A Comparative Study of Screen-Space Ambient Occlusion Methods

Bachelor Thesis

Student
Frederik P. Aalund (s093279)[*]
DTU

Supervisor
J. Andreas Bærentzen[†]
DTU

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

In this report we present a comparison between six screen-space ambient occlusion (SSAO) methods. The aim is to compare their strengths and weaknesses and ultimately to find a superior method. The methods are chosen to differ algorithmically to get the broadest overview as possible. We find that a method known as Alchemy AO is the superior approach. The report also includes discussions of topics related to the field of ambient occlusion in general.

Version 1 (February 1, 2013) Initial version handed in to supervisor.

Version 2 (May 24, 2013) Corrected typesetting errors and layout quirks.

[*] frederikaalund+ssao2012@gmail.com
[†] jab@imm.dtu.dk

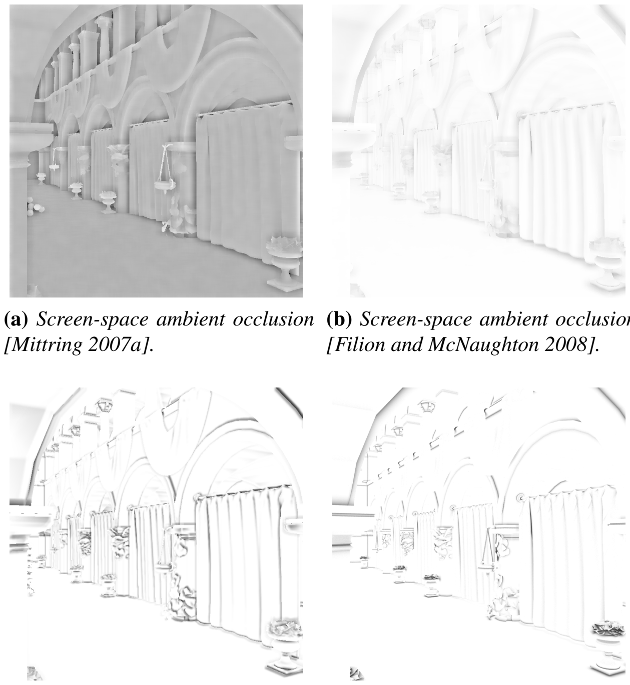
Notation

Select abbreviations

AO	Ambient Occlusion *or* Ambient Obscurance
SSAO	Screen-Space Ambient Occlusion
BRDF	Bidirectional Reflectance Distribution Function
WC	World Coordinates
EC	Eye Coordinates
CC	Clip Coordinates
NDC	Normalized Device Coordinates
SC	Screen Coordinates
TC	Texture Coordinates
GPU	Graphics Processing Unit (Graphics Card)
SPMD	Single Program Multiple Data
RGB	Red, Green, and Blue
GI	Global Illumination
GLSL	OpenGL Shading Language

Contents

List of Figures



(a) *Screen-space ambient occlusion [Mittring 2007a].* **(b)** *Screen-space ambient occlusion [Filion and McNaughton 2008].*

(c) *Screen-space ambient occlusion [McGuire et al. 2011].* **(d)** *Ray-traced ambient occlusion made with Mental Ray as reference.*

Figure 1.1: *Example renders demonstrating three screen-space approaches to ambient occlusion and a ray-traced reference.*

1 Introduction

In this section we will first provide some general motivation for the study of ambient occlusion (AO). Then we proceed to briefly outline the history of ambient occlusion methods and especially the screen-space approaches. Lastly, we give a full description of the project—the basis of this report.

1.1 Motivation

Shadows are key to our perception of the world that surrounds us. They act as a visual cue that spatially relates one object to another; a kind of glue between surfaces. Without shadows objects look as if they are flat, hovering, or both (Figure 1.2a). Studies have shown that both our impressions of shape [Langer and Zucker 1994] and depth [Langer and Bülthoff 2000] are related to the amount of light shadowed from each point on a surface—the occlusion of each point.

In the real world we take the effect of shadows for granted but in a virtual world the shadows must be computed just like everything else. The shadow computation is divided into large-scale shadows (Figure 1.2b) and small-scale shadows (Figures 1.2c and 1.2d). Simply put, ambient occlusion is a model that the describes small-scale shadows under certain conditions. Examples include the shadows between wrinkles in the skin, crevices in the wall, or between your laptop and the table. Putting ambient occlusion into effect clearly increases the visual fidelity of a virtual world. Refer to Figure 1.2 to see the effects of ambient occlusion. Notice how shape in particular is easilier perceived when the light is modulated by ambient occlusion.

1.2 Brief History

Realizations of ambient occlusion has been around in one form or the other for a long time. First using ray-tracing to generate still images . Subsuquently as a pre-processing step for use in real-time interactive applications though limited to static worlds [Möller et al. 2008]. Only since 2007 has it been proven feasible to compute ambient occlusion on-the-fly and in real-time along with other effects expected in a virtual world using a so-called *screen-space* approach [Mittring 2007a; Shanmugam and Arikan 2007; Nguyen 2007]. This allows ambient occlusion to work within dynamic environments in conjunction with animation and physics. Furthermore, screen-space methods do not need pre-calculated occlusion data which relieves both the artists and the memory budget.

Even with all of the above in favor of screen-space ambient occlusion (SSAO), it still has some drawbacks. Ray-tracing will produce superior images in general and conveniently integrates with other global illumination effects such as inter-reflections. The SSAO methods come close at times and some even include simple color-bleeding effects [Ritschel et al. 2009]. Still, they have to discard information to be fast and therefore quality will suffer. Good quality comes at a performance cost as a general rule but algorithmic advancements can help tremendously. See Figure 1.1 for a comparison of the shadows produced with several SSAO methods and a ray-traced reference.

A parallel advancement is the emergence of real-time global illumination methods which seem destined to make pure ambient occlusion routines obsolete (see [Donzallaz and Sousa 2011] and [Mittring 2012] for the state of the art). However, this is not yet the case and dedicated ambient occlusion routines are with us for the foreseeable future [McGuire et al. 2011].

1.3 The Project

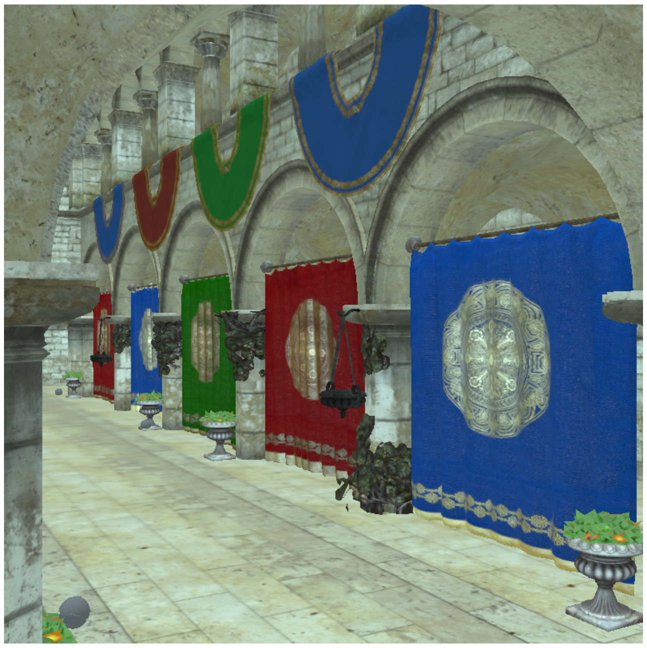
We intend to analyze the current ambient occlusion methods and implement a subset of them. In the best case, to find the superior all-around method in terms of quality versus compute time. The study will be limited to screen-space methods and not the recent global methods. The evaluated methods are chosen to be algorithmically different to get the broadest overview as possible. Nevertheless, slight variations in parameters or implementation may be pursued on a per-method basis if relevant.

SSAO algorithms are usually divided into passes. Therefore, hybrids of existing methods may be formed by combining stages from different sources. We will pursue such hybrids when applicable. In the best case, to find an abstract method from which specific methods can be derived or instead to simply categorize the passes.

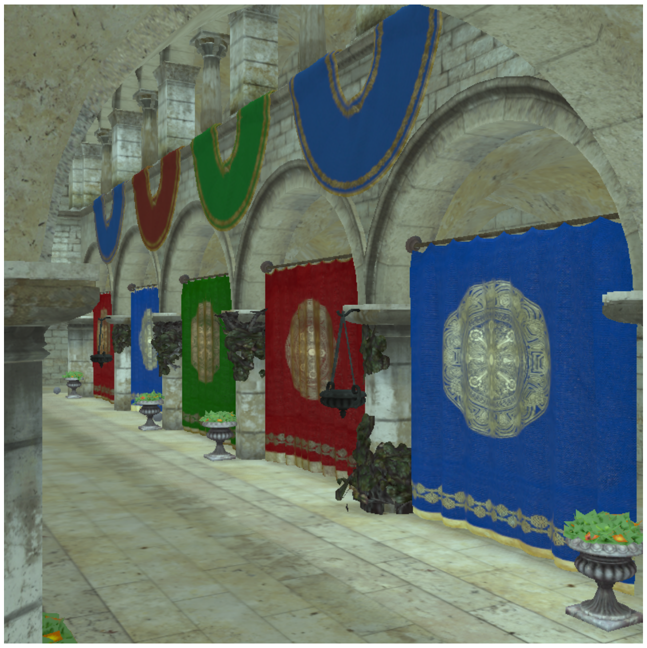
It must be mentioned that all screen-space ambient occlusion methods are embarrassingly parallel and best implemented on a GPU. Therefore, the candidate methods will be realized through shaders. We intend to use the cross-platform OpenGL API to do so.

The end product will be an interactive application that demonstrates chosen screen-space ambient occlusion methods. The evaluation of each method will be done within this application. Here, interactive means that the user will be able to freely navigate through the virtual world in real-time. Being interactive also helps when assessing quality.

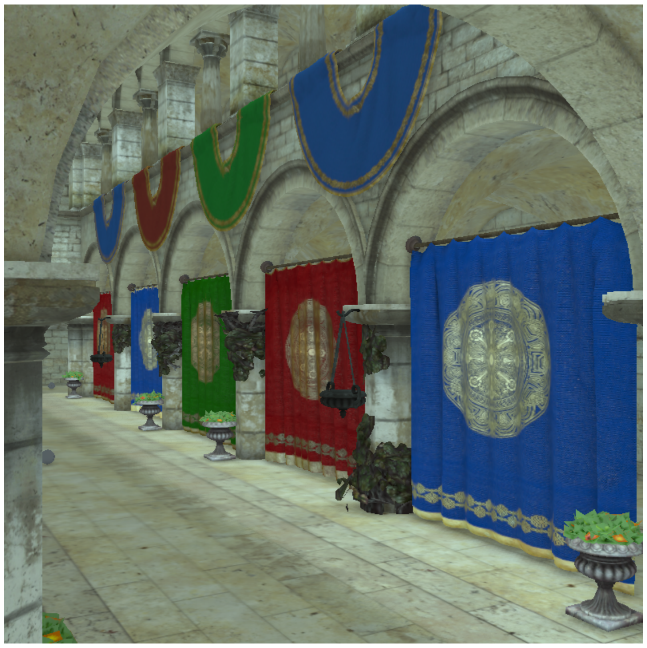
Quality is a term in the evaluation and it is best isolated by using the same reference scene for all methods. We intend to do a manual, visual inspection to evaluate quality. Systematic noise and artifacts should be categorized only when appropriate. Comparisons will be inter-method and against a ray-traced reference generated by an existing ray-tracer.



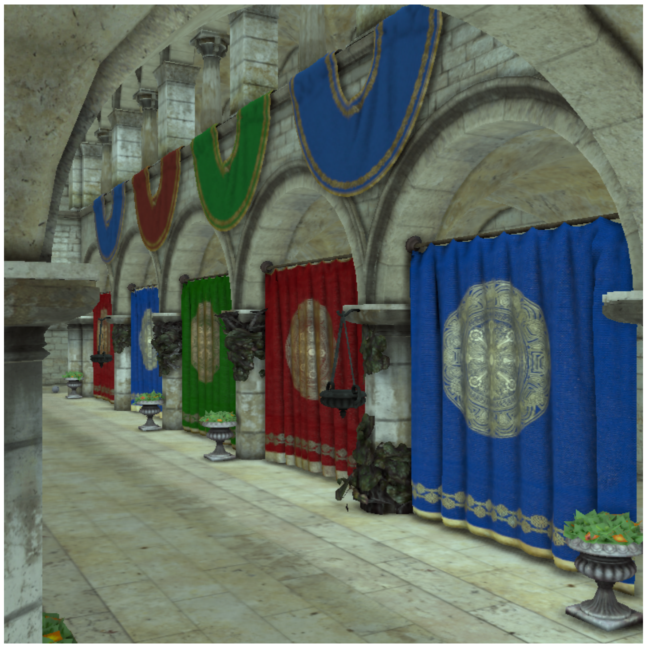
(a) *Unoccluded light*



(b) *Shadow mapping*



(c) *Ambient occlusion*



(d) *Ambient occlusion (amplified)*

Figure 1.2: *Example demonstrating the visual effect of ambient occlusion.* ***(a)*** *Unoccluded light.* ***(b)*** *Large-scale shadows implemented via shadow mapping occluding direct light but not ambient light.* ***(c)*** *Small-scale shadows occluding ambient light [Filion and McNaughton 2008].* ***(d)*** *The same as the previous image but with the ambient occlusion term raised to a power of 4 to amplify the effect.*

The reference scene will also be the base of the performance evaluations. The latter can be determined trivially via compute time and memory costs. Again, comparisons will be inter-method and also against the average industry determined render budgets.

1.4 Overview of the Report

We will first provide some basic background information regarding AO and key concepts in the field of real-time rendering. The latter will not be a substitute for a text book dedicated to the subject. We merely intend to brush up real-time rendering concepts. The reader is assumed to have knowledge and experience in this field prior to reading the report.

With the basics laid down, we continue to present and analyze a large array of SSAO methods. The goal is to provide both an overview of the available methods but also explain the underlying details as they appear. We have chosen to present each method chronologically (with few exceptions) so the reader gets insight in how an early method may influence a later one.

