) - \sin(t_\theta)$ as seen in **Eq.10**.

3.6 Alchemy Ambient Occlusion

Alchemy ambient occlusion is another ambient occlusion approximation algorithm, developed at Vicarious Visions and introduced in 2011 (McGuire et al., 2011b). Alchemy AO is part of the SSAO family of algorithms, differing with the realisation that a falloff function, chosen for aesthetics can cancel terms in a visibility integral to promote efficient operations. The Alchemy AO algorithm favours artistic freedom and performance goals over physical accuracy subsequently, addressing some of the drawbacks of screen-space AO methods that have come before. The authors elaborate on these drawbacks by listing the shortcomings of other AO algorithms, namely: Robustness, Multiscale, Artist-control and Scalable (McGuire et al., 2011b). Robustness refers mostly to the quality of the ambient occlusion output; Multiscale is defined as capturing the ambient occlusion at multiple scales, that is, producing acceptable ambient occlusion for objects within the environment with varying scales such as producing 'shadowed deep pits, corner darkening, contact shadows and wrinkles' (McGuire et al., 2011b). Artist-control is defined as providing the tools via parameters for artists to be able to control the ambient occlusion with predictably quality; four appearance parameters are provided: world-space radius and bias, and aesthetic intensity and contract to adjust the appearance of the AO. Scalable refers to the compute time taken to produce Alchemy ambient occlusion, the algorithm must compute in 3-5 ms, for both console and PC with varied quality. Alchemy AO's efficient estimator for obscurance utilises a falloff function, which reduces the influence of the occlusion with distance, creating a smooth transition between near and far-field illumination. We start with the following equation formulated by (McGuire et al., 2011a).

$$A = 1 - \int_{\Gamma} g(t(C, \hat{\omega}))\, \hat{\omega} \cdot \hat{n}\, d\hat{\omega} \tag{12}$$

The $\mathbf{\Gamma}$ is the set of all near field occluders, that is, $\mathbf{\Gamma} \subseteq \mathbf{\Omega}$ a subset of the hemisphere for which we integrate the falloff function $\boldsymbol{g(t) = u \cdot t \cdot \mathbf{max}(u, t)^{-2}}$ where $\boldsymbol{t}$ is the distance from $\boldsymbol{C}$ to a point encountered by a ray in direction $\hat{\boldsymbol{\omega}} \in \mathbf{\Gamma}$ intersects the environment. The authors mention the resemblance of the falloff function to Filion and McNaughton (2008) shifted hyperbola which was artistically desirable in *StarCraft II*. The authors substitute the function and present the following equation (McGuire et al., 2011a, p.27):

$$A = 1 - \int_{\Gamma} \frac{ut(C, \hat{\omega})}{\max\left(\mathrm{u}, \mathrm{t}(\mathrm{C}, \hat{\omega})\right)^2} \hat{\omega} \cdot \hat{n}\, d\hat{\omega} \tag{13}$$

The authors suggest simplifying the equation by defining the vector $\vec{v}(\hat{\omega}) = \hat{\omega} t(C, \hat{\omega})$ from C to the occluder (McGuire et al., 2011a, p.27).

$$A = 1 - u \int_{\Gamma} \frac{\vec{v}(\hat{\omega}) \cdot \hat{n}}{\max\,(u^2, \vec{v}(\hat{\omega}) \cdot \vec{v}(\hat{\omega}))}\, d\hat{\omega} \tag{14}$$

All that remains to do is to use *Monte Carlo integration* to numerically estimate the integral, computing the result by uniformly at random sampling a set $\hat{\boldsymbol{\omega}}_{\boldsymbol{i}}$ of $\boldsymbol{S}$ directions as described by the authors. The authors describe the computation along with some further simplifications to the equation and derives the following algebraic equation (McGuire et al., 2011a, p.27):

$$A \approx \max\left(0, 1 - \frac{2\sigma}{s} \cdot \sum_{i=1}^{s} \frac{\max\,(0, \vec{v_i} \cdot \hat{n} + zc\beta}{\hat{v_i} \cdot \hat{v_i} + \epsilon}\right)^k \tag{15}$$

Where $\boldsymbol{s}$ is the number of samples; a higher value reduces sample variance, the lower the value the faster the performance. The $\boldsymbol{\sigma}$ represents the strength multiplier, the higher the value the darker the shadows; $\boldsymbol{\beta}$ is the shadow bias, increasing $\boldsymbol{\beta}$ will reduce self-shadowing however, a higher value can cause light leaks into corners. The $\boldsymbol{\epsilon}$ simply prevents a divide-by-zero, a value small enough must be chosen in order to prevent light leaks in corners. The $\boldsymbol{k}$ is the contrast multiplier, a higher result will produce a sharper obscurance transition.

The algorithm is evaluated in screen space using the depth buffer or a positions buffer to obtain geometric information of the scene. The authors uniformly choose samples $\boldsymbol{S_n}$ in a disk with a radius of $\boldsymbol{R}$ around the point $\boldsymbol{P}$ in screen coordinates, as done in (Loos and Sloan, 2010). The samples camera-space position is recovered using the depth buffer or positions buffer constructing the sample $\boldsymbol{S_i}$ vector $\boldsymbol{S_i} = (\boldsymbol{S_x}, \boldsymbol{S_y}, \boldsymbol{S_z}(\boldsymbol{D'}))$ where $\boldsymbol{D'}$ is the depth value obtained from the depth buffer. We then calculate $\boldsymbol{v_i}$ as $\boldsymbol{v_i} = \boldsymbol{S_i} - \boldsymbol{P}$. The algorithm utilises $\boldsymbol{v_i} \cdot \hat{\boldsymbol{n}}$ to determine how in front of the surface that $\boldsymbol{P}$ lies on, the sample point $\boldsymbol{S_i}$ is. See **Figure 9** for reference. Once the previously mentioned are determined, we follow the calculations shown in **Eq.15** to compute the AO value for the sample.

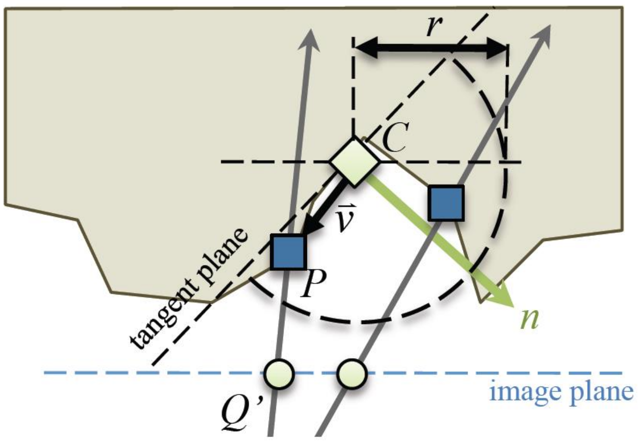


Figure 9: *(McGuire et al., 2011b, p.27) Geometric construction to demonstrate the approximation of AO using Alchemy AO.*

3.7 Overview

The previous section has discussed in detail a set of different approaches to ambient occlusion in real-time rendering. The methods that have been discussed have been displayed in **Table 2** to provide an overview of the method and their appearance in professional graphics applications.

Author	Method	Requirements	Use
(Mittring, 2007)	*CryENGINE 2 SSAO*	*Depth buffer*	*Crysis (2007)* *CryEngine 2*
(Filion and McNaughton, 2008)	*StarCraft II SSAO*	*Depth buffer* *Normal buffer*	*Starcraft II (2010)*
(Bavoil, Sainz and (Dimitrov, 2008)	*HBAO*	*Depth buffer* *Normal buffer*	*Battlefield 2042 (2021)* *Battlefield 3 (2011)* *Frostbite*
(McGuire et al., 2011b)	*Alchemy AO*	*Depth buffer* *Normal buffer*	*Alchemy Engine*

Table 2: *AO methods with their required input for calculation and some of their appearances in real-time graphics applications.*

4 Implementation

This section will cover how each of the ambient occlusion methods are implemented as shaders that can be used in real-time graphics application. The blur shader used in conjunction with the AO methods will also be explained.

4.1 Outline

The SSAO shaders are implemented in an application written in C++ which utilizes the Vulkan API. The discussed SSAO methods are implemented as shaders using *GLSL* shading language and compiled into Standard Portable Intermediate Representation (SPIR-V) format to use with the Vulkan application.

4.2 Crytek CryEngine 2 SSAO

We will begin by discussing the implementation of the ambient occlusion technique proposed by the Crytek team in 2007. The method will be implemented using (Joey De Vries, 2020) approach. However, we will demonstrate the method using the depth buffer to reconstruct view space normal and positions as mentioned in **Section 2.3.2** and not G-buffer textures, but the SSAO can also be implemented using the G-buffers.

4.2.1 Retrieving data

We will reconstruct position data and normal data from the depth buffer to demonstrate some good practices that can be used when implementing SSAO. Utilising the depth buffer instead of storing a position and normal texture will save memory. However, it is up to the reader to decide which method they would prefer to use going forward.

4.2.2 Samples and Random vectors

The samples we will use to surround our point of interest to calculate the occlusion factor are generated on the CPU using C++. This is due to the SPMD principle that shaders follow which makes random number generation difficult. The samples are generated uniformly in a sphere kernel as suggested by the authors. The generation of the random samples is shown below.

```
std::uniform_real_distribution<float> randomFloats(0.0, 1.0);
std::default_random_engine generator;
std::vector<glm::vec4> ssaoKernel;

for (unsigned int i = 0; i < 64; ++i) {

       glm::vec4 sample(
              randomFloats(generator) * 2.0 - 1.0,
              randomFloats(generator) * 2.0 - 1.0,
              randomFloats(generator) * 2.0 - 1.0, 0.0);

       sample = glm::normalize(sample);
       sample *= randomFloats(generator);

       float scale = float(i) / 64.0;
       scale = lerp(0.1f, 1.0f, scale * scale);
       sample *= scale;
       ssaoKernel.push_back(sample);
}
```

Figure 10: *Generating random samples to use to determine the occlusion factor around a fragment (de Vries, 2020).*

This will generate 64 samples, which vary the x, y and z values between -1.0 and 1.0 in order to produce a sphere kernel. The samples are distributed around the origin in the sphere kernel. Note the scaling and the use of the lerp function. The author suggests having a larger density of samples near the fragment origin. This can be achieved using the *lerp* interpolation function to distribute the samples closer to the fragment origin (Joey De Vries, 2020). The result of this places greater weight on geometry near the fragment than geometry that is further away. This eliminates the need for a distance attenuation function (Kajalin, 2009).

Similarly, the generation of random vectors is also done on the CPU. Randomness in the form of a noise texture is introduced to randomly rotate the sampling sphere around the z axis for every pixel. This is done to reduce "banding" which would occur if the same pattern of sampling were to be applied. Random vectors are generated as two-dimensional vectors ranging between -1 to 1 for the

x and y axis and the z axis remains 0 as this is the axis of rotation. The random vectors will be tiled over the screen using a small *4x4* texture in order to introduce randomness. The use of random vectors will also allow us to use a smaller number of samples to obtain an acceptable result which otherwise would require a larger number of samples, heavily impacting the performance. Randomness will allow us to better sample the surroundings of the fragment with a smaller set of samples to determine the occlusion factor and produce a realistic result.

```
std::vector<glm::vec4> ssaoNoise;

for (unsigned int i = 0; i < 16; i++) {

        glm::vec4 noise(
        randomFloats(generator) * 2.0 - 1.0,
        randomFloats(generator) * 2.0 - 1.0,
        0.0, 1.0);
        ssaoNoise.push_back(noise);
}
```

Figure 11: *Generating random vectors to reduce the number of samples required (de Vries, 2020).*

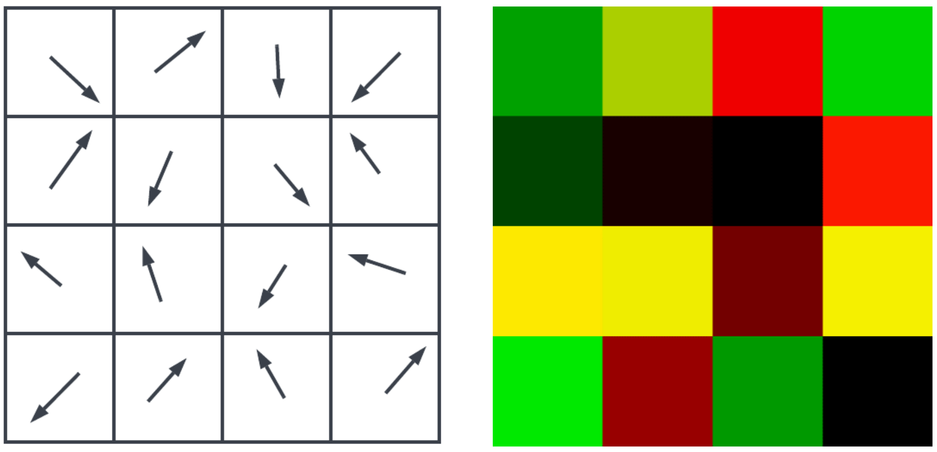


Figure 12: *4 x 4 texture containing random rotation vectors to rotate and randomise sphere sample kernel (left). Noise texture containing random vectors in GPU (right).*