Now that the methods have been presented, we will design a way to compare them. This includes picking which methods we want to study. We will present common observations regarding all methods and delve into advantages and disadvantages of each. We also categorize the methods and provide a generic pipeline for SSAO computations.

A select set of representative methods are then implemented. We will provide details about how the governing SSAO models are translated into working code.

The implemented methods provide a basis for a practical comparison that will ultimately lead to one or more superior method(s) standing out. We will tweak their parameters and see how close we can get to the ray-traced reference.

This is followed by a discussion of mainly what we didn't cover and how a future study could deepen our understanding of SSAO methods.

Lastly, we conclude the report by summarizing the work and stating what we have achieved. The impatient reader may skip the main body for now and jump directly to the conclusion to see our recommendations.

2 Background

This section will first briefly recap key rendering concepts used in the subsequent discussion. Then the theory behind ambient occlusion is presented in order to prepare the reader to the following sections. This is followed by an explanation of

2.1 Rendering Theory

The interaction between light and surfaces is complex and therefore best approximated for real-time purposes. A historically popular way to do so is to split light into *diffuse*, *specular*, and *ambient* components [Cook and Torrance 1982]. For this discussion it suffices to label the former two (diffuse and specular) as *direct light*—light that comes directly from a source. Ambient light is all the *indirect light*—everything that is not direct light. E.g. light that has undergone any number of reflections or refractions.

It is computationally difficult to calculate physically-correct indirect light for a given surface point and direction. Each surface point $\mathbf{p}$ and direction ω pair has the following associated quantities:

- L_o, the outgoing radiance at $\mathbf{p}$ for a given direction ω.
- L_i, the incomming radiance at $\mathbf{p}$ for a given direction ω.
- f, the ratio of incident irradiance along an incomming direction ω_i to outgoing radiance along ω at $\mathbf{p}$. Known as the bidirectional reflectance distribution function (BRDF).

Typically, L_o is what we are after, since it is the radiance that meets the eye from surface point $\mathbf{p}$. Both L_i and f are complex to describe. The interaction between all three quantities is even more complex [Kajiya 1986]:

$$L_o = \int f L_i \cdot (n \cdot \omega) d\omega \tag{2.1}$$

where we have left out some additional quantities not relevant for this discussion. The integral is over all directions ω in the *unoccluded* hemisphere defined by $\mathbf{p}$ and n. The dot product $n \cdot \omega$ in the integral modulates incomming radiance to account for the fact that light at shallow angles project to a larger area. See Figure 2.1 for a 2D view of the situation. All quantities depend on $\mathbf{p}$ and ω but it is common to skip those terms in the equations for brevity.

2.2 Ambient Occlusion

Ambient light is the aggregated indirect light for *all* incomming directions. That is, we have to evaluate the integral in Equation 2.1 to determine the ambient light. This makes ambient light an intractable quantity without further assumptions. So to remedy the sitation, a fair slew of assumptions are made [Cook and Torrance 1982]:

- Ambient light is uniformly incident. I.e. incomming radiance is the same from all directions (L_i is constant).
- Ambient light is independent of viewing angle. I.e. the ratio of reflected light is the same from all directions (f is constant; the surface is *Lambertian*).

Together, these approximations give us a simpler model of ambient light

$$L_{oa} = f L_{ia} \int (n \cdot \omega) d\omega \tag{2.2}$$

where L_{oa} is the outgoing radiance of the ambient light; f is the constant reflectance ratio; L_{ia} is the constant incomming radiance of the ambient light. This approximation of L_{oa} is possible due to the assumptions listed above. The key thing to notice is that both

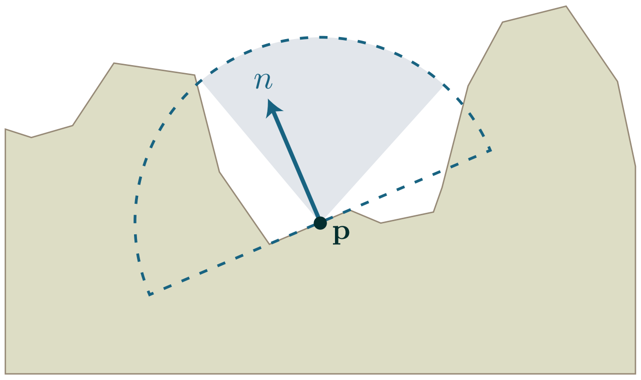


Figure 2.1: *The hemisphere at a surface point as seen from the side. The unoccluded part is shaded.*

f and L_{ia} are now constant since all directionality is defeated by the assumptions.

The integral in Equation 2.2 is the last complex term that remains. It is the visibility of the hemisphere at $\mathbf{p}$. Let us first denote the integral as

$$A(\mathbf{p}) = \frac{1}{\pi} \int (n \cdot \omega) d\omega \tag{2.3}$$

where $\mathbf{p}$ is the point being illuminated; n is the surface normal at $\mathbf{p}$; ω is a direction in the integral. The integral is over the *unoccluded* parts of the hemisphere defined by $\mathbf{p}$ and n. The dot product $n \cdot \omega$ changes meaning in this context. It modulates visibility to account for the fact that occluders at shallow angles project to a larger area.

A accounts for the occlusion of ambient light. A classical simplification is to use $A = 1$ to avoid computing the integral. With A constant, ambient light can reach any surface point $\mathbf{p}$ from any angle as if $\mathbf{p}$ was being lit in isolation. Mathematically speaking, L_{oa} would be constant and the result is dully lit flat surfaces. Modern implementations use the original definition from Equation 2.3. The latter limits incoming ambient light L_{ia} to directions that are visible from $\mathbf{p}$. In other words, Equation 2.3 takes the surroundings into account. This greatly enhances the resulting images as highlighted in Figure 1.2.

Figure 2.1 shows an example for a surface point $\mathbf{p}$ in a crevice where A is roughly 0.45. In effect, the outgoing ambient radiance L_{oa} will consist of 45 % of the reflected incomming ambient radiance $f L_{ia}$. This goes hand in hand with the intuition that the crevice should be dark. In constrast, a point at either of the two peaks in Figure 2.1 will have $A = 1.0$ as the associated hemisphere is unoccluded.

2.2.1 Defining the Hemisphere

It is easy to conceptualize the unoccluded hemisphere but we need a stronger mathematical definition for it in order to evaluate the integral. Let the function $V(\mathbf{p}, \omega)$ be the *visibility* of a ray from $\mathbf{p}$ in direction ω. V is defined as

$$V(\mathbf{p}, \omega) = \begin{cases} 0 & \text{Ray from } \mathbf{p} \text{ in direction } \omega \text{hits anything} \\ 1 & \textit{Otherwise} \end{cases}$$

Now we can extend A to evaluate over the entire hemisphere. Let the new integral be

$$AO(\mathbf{p}) = \frac{1}{\pi} \int_{\Omega} V(\mathbf{p}, \omega)(n \cdot \omega) d\omega \tag{2.4}$$

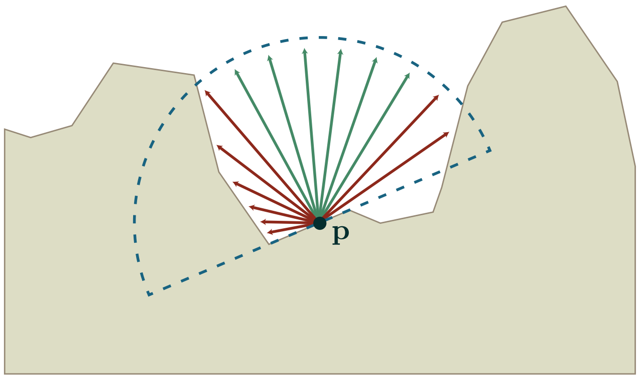


Figure 2.2: *Ray-tracing to solve the integral. Green rays are visible ($V = 1$) while red rays are occluded ($V = 0$).*

where Ω is the hemisphere defined by $\mathbf{p}$ and ω. Equation 2.4 is the definition that is commonly found in the litterature. The term ambient occlusion (AO) [Landis 2002] was coined long after the basic formulation of Equation 2.4 and yet it is the most popular today. The integral can be determined by Monte Carlo integration with a ray-tracer by proping a discrete number of directions [Landis 2002]

$$AO \approx \frac{1}{N}\sum_{n=1}^{N} V \cdot (n \cdot \omega_n)$$

where N is the number of probe rays; ω_n is a direction chosen uniformly at random for each n. See Figure 2.2 for a sideview of the situation.

2.3 Ambient Obscurance

Recall the scene from Figure 1.1d. Were it to be rendered with the formulation of AO as given in Equation 2.4 the result would be that of Figure 2.3. Notice that the result is very dark and almost impossible to recognize. The problem is that very few rays actually manage to escape the scene since there is only a small opening in the top of the scene. I.e. V will be 0 for the most part. For a fully enclosed scene, the resulting image would be entirely black. This is unwanted and we need a more practical definition of AO that gives visually pleasing results.

Ambient obscurance [Zhukov et al. 1998] is the term that we are really after. It replaces the visibility function V with an attenuation (falloff) function $\rho(d)$ based on the distance d. The function ρ is empirically chosen but should have the general properties:

- ρ is a monotonically increasing function of d.
- There exists and upper bound d_{max} such that $\rho(d) = 1$ for $d > d_{max}$.

The idea with ρ is to make distant occluders have little or no influence. This generally brightens up the resulting images and produces visually pleasing results. It should be noted that ρ is not physically-based and is entirely chosen based on aesthetics. The definition for ambient obscurance is similar to what we already have seen

$$AO^*(\mathbf{p}) = \frac{1}{\pi}\int_\Omega \rho(|\mathbf{p}\omega - \mathbf{p}|)(n \cdot \omega)d\omega \qquad (2.5)$$

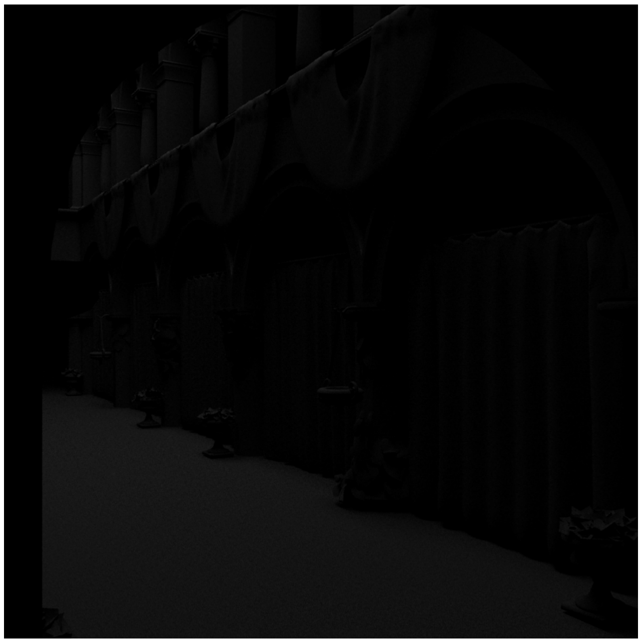


Figure 2.3: *Ambient occlusion as given by Equation 2.4. Rendered with Mental Ray.*

where $|\mathbf{p}\omega - \mathbf{p}|$ is the distance from $\mathbf{p}$ to the first surface hit by a prope ray. See Figure 1.1d for a ray-traced example with $d_{max} = 10.0$ and ρ being a simple linear falloff function (capped at d_{max}, of course).

Note that V is really just the special case of ρ where $d_{max} = \infty$.

2.4 Terminology

The two terms ambient occlusion (using V) and ambient obscurance (using ρ) are used interchangeably in the litterature. Furthermore, both terms have the unfortunate property that they abbreviate to the same two letters, AO. Anyhow, ambient occlusion in the classical sense (Equation 2.4) has little practical merit because of its difficulty dealing with (near) enclosed scenes. Effectively, many so-called ambient occlusion implementations are really based on the ambient obscurance idea (Equation 2.5). The term ambient occlusion just seems to be more popular for the resulting visual effect. In the following sections, we will continue to use the term ambient occlusion even though the more appropriate term is ambient obscurance. So for the rest of this report the following applies

$$AO = AO^*$$

2.4.1 Inversion

As you might have noted, the AO function increases with *visibility* ($AO = 1$ implies full visibility) even though the function is supposed to denote ambient *occlusion*. Consequently, some authors like to invert the definition like so

$$AO = 1 - \int \ldots$$

It is more or less a matter of taste. In this report, we use the definition given by each method. The choice should be clear from the context.

2.5 Real-Time Rendering

The theory discussed in the above sections applies to rendering in general. Real-time rendering is a specialized sub-field that focuses on delivering many frames per second which in turn enables us to have interactive virtual worlds. This section will *not* describe how to apply AO theory in a real-time context. That is left for a later section. Instead, this section will present some common terms that will make the basis of the later discussion.

2.5.1 Transformations, Spaces, and Coordinates

There are many resources on this subject so we will not go into details here. We use the terminology from the OpenGL API. Consult the OpenGL API reference for more details.

There are a key number of *coordinates* (*spaces*) in which a virtual world can be represented (in 3D). This will become important for the later discussion as many of the SSAO methods will often transform their input between spaces. The following list is meant as a quick reference and *not* a detailed explanation. It can't substitute a thourough text on the subect.

World coordinates (WC) Where world is defined with reference to an absolute origin. A so-called global space.

- The x-coordinate is right.
- The y-coordinate is up.
- The z-coordinate is into the screen.
- Transformation is represented by the *model*-matrix.

Eye coordinates (EC) Where the origin has been shifted to $(0, 0, 0)$ and the world is rotated to simulate a different viewing angle; perhaps a camera.

- The z-coordinate is now out of the screen. I.e., the eye is looking down along the negative z-axis.
- Transformation is represented by the *view*-matrix.

Clip coordinates (CC) where values have been projected and clipped.

- Transformation is represented by the *projection*-matrix and a subsequent clipping by comparing the w-coordinate.

Normalized device coordinates (NDC) Where values have been divided by the homogeneous w-coordinate.

- $(x, y, z) \in [-1; 1]^3$.
- Transformation is the division by the w-coordinate.

Screen coordinates (SC) Where values have been translated into the window range.

- $x \in [0, \text{width}]$.
- $y \in [0, \text{height}]$.
- $z \in [0, 1]$ but *non-linearly*.
- Transformation is simple translation and scaling.

Transformations are meant to be applied in the order given above in order to go from the world coordinates to screen coordinates. It is sometimes convenient to use a space somewhat in between NDC and SC:

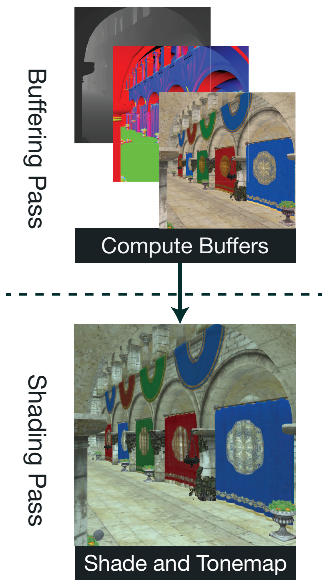


Figure 2.4: *Basic deferred shading pipeline.*

Texture coordinates (TC) Where values have been translated into the texture look-up range.

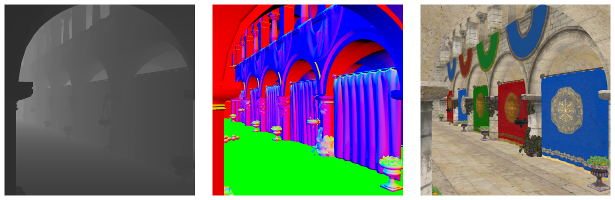
- $(x, y, z) \in [0; 1]^3$. Again, the z-coordinate is stored *non-linearly*.
- Transformation is simple translation and scaling.

2.5.2 Deferred Rendering

Many modern real-time applications use the concept of deferred rendering to get from polygons to pixels [Filion and McNaughton 2008; Smedberg and Weight 2009; Mittring 2012; Kaplanyan 2010]. The classical motivation is to conquer a technical nuisance known as overdraw. I won't go into details here as this is not within the scope of this report. The basic remedy is to defer the actual shading of geometry (as governed by Equation 2.1) and instead output intermediate computations to buffers. The intermediate information can be surface positions, normals, color, BRDF-parameters, or anything the might suit the particular implementation of Equation 2.1. A basic pipeline for deferred shading can be seen in Figure 2.4. The figure is simplified for the discussion. Normally, shading and tonemapping would also be separate. The key thing to note here is that shading is decoupled from the rasterization of polygons. In other words, the shading occurs in screen-space. The key benefit is that Equation 2.1 is applied (in some form or another) using the buffers as the only input.

We will now briefly present some commonly used buffers.

Positions and Depths One popular choice is to buffer the surface depths as seen in EC. The result can be seen in Figure 2.5a where the colors black and white denote positions near and far from the viewer, respectively. Given the scene depth, it is trivial to reconstruct the surface positions in both eye and world coordinates [Mittring 2007a; Kasyan et al. 2011]. This saves a lot of bandwidth as only a single depth value has to be stored instead of 3 position coordinates. Though you would generally need a high-precision depth buffer to reconstruct the positions without artifacts.



(a) *Depths* **(b)** *Normals* **(c)** *Diffuse colors*

Figure 2.5: *Buffers commonly used in deferred rendering.*

Normals Normals can also be explicitly stored as seen in Figure 2.5b where each axis is represented by a color. However, it is also possible to reconstruct the normals from position information [McGuire et al. 2012; Bukowski et al. 2012]. This will cut the entire normal buffer from the memory budget but it does lead to some artifacts along depth discontinuities [Bukowski et al. 2012].

Colors Surface material properties such as diffuse color are often buffered as they can't really be reconstructed. See Figure 2.5c for a diffuse color buffer.

Miscellaneous BRDF-properties and the like are usually stored within any left-over space from the above-mentioned buffers.

2.5.3 Shaders

Both the buffering and shading pass of Figure 2.4 are typically implemented in *shaders*. A shader is a program that runs on the GPU and evaluates on different input data, depending on its type. Originally, shaders worked in a directly connected pipeline. A *vertex shader* took raw polygon data and produced the position, normal, color, etc. for the next stage. A *fragment shader* immediately received this input and combined them to a pixel value. Along the line came the ability to read buffers in shaders. This feature was originally intended for *texturing*. Deferred shading bent this purpose a bit by introducing buffers. A *vertex shader* takes raw polygon data and processes it into the aforementioned buffers. A *fragment shader* reads from the buffers and perform shading calculations on them to output final colors.

Shaders implement the Single Program Multiple Data (SPMD) principle which is why GPUs can execute them in parallel and achieve great speeds. A deeper discussion quickly gets involved. We advice the reader to consult a text dedicated to shaders for further reference. The key thing to remember is that shaders can be exploited beyond their initial purpose to perform computations for every pixel several times per second. In the following, we will assume that the reader has experience with the use of shaders in a deferred pipeline.

The theory of AO combined with concepts from real-time rendering now enables us to uncover SSAO.

3 Analysis

This section will first present previous work concerning real-time AO and SSAO in particular. We consider each method chronologically and explain the underlying details as they come. The methods are also outlined in Table 1. We encourage the reader to use this table to get a historical overview.

3.1 Ray-Tracing and Rasterization

Historically, AO has been a feature reserved for ray-tracers because the definition (Equation 2.5) lends itself easily to such an approach [Landis 2002]. However, ray-tracers are not yet used for interactive real-time rendering on a large commercial basis. The key problem is that a ray-tracing is not compatible with the rasterization approach GPUs utilize. With ray-tracing, the entire scene representation must be in available at all times for the intersection tests to work. With rasterization, only a few polygons are considered at a time.

3.2 Previous Work

Despite the abovementioned difficulties, authors have found ways to bring AO into the real-time domain. We will uncover them below. Note that the focus here is on the so-called screen-space approaches but other methods may be presented if relevant.

3.2.1 Disk Proxies

The algorithm requires as input every polygon mesh of the scene decompositioned into disk elements. The idea is to then see how these disks occlude geometry in the scene [Pharr and Fernando 2005]. It is the first proposed alternative to ray-tracing for implementing AO. The method was later improved by smoothing the result [Nguyen 2007]. As we will soon discover, this is a trait that is shared with many screen-space methods. That being said, the smoothing is not just implemented a post-process filter but as an algorithmic change that linearly blends two stages instead of a hard cut as in the originally proposed method.

The method does not classify itself as a screen-space approach and we won't go into further detail with it. It is mentioned here simply because it is the first (near) real-time AO method.

3.2.2 Unsharp Masking

We assume that the reader is familiar with the principle of applying filter kernels to images. E.g. using a guassian blur kernel to blur an image. This principle has a surprising application with regards to AO. The idea is to apply a so-called unsharpening filter kernel to the depth buffer [Luft et al. 2006]. The result is not a direct AO computation but a post-process effect that mimics it. I.e. there is no real physical motivation behind the approach but it produces visually similar results. However, the method is not in the scope of this project and we won't go into details with it. One thing that is worth noting is the use of the depth buffer. This becomes a recurring element of the future methods.

3.2.3 First Screen-Space Approaches

Depth Buffer as a Scene Approximation As noted with unsharp masking, the depth buffer has some uses beyond its original purpose. It can also serve as a coarse representation of the scene [Mittring 2007a; Shanmugam and Arikan 2007]. See Figure 3.1 for a sideview of the situation. Here, the original scene is denoted by the dashed line and the shaded object is the depth buffer representation of it. Figure 3.1 could be a scanline through the depth buffer of Figure 2.5a. This implies that we actually *have* a representation of the entire scene available with rasterization. However, the nature of the depth buffer renders it only a rough approximiation. We will go deeper into this later.

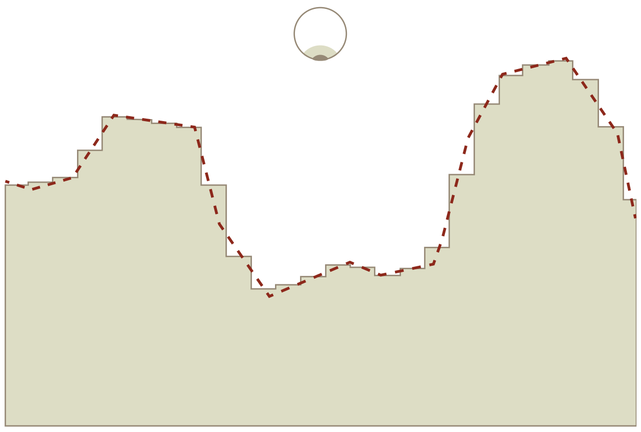


Figure 3.1: *Depth buffer as a scene approximation.*

Extended Pipeline Another pass must be included in the shader pipeline for the AO computations (See Figure 3.2a). Notice how the AO pass uses the depth buffer as its only input for now. It must be mentioned that it *is* possible to compute AO in the shading pass since it also gets the depth buffer as input. However, it is generally a better idea to calculate AO in its own pass. The reason being that it makes it straight-forward to compute the AO pass at a lower resolution. We will dicuss some approaches that involves this later on. It is also easier to isolate performance characteristics when the AO has a dedicated pass. This is very relevant in the context of this project.

The shader in the AO pass works on buffer input just like the shader in the shading pass. This makes the AO shader a fragment shader which executes for every pixel on the screen. It is the reason why these kind of AO algorithms categorized as working in screen-space.

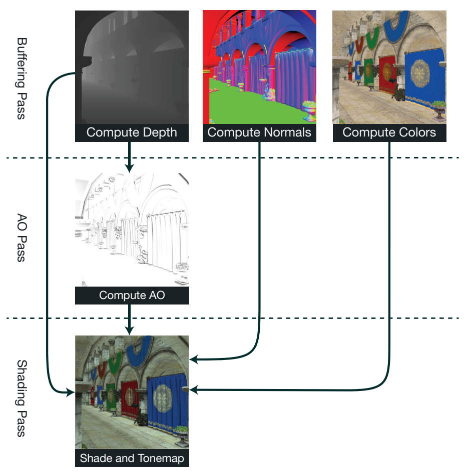
Screen-Space Integration Now that the scene is present through the depth buffer, we have information available to the shader beyond the pixel currently being shaded. This allows for some interesting approaches to solving the integral of Equation 2.5.

One way is to approximate AO as the ratio of visible to occluded nearby sample points [Mittring 2007a; Mittring 2007b; Dachsbacher and Kautz 2009]. The integral then becomes

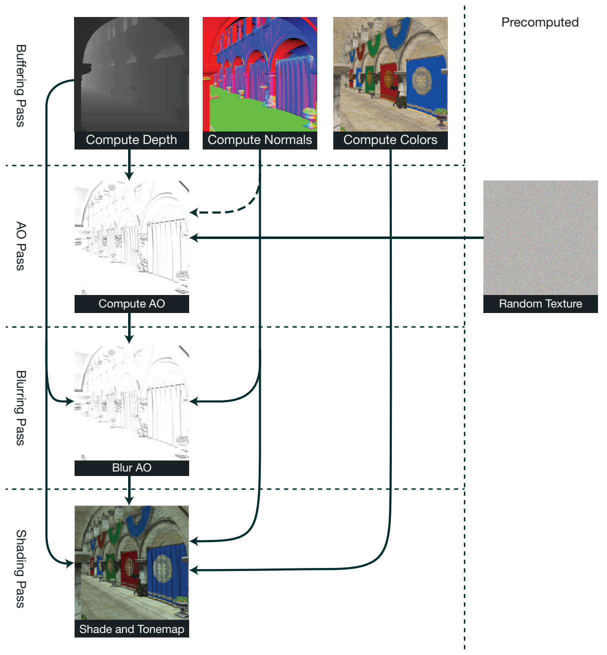
$$AO \approx \frac{1}{N}\sum_{n=1}^{N} V'(\mathbf{s}_n) \tag{3.1}$$

where N is the number of sample points; $\mathbf{s}_n$ is the n'th sample point; V' is 1 if s_n is visible and 0 otherwise. The s_n sample points are scattered in a sphere around $\mathbf{p}$ with a radius of r. See Figure 3.3 for a sideview of the situation.

The current position $\mathbf{p}$ must first be reconstructed. The chosen approach is to reconstruct $\mathbf{p}$ in screen coordinates which is computationally inexpensive. $\mathbf{p}_{xy}$ are the current pixel coordinates and $\mathbf{p}_z$ can be read directly from the depth buffer. With $\mathbf{p}$ determined, it should be trivial to choose sample points in the surrounding sphere. However, as the authors of the method note, this can be a source of artifacts. We will come back to this in a moment.



(a) *Basic overview. Compare it to the deferred shading pipeline of Figure 2.4.*



(b) *Extended overview. The dashed connection from the normals buffer ot the AO computation indicates that it is optional for many methods.*

Figure 3.2: *The ambient occlusion pipeline.*

In Figure 3.3 it is intuitive to see that samples below the surface are occluded. The intuition follows closely into the actual definition of V'. A sample position $\mathbf{s}$ is in screen coordinates just like $\mathbf{p}$ so it has an associated $\mathbf{s}_z$ coordinate. The surface depth above $\mathbf{s}$ can be read from the depth buffer as $\mathbf{s}_d = depth(\mathbf{s}_{xy})$. All that is left is to do a simple comparison

$$V'(\mathbf{s}) = \begin{cases} 1 & \mathbf{s}_d > \mathbf{s}_z \\ 0 & Otherwise \end{cases}$$

and the visibility is determined.

Also note that method (Equation 3.1) samples in a sphere around $\mathbf{p}$ and not a hemisphere as in the original formulation (Equation 2.5). I.e. we must expect half of the samples to be occluded; even for a flat surface.

All is good in theory but the sample distribution issue remains. If the sample positions are fixed for each $\mathbf{p}$ the resulting AO will have banding artifacts. Instead, sample points are distributed uniformly at random which removes the banding artifacts but introduces high-frequency noise. The authors circumvents the noise by applying a geometry-aware blur over AO calculations. Unfortunately, due to the nature of GPUs, the blur must be done a shader pass of its own. Another hurdle is that shader programs can't generate random variables. Instead, a buffer of pre-computed random variables are supplied to the AO shader. These extra steps requires an extended ambient occlusion pipeline (See Figure 3.2b).