4.2.3 Determine Occlusion factor

The occlusion factor is calculated for each fragment within a for loop

```
for (int i = 0; i < samples; i++)
{
        // calculate occlusion factor for current fragment
}
```

Where the *kernelSize* is the number of samples generated. The author suggests reflecting the sample positions on a plane defined by the per-fragment random directions in order to reduce the number of samples.

```
const vec2 noiseScale = vec2(Width/4.0, Height/4.0);
float occlusion = 0.0;
//(Joey De Vries, 2020) Create a TBN: Convert the sample from tangent to view-space
vec3 tangent = normalize(randomVec - normal.xyz * dot(randomVec, normal.xyz));
vec3 bitangent = cross(normal.xyz, tangent);
mat3 TBN = mat3(tangent, bitangent, normal.xyz);

vec3 plane = texture(texNoise,  uvCoords) * noiseScale).xyz - vec3(1.0);
for(int i = 0; i < kernelSize; i++) {
        vec3 samplePos = reflect(samples[i].xyz, plane);
        samplePos = TBN * samplePos;
}
```

We iterate over the kernel samples, obtaining a sample and use a tangent, bitangent and normal matrix (TBN) to convert to sample to view-space and utilize the *reflect* function to reflect the sample. Where the *reflect* function is provided as part of the GLSL specification. The TBN matrix will allow us to rotate the sphere in such a way where the normal will be aligned with the normal of the point that we are sampling around. The TBN matrix is calculated as follows:

$$\vec{T} = \vec{R} - \vec{N} \times (\vec{R} \cdot \vec{N})$$
$$\vec{B} = \vec{N} \times \vec{T}$$
$$TBN = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

Where N is a normal in view-space and $\vec{R}$ is the random rotation vector from the noise texture.

Note the *noiseScale* variable. This is used to correctly tile the noise over the screen; dividing the screens dimensions by the size of the noise texture will provide a scaling value which we will scale *uvCoords* by to correctly tile the noise texture. Next, we add this sample to the current fragment position and introduce a sample radius of 0.5 which we multiply by, defining the sampling radius for the SSAO.

```
float radius = 0.5;
samplePos = viewSpacePositions + samplePos * radius;
```

In order to obtain the position and the depth value of the sample correctly, that is, as if the sample was being rendered directly to the screen, we need to transform the sample to screen-space. This is first achieved by transforming the sample to clip-space via the projection matrix. Then we perform a perspective divide in order to transform from clip-space to normalized device coordinates (NDC).

```
vec4 offset = vec4(sample, 1.0);

offset = camera.proj * offset; //transform sample to clip space

offset.xyz /= offset.w; // perspective divide
```

We transform the sample from NDC to texture coordinate in the range [0.0, 1.0] and use this to sample the position G-buffer to obtain the depth value of the sample.

```
offset.xyz = offset.xyz * 0.5 + 0.5; // transform to range 0-1

float sampleDepth = texture(gPosition, offset.xy).z;
```

All that remains is to determine the visibility of the sample. Before doing this the author suggests performing a range check, as in some scenarios the method can produce false occlusion from nearby objects to far objects (See **Figure 13**).

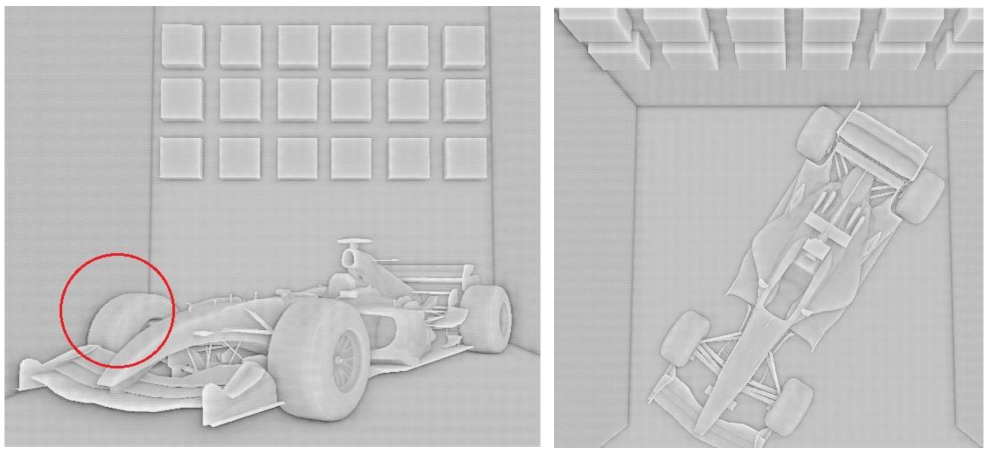


Figure 13: *SSAO false occlusion without range check [No Blur].*

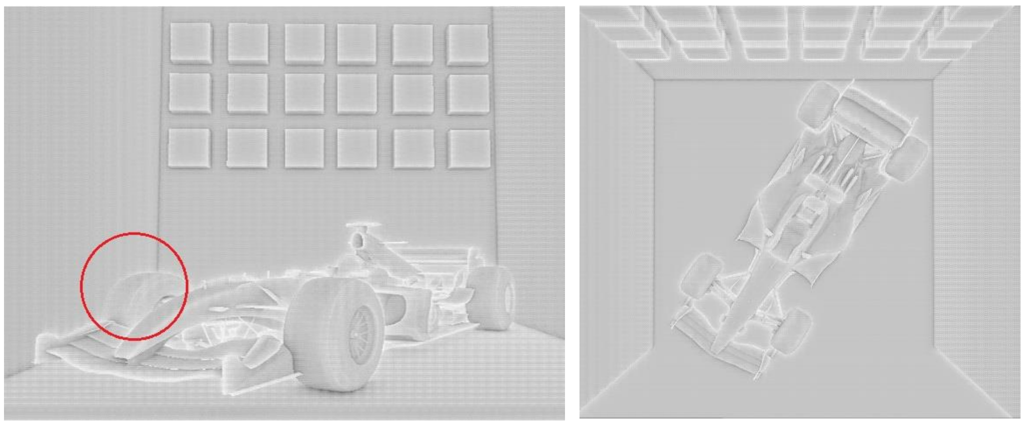


Figure 14: *SSAO with range check [No Blur].*

The range check aims to remedy this issue by zeroing contributions that are outside the sampling radius (**Figure 14**).

```
float rangeCheck = (viewSpacePositions.z - sampleDepth) < radius ? 1.0 : 0.0;
```

We can now check if the sample is visible or within geometry, we perform a check to see if the samples depth value is greater than the samples z value. If the depth value is greater, the sample is visible and if not, the sample lies within geometry; the result is then added to the contribution factor.

```
occlusion += (sampleDepth >= sample.z ? 1.0 : 0.0) * rangeCheck;
```

Once all the samples have been gathered, and the contribution is determined, the final occlusion contribution is then normalized by dividing it by the size of the kernel and 1.0 is subtracted to allow us to scale the ambient lighting component.

4.3 Horizon-Based Ambient Occlusion

HBAO requires sampling steps and sampling directions to be defined, where the sampling steps is the number of steps taken along the ray when raymarching, occurring across multiple directions defined by the number of sampling directions which defines the number of samples. We will begin by defining the relevant parameters. The number of samples and sampling steps has been defined as 6 as the authors found this to be good values to produce acceptable results (Bavoil, Sainz and Dimitrov, 2008). In addition, HBAO requires normal in view-space. This can be reconstructed from the depth buffer or rendered to a texture as previously mentioned. We will use the depth buffer to reconstruct normal. The HBAO implementation is based on (jhk2, 2019) implementation with improvements including sample jittering and rotating directions by a random angle to produce the complete AO effect.

```
/* Based on jhk2 HBAO implementation (jhk2, 2019) - Changes needed to be made to complete the
algorithm such as rotating the direction angle and jittering samples the shader was re-worked to
become compatible with Vulkan. Parameter to control darkness was also introduced
https://github.com/jhk2/glsandbox/blob/master/kgl/samples/ao/hbao.glsl */

layout(binding = 1) uniform sampler2D texNoise;

layout(binding = 2) uniform sampler2D gNormal;

float RADIUS = 0.5;

float NUMBER_OF_SAMPLING_DIRECTIONS = 6;

float STEP = 0.004;

float NUMBER_OF_STEPS = 6;

float TANGENT_BIAS  = 0.3;
```

As previously mentioned, HBAO requires view-space normal. We will handle this next by reconstructing the required normal from the depth buffer.

```
vec4 depthToNormal(vec2 tc)
{
   float depth = texture(depthMap, tc).x;
   vec4 clipSpace = vec4(tc * 2.0 - 1.0, depth, 1.0);
   vec4 viewSpace = inverse(camera.proj) * clipSpace;

   viewSpace.xyz /= viewSpace.w;
   vec3 pos = viewSpace.xyz;
   vec3 n = normalize(cross(dFdx(pos), dFdy(pos)));
   n *= - 1;
   return vec4(n, 1.0);
}
vec4 normal = depthToNormal(uvCoords);
```

Next, the point of interest $\boldsymbol{P}$ needs to be obtained and reconstructed in eye-space. This can be achieved by sampling the depth buffer to obtain the depth value which can then be used with the texture coordinates to define the position of the point. We will then convert the point normalised device coordinates (NDC).

```
vec3 pos = vec3(uvCoords, texture(depthMap, uvCoords).r;
vec3 NDC_POS = (2.0 * pos) - 1.0; // normalized device coordinates
```

Once we have the point $\boldsymbol{P}$ in normalized device coordinates we can utilize the inverse of the projection matrix and perform a perspective divide in order to convert the point from NDC to a view-space position.

```
vec4 unprojectPosition = inverse(camera.proj) * vec4(NDC_POS, 1.0);
vec3 viewPosition = unprojectPosition.xyz / unprojectPosition.w;
```

Once we have the point in view-space, we can now begin to prepare for the raymarching. First, we must uniformly distribute the directions per pixel as suggested by the authors. We set up the noise texture which we will use in order to rotate directions and jitter samples by a random offset as suggested by the authors. This is so that we can trade banding for noise which we can later remedy with a blur filter for a better result. After rotating the directions, we can calculate the sample direction and obtain the tangent and horizon angle.

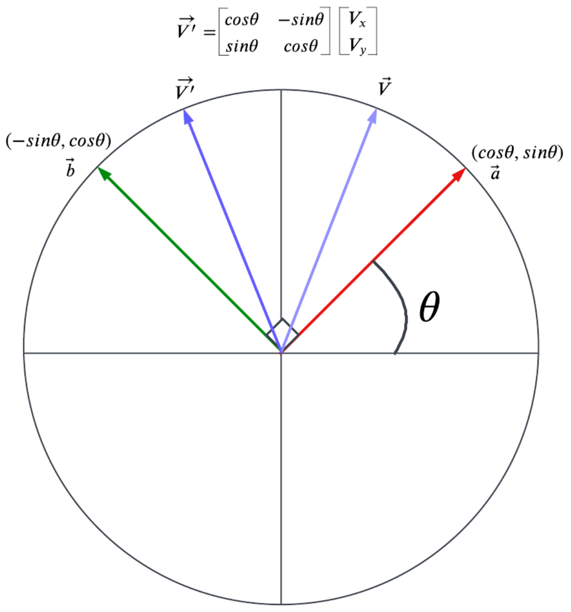
```
float samplingDiskDirection = 2 * PI / NUMBER_OF_SAMPLING_DIRECTIONS;

vec3 sampleNoise = texture(texNoise, uvCoords) * noiseScale).xyz;

sampleNoise.xy = sampleNoise.xy * 2.0 - vec2(1.0);

vec4 Rand = GetJitter();
```

We can obtain a new direction and rotate the newly obtained direction by using the noise we set up earlier. **Figure 15** demonstrates this process.



$\vec{V'} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} V_x \\ V_y \end{bmatrix}$
$\vec{V'}$
$\vec{V}$
$(-sin\theta, cos\theta)$
$\vec{b}$
$(cos\theta, sin\theta)$
$\vec{a}$
θ

Figure 15: *Rotating direction by angle where $\vec{V}$ is the vector of the direction angle and $\vec{A}$ is offset vector with a given angle which will be used to rotate the direction vector to create $\vec{V'}$and $\vec{B}$ is the vector $\vec{A}$ rotated $\frac{\pi}{2}$ to construct a coordinate system.*

```
for(int i = 0; i < NUMBER_OF_SAMPLING_DIRECTIONS; i++) {

        // use i to get a new direction

        float samplingDirectionAngle = i * samplingDiskDirection;

        //Rotate direction by random angle

        vec2 samplingDirection = RotateDirectionAngle(vec2(cos(samplingDirectionAngle),
        sin(samplingDirectionAngle)), Rand.xy);

}
```

Once the sampling direction has been obtained, we can define our tangent θ and our horizon θ. The horizon θ is set to the tangent angle to begin with. The tangent angle can be obtained from the view-space normal by performing $(D \cdot N)$ and calculating the inverse cosine by utilising the $\boldsymbol{acos}$ function.

```
//tangent angle : inverse cosine from dot product of direction and normal

float tangentAngle = acos(dot(vec3(samplingDirection, 0.0), normal.xyz)) - (0.5 * PI) +
TANGENT_BIAS;

//set the horizon angle to the tangent angle to begin with

float horizonAngle = tangentAngle;

vec3 LastDifference = vec3(0);
```

Note the use of the *TANGENT_BIAS;* the authors suggest adding a tangent bias in order to prevent false occlusion. The tangent bias ignores occlusion near the tangent plane which produces the false occlusion (Bavoil and Sainz, 2008). The differences can be seen below.