The authors also note a fair slew of other shortcommings that we will address in the later comparison.

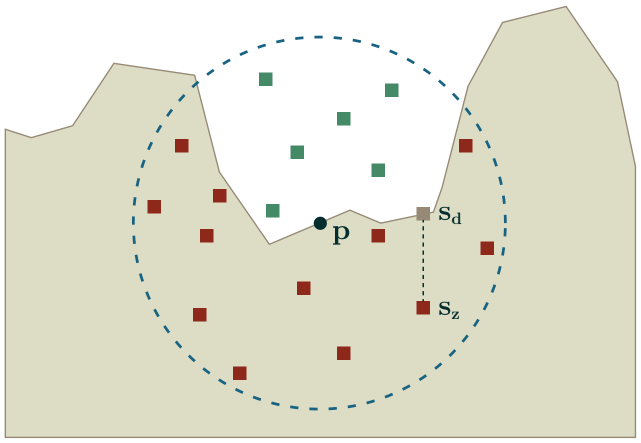


Figure 3.3: *Approximation of AO [Mittring 2007a]. The green and red rectangles are the visible and occluded samples, respectively. One sample is also shown projected onto the surface.*

Variations There exists a variation [Filion and McNaughton 2008] to the abovementioned method whose differences will be outlined below.

The sample positions $\mathbf{s}$ are found in world coordinates and then projected into screen coordinates for the depth comparison. This

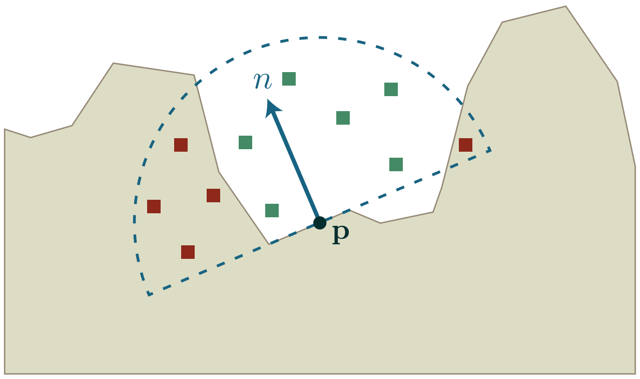


Figure 3.4: *Approximation of AO [Filion and McNaughton 2008]. The green and red rectangles are the visible and occluded samples, respectively.*

has some implications that we will later uncover. The authors also re-introduce an attenuation function, ρ, whereas a simple comparison function, V', was used before. This alleviates over-occlusion due to foreground objects with high $\mathbf{s}_d$ values. We will also treat this in the comparison later. Self-occlusion is also treated differently. As seen in Figure 3.3, samples are found in a sphere around $\mathbf{p}$ and half the samples are treated as occluding per default. The alternative approach is to first find samples in the sphere but then flip them back into the hemisphere according to the surface normal, n. See Figure 3.4 for the differences. Effectively, only half as many samples are needed with this approach. There are other subtle differences but the above are the most prominent.

Spherical Proxies Around the same time as [Mittring 2007a] came an independently developed method which also works in screen-space and shares many of the same traits. It approximates AO by averaging the occlusion of nearby surfaces represented by spherical proxies [Shanmugam and Arikan 2007]. The method reconstructs $\mathbf{p}$ in screen coordinates and distributes N samples $\mathbf{s}_n$ in a *disk* around $\mathbf{p}$ in screen coordinates as well—similarly to [Mittring 2007a]. This is were the similarities stop, as each sample $\mathbf{s}$ is now projected to the surface by setting $\mathbf{s}_z = \mathbf{s}_d$ (See Figure 3.3 for reference) followed by a transformation into world coordinates. A sphere is positioned at each world-space $\mathbf{s}$, which is intended to approximate the nearby surfaces. The radius of the sphere is a function of $\mathbf{s}_d$. AO is now approximated as the sum of how much each sample sphere occludes the hemisphere at $\mathbf{p}$. Note that this is not just a comparison function like V' as found in [Mittring 2007a]. The hemisphere is defined by the position $\mathbf{p}$ and normal n in world coordinates. The latter can be read from the normal buffer as seen in Figure 3.2b. The paper goes on to approximate AO for distant surfaces as well. Since this is not accomplished within screen-space, we won't go into further details with it.

The method described above is very involved and approaches the AO approximation very differently compared to other proposals. It is an interesting albeit overly-complex technique that hasn't got much mention since its inception. We have therefore chosen not to pursue this method any further. It is mentioned here because it was one of the first screen-space techniques and because of its similarities with [Mittring 2007a].

3.2.4 A Horizon-Based Approach

Another way to approximate AO is by determining the angle of the visible horizon, h, on the hemisphere around $\mathbf{p}$ [Bavoil et al. 2008a; Bavoil et al. 2008b; Bavoil and Sainz 2008; Dachsbacher and Kautz 2009]. That is, we want to find the angle of the shaded area in Figure 2.1. Under the assumption that the heightfield is continuous, all rays traced beyond this angle should be occluded (See Figure 2.2). We will discuss the implications of this assumption later. Further assume that we can find the horizon angle $h(\theta)$ for a given angle around the view vector, θ. The sideview in the figures represent the situation for *one* such angle, θ. We want to find the average h for *all* θ around the view vector. The authors of the method present the following equation to do so

$$AO = 1 - \frac{1}{2\pi}\int_{\theta=-\pi}^{\pi} AO_{horizon}(\theta)d\theta \tag{3.2}$$

where the integral goes through all angles θ around the view vector. Notice the inversion of the AO definition.

Now we need a way to evaluate $AO_{horizon}(\theta)$ for a given θ. In Equation 2.5 we should trace rays in every direction from the tangent to the normal weighed by ρ. However, we already know that all rays below $h(\theta)$ are occluded so there is no need to actually trace them. Symmetrically, all rays above $h(\theta)$ are visible, and don't need to be traced either. Instead, we can find the AO contribution as the integral between the tangent angle $t(\theta)$ and $h(\theta)$. The tangent angle $t(\theta)$ can be easily derived from n and θ. The authors put it as follows

$$AO_{horizon} = \int_{\alpha=t(\theta)}^{h(\theta)} \rho(d)\cdot\cos(\alpha)d\alpha \tag{3.3}$$

where $\cos(\alpha)$ is the dot product from Equation 2.5. $\rho(d)$ is a little more involved and will be explained later. The authors further evaluate Equation 3.3 to

$$AO_{horizon} = \rho(d)(\sin(h(\theta)) - \sin(t(\theta)))$$

We can now insert Equation 3.3 into 3.2. For computational purposes, it is also useful to use Monte Carlo integration to solve the integral

$$AO = 1 - \frac{1}{N}\sum_{n=1}^{N} \rho(d_n)(\sin(h(\theta_n)) - \sin(t(\theta_n))) \tag{3.4}$$

where N is the number of samples.

Screen-Space Evaluation The key to implement Equation 3.4 is to ray-march the depth buffer in screen-space to find $h(\theta)$ as proposed by the same authors [Bavoil et al. 2008a]. The idea is illustrated in Figure 3.5. The a ray is created from $\mathbf{p}$ in direction θ in screen coordinates. The ray traverses the depth buffer and samples at distinct intervals. Each sample is projected onto the scene surface by setting $\mathbf{s}_z = depth(\mathbf{s}_{xy})$. The sample's AO contribution is computed from the occluded angle $\text{acos}(t^*, \text{normalize}(\mathbf{s}-\mathbf{p}))$ and weighed by $\rho(|\mathbf{s}-\mathbf{p}|)$. This is how ρ is integrated. t^* is initially the tangent vector but is reassigned to each successive samples' $\mathbf{s}-\mathbf{p}$ vector. This allows each sample to contribute and avoids overocclusion. A sample that is below the currently defined t^* is rejected since it is not visible from $\mathbf{p}$. See Figure 3.5 for reference.

Readers familiar with real-time rendering might note the strong parallel to parallax occlusion mapping regarding the screen-space ray-marching.

Furthermore, the authors suggests to jitter the step size of the ray-marching and choose directions randomly to avoid systematic noise. This is similar to the artifacts encountered with [Mittring 2007a].

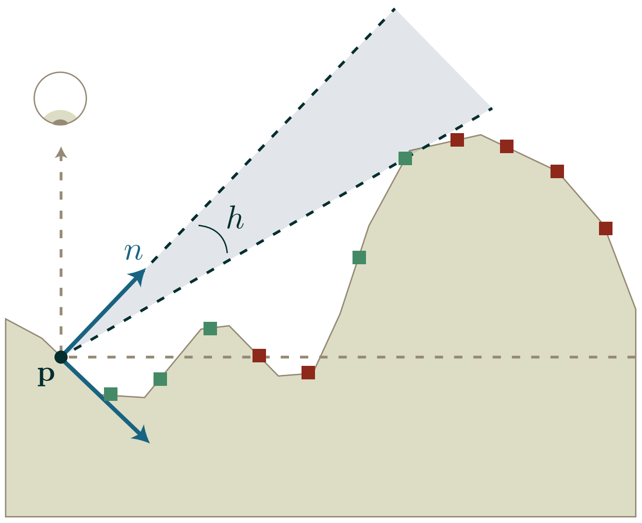


Figure 3.5: *Approximation of AO [Bavoil et al. 2008a]. The green and red rectangles are the used and rejected samples, respectively. The rightmost used sample determines the horizon angle, h. This sideview shows the screen-space ray-marching for one angle θ around the view vector (arrow pointing towards the eye). Notice that all samples are projected onto the scene surface by means of the depth buffer.*

Variations It is also suggested to trace the rays in eye coordinates and then project them into screen coordinates every step to sample the buffers [Dimitrov et al. 2008; Sainz 2008]. This is known as Horizon-Split AO (HSAO) whereas the method described in [Bavoil et al. 2008a] is known as Horizon-Based AO (HBAO). The added projections imply extra computations which makes HBAO superior in terms of performance. The papers backing both HSAO and HBAO are by the same authors but the latter paper dates the former by half a year. We think it is safe to conclude that HBAO is an improvement upon HSAO. Consequently, we will not go further into detail with HSAO.

3.2.5 Directional Occlusion

In between all the SSAO methods came a slightly different take on things known as Screen-Space Directional Occlusion (SSDO) [Ritschel et al. 2009; Dachsbacher and Kautz 2009]. SSDO is not another take on how to calculate ambient occlusion in screen-space. It is an extension of the core SSAO idea.

SSDO does not only compute AO but also the incoming radiance at the surface point belonging to $\mathbf{p}$. This allows for colored and directed shadows in the result. Furthermore, SSDO has an optional second pass that approximates one diffuse indirect bounce of light. This is done by sampling nearby values of direct radiance gathered in the first pass. Color bleeding effects as known from ray-traced global illumination solutions are thereby possible.

We include SSDO here because it is an interesting extension that can be built on top of most SSAO method. However, as SSAO is the focus of this report, we will not go into great detail about SSDO integration.

One thing worth noting with respect to SSAO methods is the treatment of screen-space related artifacts. The authors address the issue of missing scene information due to the fact that the depth buffer only contains information about the frontmost objects. They suggest to use *depth peeling*. Simply put, this involves a layered depth buffer that records the depths of multiple objects per pixel. With this additional information, the scene is better represented and some artifacts dissapear. Alternatively, the authors suggest to use the collective depth information from multiple views. We will revisit this topic later.

3.2.6 Multi-Layer and Multi-Resolution

Another SSAO extension came around the same time. [Bavoil and Sainz 2009] presents two enhancements to a generic SSAO algorithm. I.e. the authors do not present a new SSAO method altogether but merely suggest two improvements to existing and future methods.

The first improvement is on quality. The authors suggest to use depth peeling as already presented in [Ritschel et al. 2009]. Additionally, [Bavoil and Sainz 2009] also suggests to use an enlarged field (a guard band) of view when calculating AO in order to fix artifacts near scene edges. The problem is that there is no scene information outside the buffers range. Consequently, the SSAO algorithms fail collectively near borders. The remedy is to provide the borderline scene information by rendering buffers that are larger than the final image.

The second improvement focuses on performance without sacrificing quality. The idea is to split the AO calculation into two passes: One at half-resolution and one at full-resolution. The former pass is quickly rendered because it operates on $\frac{1}{4}$ of the pixels. The latter pass then either simply up-scales the half-resolution results or recalculates the AO if it is found that more precision is needed. The details are rather involved and we won't present them here.

Variations As shown in the pipeline of Figure 3.2b, it seems obligatory to blur the AO computation to remove the noise introduced by the random sampling patterns. However, using multiple resolutions the blurring step can be skipped [Hoang and Low 2010]. There are 2 phases to the algorithm. AO is first calculated at increasingly lower resolutions using symmetric lower-resolution input (depth and normal buffers). Then the different AO values are combined into a final full-resolution render. The idea seems similar to the dual-resolution method developed in [Bavoil and Sainz 2009] but there are some key differences. Firstly, the authors suggest to down-sample the input depths and normals to use for calculating AO. This roughens the already coarse approximation of the scene. Secondly, they do not propose an intelligent combination scheme to combine the different AO values but merely suggest to blend them together using an edge-aware bilateral filter. Despite the rough approximations the authors get good results. The multi-resolution input and output buffers are easily represented on modern hardware through mipmaps.

The same authors revisit their initial implementation two years later with several improvements [Hoang and Low 2012]. This time around, each pass is explained in detail and with mathematical motivation. Also, the authors present methods to intelligently down-sample both the input buffers and combine the different AO values into one result. As such, the proposed method is no longer only aesthetically motivated but as sound mathematical reasoning behind it. However, the details are still involved and we won't pursue these methods further.

3.2.7 A Temporal Approach

One way to improve the quality of the AO computation is to reuse the information from previous frames [Smedberg and Weight 2009]. The authors present an SSAO algorithm that basically follows Equation 3.1 but with the temporal extension. As such, the temporal extension can be applied to other SSAO methods as well.

A *temporal filter* is presented which improves the result of moving scenes. The motivating observation is that the view (camera) changes gradually from frame to frame as do scene animations in regular cases. The idea is to filter AO of frame $n-1$ with AO of frame n by blending the two terms together *over time*—effectively removing noise artifacts under movement. For this to work, the view and animation state of frame $n-1$ must be available at frame n which incurs a memory overhead. Since $\mathbf{p}$ in screen coordinates is likely to be different between the two frames, it must be calculated for both frame $n-1$ and frame n from $\mathbf{p}$ in world coordinates. The saved view and animation state makes this possible—the authors call it *reprojection*. Now AO can be sampled from frame $n-1$ in frame n.

It should be noted that when there is not movement this extra temporal filter does not provide a quality advantage but only incurs a performance cost. This is because the filter blends between frames *over time*. In a static scene, where AO doesn't change between frames, the filter has no effect. Even in dynamic scenes, streaking artifacts can occur depending on the rate of convergence between old and new AO values.

The authors also explains that the random sample look-ups trash the texture cache if the sample radius is too large. This quickly becomes a performance problem. Clamping the radius to a low value improves the spatial locality of the texture samples in the cache. As we will soon uncover, later algorithmic advances will tackle this problem more elegantly.

Variations An alternative take is to not simply blend AO computations between frames but actually re-use the samples to refine AO over time [Mattausch et al. 2010; Scherzer et al. 2010]. That is, to aggregate the samples of older frames in order to converge to a better result. This has the implication that the number of samples must additionally be stored per pixel. The upside is that the quality will be improved even for static scenes (as opposed to the method proposed in [Smedberg and Weight 2009]). This method was developed independently of [Smedberg and Weight 2009].

The authors go further into how to integrate the reprojection into a deferred pipeline and how to deal with disocclusion. The latter is when background objects become visible (disoccluded) because of foreground movement. The general idea is to see if the relative depth difference of $\mathbf{p}$ in world coordinates between frame $n-1$ and n is above a given thresshold. If so, it is most likely that $\mathbf{p}$ now represents the surface of the background object and AO must be calculated anew.

3.2.8 The First Hybrid

A combination of the methods found in [Bavoil et al. 2008a] and [Shanmugam and Arikan 2007] forms the first proposed hybrid method [Song et al. 2010]. The latter method has over-occlusion issues because it fails to reject invalid occluders in all situations. The idea is to approximate nearby surfaces by sphere proxies (as done in [Shanmugam and Arikan 2007]) but simultaneously calculate the horizon-angle (as done in [Bavoil et al. 2008a]) in order to reject sphere locations $\mathbf{s}$ that are not visible from $\mathbf{p}$ due to an occluding horizon. This way over-occlusion does not occur.

We mention this method because it is the first hybrid method. As previously told, we will not go into further detail with the method of [Shanmugam and Arikan 2007] and its derivatives.

3.2.9 A Volumetric Approach

A new yet seemingly familiar term, Volumetric Obscurance (VO), is presented which is the ratio of unoccluded to occluded volume of a 3D neighborhood around $\mathbf{p}$ (e.g. a sphere) [Loos and Sloan 2010]. They formalize VO as

$$VO\left(\mathbf{p}\right) = \int_X \rho(|\mathbf{x}-\mathbf{p}|)O(\mathbf{x})d\mathbf{x} \qquad (3.5)$$

where X is a 3D volume around $\mathbf{p}$; $\mathbf{x}$ is a point in X; ρ is as before; $O(\mathbf{x})$ is an *occupancy function* which is either 1 if there is matter at $\mathbf{x}$ and 0 otherwise. They use Equation 3.5 in place of 2.5. That is, they propose to set

$$AO = VO$$

for the purpose of calculating AO. As the authors note, the above equation is valid under the assumption that a ray from $\mathbf{p}$ in any direction will only intersect a single surface.

In a sense, the methods of [Mittring 2007a] and [Filion and McNaughton 2008] can be thought of as solving the integral in Equation 3.5 using point samples. Each point sample represents a part of the probed volume X; be it a sphere or a hemisphere. When the number of samples N approaches ∞ then the samples cover all of X and we have an equal formulation. This is under the assumption that $V' = O$, of course.

The authors generalize the use of point samples to using line or area sampling to solve the integral. Hereby, they get a better volume approximation which improves the overall result. The idea of line sampling is outline in Figure 3.6. The integral is approximated as the ratio of visible to occluded line lenghts. The sample positions $\mathbf{s}_n$ themselves are found in a disk around $\mathbf{p}$ in screen coordinates. It is trivial to compute the visible part of the line length. The crossing point from visible to occluded is indicated by a simple depth buffer lookup, $\mathbf{s}_d = depth(\mathbf{s}_{xy})$. The height of the sphere at $\mathbf{s}_{xy}$ can be found with simple trigonometry, $\sqrt{r^2 - (\mathbf{s}_x^2 + \mathbf{s}_y^2)}$, where r is the radius of the sphere. The proposed benefit of line sampling over point sampling is that it deals better with movement. Point samples are either occluded or not (tested with V'), and thus pop-in and pop-out can occur for slight movement. Line samples have a visible to occluded ratio *per sample* which enables them to fade smoothly under movement.

Another novel idea is to combine the AO or VO calculations of a small sphere and a large sphere. This technique captures both low-scale as well as broad-scale details.

Variations There are many ways to choose the sample points in a disk around $\mathbf{p}$ using both randomization and rotation [Ownby et al. 2010]. The authors propose several variants and analyze the differences. One proposal is to use an outward spiral pattern to better get all ranges. However, as previously mentioned most fixed-formation sampling patterns results in banding—the spiral distribution included. In the end, they randomly rotate each sample in the spiral around the view vector.

The authors also propose to use paired sample locations (symmetric around $\mathbf{p}$) to better estimate missing sample information due to the rough scene approximation given by the depth buffer. We will look at this in detail later.

Another suggestion actually used in a large scale production is to use line sampling [Loos and Sloan 2010] together with temporal refinement [Mattausch et al. 2010] proving that the latter can work as an extension of the former [Kaplanyan 2010].

3.2.10 Poisson Sampling Approaches

Samples that are distributed on a surface where the distance between any two samples is over a given threshold are said to be Poisson distributed. Formally, this distribution property can be stated

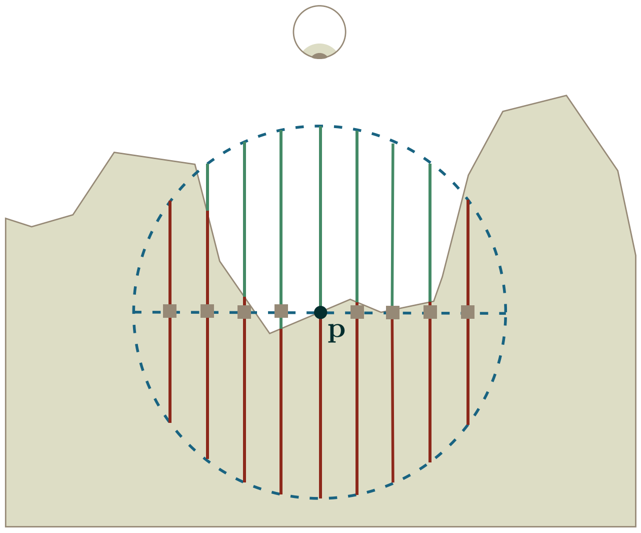


Figure 3.6: *Approximation of AO [Loos and Sloan 2010] The grey rectangles are the N sample points $\mathbf{s}_n$ which define the line segments samples together with the view vector and sphere. The green and red line segments are the visible and occluded parts of each sample. Notice how $\mathbf{p}$ itself contributes as a sample.*

as

$$\forall \mathbf{s}_i \forall \mathbf{s}_j \left(|\mathbf{s}_j - \mathbf{s}_i| > d | i \neq j \right)$$

for sample positions $\mathbf{s}$ and threshold d.

The idea of using poisson distributed samples on a disk was introduced as part of a hybrid of existing methods [Sourimant et al. 2011]. The authors propose to ray-march the depth buffer as a height-field as proposed in [Bavoil et al. 2008a]. The difference is that they are not measuring the angle of the free horizon but instead aim to find a better sample occluder distance by ray-marching. The samples themselves are chosen from a Poisson distribution on a disk surface and projected into the hemisphere defined by $\mathbf{p}$ and n. The AO computation itself is akin to [Mittring 2007a] and [Filion and McNaughton 2008].

There exists another algorithm that shares many of the above steps but differs in the details [Aksoy and Phaneuf 2011]. It starts by projecting Poisson distributed sample positions from disk onto the depth buffer (scene surface) in screen coordinates. If a sample position is behind the plane defined by $\mathbf{p}$ and the normal n then it is occluded. If not, then a function $\rho(|\mathbf{s} - \mathbf{p}|)$ weighs the AO contribution such that distant samples contribute less.

The authors note that they did not have to blur the result with 32 samples per fragment at interactive rates. Though it should be kept in mind that only key characters were shaded using the method. Large portions of the frame (e.g. the background geometry) did not recieve this kind of AO calculation.

Similarly to [Loos and Sloan 2010], the authors suggest to use two hemi-spheres with different radii to capture both fine-scale and broad-scale details.

3.2.11 A Curvature-Based Approach

Similarly to [Luft et al. 2006], the authors follow a novel strategy to approximate AO [Hattori et al. 2010; Hattori et al. 2011]. They approximate AO from the curvature of the geometry in the neighbourhood around $\mathbf{p}$. The language in both papers is hard to follow and the results have either a very distinct, stylish look or suffer from banding artifacts. Nevertheless, the method performs computationally well.

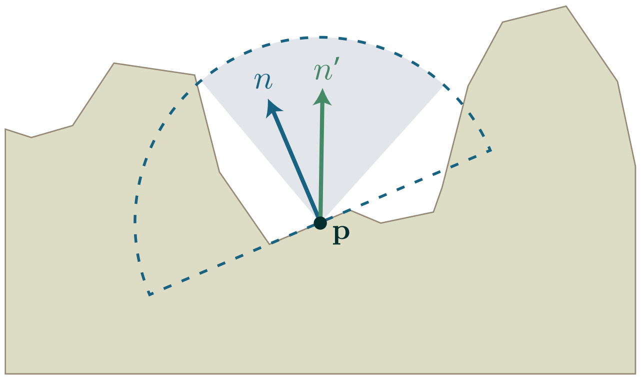


Figure 3.7: *The normal, n, and bent normal, n', at $\mathbf{p}$. Notice that the bent normal points in the average unoccluded direction.*

This method is algoritmically very distinct from the others proposed. Even with the performance in favor of it, the artifacts or the stylish look can't be overseen. We won't pretend to fully comprehend the method and we will not go into further detail with it. It is mentioned here to note that radically different approaches still appear.

3.2.12 Bent Normals

The concept of *bent normals* has been used since the inception of the term ambient occlusion [Landis 2002]. It is the average unoccluded direction (See Figure 3.7). However, it would take a couple of years before it first appeared in combination with screen-space approaches [Kasyan et al. 2011; Donzallaz and Sousa 2011]. The standard AO calculation is done as usual but with the addition of a bent normal that denotes the average unoccluded direction. This bent normal is then saved for each pixel just as AO and used in the following shading computations. E.g. with the Blinn Phong model. Just as in [Ritschel et al. 2009] this allows for directed shadows on a small-scale—what the authors call *contact shadows*. This is an SSAO *extension* so we will not go into too much detail about it. We include it here because it works on top of SSAO and generates visually pleasing results.

An extension to bent normals is *bent cones* which basically is a bent normal with an associated *angle* denoting the size of the cone [Klehm et al. 2011]. The angle is based on the variance between the sample directions. Having the *bent cone* available allows for more complex illumination methods that can evaluate the amount of light that a point recieves through a spherical cap (defined by the cone). The authors suggests to use interleaved sampling in order to reduce the noise found among the bent normals.

3.2.13 Alchemy Ambient Occlusion

Named after the article in which the method first appeared—there is no relation to craft of the same name. The idea is to intelligently choose a falloff function, ρ, so that some terms cancel out in Equation 2.5 [McGuire et al. 2011]. The ρ they choose is

$$\rho(d) = \frac{u \cdot d}{\max(u, d)^2}$$

where d is he sample distance; u is a user-specified parameter to choose the exact shape. It should already be apparent that their choice of ρ implies that they use the inverse definition of AO. I.e. $\rho \to 0$ and not 1 for $d \to \infty$. We will restate inverse AO function to make the discussion easier

$$AO = 1 - \frac{1}{\pi} \int_{\Omega} \rho(d)(n \cdot \omega) d\omega \tag{3.6}$$

Inserting ρ into Equation 3.6 does not yet yield an interesting result

$$AO = 1 - \frac{u}{\pi} \int_{\Omega} \frac{d \cdot (n \cdot \omega)}{\max(u, d)^2} d\omega \tag{3.7}$$

despite the fact that one u can be moved outside the integral. Then the authors suggest to define the vector $v = \omega \cdot d$ and simplify Equation 3.7 to

$$AO = 1 - \frac{u}{\pi} \int_{\Omega} \frac{v \cdot n}{\max(u, d)^2} d\omega \tag{3.8}$$

Furthermore, it is safe to assert that $\max(u, d)^2 = \max(u^2, d^2)$ under the assumption that both $u > 0$ and $d > 0$. The authors also note that $v \cdot v = |v|^2 = d^2$ and simplify Equation 3.8 to

$$AO = 1 - \frac{u}{\pi} \int_{\Omega} \frac{v \cdot n}{\max(u^2, v \cdot v)} d\omega \tag{3.9}$$

All there is left to do is to use Monte Carlo integration to get a computational formula. The authors do that along with some extra simplifications and end up with

$$AO = 1 - \frac{1}{N} \sum_{n=1}^{N} \frac{\max(v_n \cdot n + \beta, 0)}{v_n \cdot v_n + \epsilon} \tag{3.10}$$

where N is the number of samples; v_n is a given sample vector ($\mathbf{s}_n - \mathbf{p}$); β is a parameter that controls the extent of the occlusion; ϵ is a small number to avoid division with 0. Note that the max function in the denominator has been replaced, since the function of u^2 really was to avoid division with 0. Additionally, the dot product in the nominator has been clamped to 0 because sample vectors v_n may unintentionally go below the hemisphere. This is an artifact of the sample generation which is outlined next.