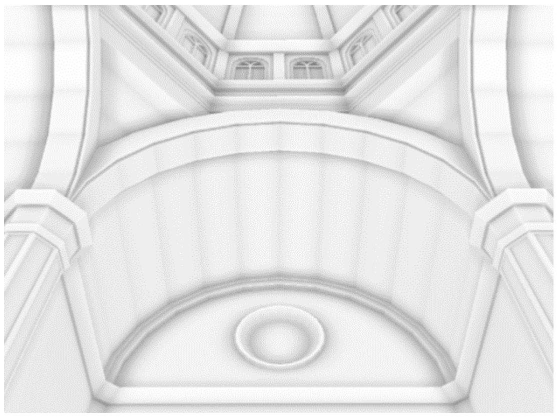


Figure 17: *No tangent bias.*

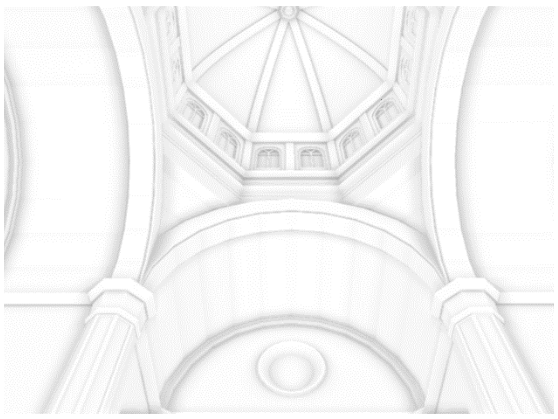


Figure 16: *Tangent bias 0.3.*

We are now ready to begin ray-marching along the sample directions in order to determine the horizon angle. The ray-marching beings in the nested for loop. Once we have found the horizon angle for a given sample direction, we will convert the sample point to view-space in order to compare the distance between the sample point and the position ***P*** to ensure it is with the radius of influence. The horizon angle for a sample direction angle can be found and converted to view-space as follows:

```
// for each direction we step in the direction of that sampling direction to sample

for(int j = 0; j < NUMBER_OF_STEPS; j++){

        vec2 stepForward = (Rand.z + float(j+1)) * STEP * samplingDirection;

        order to move to that location

        vec2 stepPosition = vec2(1.0 - uvCoords.x, uvCoords.y) + stepForward;

        float steppedLocationZ = texture(depthMap, stepPosition.st).x;

        vec3 steppedLocationPosition = vec3(stepPosition, steppedLocationZ);

        vec3 steppedPositionNDC = (2.0 * steppedLocationPosition) - 1.0;

        vec4 SteppedPositionUnProj = inverse(camera.proj) * vec4(steppedPositionNDC,
        1.0);

        vec3 viewSpaceSteppedPosition = SteppedPositionUnProj.xyz /
        SteppedPositionUnProj.w;

}....
```

Since we now have the view space position of the offset point, we need to check to see if the point is within the radius of influence in order to be considered. We can ensure the point is within the radius of influence by calculating the difference between the offset point and the position $\boldsymbol{P}$. We can then use this value to perform a comparison with the radius value and compare the distance to the sample radius to ensure the offset point lies within the radius of influence.

```
vec3 diff = viewSpaceSteppedPosition.xyz - viewPosition;
if (length(diff) < RADIUS) {
        // ….
 }
```

If the sample is within the radius of influence, its horizon angle is considered as a candidate for the maximum horizon angle and the horizon angle variable is updated to account for the newly found horizon angle.

```
if (length(diff) < RADIUS) {
  LastDifference = diff;
  float FoundElevationAngle = atan(diff.z / length(diff.xy));
  // update horizon angle if new found elevation angle is larger
  horizonAngle = max(horizonAngle, FoundElevationAngle);
 }
```

Once the horizon angle has been found, we can begin to implement and apply the attenuation function. In order to implement the attenuation function, we first need to obtain the normalized distance which is the distance from the sample to the point of interest position $\boldsymbol{P}$ divided by the sampling radius.

```
vec3 diff = viewSpaceSteppedPosition.xyz - viewPosition;
float norm = length(LastDifference) / RADIUS;
```

(Bavoil, Sainz and Dimitrov, 2008) use a quadratic attenuation instead of a linear attenuation function. A quadratic attenuation function is used in order to attenuate the samples near the middle of the radius less (Bavoil and Sainz, 2009). The following quadratic attenuation function is used; $\boldsymbol{W(r) = 1 - r^2}$ and so the attenuation can be calculated as:

```
float attenuation = 1 – norm *norm;
```

Finally, we can compare the horizon angle to the tangent angle in order to obtain the ambient occlusion and add the occlusion contribution the total. Once we have added the occlusion contribution, all that remains to do is to normalize the occlusion contribution by the number of sampling directions in order to obtain an approximation of the ambient occlusion. We subtract 1.0

from the occlusion result in order to scale the ambient lighting component using the occlusion factor.

```
float occlusion = clamp(attenuation * (sin(horizonAngle) - sin(tangentAngle)),
0.0, 1.0);

sum += 1.0 - occlusion;

sum /= NUMBER_OF_SAMPLING_DIRECTIONS;
```

4.4 Alchemy Ambient Occlusion

Alchemy Ambient occlusion is a mixture of previous ambient occlusion methods, utilizing the techniques of other AO algorithms such as the projection of samples to the scenes surface (Bavoil, Sainz and Dimitrov, 2008). Alchemy shares some similarities to previously discussed AO methods to derive Alchemy AO hence we will focus on the details specific to the Alchemy AO method as many of the surrounding details have been covered in our previous discussions regarding the other candidate ambient occlusion methods. We follow the steps in (McGuire et al., 2011b, p.27) to set up the fundamentals for the Alchemy AO algorithm and utilise our understanding from (Timurson, 2021) implementation of *Scalable Ambient Occlusion* (McGuire, Mara and Luebke, 2012b) to complete the final calculations for **Eq.10** in (McGuire et al., 2011b, p.27).

As we have done before, we first need to obtain the positions and the normal information in view-space. In this implementation we use the depth buffer and our previously mentioned, *depthToPositions* and *depthToNormal* functions to obtain the position and normal data from the depth buffer.

```
layout(binding = 1) uniform sampler2D texNoise;

layout(binding = 2) uniform sampler2D gPosition;

layout(binding = 3) uniform sampler2D depthMap;

// position in view-space from depth buffer

vec3 Position = depthToPosition(uvCoords);

// normals in view-space from depth buffer

vec3 Normal = depthToNormal(uvCoords).xyz;
```

We will define our Alchemy AO helper variables which we can use to adjust the ambient occlusion method as seen in **Eq.15.**

```
float sampleRadius = 1.0;

int samples = 12

float bias = 0.001;

float shadowScalar = 0.5;

float shadowContrast = 0.5;

const float epsilon = 0.0001;
```

We generate random values between [0,1] using a random hash value generating function. The value will be used to obtain a sample point within a disk.

```
//https://stackoverflow.com/a/4275343/14723580

// Generate random values

vec2 RandomHashValue()

{

   return fract(sin(vec2(RANDOMVAUE += 0.1, RANDOMVAUE += 0.1)) * vec2(43758.5453123,
   22578.1459123));

}
```

The author suggests to uniformly select a sample point within the disk. We create a function to return a point within a disk *DiskPoint* which takes in the x, y value of the random value we generated and returns a point within the disk.

```
//https://stackoverflow.com/a/50746409/14723580

// Obtain a random sample from the disk, returned as cartesian coordinates

vec2 DiskPoint(float sampleRadius, float x, float y)

{

   float r = sampleRadius * sqrt(x);

   float theta = y * (2.0 * PI);

   return vec2(r * cos(theta), r * sin(theta));

}
```

Note that we $\sqrt{x}$ when multiplying by the radius. This is because we want the number of points to linearly increase with the disk radius of r, that is, the probability density function will be linear to ensure we are uniformly sampling within the disk.

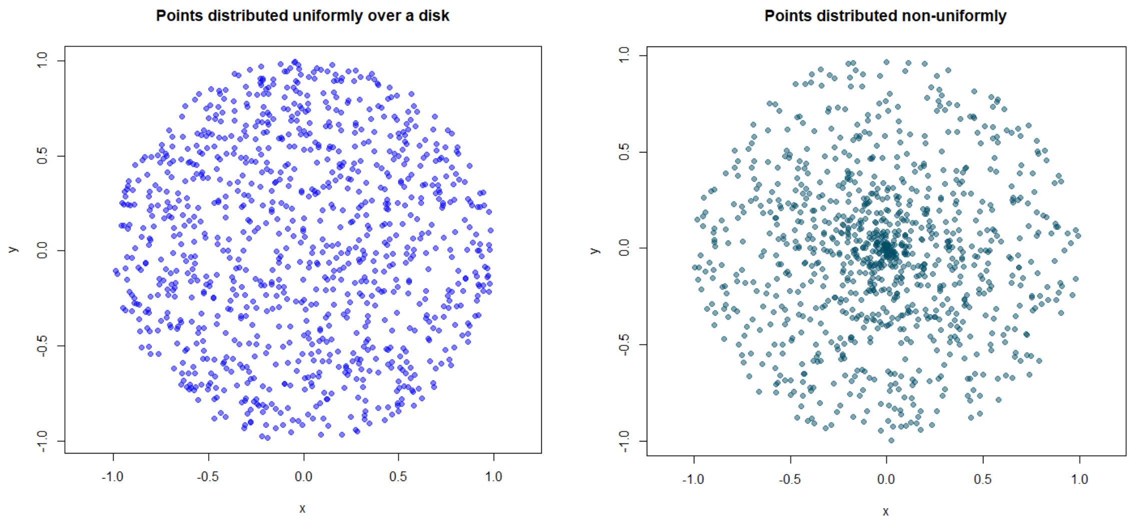


Figure 18: *Distribution of samples: Uniform sampling in disk (left) non-uniform sampling in disk (right).*

We offset *uvCoords* with the sample to obtain the sample position. We subsequently utilise the *depthToPositions* function to conver the same position to view-space. This will be used to obtain the distance from our fragment to the sample point.

```
vec2 RandomValue = RandomHashValue();
vec2 disk = DiskPoint(radius, RandomValue.x, RandomValue.y);
vec2 samplepos = uvCoords + disk.xy * screen_radius;
vec3 P = depthToPosition(samplepos.xy);
```

We can then calculate the vector $\vec{v}$ which will be the distance from our current fragment to the sample position. We have already obtained the position of our current fragment from the depth-buffer. With our newly calculated sample position we can obtain the distance, providing us with the projected area of the sample point.

```
vec3 V = P - Position;
float Len = dot(V, V);
float Heaviside = step(length(dot(V, V)), sampleRadius);
```

The Heaviside step function is implemented here using the ***step*** function which is part of the GLSL specification to implement distance attenuation, ensuring samples are within the defined radius. Next, with the defined depth threshold, we can implement the Bias distance of the algorithm which will allow us to tune for appearance to reduce self-shadowing and light leaks in corners. The *Bias* is scaled by $\boldsymbol{z}$ as suggested by the authors in order to prevent dot products from becoming sensitive to errors. Then we begin to perform the summation of the obscurance factor.

```
float dD = bias * Position.z;
float c = 0.1 * sampleRadius;
// Summation of the obscurance
ambientValue += (max(0.0, dot(Normal, V) + dD) * Heaviside) / (dot(V, V) + epsilon);
```

Finally, we can apply the final scalar multiplications in order to average and intensify the shadowing in the final output.

```
ambientValue *= (2.0 * shadowScalar) / samples;
ambientValue = max(0.0, 1.0 - pow(ambientValue, shadowContrast));
```

4.5 Blur

The raw output from our SSAO methods will produce a noisy output because of the consistent randomness provided by the tiled random vectors(noise) which we used to trade banding for noise (see **Figure 19**). In order to mitigate the effect of the noise on the final output, we can blur the SSAO output to obtain a more acceptable and pleasing result.