The authors suggest an interesting approach to sample generation; they aim to limit the sampling to occluders only. This contrasts the common strategy of volumetric methods were samples can be either visible or occluded. The authors proposal is therefore more akin to [Bavoil et al. 2008a] where everything below a the horizon angle h is deemed occluding.

The authors choose samples $\mathbf{s}_n$ in a disk around $\mathbf{p}$ in screen coordinates as done in [Loos and Sloan 2010]. They then project the samples onto the scene surface as done in [Bavoil et al. 2008a]. Now they can construct the sample vectors, v_n as $\mathbf{s}_n - \mathbf{p}$. See Figure 3.10 for reference. This is all the information that is needed for Equation 3.10. Note that samples below the hemisphere gets automatically rejected by the max function in the nominator of Equation 3.10.

They also present the interesting idea to vary the number of samples with distance such that background objects use fewer samples. This improves performance since fewer samples are used overall.

3.2.14 A Separable Approach

The core idea is to separate the AO computation into a vertical and horizontal pass [Huang et al. 2011]. This presents an additional layer of approximation to a generic SSAO algorithm. There is a strong analogy with separable blur filters where the blur kernel is first evaluated horizontally and subsequentially vertically. The

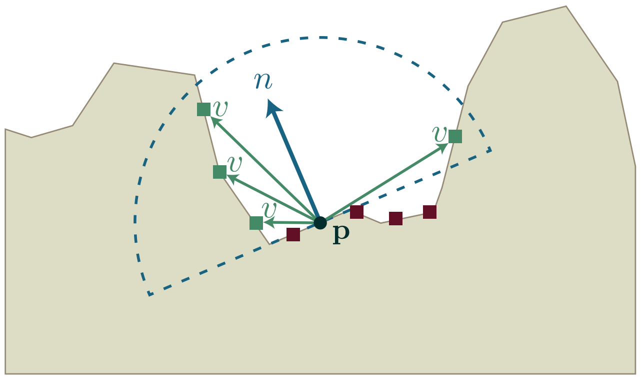


Figure 3.8: *Approximation of AO [McGuire et al. 2011]. The green and red line segments are the contributing and rejected samples, respectively. The sample vectors, v, have been drawn for the contributing samples. Notice how each sample has been projected onto the scene surface.*

benefit is improved performance due to less samples overall. We will return to separable blurs later.

In the basic form presented above the method does not deal with diagonally aligned occlusion. E.g. a flagpole will not have a round shadow at its base but a cross. To circumvent this problem, the authors propose to randomly rotate the sample frame for each pixel. I.e. the two "horizontal" and "vertical" directions are still orthogonal but they are rotated together relative to the image frame. This effectively hides the areforementioned "cross" artifacts.

It should be noted that a blur phase is needed to hide the jitter introduced by the randomly rotated sample frames.

3.2.15 Another Horizon-Based Approach

Instead of attempting to approximate the precise horizon angle h (e.g. with ray-marching [Bavoil et al. 2008a]), many sample angles are sampled and averaged [Mittring 2012]. As already noted, there is a strong parallel to the aforementioned angle-based algorithm. This method can be seen as a late variant of the earlier algorithm.

A set of paired sample positions $\langle \mathbf{s}_1, \mathbf{s}_2 \rangle_n$ are found around $\mathbf{p}$ in screen coordinates. Paired in this context mains that each pair of sample positions are symmetrically positioned around $\mathbf{p}$. Each pair is projected into the scene by setting $\mathbf{s}_z = depth(\mathbf{s}_{xy})$ for both samples $\mathbf{s}_1$ and $\mathbf{s}_2$ in the pair. The angle of the free horizon can now be estimated for the pair as the angle between the vectors $\mathbf{s}_1 - \mathbf{p}$ and $\mathbf{s}_2 - \mathbf{p}$. The free horizon angle is approxmiated as the mean angle found from the pairs. The normal can be used to clamp each $\mathbf{s} - \mathbf{p}$ vector to the tangent plane.

3.2.16 A Scalable Approach

The authors tune their previous method [McGuire et al. 2011] to scale with high resolutions as well as large sampling radii [McGuire et al. 2012; Bukowski et al. 2012].

The first improvement is to cut on memory bandwidth by only relying on the depth buffer. From the depth buffer, positions and normals can be derived—the latter via spatial derivatives. Another key change is to use a mipmapped depth buffer similar to the work done in [Hoang and Low 2010]. This improves the performance of texture fetches from the depth buffer which enables the algorithm

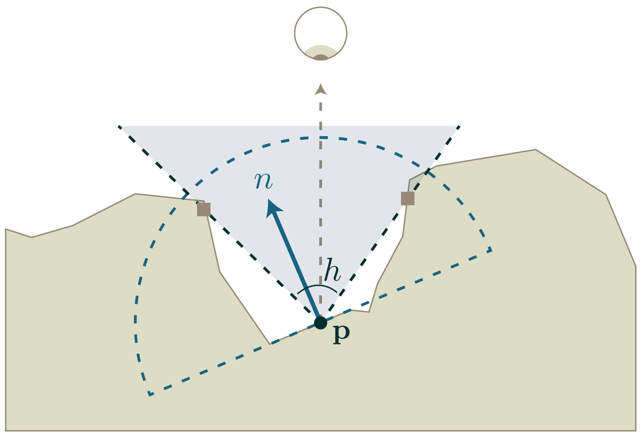


Figure 3.9: *Approximation of AO [Mittring 2012]. The sample pair is denoted by rectangles. Together the samples approximate the angle of the free horizon, h. The sideview denotes the situation for one angle θ around the view vector.*

to scale with large sample radii. The fetched mipmap-level depends on how far $\mathbf{s}$ is from $\mathbf{p}$ such that distant samples fetch from coarser depth buffer levels. I.e. fine details are preserved around $\mathbf{p}$ where the high-resolution depth buffer is sampled while long-range occlusion is also taken into account albeit through coarse depth buffer levels.

It should be noted that in both [Hoang and Low 2010] and [Hoang and Low 2012] the output AO is also calculated at multiple resolutions corresponding to the resolutions of the input buffers. Then AO is subsequently combined in a later pass. In [McGuire et al. 2012], AO is always calculated at the highest resolution even though it uses a mipmapped depth buffer.

The authors also provide a lot of technical details to improve performance on contemporary hardware. These are very low-level and I won't go into details here.

3.3 Overview

The previous section has uncovered a vast number of approaches to AO with detailed explanations to many of them. We have produced an overview in Table 1. The table lists the reference that presents or explains the method, the name of the method, the input the method requires (as denoted in the reference), and a non-exhausting list of industry uses (as denoted in the reference). Each table entry has an associated note that explains either the key points of the method or details interesting ideas from the reference.

Table 1: *Overview of Methods*

Reference	Method	Input	Industry Use
[Pharr and Fernando 2005]	*Dynamic AO*	*Polygon mesh decompositioned into disk elements*	
Not really a screen-space approach. Here because it is the first (near) real-time AO method for rasterized scenes.			
[Luft et al. 2006]	*Unsharp Masking*	*Depth buffer*	
Apply a filter kernel to the depth buffer. The result is not a direct AO computation but a post-process effect that mimics it.			
[Mittring 2007a; Mittring 2007b; Dachsbacher and Kautz 2009]	*CryENGINE 2 SSAO*	*Depth buffer*	*Crysis (2007)*
Approximate AO as the ratio of visible to occluded nearby sample points $\mathbf{s}$. Samples are chosen at randomly in a sphere around $\mathbf{p}$ in screen coordinates. In [Mittring 2007b] the authors further reveal that a smart blurring scheme (depending on the depth buffer) is needed to reduce noise. Additionally, they suggest that the sample distribution (the sphere radius) should be scaled with the distance into the scene.			
[Shanmugam and Arikan 2007]	*Image-space approach for high-frequency AO*	*Depth buffer* *Normal buffer*	
Approximate AO by averaging the occlusion of nearby surfaces represented by proxy spheres.			
[Nguyen 2007]	*Improved dynamic AO*	*Polygon mesh decompositioned into disk elements*	
An improvement over the ealier method found in [Pharr and Fernando 2005]. Smooths the result which is a trait that is also found in many screen-space methods.			
[Bavoil et al. 2008a; Bavoil et al. 2008b; Bavoil and Sainz 2008; Dachsbacher and Kautz 2009]	*HBAO*	*Depth buffer* *Normal buffer*	
Approximate AO by the size of the unoccluded horizon in the hemisphere defined by $\mathbf{p}$ and n. The method ray-marches the depth buffer to get the horizon estimates. Furthermore, the authors suggests to jitter the step size of the ray-marching and choose directions randomly to avoid systematic noise.			
[Dimitrov et al. 2008; Sainz 2008]	*HSAO*	*Depth Buffer* *Normal Buffer*	
Very similar to [Bavoil et al. 2008a]. The only notable difference is that the rays are traced in eye coordinates and then projected into screen coordinates.			
[Filion and McNaughton 2008]	*Starcraft II SSAO*	*Depth Buffer* *(Normal buffer)*	*Starcraft II (2010)*
Similar to [Mittring 2007a] though the authors claim to have come up with the method on their own. The differences are in the details. Most notable is that point samples are found in world coordinates and not screen coordinates.			
[Ritschel et al. 2009; Dachsbacher and Kautz 2009]	*SSDO (Screen Space Directional Occlusion)*	*Depth Buffer* *Normal buffer*	
SSDO is not another take on how to calculate ambient occlusion in screen-space. It is an extension of the core SSAO idea. It allows for colored and directed shadows in the result. Furthermore, SSDO has an optional second pass that approximates one diffuse indirect bounce of light.			
[Bavoil and Sainz 2009]	*Multi-Layer Dual-Resolution SSAO*	*[Any required by the basis method]*	
This paper presents enhancement to any other SSAO algorithm. The authors propose to use depth peeling and/or multiple-view depth buffers. They also suggests to calculate AO at different resolutions to get better performance.			
[Dachsbacher and Kautz 2009]			
A good presentation of the *Dynamic AO, CryENGINE2 SSAO*, *HBAO* and *SSDO*. While the authors do not present any new information about the various methods they do provide a good overview. The presentation is a good starting point when learning about SSAO (and global illumination).			
[Smedberg and Weight 2009]	*TSSAO (Temporal SSAO)*	*[Any required by the basis method]* *AO buffer (from previous frame)*	*Gears of War 2*
Presents the idea of re-using AO computations of previous frames in order to smooth out the overall result.			
[Mattausch et al. 2010; Scherzer et al. 2010]	*TSSAO (Temporal SSAO)*	*[Any required by the basis method]* *AO buffer (from previous frame)*	
The same method as presented in [Smedberg and Weight 2009] but developed independently. The authors also suggest to use *temporal refinement*, which involves re-using actual samples and not just blending AO results between frames.			
Reference	**Method**	**Input**	**Industry Use**

Reference	Method	Input	Industry Use
[Song et al. 2010]	*HBAO using Mixture-Sampling*	*Depth Buffer* *Normal Buffer*	
This is a combination of the methods found in [Bavoil et al. 2008a] and [Shanmugam and Arikan 2007]. The first hybrid method.			
[Loos and Sloan 2010]	*VO (Volumetric Obscurance)*	*Depth Buffer* *(Normal Buffer)*	
The authors define the term Volumetric Obscurance (VO) and use it to find AO. Simply put, they use the ratio of visible to occluded volume around $\mathbf{p}$. Another new idea is to combine the AO calculations of a small sphere and a large sample sphere. This technique captures both low-scale as well as broad-scale details.			
[Ownby et al. 2010]	*VO (Volumetric Obscurance)*	*Depth Buffer* *(Normal Buffer)*	*Toy Story 3 (Video Game, 2010)*
Based on the line integrals of [Loos and Sloan 2010]. The authors goes into details of how to choose effective sample positions $\mathbf{s}$ using randomization and rotation. They also propose to use paired sample locations (symmetric around $\mathbf{p}$) to better estimate missing sample information due to the rough scene approximation given by the depth buffer.			
[Kaplanyan 2010]	*[Production Integration]*		*Crysis 2 (2011)*
Though the authors present no new methods they show that it is possible to combine basic methods with extensions in a production. They use the volumetric obscurance method [Loos and Sloan 2010] in combination with temporal refinement [Mattausch et al. 2010]. It is interesting to note that they still suggest calcualting $\mathbf{p}$ at half-resolution *and* blurring the result.			
[Hoang and Low 2010]	*MSSAO*	*[Any required by the basis method]* *Mipmaps*	
The same idea of using multiple resolutions as in [Bavoil and Sainz 2009] but taken a step further.			
[Sourimant et al. 2011]	*Poisson Disk Ray-Marched AO*	*Depth Buffer* *Normal Buffer*	
Introduces the concept of Poisson distributed samples and presents a method for computing AO using combinations of previous methods.			
[Aksoy and Phaneuf 2011]	*Poisson Disk Projected AO*	*Depth Buffer* *Normal Buffer*	*EA Sports MMA (2010)*
Another approach involving Poisson distributed samples though the authors use a different AO computation. Similarly to[Loos and Sloan 2010], the authors suggest to use two hemispheres with different radii to capture both fine-scale and broad-scale details.			
[Hattori et al. 2010; Hattori et al. 2011]	*Curvature-based AO*	*Depth Buffer* *Normal Buffer*	
An unique method that mimics AO by analyzing the scene curvature.			
[White and Barré-Brisebois 2011]	*[Production Integration]*		*Battlefield 3 (2011)* *Need For Speed: The Run (2011)*
A presentation of how AO effects fit into a production. The authors use [Bavoil et al. 2008a] for the best quality and [Loos and Sloan 2010] for a performant alternative that runs interactively on consoles. They go into details on how to tweak performance out of the latter method. This is mostly low-level and technical tricks.			
[Kasyan et al. 2011; Donzallaz and Sousa 2011]	*[Extension]*	*[Any required by the basis method]*	*Crysis 2 (2011)*
Introduces the concept of bent normals into the SSAO domain.			
[McGuire et al. 2011]	*Alchemy AO (Ambient Obscurance)*	*Depth Buffer* *Normal Buffer*	
A unique method that uses a carefully chosen ρ function to get a simple formula for AO.			
[Huang et al. 2011]	*[Extension]*	*[Any required by the basis method]*	
An extension to a generic AO algorithm that splits the computations into two passes like a separable blur filter.			
[Klehm et al. 2011]	*[Extension]*	*[Any required by the basis method]*	
Extends the concept of bent normals to bent cones.			
[Ritschel et al. 2012]	*[Overview]*		
Provides a good, short overview of different ambient occlusion methods (screen space and not). We have used it to find many of the previous methods. However, the authors do not explain the methods in any detail at all. They do roughly categorizone them based on whether they approximate AO by projecting proxy geometry, finding unoccluded volume, or finding unoccluded directions.			
Reference	**Method**	**Input**	**Industry Use**

Reference	**Method**	**Input**	**Industry Use**
[Hoang and Low 2012]	*MSSAO*	*[Any required by the basis method]* *Mipmaps*	
Similar to the earlier work by the same authors in [Hoang and Low 2010]. However this time, each pass is explained in detail and with mathematical motivation.			
[Mittring 2012]	*Unreal Engine 4 SSAO*	*Depth Buffer* *(Normal Buffer)*	
This method is based on finding the free angle of the free horizon as in [Bavoil et al. 2008a]. However, it is differs in that it doesn't use ray-marching but instead sample pairs projected onto the scene surface.			
[McGuire et al. 2012; Bukowski et al. 2012]	*SAO (Scalable Ambient Obscurance)*	*Depth Buffer*	
The method is closely based on the earlier work of the same authors presented in [McGuire et al. 2011]. The authors tune their previous method to scale with high resolutions as well as large sampling radii.			
Reference	**Method**	**Input**	**Industry Use**

4 Design

The reader should now have a general idea of the different approaches to SSAO. We will now outline some common observations and present a framework for the comparison.