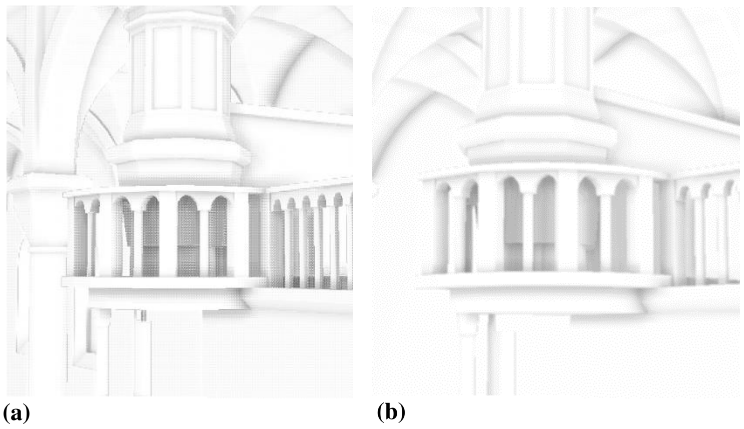


Figure 19: *(a) Without blur, random noise pattern is visible. (b) Blur filter mitigates the random texture noise.*

The blur implementation is based on (Joey De Vries, 2020) blur shader. In order to blur the SSAO result we need to take the output result from the SSAO pass as an input into our blur shader.

```
layout(binding = 0) uniform sampler2D ssao;
```

We will first need to obtain the size of a single texel using the ***textureSize*** function which is part of the *GLSL* specification. We will later use the texel value to convert to texture coordinates [0,1] to correctly offset from our current uvCoords.

```
vec2 texelSize = 1.0 / vec2(textureSize(ssao, 0));
```

Since a 4x4 noise texture which was tiled across the screen, we need to loop through the 4x4 square texture and blur the texture by calculating an average value for the current fragment. Two for loops are used to traverse the width and height of the 4x4 texture.

```
vec2 texelSize = 1.0 / vec2(textureSize(ssao, 0));

for (int x = -2; x < 2; ++x)

{

   for(int y = -2; y < 2; ++y) { ... }

}
```

The start of a column is given by the x value and traverse the column using y. We obtain an offset which is multiplied by the *texelSize* to convert within the range [0,1]. We can use this offset by adding it to our current uvCoords and sampling the SSAO input texture at that point to obtain the value of the texel. A *result* variable is used to accumulate the sample values.

```
vec2 texelSize = 1.0 / vec2(textureSize(ssao, 0));
for (int x = -2; x < 2; ++x)
{
   for(int y = -2; y < 2; ++y) {
        vec2 offset = vec2(float(x), float(y)) * texelSize;
        result += texture(ssao, uvCoords + offset).r;
   }
}
```

We calculate an average value to produce the blur effect on the given fragment of the SSAO input which will remove the noise created by the random vector noise texture.

```
fragColor = result / (4.0 * 4.0);
```

5 Results & Analysis

This chapter will present and compare the results from testing the various ambient occlusion methods and discuss which method proved to be superior. We will begin by outlining the testing methodology and then walk the reader through the results obtained.

5.1 Testing Methodology

In order to evaluate which AO method is superior, the ambient occlusion methods will be tested. The tests and results will be obtained from a desktop PC with an Nvidia GTX 980Ti GPU utilising a resolution of 1920x1080. The testing has been divided into two categories. The first category is **Performance configuration** which aims to display the ambient occlusions output quality with adjusted parameters to match a ray-traced reference while maximising performance for real-time purposes; the intent is to produce a good quality output result from the AO method while maintaining acceptable performance. A method must compute the AO pass in $\leq 3ms$ to be considered performant. The second test category is **Quality configuration** which aims to capture and display the performance of the AO methods when adjusted to run at their higher quality settings. The AO parameters will be adjusted to demonstrate the impact of the AOs input parameters on the compute time of the AO pass and the rendering frame. The compute time for the AO pass, blur pass, and the frame time will be captured using *Nvidia NSight Graphics* which will display the compute time taken for the individual passes as well as the total time taken to produce the frame. **Figure 20** shows the testing methodology for a single AO method.

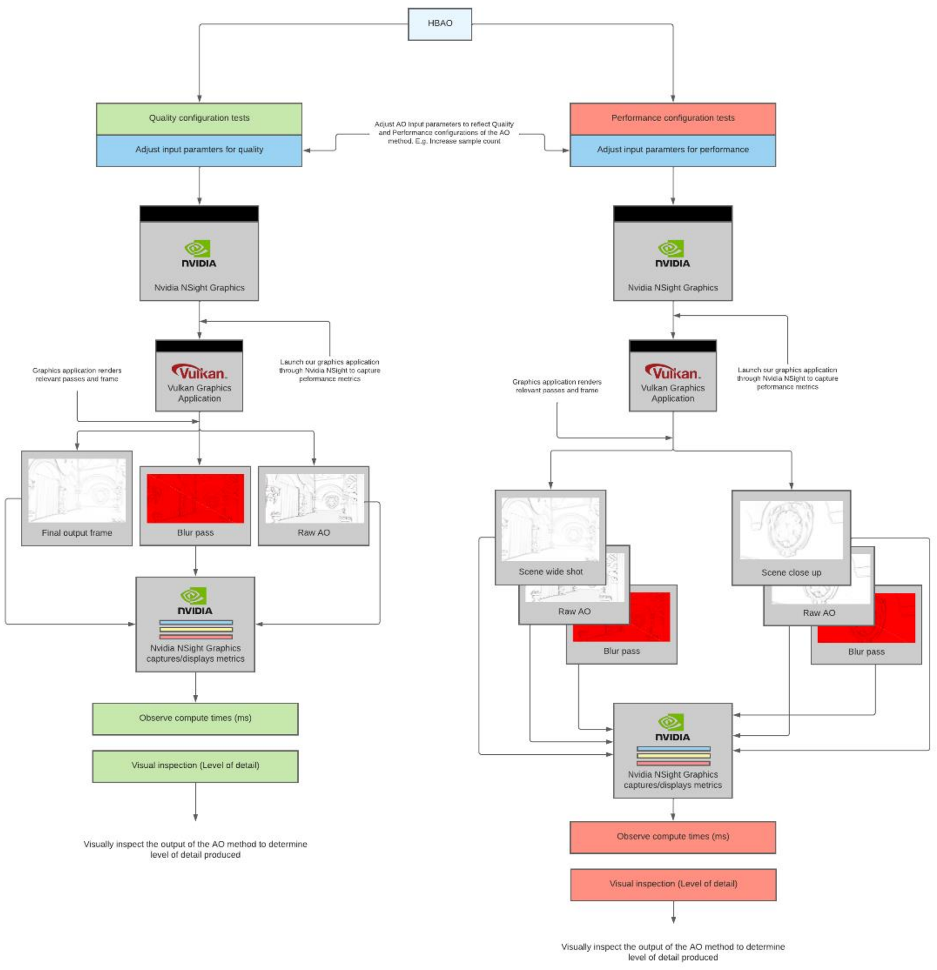

HBAO
Quality configuration tests
Adjust input paramters for quality
Adjust AO Input parameters to reflect Quality and Performance configurations of the AO method. E.g. Increase sample count
Performance configuration tests
Adjust input paramters for performance
NVIDIA
Nvidia NSight Graphics
Graphics application renders relevant passes and frame
Vulkan
Vulkan Graphics Application
Launch our graphics application through Nvidia NSight to capture peformance metrics
Final output frame
Blur pass
Raw AO
Nvidia NSight Graphics captures/displays metrics
Observe compute times (ms)
Visual inspection (Level of detail)
Visually inspect the output of the AO method to determine level of detail produced
Scene wide shot
Scene close up
Raw AO
Blur pass

Figure 20: *Testing methodology visualized for a single AO method (HBAO). We test the AO method under a quality configuration setting (left) and gather the performance metrics using Nvidia Nsight Graphics. We also observe the visual output of the method under the quality configuration to see the level of detail produced and compare it to the ray traced reference. Similarly, we test the AO method under its performance configuration (right) and gather the performance metrics using Nvidia Nsight Graphics and observe the visual output produced by the AO method, also comparing it to the ray traced reference.*

5.1.1 Table Results

We will discuss the structure of the tables displaying the results from our testing. The **Performance configurations** (**Table 3** and **Table 4**) output images are displayed and presented as follows: the rightest image is the AO output without any adjustments or blur applied, it is the **Raw** output; the middle image is the **ray-traced** reference output from *Arnold*; the left image is the output of the AO method with the displayed values in the table and the blur applied. **Quality configuration** (**Table 5**) follows a similar layout with the centre image being the **ray-traced** reference and the left most image as the output using the presented values show in the table.

5.2 Performance Configurations

We will first look at the image-quality the methods produce, adjusting the parameters of the method to produce an output image that matches closely to the ray-traced reference rendered in Maya using Arnold while maintaining an acceptable performance for the ambient occlusion method. The output image will utilize the settings shown in the ***Parameter/Value*** table column. A ***wide shot*** of the scene as well as a ***close-up*** of scene objects are tested and presented to display the quality and impact of the ambient occlusion method on the scene. See **Table 3** for results.

The level of detail captured by (Mittring, 2007) AO compared to both (Bavoil and Sainz, 2008) and (McGuire et al., 2011b) in their performance configurations is very apparent. (Mittring, 2007) suffers from over-occlusion on the scene's geometry and noticeable halos, providing an output far from matching the ray-traced reference. (Bavoil and Sainz, 2008) however, offers better visual detail, matching closely to the ray-traced reference at the expense of compute time. (Bavoil and Sainz, 2008) computes the ***wide-shot*** AO pass in *3.16ms* with a frame time of *6.61ms* compared to (Mittring, 2007) computing in *1.34ms* with a frame time of *4.93ms*. While (Bavoil and Sainz, 2008) is *57.6%* slower to compute, the visual output is superior to (Mittring, 2007), better matching the ray-traced reference. Alchemy AO (McGuire et al., 2011b) in our testing computes the ***wide-shot*** AO pass in *1.10ms* with a frame time of *4.55ms*, computing *65%* faster than HBAO, while also offering a detailed output result, closely matching the quality of (Bavoil and Sainz, 2008). However, the Alchemy AO method did suffer with over-occlusion on some geometry which is not present in the ray-traced reference or (Bavoil and Sainz, 2008) method. However, this issue can be overcome by increasing the sampling radius.

5.2.1 Close-up

The close-up results present similar findings as previously discussed where Alchemy AO dominates with its performance, computing in *1.17ms* while HBAO computes in *3.16ms* and (Mittring, 2007) computing in *1.45ms*. While (Mittring, 2007) is faster than HBAO, the visual result is again inferior to that of HBAO and Alchemy AO. Despite HBAOs longer compute time, in this test the visual quality between Alchemy and HBAO appear similar, with Alchemy achieving a detailed result with a faster compute time. Overall, compute times have increased compared to the wide-shot results. We attribute this to the fact that more fragment shader executions would need to occur as the object takes up a larger portion of the screen space. While Alchemy AO dominates in performance, the visual output of HBAO appears to match closer to the reference image than both Alchemy and (Mittring, 2007) AO, with Alchemy AO suffering from some over occlusion around the lion head. However, this darker occlusion produced by Alchemy AO is not visually unpleasing and maybe artistically more desirable. (Mittring, 2007) visual output appears to be the output furthest from matching the reference, producing a visually unpleasing result.

5.3 Quality Configurations

We will now look at the Quality configuration of the ambient occlusion methods. In this test we aim to uncover how well the method performs and looks when adjusting the input parameters that aim to produce an *Ultra* quality configuration of the method. This will be achieved by adjusting the sample count of the ambient occlusion method and radius. We will measure the performance of the AO method without the blur to demonstrate the raw performance of the method when utilised at their highest quality. See **Table 5** for results.

(Mittring, 2007) AO method at a high-quality setting was tested with *64 samples*, computing in *13.84ms* for the AO pass while producing an image result considerably inferior to that of the ray-traced reference, despite the increase in samples and radius. This appears to be primarily because of the methods use of a sphere, to sample within the radius of influence which produces the noticeable self-occlusion. Despite this, a compute time of 1*3.84ms* and a frame time of *16.70ms* for the level-of-detail produced makes this method far from ideal to be considered a superior AO method in our quality configuration test.

HBAO was tested at its high quality with an increase of samples and radius: *16 samples (20 ray-marched steps)* and *radius 2.0*. HBAO produces a visually-pleasing result, however, this is at the expense of a considerably large amount of performance, computing slower than (Mittring, 2007) with a compute time of *30.02ms* and a frame-time of *33.63ms*. While HBAO offers a visually pleasing result, the price paid in performance is far too great.

From the results it is immediately noticeable that the methods produce much darker results under their quality configurations, an expected result due to the higher sample count. However, what is important to note is the quality and performance characteristics of the methods, especially that of the Alchemy AO method running in its quality settings. While (Mittring, 2007) computes faster, the result is inferior to HBAO and Alchemy. HBAO produces a pleasing visual result however, it takes considerably longer to compute, computing in *30.02ms.* Alchemy AO however, while running at a high-quality setting with *64 samples*, takes only an astonishing *5.88ms* to compute while producing an image quality superior to that of (Mittring, 2007) AO and matching and excelling in some areas to HBAOs detailed output with significantly less compute time, computing *80.4%* faster than (Bavoil and Sainz, 2008). Alchemy AO also fares the best when matching the ray-traced reference, producing AO detail that closely matches the reference render from *Arnold*. Alchemy AO at a high-quality setting produces a visually detailed output while maintaining an acceptable compute time compared to the other candidate methods, defining itself as the superior model in the quality configuration test.

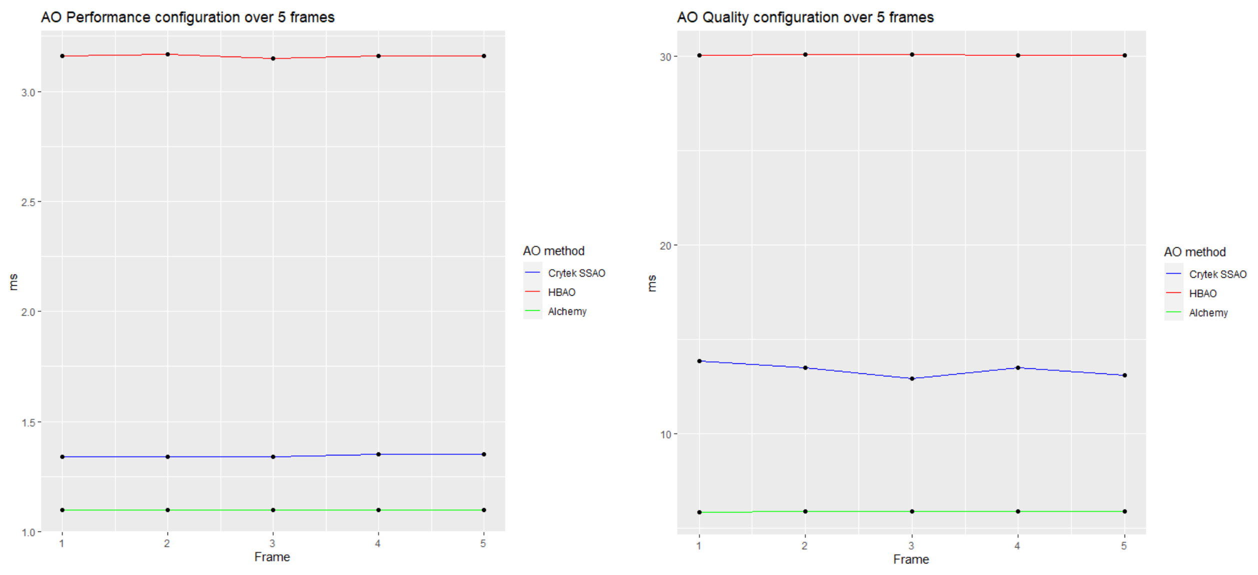


Figure 21: *Performance of AO methods captured over 5-frames. Performance configuration (left) Quality configuration (right). All AO methods hold a relatively consistent framerate when running in real-time. We attribute the consistence frame rates largely to the fact that the observed scene is still, with no animation or movement to cause the AO method to significantly shift in compute time per-frame thus we leave measurement of framerates over multiple frames in real-time out of the testing methodology when evaluating the AO methods.*

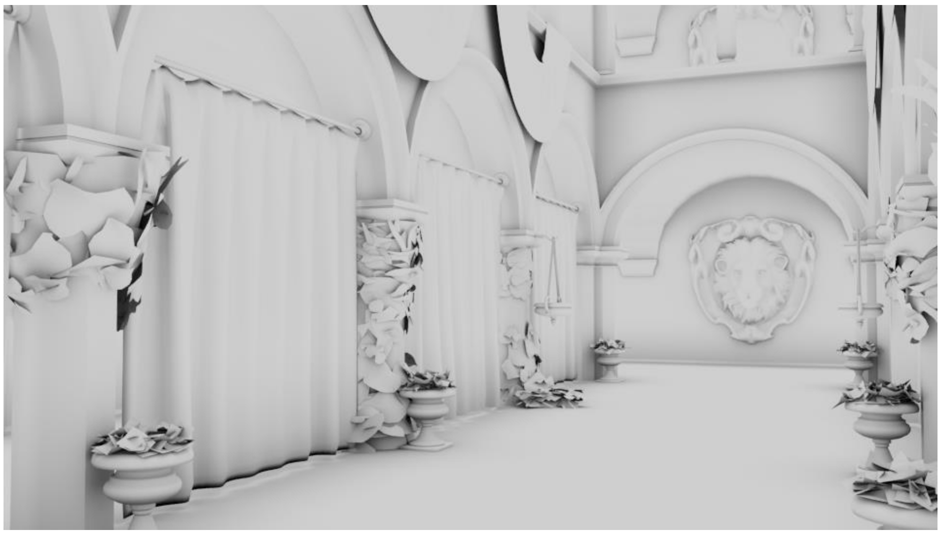


Figure 23: *Ray traced reference rendered using Arnold in Maya. SSAO methods will be compared against this reference render (wide shot).*

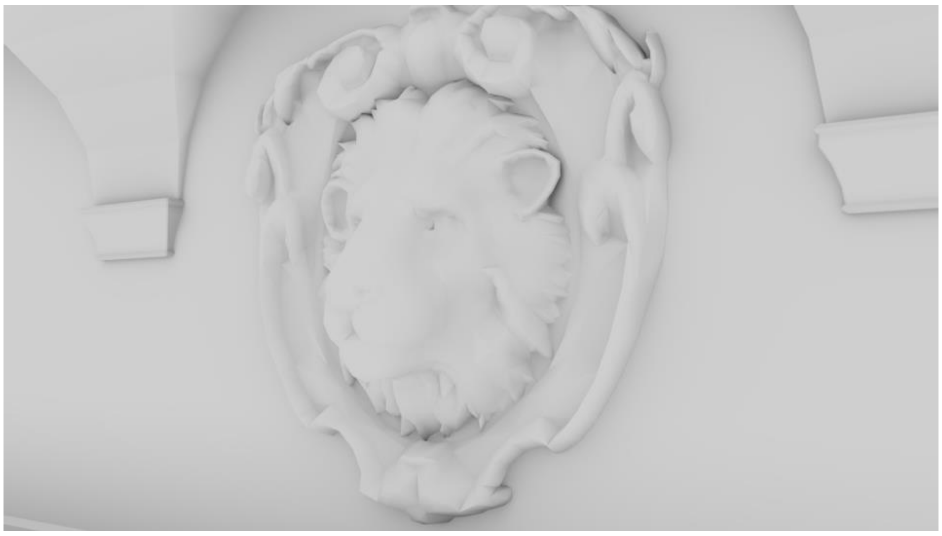
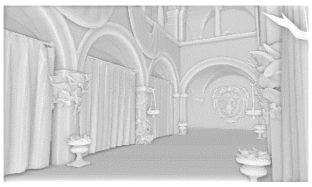
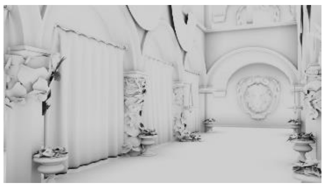
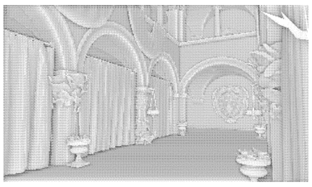
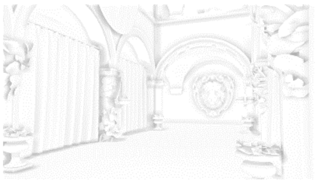
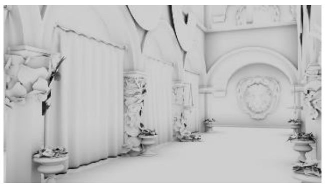
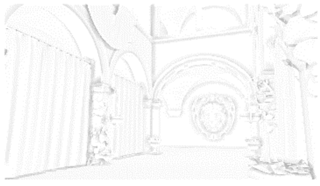


Figure 22: *Ray traced reference rendered with Arnold in Maya for close-up comparison for SSAO methods.*

Parameter/Value	Output	Reference	Raw
Samples: 16 Radius: 0.299 Blur: 0.20ms Performance: AO(1.34ms) Frame time: 4.93ms			
Samples: 6(6 ray-march steps) Radius: 0.549 Bias: 0.3 Blur: 0.21ms Performance: AO(3.16ms) Frame time: 6.61ms			

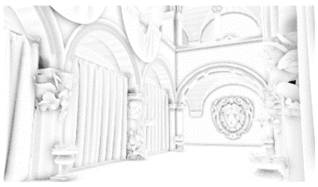
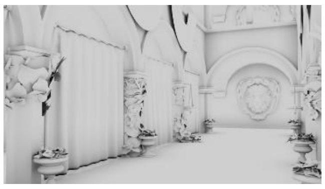
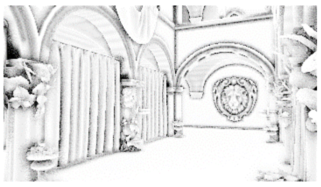
Samples: 12 Radius: 0.549 Bias: 0.3 Blur 0.21ms Performance: AO(1.10ms) Frame time: 4.55ms Sigma: 0.128 Kappa: 1.560			

Table 3: *Performance configuration [Match ray-traced reference]: Wide shot.*

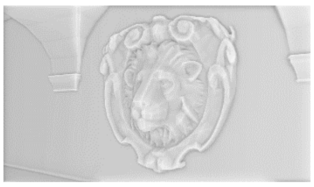
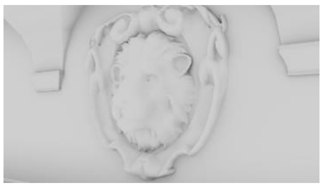
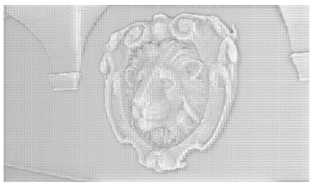
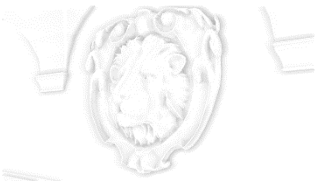
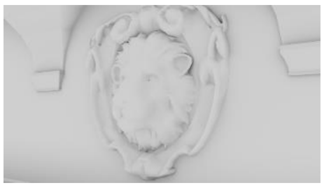
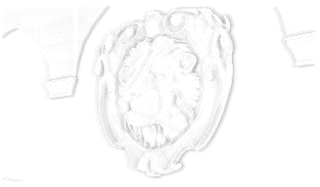
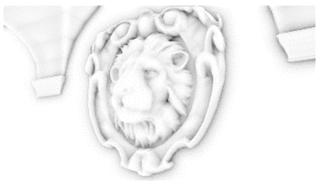
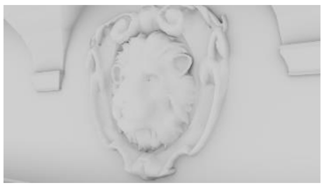
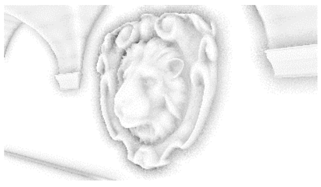
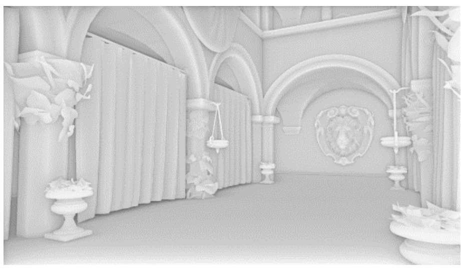
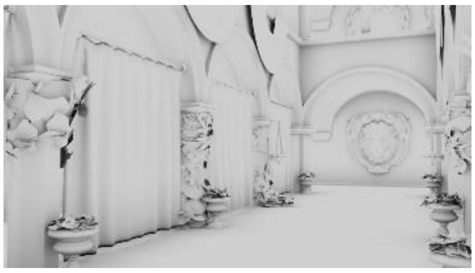
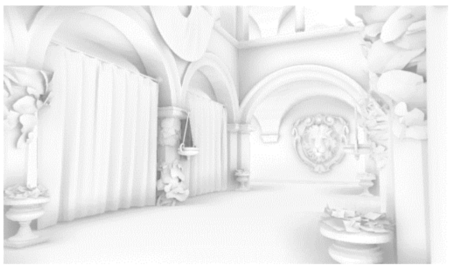
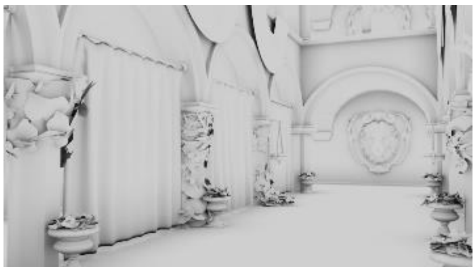
Parameter/Value	Output	Reference	Raw
Samples: 16 *Radius: 0.299* *Blur: 0.20ms* *Performance AO(1.45ms)* *Frame time: 3.30ms*			
Samples: 6(6 ray-march steps) *Radius: 0.5* *Bias: 0.3* *Blur: 0.21ms* *Performance: AO(3.16ms)* *Frame time: 6.26ms*			
Samples: 12 *Radius: 0.5* *Blur 0.21ms* *Performance: AO(1.17ms)* *Frame time: 4.30ms* *Sigma: 0.128* *Kappa: 1.560*			