4.1 Common Observations

4.1.1 Depth Buffer Discontinuities

In the previous section we have thus far assumed that the depth buffer is a good scene approximation. This is not always the case as seen in Figure 4.1. The depth buffer only stores the frontmost surfaces and all other surface information is lost. This may result in erroneous AO computations. Take for instance the point $\mathbf{p}$ in Figure 4.1 which is in top of a bump and should this be fully visible. However, the floating object to its right causes a sharp rise in the depth buffer. Consequently, $\mathbf{p}$ may be deemed partly occluded by some the previous algorithms. The resulting artificat is dark halos along depth discontinuities.

Falloff Function The easiest remedy is to include the falloff function $\rho(d)$ in the AO approximation. Foreground and background objects tend to be separate by quite a distance; certainly over d_{max} for the most part. With an appropriately set d_{max}, any samples to the left of point $\mathbf{p}$ in Figure 4.1 will be treated as visible. There exists a delicate balance when setting d_{max} that can only be found empirically. After all, ρ was introduced with aesthetical motivation [Zhukov et al. 1998]. However, this is not a perfect solution. Sometimes samples are occluded by a surface hidden from the view of the depth buffer. In this case it is wrong to treat the sample as visible.

Rejection A simpler but related alternative is to reject samples whose depth value $\mathbf{s}_d$ are too far from $\mathbf{p}_d$. That is, we measure the difference $\mathbf{p}_d - \mathbf{s}_d$ and not the distance $|\mathbf{s} - \mathbf{p}|$. This can be computationally cheaper for methods that calculates the former difference anyways but not the latter distance. Simply discarding like this samples may lead to undersampling [Ownby et al. 2010]. E.g. if a most samples are beyond the threshold then few remain for the actual AO calculation—maybe too few.

Depth Layers and Multiple Views There is also the idea to use a layered depth buffer (or depth peeling) [Ritschel et al. 2009; Bavoil and Sainz 2009]. Each layer starts at an increasing distance into the scene and hopes to capture surfaces that are hidden by the frontmost layers. If such a layer was available between the floating object and the ground in Figure 4.1, then there wouldn't be a problem. Alternatively, multiple view angles can be used to generate depth buffers from different perspectives [Ritschel et al. 2009]. Hopefully, a surface hidden from one angle may be visible from another and thereby the problem disappears. Both the ideas of depth layers and multiple viewing angles will eat into the memory budget which may hurt performance. Additionally, each method must incorporate the computations required to facilitate the extra scene information.

Paired Sampling Lastly, is the idea of using sample pairs to increase the changes to recover missing depth information [Ownby et al. 2010]. This approach uses the assumption that surfaces are flat on average. For many scenes, this is not an unreasonable assumption. If one sample in a pair is rejected (say by the difference $\mathbf{p}_d - \mathbf{s}_d$) then its partner may be used to reconstruct the hidden information (See Figure 4.2). In the figure, the green sample has been rotated around $\mathbf{p}$ and reconstructed the missing information. This approach lends itself naturally to methods that use paired

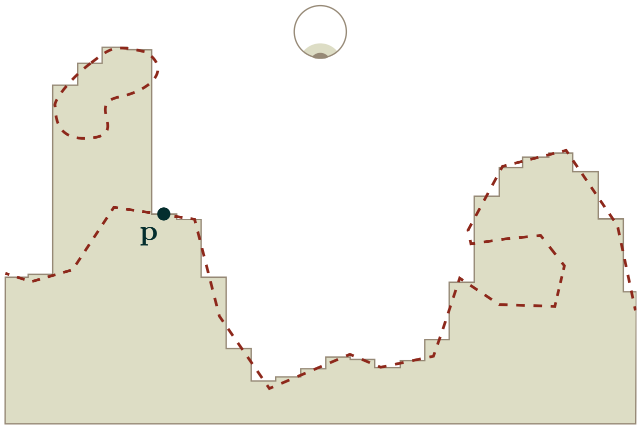


Figure 4.1: *Problems encountered with a scene represented as a depth buffer. The real scene's surface is the dashed line and the shaded bars are the depth buffer representation. Notice that only the frontmost (towards the eye) surfaces are stored in the depth buffer.*

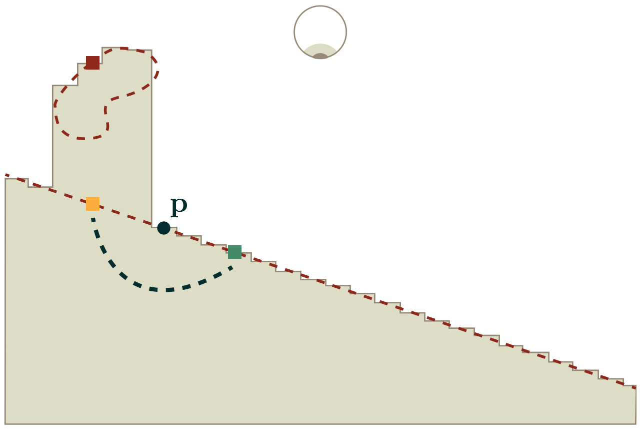


Figure 4.2: *Depth discontinuities resolved with paired samples [Ownby et al. 2010]. The red and green rectangles are the paired samples projected onto the depth buffer. The orange sample is the reconstructed position that the red sample should have had. Note the assumption of a flat surface.*

samples by default. Furthermore, many SSAO methods can be extended to use paired sampling.

Comparison All of the above remidies for depth buffer discontinuities can theoretically be applied between many of the methods. However, each method lends itself more naturally to a specific remedy either by the governing model or by the data it computes. How each method treats depth buffer discontinuities is an interesting variable for comparison. The chosen approach greatly affects the resulting image but also the computation time.

Note that the definition of the falloff function ρ is vague and thus a potential variable itself for the methods that use it. We have chosen to go with the definitions that recommended by the papers behind each method.

4.1.2 Guard Bands

As we have previously mentioned, some authors recommend the use of a guard band to artifacts near the screen edges [Bavoil and

Sainz 2009]. However, this comes at a performance cost since more pixels are needed to be rendered. More interesting is it that the use of guard bands are never mentioned in production use cases (as far as we can see) but only in academic papers. We ponder that the reason for this is that the edge artifacts are not noticed since the focus is mostly on the center of the screen. Consequently, we have chosen not to use guard bands. This variable will therefore not be in the comparison.

4.1.3 Memory Budget Impact

When the study initally began, we assumed that each method would have different memory requirements. However, it turns out that all methods can be implemented with the depth buffer as the only input. The reason is that positions can be reconstructed from depth [Mittring 2007a; Kasyan et al. 2011] and normals can be reconstructed from positions [McGuire et al. 2012; Bukowski et al. 2012]. Consequently, there is potentially no difference in memory requirements between the methods. It should be noted that the reconstruction does come with a slight performance hit though nothing major on modern hardware. Furthermore, the quality of reconstructed normals is debatable [Bukowski et al. 2012] and some authors prefer the real normals. Nevertheless, we have chosen to keep the memory budget aspect out of the comparison.

4.1.4 Blurring

A key point in the AO pipeline of Figure 3.2b is the use of a blur to remove high-frequency noise introduced by random sampling. All of the aforementioned SSAO methods (unless explicitly noted in the analysis) recommend a blurring pass. That is not to say, that every method absolutely need one. Some methods can indeed produce quality results if a high enough number of samples N is used. The problem is that performance goes down quickly when N increases. However, each method scales differently and thus this becomes an interesting point of comparison. We will attempt to discover how each selected method respond to skipping the blur pass entirely while maintaining quality output. We expect that some will suffer more than others in terms of performance.

The Blur Itself Common to all recommendations is that the blur must be geometry-aware. That is, it must not blur accross unrelated surfaces. Some authors suggest to use the depth buffer [Mittring 2007a] and others propose to additionally use the surface normals [Filion and McNaughton 2008]. However, the goal of all methods seem to be the same: Remove the high frequency noise while maintaining surface definition. In light of this, we have chosen to use the *same* blur pass for *all* methods. This reduces the dimensions of the comparison but more importantly focuses the discussion on the SSAO details. That being said, some methods require broader blurs than others. This will remain as a factor in the comparison.

We use a separable blur filter that uses both depth and normal information to maintain surface definition. This should ensure that all methods retrieve a high-quality blur. See the implementation section for more details.

4.1.5 Noise and Sample Patterns

It is inherent to all the methods that the choice of sample pattern will be reflected in the result. When a fixed pattern is used, banding occurs. When a random pattern is used, high frequency noise is added [Mittring 2007a]. Generally speaking, authors prefer the high frequency noise to banding artifacts. Each method also has a recommended sample pattern—often strongly linked with the way the method works. Thus it is infeasible switch sample patterns between methods. Consequently, sample patterns will not be a variable in the comparison.

4.1.6 Self-Occlusion

Some methods use samples distributed in a sphere around the position in question ([Mittring 2007a] and [Loos and Sloan 2010]). The problem is that even on a flat surface half of the samples are expected to be below the surface (See Figures 3.3 and 3.6). This is known as self-occlusion. Methods that self-occlude have a very distinct, almost stylish look. The remedy is simply to transform the AO approximation by parameters in order to remove the look. Parameters will be presented shortly.

If we recall the original definition of AO (Equation 2.5) we will note that it uses a hemisphere to gather the samples. Methods that follow this definition will self-occlude.

4.1.7 Varying the Resolution

Many authors suggests to calculate AO at a lower resolution for performance reasons and then rescale it to fit the screen [Bavoil and Sainz 2008; Bavoil et al. 2008b; Smedberg and Weight 2009; Bavoil and Sainz 2009]. Other authors make a dedicated and involved effort to compute AO at multiple resolutions [Bavoil and Sainz 2009; Hoang and Low 2010; Hoang and Low 2012; McGuire et al. 2012]. Some are smart about the rescaling and resamples AO when precision is lacking [Bavoil and Sainz 2009] while others favor performance and skips the resampling [Smedberg and Weight 2009].

We have chosen to always compute AO at one specific resolution (800×800 pixels) to leave this factor out of the comparison. It puts every approach on equal footing so differences in quality can't be attributed to rescaling.

4.1.8 Scale

A prime variable for the comparison is the scale at which the AO effect applies. All methods have a radius of influence or similar either to control the radius of a sample sphere or sample disk. In Equation 2.5 it is implicitly defined in the falloff function ρ by the parameter d_{max}. The ray-traced reference scales as expected. It will be interesting to see how the candidate methods scale in comparison.

Number of Samples The number of samples N used in solving the AO integral is also a good candidate variable for the comparison. Each method scales differently with N but common for them all is that they get slow for large N. We will attempt to find some good compromises. This will be explained in a later chapter along the comparison framework.

4.1.9 Computational Performance

Performance is profiled as the time of execution in milliseconds (hopefully). Such profiling is easily performed in the software itself. It is important, however, that the application as a whole is not profiled. Only the shader passes computing AO and the associated blurs are relevant.

We use an Nvidia GeForce 260 GTX GPU during profiling. This card came around 2008-2009 and thus it is somewhat dated by todays standards. However, this ensures that if our implementation performs well in our tests then they ought to perform even better using more recent hardware.

Industry Average Render Budget There doesn't exist a standard reference frame for how long an AO computation should take. However, we can compute an average from the render budgets used in the industry:

- AO Computation: 1.2 ms [Kaplanyan 2010].
- AO Computation and GI[1]: 2 ms [Donzallaz and Sousa 2011].
- AO Computation and Blur): 2-4 ms [McGuire et al. 2011].
  - Between 3-5 ms when targeting multiple platforms.

Unfortunately, two the render budgets we found (above) are not purely for AO computations. Also, the authors did only mention the GPU used in some of the cases. However, we think it is fair to choose 2 ms based on the above information. The performance comparison is supposed to be inter-method anyhow. The industry's render budget is only included as a reference.

4.1.10 Quality

Quality is assessed by a manual visual comparison. We have produced a ray-traced reference which is used as the golden standard. We aim to tweak each method to resemble the reference as closely as possible.

4.1.11 Reference Scene

It is important to use the same scene (virtual world) and camera position when doing the comparison. The versed reader may have noticed that we use the Sponza model developed by Crytek. It is the slightly modified version as found on `http://graphics.cs.williams.edu/data/meshes.xml`. The reference will be rendered using the existing ray-tracing implementation in the Mental Ray renderer. We have consulted the documentation and noted that Mental Ray can in fact be configured to implement equation 2.5. We used 64 rays per pixel in the full hemisphere with a linear falloff function, ρ.