Table 4: *Performance configuration [Match ray-traced reference]: Close-up.*

Parameter/Value	Output	Reference
Samples: 64 *Radius: 0.391* *Blur: N/A* *Performance: AO (13.84ms)* *Frame time: 16.70ms*		
Samples: 16(20 ray-march steps) *Radius: 2.0* *Bias: 0.3* *Blur: N/A* *Performance: AO (30.02ms)* *Frame time: 33.63ms*		

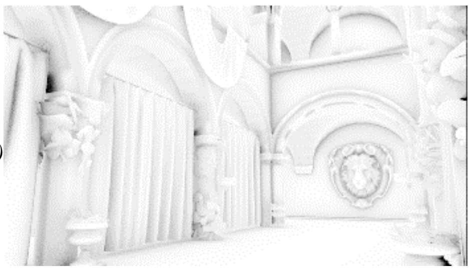
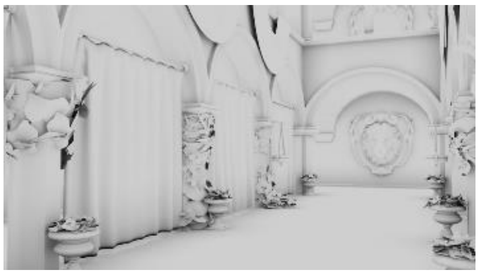
Samples: 64 *Radius: 0.5* *Bias: 0.3* *Blur N/A* *Performance: AO(5.8ms)* *Frame time: 8.77ms* *Sigma = 0.139* *Kappa = 1.441*		

Table 5: *Quality configuration [Match ray-traced reference].*

5.4 Superior method

Following the tests and the discussion above, it is apparent which method among the commonly used ambient occlusion methods in real-time graphics applications is superior. We previously defined a superior technique as one that can offer both acceptable performance in its performance configuration and a high level of detail. We define a method to be considered performant if it can compute it's AO pass in $\leq 3ms$. (Mittring, 2007) proved to be a strong competitor because of its performance however, the visual output is far from a realistic ambient occlusion result and therefore, not the superior method. While HBAO proves to be a strong competitor for the superior method, because of the high level of detail output and near-acceptable compute time in its performance configuration, just exceeding our desired limit by $0.16ms$, it is the Alchemy ambient occlusion method however, that is undoubtedly the superior ambient occlusion method for real-time graphics applications. Alchemy AO greatly outperforms HBAO in both our performance and quality configuration tests. HBAO in its performance configuration computed in *3.16ms*; a compute time *65%* slower than Alchemy AOs performance configuration which computed the AO pass in an astonishing *1.10ms* while offering the defined detail provided by HBAO and artistic control parameters. In addition to Alchemy AOs faster compute time in the performance configuration, the quality configuration tests presented similar results with HBAO computing in *30.02ms* in its quality configuration compared to Alchemy AOs *5.88ms*. HBAO is again the slower ambient occlusion method, computing *80.4%* slower than Alchemy AO. The level-of-detail and performance of Alchemy AO under both quality and performance configurations makes Alchemy, undoubtedly the superior AO method.

5.5 Future work

An extension to the research carried out in this thesis would be to explore more recent strategies for real-time ambient occlusion such as another screen-space approach called Ground Truth Ambient Occlusion (GTAO) (Jimenez et al., 2016). This technique builds upon the strong mathematical and practical foundations laid out by HBAO to produce more accurate indirection occlusion. Furthermore, exploration of techniques that have become more feasible in real-time as a result of hardware innovations such as real-time raytracing, introducing ray-traced ambient occlusion. The exploration of ray-traced ambient occlusion would be a natural step forward to further investigate the space of ambient occlusion techniques for real-time rendering since ray traced results often provide a superior output which may render screen-space ambient occlusion methods obsolete. An investigation into recent innovations surrounding ambient occlusion such as GTAO and ray-traced ambient occlusion will provide a more complete comparison and understanding of the ambient occlusion techniques that can be used in real-time graphics applications and offer meaningful details regarding the newer methods, showcasing what they offer compared to the most commonly used methods discussed in this thesis.

Based on the research presented in this thesis, it is not entirely clear the impact the various ambient occlusion (AO) methods have on the final image quality and the compute time associated. The discussion remains focused on the compute time of each AO pass and the rendered frame without including lighting, shading, atmospheric effects, post-processing effects and more, which would be commonly found within a real-time graphics application. This is another avenue for further research on the work presented in this thesis; presenting tests and results which demonstrate the performance and quality of the AO with the inclusion of shading and post-processing techniques to give a more meaningful demonstration of the performance and quality impact of the AO methods.

In addition to this, another extension to the research that is carried out in this thesis would be to test the AO methods in scenes with movement. The testing in this thesis focuses on a still scene which disallows us to understand how the AO methods perform with movement within the scene such as character animations, evident in **Figure 21**. Including this approach within the testing will provide a more detailed insight into the AO method as issues can arise with movement within the scene such as visible artifacting (Waldner, 2AD). Testing the AO methods with animated movement will prove meaningful to understand how the AO methods compare when utilised in a moving environment which will change the scene information to produce the AO. This will be a meaningful extension as many real-time graphics applications will include some form of movement and animation.

5.6 Conclusion

In this thesis, we investigated and discussed the most commonly used screen-space ambient occlusion methods in real-time rendering for applications such as video games and interactive visualisations and discovered which AO method among those commonly used is the most superior method. We defined a method to be superior if it could offer both acceptable performance and a pleasing visual result which matched a ray traced reference where performance was considered acceptable if the AO pass was computed in $\leq 3ms$. In order to find the most superior method, the AO methods were evaluated by capturing their performance and visual output under a under a quality configuration and a performance configuration which were attuned using their respective AO parameters to match the ray traced reference. The methods were tested in varying scenarios such as a wide shot of a scene and a close-up of scene objects. The results obtained were analysed and compared with each other in order to determine which among the candidate methods proved to be superior through our testing. The aim to find the superior method was achieved. From our testing results we uncovered that (Mittring, 2007) fared well in our performance configuration tests, computing faster than HBAO, however, it produced a visually unpleasing result which was far from matching the ray traced reference, HBAO, and Alchemy AO's visual output. The results indicated HBAO to be a strong candidate for the superior method, providing near-acceptable performance in the performance configuration tests, and a detailed AO output which closely matched our ray-traced reference. However, the visually detailed output came at a performance cost and became apparent in our quality configuration test where the AO method was computing much slower than the other respective AO methods. While HBAO was a strong contender for the superior method due to its visual output and near-acceptable performance in its performance configuration, the Alchemy AO method proved to be far superior, producing a visual output result that closely matched that of HBAO and the ray traced reference image, while rendering with a significantly lower compute time in both the performance configuration tests and the quality configuration test. The incredible performance observed from the Alchemy AO algorithm in our testing proved that the Alchemy AO method is the most superior method for AO in real-time graphics applications, among those commonly used in real-time graphics.

6 References

Abi-Chahla, F. (2009). *When Will Ray Tracing Replace Rasterization*. tomshardware.com.

Bavoil, L. and Sainz, M. (2008). *Image-Space Horzion-Based Ambient Occlusion*. https://developer.download.nvidia.com/presentations/2008/SIGGRAPH/HBAO_SIG08b.pdf.

Bavoil, L. and Sainz, M. (2009). *ShaderX7: Advanced Rendering Techniques*.

Bavoil, L., Sainz, M. and Dimitrov, R. (2008). *Image-Space Horizon-Based Ambient Occlusion*.

Caulfield, B. (2019). *What's the Difference Between Ray Tracing, Rasterization? | NVIDIA Blog*. [online] The Official NVIDIA Blog. Available at: https://blogs.nvidia.com/blog/2018/03/19/whats-difference-between-ray-tracing-rasterization/.

Chapman, J. (2013). *john-chapman-graphics: SSAO Tutorial*. [online] john-chapman-graphics. Available at: http://john-chapman-graphics.blogspot.com/2013/01/ssao-tutorial.html.

Cook, R.L. and Torrance, K. (1981). A Reflectance Model for Computer Graphics.

de Vries, J. (2020). *LearnOpenGL - SSAO*. [online] learnopengl.com. Available at: https://learnopengl.com/Advanced-Lighting/SSAO.

DutréP., Kavita Bala and Philippe Bekaert (2009). *Advanced global illumination*. Wellesley, Mass: Ak Peters, [Ca.

Farris, J. (2020). *Forging new paths for filmmakers on The Mandalorian*. [online] Unreal Engine. Available at: https://www.unrealengine.com/en-US/blog/forging-new-paths-for-filmmakers-on-the-mandalorian.

Filion, D. and McNaughton, R. (2008). *Starcraft II Effects & Techniques*.

Insight, A. (2022). *Why Are Most Games Coded in C++?* [online] analyticsinsight. Available at: https://www.analyticsinsight.net/why-are-most-games-coded-in-c/.

Jimenez, J., Wu, X.-C., Pesce, A. and Jarabo, A. (2016). *Practical Real-Time Strategies for Accurate Indirect Occlusion*.

Joey De Vries (2020). *Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion*. Kendall & Welling.

Kajalin, V. (2009). *ShaderX 7 : advanced rendering techniques*. Boston, Mass.: Charles River Media.

Kajiya, J.T. (1986). *The Rendering Equation*.

Karabelas, P. (2022). *Spartan Engine*. [online] GitHub. Available at: https://github.com/PanosK92/SpartanEngine.

Loos, B.J. and Sloan, P.-P. (2010). *Volumetric Obscurance*.

McGuire, M., Mara, M. and Luebke, D. (2012). *Scalable Ambient Obscurance*.

McGuire, M., Osman, B., Bukowski, M. and Hennessy, P. (2011a). *The Alchemy Screen-Space Ambient Obscurance Algorithm*. [online] Available at: http://graphics.cs.williams.edu/papers/AlchemyHPG11/.

McGuire, M., Osman, B., Bukowski, M. and Hennessy, P. (2011b). *The Alchemy Screen-space Ambient Obscurance Algorithm*.

Mittring, M. (2007). *Advanced Real-Time Rendering in 3D Graphics and Games Course -SIGGRAPH 2007 Finding Next Gen -CryEngine 2*. [online] Available at: https://developer.amd.com/wordpress/media/2012/10/Chapter8-Mittring-Finding_NextGen_CryEngine2.pdf [Accessed 1 Nov. 2021].

Möller, T., Haines, E., Naty Hoffman, Pesce, A., Iwanicki, M. and Sébastien Hillaire (2018). *Real-time Rendering*. Boca Raton: Crc Press, Taylor & Francis Group.

McGuire, M., Mara, M. and Luebke, D. (2012b). *Scalable Ambient Obscurance*. [online] Available at: https://research.nvidia.com/sites/default/files/pubs/2012-06_Scalable-Ambient-Obscurance/McGuire12SAO.pdf.

Owens, B. (2013). *Forward Rendering vs. Deferred Rendering*. [online] Game Development Envato Tuts+. Available at: https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342.

Parker, B. (2014). *All you need to know about Ambient Occlusion*. [online] Bobby Parker. Available at: https://bobby-parker.com/architectural-rendering-blog/all-you-need-to-know-about-ambient-occlusion [Accessed 24 Oct. 2021].

Rakos, D. (2022). *SIMD in the GPU world – RasterGrid*. [online] www.rastergrid.com. Available at: https://www.rastergrid.com/blog/gpu-tech/2022/02/simd-in-the-gpu-world/.

Scratchapixel (2016). *Monte Carlo Methods in Practice (Monte Carlo Integration)*. [online] www.scratchapixel.com. Available at: https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/monte-carlo-integration.

Seymour, M. (2011). *Ben Snow: the evolution of ILM's lighting tools*. [online] fxguide. Available at: https://www.fxguide.com/fxfeatured/ben-snow-the-evolution-of-ilm-lighting-tools/ [Accessed 24 Oct. 2021].

Shanmugam, P. and Arikan, O. (2007). *Hardware Accelerated Ambient Occlusion Techniques on GPUs*.

Technologies, U. (2021). *Understand Real-Time Rendering In Both 3D & 2D with Unity*. [online] unity.com. Available at: https://unity.com/how-to/real-time-rendering-3d.

Thaler, J. (2011). *Deferred Rendering*.

turanszkij (2019). *Improved normal reconstruction from depth*. [online] Wicked Engine Net. Available at: https://wickedengine.net/2019/09/22/improved-normal-reconstruction-from-depth/ [Accessed 28 Dec. 2021].

Timurson, R. (2021). *AlchemyAO*. [online] GitHub. Available at: https://github.com/timurson/AlchemyAO/blob/master/shaders/ambientOcclusion.glsl.

Ubisoft (2020). *What is Vertex and Pixel Shading? | Ubisoft Help*. [online] www.ubisoft.com. Available at: https://www.ubisoft.com/en-us/help/gameplay/article/what-is-vertex-and-pixel-shading/000072849.

Usher, W. (2015). *Screen Space Ambient Occlusion*. [online] GitHub. Available at: https://github.com/Twinklebear/ssao [Accessed 23 Dec. 2021].

Vermeer, J., Scandolo, L. and Eisemann, E. (2021). *Stochastic-Depth Ambient Occlusion*.

Waldner, F. (2AD). *Real-time Ray Traced Ambient Occlusion and Animation*. [online] Available at: http://kth.diva-portal.org/smash/get/diva2:1574351/FULLTEXT01.pdf.

ZaOniRinku (2021). *NeigeEngine*. [online] GitHub. Available at: https://github.com/ZaOniRinku/NeigeEngine/blob/main/src/graphics/shaders/ssao/depthToPositions.frag.

7 Bibliography

Abi-Chahla, F. (2009). *When Will Ray Tracing Replace Rasterization*. tomshardware.com.

Arrskog, T. (2021). *shadow-swan*. [online] GitHub. Available at: https://github.com/topfs2/shadow-swan.

Babey, A. (2020). *VulkanScene*. [online] GitHub. Available at: https://github.com/ExtinctionHD/Vul-kanScene.

Bavoil, L. and Sainz, M. (2008). *Image-Space Horzion-Based Ambient Occlusion*. https://devel-oper.download.nvidia.com/presentations/2008/SIGGRAPH/HBAO_SIG08b.pdf.

Bavoil, L. and Sainz, M. (2009). *ShaderX7: Advanced Rendering Techniques*.

Bavoil, L., Sainz, M. and Dimitrov, R. (2008). *Image-Space Horizon-Based Ambient Occlusion*.

Caulfield, B. (2019). *What's the Difference Between Ray Tracing, Rasterization? | NVIDIA Blog*. [online] The Official NVIDIA Blog. Available at: https://blogs.nvidia.com/blog/2018/03/19/whats-difference-between-ray-tracing-rasterization/.

Chapman, J. (2013). *john-chapman-graphics: SSAO Tutorial*. [online] john-chapman-graphics. Available at: http://john-chapman-graphics.blogspot.com/2013/01/ssao-tutorial.html.

Cook, R.L. and Torrance, K. (1981). A Reflectance Model for Computer Graphics.

de Vries, J. (2020). *LearnOpenGL - SSAO*. [online] learnopengl.com. Available at: https://learno-pengl.com/Advanced-Lighting/SSAO.

DutréP., Kavita Bala and Philippe Bekaert (2009). *Advanced global illumination*. Wellesley, Mass: Ak Peters, [Ca.

Farris, J. (2020). *Forging new paths for filmmakers on The Mandalorian*. [online] Unreal Engine. Available at: https://www.unrealengine.com/en-US/blog/forging-new-paths-for-filmmakers-on-the-mandalorian.

Filion, D. and McNaughton, R. (2008). *Starcraft II Effects & Techniques*.

FlaxEngine (2022). *Flax Engine*. [online] GitHub. Available at: https://github.com/FlaxEn-gine/FlaxEngine.

Insight, A. (2022). *Why Are Most Games Coded in C++?* [online] analyticsinsight. Available at: https://www.analyticsinsight.net/why-are-most-games-coded-in-c/.

IvanKutenev (2019). *QGE (Quantun Game Engine)*. [online] GitHub. Available at: https://github.com/IvanKutenev/QGE.

Jessup, B. (2022). *Arcane-Engine*. [online] GitHub. Available at: https://github.com/Er-shany/Arcane-Engine.

Jimenez, J., Wu, X.-C., Pesce, A. and Jarabo, A. (2016). *Practical Real-Time Strategies for Accurate Indirect Occlusion*.

Joey De Vries (2020). *Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion*. Kendall & Welling.

jhk2 (2019). *glsandbox*. [online] GitHub. Available at: https://github.com/jhk2/glsandbox/blob/master/kgl/samples/ao/hbao.glsl.

Kajalin, V. (2009). *ShaderX 7 : advanced rendering techniques*. Boston, Mass.: Charles River Media.

Kajiya, J.T. (1986). *The Rendering Equation*.

Keutel, M. (2010). *Ambient Occlusion – approaches in screen space (SSAO) : Michael Keutel | Portfolio*. [online] www.michaelkeutel.de. Available at: http://www.michaelkeutel.de/computer-graphics/ambient-occlusion/ssao-screen-space-approaches-to-ambient-occlusion/.

Kubisch, C. (2015). *gl_ssao*. [online] GitHub. Available at: https://github.com/nvpro-samples/gl_ssao/blob/master/hbao.frag.glsl.

Karabelas, P. (2022). *Spartan Engine*. [online] GitHub. Available at: https://github.com/PanosK92/SpartanEngine.

Loos, B.J. and Sloan, P.-P. (2010). *Volumetric Obscurance*.

Lenaerts, D. (2013). *An alternative implementation for HBAO | Der Schmale - Real-time 3D programming*. [online] https://www.derschmale.com/2013/12/20/an-alternative-implementation-for-hbao-2/. Available at: https://www.derschmale.com/2013/12/20/an-alternative-implementation-for-hbao-2/.

McGuire, M., Mara, M. and Luebke, D. (2012). *Scalable Ambient Obscurance*.

McGuire, M., Osman, B., Bukowski, M. and Hennessy, P. (2011a). *The Alchemy Screen-Space Ambient Obscurance Algorithm*. [online] Available at: http://graphics.cs.williams.edu/papers/AlchemyHPG11/.

McGuire, M., Osman, B., Bukowski, M. and Hennessy, P. (2011b). *The Alchemy Screen-space Ambient Obscurance Algorithm*.

Mittring, M. (2007). *Advanced Real-Time Rendering in 3D Graphics and Games Course -SIGGRAPH 2007 Finding Next Gen -CryEngine 2*. [online] Available at: https://developer.amd.com/wordpress/media/2012/10/Chapter8-Mittring-Finding_NextGen_CryEngine2.pdf [Accessed 1 Nov. 2021].

Möller, T., Haines, E., Naty Hoffman, Pesce, A., Iwanicki, M. and Sébastien Hillaire (2018). *Real-time Rendering*. Boca Raton: Crc Press, Taylor & Francis Group.

McGuire, M., Mara, M. and Luebke, D. (2012b). *Scalable Ambient Obscurance*. [online] Available at: https://research.nvidia.com/sites/default/files/pubs/2012-06_Scalable-Ambient-Obscurance/McGuire12SAO.pdf.

Owens, B. (2013). *Forward Rendering vs. Deferred Rendering*. [online] Game Development Envato Tuts+. Available at: https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342.

Overvoorde, A. (2016). *Introduction - Vulkan Tutorial*. [online] vulkan-tutorial.com. Available at: https://vulkan-tutorial.com/.

Parker, B. (2014). *All you need to know about Ambient Occlusion*. [online] Bobby Parker. Available at: https://bobby-parker.com/architectural-rendering-blog/all-you-need-to-know-about-ambient-occlusion [Accessed 24 Oct. 2021].

Pharr, M. and Green, S. (2004). *Chapter 17. Ambient Occlusion*. [online] NVIDIA Developer. Available at: https://developer.nvidia.com/gpugems/gpugems/part-iii-materials/chapter-17-ambient-occlusion.

Rakos, D. (2022). *SIMD in the GPU world – RasterGrid*. [online] www.rastergrid.com. Available at: https://www.rastergrid.com/blog/gpu-tech/2022/02/simd-in-the-gpu-world/.

Rasquinha, S. (2022). *Lumen Engine*. [online] GitHub. Available at: https://github.com/swr06/Lumen-3D/blob/main/Source/Core/Shaders/DiffuseTrace.glsl.

Scratchapixel (2016). *Monte Carlo Methods in Practice (Monte Carlo Integration)*. [online] www.scratchapixel.com. Available at: https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/monte-carlo-integration.

Seymour, M. (2011). *Ben Snow: the evolution of ILM's lighting tools*. [online] fxguide. Available at: https://www.fxguide.com/fxfeatured/ben-snow-the-evolution-of-ilm-lighting-tools/ [Accessed 24 Oct. 2021].

Shanmugam, P. and Arikan, O. (2007). *Hardware Accelerated Ambient Occlusion Techniques on GPUs*.

S, E. (2022). *Build*. [online] GitHub. Available at: https://github.com/hotstreams/limitless-engine.

Technologies, U. (2021). *Understand Real-Time Rendering In Both 3D & 2D with Unity*. [online] unity.com. Available at: https://unity.com/how-to/real-time-rendering-3d.

Thaler, J. (2011). *Deferred Rendering*.

turanszkij (2019). *Improved normal reconstruction from depth*. [online] Wicked Engine Net. Available at: https://wickedengine.net/2019/09/22/improved-normal-reconstruction-from-depth/ [Accessed 28 Dec. 2021].

Timurson, R. (2021). *AlchemyAO*. [online] GitHub. Available at: https://github.com/timurson/AlchemyAO/blob/master/shaders/ambientOcclusion.glsl.

Ubisoft (2020). *What is Vertex and Pixel Shading? | Ubisoft Help*. [online] www.ubisoft.com. Available at: https://www.ubisoft.com/en-us/help/gameplay/article/what-is-vertex-and-pixel-shading/000072849.

Usher, W. (2015). *Screen Space Ambient Occlusion*. [online] GitHub. Available at: https://github.com/Twinklebear/ssao [Accessed 23 Dec. 2021].

Vermeer, J., Scandolo, L. and Eisemann, E. (2021). *Stochastic-Depth Ambient Occlusion*.

Waldner, F. (2AD). *Real-time Ray Traced Ambient Occlusion and Animation*. [online] Available at: http://kth.diva-portal.org/smash/get/diva2:1574351/FULLTEXT01.pdf.

ZaOniRinku (2021). *NeigeEngine*. [online] GitHub. Available at: https://github.com/ZaOniRinku/NeigeEngine/blob/main/src/graphics/shaders/ssao/depthToPositions.frag.

8 Appendix

Crytek SSAO

```glsl
#version 450
layout(binding = 0) uniform SSAOubo {
    vec4 samples[64];
    int sample_amount;
    float radius;
    int hbaoSampleDirection;
    float hbaoSteps;
    int hbaoNumberOfSteps;
    float hbaoAmbientLightLevel;
    int alchemySampleTurns;
    float alchemySigma;
    float alchemyKappa;
}ssao;
layout(binding = 4) uniform CameraProjection {
    mat4 model;
    mat4 view;
    mat4 proj;
}camera;
layout(binding = 1) uniform sampler2D texNoise;
layout(binding = 2) uniform sampler2D gPosition;
layout(binding = 3) uniform sampler2D depthMap;
layout(location = 0) in vec3 fragColor;
layout(location = 1) in vec2 uvCoords;
layout(location = 0) out float outColor;
const vec2 noiseScale = vec2(1920.0/4.0, 1080.0/4.0);
//(ZaOniRinku, 2021) Use depth to obtain normal data
vec3 depthToPositions(vec2 tc)
{
    float depth = texture(depthMap, tc).x;
    vec4 clipSpace = vec4(tc * 2.0 - 1.0, depth, 1.0);
    vec4 viewSpace = inverse(camera.proj) * clipSpace;
    return viewSpace.xyz / viewSpace.w;
}
//(ZaOniRinku, 2021) Use depth to obtain normal data

vec4 depthToNormals(vec2 tc)
{
    float depth = texture(depthMap, tc).x;

    vec4 clipSpace = vec4(tc * 2.0 - 1.0, depth, 1.0);
    vec4 viewSpace = inverse(camera.proj) * clipSpace;
    viewSpace.xyz /= viewSpace.w;

    vec3 pos = viewSpace.xyz;
    vec3 n = normalize(cross(dFdx(pos), dFdy(pos)));
    n *= - 1;

    return vec4(n, 1.0);
}
```

```
void main()
{
    float radius = ssao.radius;
    float bias = 0.025;

    // Obtain the fragment view space position
    vec3 viewSpacePositions = depthToPositions(uvCoords);
    // Obtain the fragment normal position from view space
    vec4 viewSpaceNormals = depthToNormals(uvCoords);
    // Sample random vector from texture. Apply noise scale to scale across
width and height of display
    vec3 randomVec = texture(texNoise, uvCoords * noiseScale).xyz;

    //(Joey De Vries, 2020) Create a TBN matrix to convert the sample from
tangent-space to view-space
    vec3 tangent = normalize(randomVec - viewSpaceNormals.xyz * dot(randomVec,
viewSpaceNormals.xyz));
    vec3 bitangent = cross(viewSpaceNormals.xyz, tangent);
    mat3 TBN = mat3(tangent, bitangent, viewSpaceNormals.xyz);

    // create the plane to reflect as suggested by (Mittring, 2007)
    vec3 plane = texture(texNoise, uvCoords * noiseScale).xyz - vec3(1.0);

    float occlusion = 0.0;
    for(int i = 0; i < ssao.sample_amount; ++i) {

        vec3 samplePos = reflect(ssao.samples[i].xyz, plane); //reflect the
sample
        samplePos = TBN * samplePos; // convert sample to view-space
        samplePos = viewSpacePositions + samplePos * radius; //offset current
position with sample pos
        vec4 offset = vec4(samplePos, 1.0);
        offset = camera.proj * offset; //convert to clip space
        offset.xyz /= offset.w; // perspective divide
        offset.xy = offset.xy * 0.5 + 0.5; // convert to texture coordinate
(0,1)
        float sampleDepth = texture(gPosition, offset.xy).z; // obtain sample
pos depth value
        float rangeCheck = (viewSpacePositions.z - sampleDepth) < radius ? 1.0
: 0.0; // range check to ensure within radius
        occlusion += (sampleDepth >= samplePos.z + bias ? 1.0 : 0.0) *
rangeCheck; // check if depth > z pos to determine occlusion
    }
    // subtract 1.0 to allow AO to be used with other lighting calculations
    occlusion = 1.0 - (occlusion / ssao.sample_amount);
    outColor = occlusion;
}
```

HBAO

```
layout(binding = 0) uniform SSAOubo {
    vec4 samples[64];
    int sample_amount;
    float radius;

    int hbaoSampleDirection;
    float hbaoSteps;
    int hbaoNumberOfSteps;
    float hbaoAmbientLightLevel;

    int alchemySampleTurns;
    float alchemySigma;
    float alchemyKappa;
}ssao;

layout(binding = 4) uniform CameraProjection {
    mat4 model;
    mat4 view;
    mat4 proj;
}camera;

layout(binding = 1) uniform sampler2D texNoise;
layout(binding = 2) uniform sampler2D gPosition;
layout(binding = 3) uniform sampler2D depthMap;

layout(location = 0) in vec3 fragColor;
layout(location = 1) in vec2 uvCoords;

layout(location = 0) out vec4 outColor;

#define PI 3.1415926535897932384626433832795

float RADIUS = ssao.radius;
float NUMBER_OF_SAMPLING_DIRECTIONS = ssao.hbaoSampleDirection;
float STEP = ssao.hbaoSteps; //0.04
float NUMBER_OF_STEPS = ssao.hbaoNumberOfSteps;
float TANGENT_BIAS = 0.3;

float ao = 0.0;
float occlusion = 0.0;

const vec2 noiseScale = vec2(1920.0/4, 1080.0/4);
```

```
//(ZaOniRinku, 2021) Use depth to obtain position data
vec4 depthToPosition(vec2 uv) {

    float depth = texture(depthMap, uv).x;
    vec4 clipSpace = vec4(uv * 2.0 - 1.0, depth, 1.0);
    vec4 viewSpace = inverse(camera.proj) * clipSpace;
    viewSpace.xyz /= viewSpace.w;

    return vec4(vec3(viewSpace), 1.0);
}
//(ZaOniRinku, 2021) Use depth to obtain normal data
vec4 depthToNormal(vec2 tc)
{
    float depth = texture(depthMap, tc).x;

    vec4 clipSpace = vec4(tc * 2.0 - 1.0, depth, 1.0);
    vec4 viewSpace = inverse(camera.proj) * clipSpace;
    viewSpace.xyz /= viewSpace.w;

    vec3 pos = viewSpace.xyz;
    vec3 n = normalize(cross(dFdx(pos), dFdy(pos)));
    n *= - 1;

    return vec4(n, 1.0);
}

//Using noise texture to produce a noise jitter value (Kubisch, 2015)
vec4 GetJitter()
{
    // Get single texel
    return texture(texNoise, (gl_FragCoord.xy / 4));
}

// Slight modified version of (Kubisch, 2015), we use rotation matrix
vec2 RotateDirectionAngle(vec2 direction, vec2 noise)
{
    // contruct a rotation matrix to rotate the direction
    mat2 rotationMatrix = mat2(vec2(noise.x, -noise.y), vec2(noise.y,
noise.x));
    return rotationMatrix * direction;
}

void main()
{
    // position of current fragment
    vec3 pos = vec3(uvCoords, texture(depthMap, uvCoords).r);
    vec4 normal = depthToNormal(uvCoords);
    normal.y = -normal.y;
    vec3 NDC_POS = (2.0 * pos) - 1.0; // normalized device coordinates
    vec4 unprojectPosition = inverse(camera.proj) * vec4(NDC_POS, 1.0);
    vec3 viewPosition = unprojectPosition.xyz / unprojectPosition.w;

    // paper suggests to jitter samples by random offset
    vec3 sampleNoise = texture(texNoise, uvCoords * noiseScale).xyz;
    sampleNoise.xy = sampleNoise.xy * 2.0 - vec2(1.0);

    // A single direction
    float samplingDiskDirection = 2 * PI / NUMBER_OF_SAMPLING_DIRECTIONS;
    vec4 Rand = GetJitter();

    for(int i = 0; i < NUMBER_OF_SAMPLING_DIRECTIONS; i++) {

        // use i to get a new direction by * given direction
        float samplingDirectionAngle = i * samplingDiskDirection;
        //jitter direction
        vec2 samplingDirection =
RotateDirectionAngle(vec2(cos(samplingDirectionAngle),
sin(samplingDirectionAngle)), Rand.xy);

        //tangent angle : inverse cosine
        float tangentAngle = acos(dot(vec3(samplingDirection, 0.0),
normal.xyz)) - (0.5 * PI) + TANGENT_BIAS;
        float horizonAngle = tangentAngle; //set the horizon angle to the
tangent angle to begin with

        vec3 LastDifference = vec3(0);

        // for each direction we step in the direction of that sampling
direction to sample
        for(int j = 0; j < NUMBER_OF_STEPS; j++){

            // step forward in the sampling direction
            vec2 stepForward = (Rand.z + float(j+1)) * STEP *
samplingDirection;
            // use the stepforward position as an offset from the current
fragment position in order to move to that location
            vec2 stepPosition = uvCoords + stepForward;
            // sample at the stepped location to get the depth value
            float steppedLocationZ = texture(depthMap, stepPosition.st).x;
            // complete sample position
            vec3 steppedLocationPosition = vec3(stepPosition,
steppedLocationZ);
            //convert to NDC
            vec3 steppedPositionNDC = (2.0 * steppedLocationPosition) - 1.0;
            vec4 SteppedPositionUnProj = inverse(camera.proj) *
vec4(steppedPositionNDC, 1.0);
```

```
            vec3 viewSpaceSteppedPosition = SteppedPositionUnProj.xyz /
SteppedPositionUnProj.w;

            // Now that we have the view-space position of the offset sample
point
            // We can check the distance from our current fragment to the
offset point

            vec3 diff = viewSpaceSteppedPosition.xyz - viewPosition;
            // If the distance is less than the set radius
            if(length(diff) < RADIUS){

                LastDifference = diff;
                float FoundElevationAngle = atan(diff.z / length(diff.xy));
                // update horizon angle if new found elevation angle is larger
                horizonAngle = max(horizonAngle, FoundElevationAngle);
            }
        }

            float norm = length(LastDifference) / RADIUS;
            float attenuation = 1 - norm * norm;

            occlusion = clamp(attenuation * (sin(horizonAngle) -
sin(tangentAngle)), 0.0, 1.0);
            ao += 1.0 - occlusion * ssao.hbaoAmbientLightLevel; //control AO
darkness
    }

    ao /= NUMBER_OF_SAMPLING_DIRECTIONS;

    outColor = vec4(sum, sum, sum, 1.0);

}
```

Alchemy AO

```glsl
const float PI = 3.14159265359;

layout(binding = 0) uniform SSAOubo {
    vec4 samples[64];
    int sample_amount;
    float radius;

    int hbaoSampleDirection;
    float hbaoSteps;
    int hbaoNumberOfSteps;
    float hbaoAmbientLightLevel;

    float alchemySigma;
    float alchemyKappa;
}ssao;

layout(binding = 4) uniform CameraProjection {
    mat4 model;
    mat4 view;
    mat4 proj;
}camera;

layout(binding = 1) uniform sampler2D texNoise;
layout(binding = 2) uniform sampler2D gPosition;
layout(binding = 3) uniform sampler2D depthMap;

layout(location = 0) in vec3 fragColor;
layout(location = 1) in vec2 uvCoords;


layout(location = 0) out vec4 outColor;

float sampleRadius = ssao.radius;
int samples = ssao.sample_amount;
float bias = 0.001;
float shadowScalar = ssao.alchemySigma;
float shadowContrast = ssao.alchemyKappa;
const float epsilon = 0.0001;
//(ZaOniRinku, 2021) Use depth to obtain position data
vec3 depthToPosition(vec2 tc)
{
    float depth = texture(depthMap, tc).x;
    vec4 clipSpace = vec4(tc * 2.0 - 1.0, depth, 1.0);
    vec4 viewSpace = inverse(camera.proj) * clipSpace;
    return viewSpace.xyz / viewSpace.w;
}
```

```
//(ZaOniRinku, 2021) Use depth buffer to obtain normal data
vec4 depthToNormal(vec2 tc)
{
    float depth = texture(depthMap, tc).x;
    vec4 clipSpace = vec4(tc * 2.0 - 1.0, depth, 1.0);
    vec4 viewSpace = inverse(camera.proj) * clipSpace;
    viewSpace.xyz / viewSpace.w;

    vec3 pos = viewSpace.xyz;
    vec3 n = normalize(cross(dFdx(pos), dFdy(pos)));
    n *= -1;

    return vec4(n, 1.0);
}
float RANDOMVALUE = 0.0f;
//https://stackoverflow.com/a/4275343/14723580:Generate random values
vec2 RandomHashValue()
{
    return fract(sin(vec2(RANDOMVAUE += 0.1, RANDOMVAUE += 0.1)) *
vec2(43758.5453123, 22578.1459123));
}
//https://stackoverflow.com/a/50746409/14723580
// Obtain a random sample from the disk, returned as cartesian coordinates
vec2 DiskPoint(float sampleRadius, float x, float y)
{
    float r = sampleRadius * sqrt(x); // square root number to map point to
radius
    float theta = y * (2.0 * PI); // theta is angle in radians inside disk
    return vec2(r * cos(theta), r * sin(theta));
}
void main()
{
// Randon value updating based on (Rasquinha, 2022) random hash implementation
    RANDOMVALUE = (uvCoords.x * uvCoords.y) * 64.0;
    RANDOMVALUE += fract(ssao.time) * 64.0f;
    // Normals and positions in view-space
    vec3 Position = depthToPosition(uvCoords);
    vec3 Normal = depthToNormal(uvCoords).xyz;
    float ambientValue = 0.0;
    //(Timurson, 2021) projection of a ball around the point
    float screen_radius = (sampleRadius * 0.75 / Position.z);
    for (int i = 0; i < samples; ++i)
    {
        vec2 RandomValue = RandomHashValue();
        vec2 disk = DiskPoint(RandomValue.x, RandomValue.y);
        vec2 samplepos = uvCoords + (sampleRadius * disk.xy) * screen_radius;
        //(Timurson, 2021)
        vec3 P = depthToPosition(samplepos.xy);
        //(Timurson, 2021) built understanding for final calculations from
different AO method implementation
        vec3 P = depthToPosition(samplepos.xy); // convert sample to view-
space
        vec3 V = P - Position; //calculate distance from current frag position
to sample position
        float Heaviside = step(length(dot(V, V)), sampleRadius); //Ensure it
is within the radius of influence using step
        float dD = bias * Position.z; //multiply bias by z as suggested by pa-
per, to prevent dot products becoming sensitive

        // Apply summations calculations following equation 10 from paper
        ao += (max(0.0, dot(Normal, V) + dD) * Heaviside) / (dot(V, V) + epsi-
lon);
    }

    // Apply final calculations for AO following equation 10 from paper
    ao *= (2.0 * shadowScalar) / samples; //apply shadow scalar as show in the
equation of the paper
    ao = max(0.0, 1.0 - pow(ao, shadowContrast)); //final ambient value

    outColor = vec4(ao, ao, ao, 1.0);
}
```

Blur

```
#version 450


layout(binding = 0) uniform sampler2D ssao;

layout(location = 0) in vec3 fragColor;
layout(location = 1) in vec2 uvCoords;


layout(location = 0) out float outColor;

// Based on (Joey De Vries, 2020) blur implementation.
void main()
{
    vec2 texelSize = 1.0 / vec2(textureSize(ssao, 0));
    float result = 0.0;
    for(int x = -2; x < 2; ++x)
    {
        for(int y = -2; y < 2; ++y)
        {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(ssao, uvCoords + offset).r;
        }
    }

    outColor = result / (4.0 * 4.0);
}
```