We use two different camera positions for the comparisons. One that looks down along the hall, and one that focuses on the lion head on the stone wall. The former tests the methods in a setting with many depth discontinuities (pillars, curtains) while the latter has a focus on small-scale details (curvature and creases of the lion head). This will also be a variable for comparison. That is, whether the method is best at captuing large-scale effects, low-scale effects, or both. This comparison parameter was proposed in [McGuire et al. 2011].

4.1.12 Parameters

Each method has its own set of parameters that can be tweaked to the users liking. Sometimes the governing AO model does not cover all the details which gives the implementer a certain degree of freedom. However, there exists a set of parameters to tweak the result that shared between all of the methods. They are applied via a post-processing step as

$$AO'(AO) = (b \cdot (AO + a))^c$$

where a, b, and c are user-defined. b and c can be used to control the brigthness and contrast of the AO computation. a can be used to remedy self-occlusion (see above) by simply adding an expected 0.5 occlusion amount to the result ($a = 0.5$) and clamp. This removes the stylish look that self-occluding methods otherwise produce.

4.2 Candidate Methods

The methods chosen for comparison are in Figure 4.3. They are meant to represent SSAO methods that differ both algorithmically and chronologically. The former enables us to cover different governing AO approximations. The latter lets us observe how the early methods have influenced the later methods and the general trends.

Furthermore, we have found that the methods can be categorized based on a combination ofAO approximation and how they solve the integral. This allows us to quickly assess each methods' algorithmic basis. Thus methods from differenct categories (which we will uncover below) should be equally represented.

4.2.1 Categorization of Methods

Point Sampled AO These groups of methods are key to all SSAO approaches because they were the first to popularize the use of SSAO. They are based on intuitive reasoning and have seen much industry use. Even recent video game titles consoles have used them as their basic method [Smedberg and Weight 2009]. The basic premise is to probe sample locations in a volume around each point in quest and use the ratio of visible to occluded samples as the AO approximation. As far as we can see, the two most popular candidates are [Mittring 2007a] and [Filion and McNaughton 2008] which both keep getting mentions in recent SSAO-related papers [Hoang and Low 2012]. Therefore, we have chosen to implement both (Figures 4.3a and 4.3b). Between them, there are some key differences which vastly affect the result. Therefore, it is relevant to include both in the study.

Horizon-Based AO The next iteration of popular AO approximations. The horizon-based AO methods take an algorithmically different approach altogether and yet produce a convincing result. They aim to find the angle of the free horizon instead of directly probing samples for visibility. Therefore, they are interesting to study and two candidates have been included in the comparison. The first method [Bavoil et al. 2008a] presented the horizon-based approach and the second rely on the same concept but is implemented differently [Mittring 2012]. As seen in Figures 4.3c and 4.3f, the methods are also greatly split by time. It will be interesting to find whether the later method can be seen as an improvement of the earlier method—if only in some variables of the comparison.

[Mittring 2012] lends itself so easily to using paired samples to solve depth discontinuities [Ownby et al. 2010]. Therefore, we have chosen to add this feature to our implementation (see next chapter). This choice, however, makes the method differ from the one proposed in the reference if only ever so slightly. We argue that the resulting comparison only becomes more interesting with the added variation.

Volumetric AO (Line Sampled AO) Technically speaking, the point sampled methods also belong to this category as mentioned before. However, we choose to split them into two; the first one already being mentioned and the other one explained here. With the inception of the VO term also came the idea to use line sampling [Loos and Sloan 2010] to solve the integral as opposed to point sampling (Figure 4.3d). Therefore, the two terms have become synonymous in the literature and we follow that way of thinking here. Consequently, this category is for methods that solve the integral using line samples. The name of the category is chosen to be compatible with the existing literature.

The prominent VO method [Loos and Sloan 2010] has seen production use [Kaplanyan 2010] and some authors present significant advantages in terms of both performace and quality over the point sampling counterparts. It will be interesting to see if this is also reflected in our comparison.

This method is also a good candidate for using paired sampling to

[1]Global Illumination

solve depth discontinuities. In fact, the original paper on the idea used the VO method as basis [Ownby et al. 2010].

Alchemy AO We have chosen to create a category solely for the alchemy method [McGuire et al. 2011] (See Figure 4.3e). On some terms it shares many similarities with both the point sampling and the horizon-based approaches and yet it seems entirely different when put together. One could argue that it is in fact a hybrid but we think that the inclusion of an intelligently chosen ρ function and the reduction of terms in Equation 3.10 is enough to make it unique. It also fits nicely on the timeline between the other methods.

4.2.2 Common Attributes

Common to all the candidate methods is that they require uniformly distributed unit vectors to implement their sample patterns. I.e. every method can be supplied the same random texture (See Figure 3.2b).

The methods that require normals will have them explicitly provided by the normal buffer. So in our overall design of an SSAO algorithm, the dashed line in Figure 3.2b is used.

4.3 Comparison Framework

We have presented variables to use in the comparison as well as the candidate method in the sections above. This section will combine the variables and candidates into a framework that will be the basis of the comparison. I.e. the methodology of the comparison.

4.3.1 Configurations

We will make two different configurations for each candidate method tuned for two different purposes.

Performance The first configuration is intended to showcase how performant the method can be. Some level of quality must be maintained and we aim to find a good compromise in visual quality that is comparable between the methods. This is a task prone to subjective opinion. We will provide sample renders so that the reader can apply his or her own conclusions as well.

Quality The second configuration is on quality. We aim here to find the set of settings that aligns the result with the reference image. We also intend to disable the blur pass for this configuration in order to highlight deficiencies that are inherent to the method. To keep things reasonable, we will aim to find the point where the AO term converges. That is, we will seek the set of settings where increasing parameters further will only return diminishing results. Again, this is a task prone to subjective opinion and we will provide sample renders for the reader to inspect.

4.3.2 Scalability

We will then test the scalability of the method by varying the sampling radius while keeping the other parameters constant. This includes not changing the number of samples. The results will show how each method scales in terms of quality and performance.

We will then conduct the same experiment but this time increase the sampling radius along with the number of samples as well. The aim is to find a relation between the two variables.

4.3.3 Views

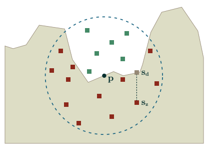
When relevant, the comparisons will be done for two different views (camera angles) of the scene as discussed in section 4.1.11.

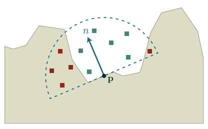
4.3.4 Overview

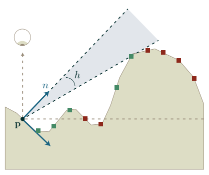
We will list the comparison variables here to provide an overview for the reader:

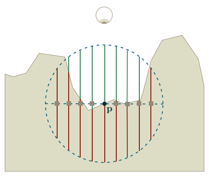
- Sampling radius (world-space units).
- Number of samples.
- Parameters a, b, and c.
- Blur kernel size (pixels)
- Performance (milliseconds)
- Quality

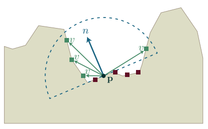
Other variables (e.g. the sample distribution) are either inherent to the each method or have been left out of the comparison to keep the complexity reasonable.


(a) *Point sampled AO in sphere, screen coordinates [Mittring 2007a].*


(b) *Point sampled AO in hemisphere, world coordinates [Filion and McNaughton 2008].*


(c) *Horizon-based AO with ray-marching [Bavoil et al. 2008a].*


(d) *Volumetric AO using line samples [Loos and Sloan 2010].*


(e) *Alchemy AO with projected samples and special ρ function [McGuire et al. 2011].*

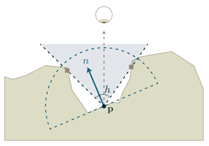

(f) *Horizon-based AO with paired samples [Mittring 2012].*

Figure 4.3: *The selected SSAO methods.*

5 Implementation

This section will explain how each of the candidate methods are translated into working shaders. We will also explain the blur shader that is shared between all the methods.

5.1 Overall

The application in which the SSAO methods are implemented is written in C++. It is project that has been used earlier by one of the authors of this paper for other real-time rendering experiments. It is freely available and can be found on GitHub at `https://github.com/frederikaalund/sfj`. It is based on standard implementations, notably the OpenGL API. Consequently, the SSAO methods are implemented as shaders in the associated shading language, GLSL. The shaders are also available at the aforementioned GitHub address. They can also be found in full length in the appendix of this report.

5.1.1 Prefixes

We use the coordinate (space) abbreviations found in section 2.5.1 as a prefix to variables. This makes it easier to see which space the associated variable belongs to. E.g. `vec3 wc_position` would be a position in World Coordinates (WC) whereas `vec2 tc_sample` denotes the a sample position in Texture Coordinates (TC).

5.2 The Candidate Methods

Many of the candidate methods share similar steps. We will explain how these steps are implemented when they first appear and refer back to previous explanations when needed. This way, we won't repeat ourselves. The discussion is also limited to the AO *fragment* shaders since the accompanying vertex shaders are trival—they just forward a full-screen quad. We will not present the full shader code here but only relevant snippets. Please refer to the appendix for the full source code of each shader.

5.2.1 [Mittring 2007a]

The method in [Mittring 2007a] (Figure 4.3a) presents some challenges which we will conquer in turn. We will be extra detailed when describing this method since it is the first and a lot of basics have to be laid down. Subsequent descriptions will only cover key parts.

Random Vectors Shaders follow the SPMD principle and thus have real difficulty generating random numbers different for each data set. We remedy the situation by providing pre-computed random vectors in a texture. The vectors are distributed in a uniformly in a circle (generated with rejection sampling). They are then subsequently converted to the $[0;1]^3$ range to preserve precision. Therefore, when we read a random direction from the texture, we must remember to scale it back into the $[-1;1]^3$ range

```
vec3 random_direction = texture(random_texture,
      tc_random_texture).xyz;
random_direction = normalize(random_direction * 2.0 - 1.0);
```

Sampling The sampling itself works in a for-loop

```
for (int i = 0; i < samples; ++i)
{
    ... // Use sample i
}
```

where $samples$ is the number of samples used. Every sample requires a random direction. That is, we need $samples \times pixels$ random directions. The random texture take up way too much memory if it were to have that many entries. Instead, the authors of the method propose to reflect each sample's random direction in the plane defined by a per-fragment random direction.

```
vec3 sample_random_direction = // Fetch rand vector based on
      i
sample_random_direction = sample_random_direction * 2.0 -
      1.0;
sample_random_direction = reflect(sample_random_direction,
      random_direction);
```

where the $reflect$ function is already part of the GLSL specification. Now we proceed to use the random direction to sample a sphere in screen coordinates. That should rather be *texture coordinates* as they are the ones the buffer sampler function expects. Remember that the difference between screen coordinates (range $[0;width]$, $[0;height]$, $[0;1]$) and texture coordinates (range $[0;1]^3$) is just a question about scale. The sample position is found as follows

```
vec3 tc_sample_pos = vec3(tc_depths.xy, scene_depth)
         + vec3(sample_random_direction.xy *
      projection_scale_xy, sample_random_direction.z *
      scene_depth * projection_scale_z) * radius;
```

where the $projection\_scale$ factors are there because the sample radius is given in world coordinates. I.e. the sample radius must be scaled to account for projection. It is done as follows

```
float projection_scale_xy = 1.0 / ec_depth_negated;
float projection_scale_z = 100.0 / z_far *
      projection_scale_xy;
```

where $ec\_depth\_negated$ is the depth read from the depth buffer. It is negated because eye coordinates have the z-axis pointing out of the screen.

Now we have a sample position and are ready to find the actual depth, $\mathbf{s}_d$, of the surface above or below it (See Figure 3.3). It can be found by a read from the depth buffer

```
float sample_depth = texture(depths, tc_sample_pos.xy).x;
```

What remains is simply to make the test of V' and add the contribution

```
ambient_occlusion.a += float(sample_depth > tc_sample_pos.z);
```

When all samples are gathered, and the execution returns from the for loop, then the sum of contributions is divided by the number of samples

```
ambient_occlusion.a /= float(samples);
```

just as it is done in Equation 3.1.

5.2.2 [Filion and McNaughton 2008]

This method shares many of the same traits of the above. One difference is that it finds samples in the hemisphere aound $\mathbf{p}$ and not in a circle (See Figure 3.4). This is achieved by mirroring the samples of the negative hemisphere around the plane defined by the surface normal into the positive hemisphere.

```
sample_random_direction = faceforward(sample_random_direction
      , sample_random_direction, -wc_normal);
```

where the $faceforward$ function is part of the GLSL specification. The observant reader may have noticed the other key difference already: samples are chosen in world coordinates. However, this implies that each sample must be projected into texture coordinates for the subsequent depth buffer look-up. It is done by mimicking the OpenGL transformation pipeline

```
vec3 wc_sample = wc_position + sample_random_direction *
      radius;
vec3 ec_sample = (view_matrix * vec4(wc_sample, 1.0)).xyz;
vec4 cc_sample = view_projection_matrix * vec4(wc_sample,
      1.0);
vec3 ndc_sample = cc_sample.xyz / cc_sample.w;
vec2 tc_sample = (ndc_sample.xy + vec2(1.0)) * 0.5;
```

where our prefix notation comes to good use. We also implement this method with a falloff function ρ as follows

```
float rho = clamp((depth_difference - radius) /
      depth_difference, 0.0, 1.0);
ambient_occlusion.a += rho;
```

We note that it would be easy to integrate the concept of bent normals [Klehm et al. 2011] now as

```
bent_normal += normalize(sample_random_direction) * rho;
```

though we don't use this in the actual implementation.

5.2.3 [Bavoil et al. 2008a]

In this case, the number of samples refer to the number of directions in which the algorithm marches along a ray. When choosing an angle θ around the view vector, we use a random direction as before but simply discard the z-coordinate. I.e. we don't first choose an angle and then convert it to a direction but simply use one of the random direction vectors up front. The algorithm then proceeds to find the increments in which it marches along the ray from $\mathbf{p}$ in direction θ.

```
const int steps = 6;
vec2 tc_step_size = tc_sample_direction * projected_radius /
      float(steps);
vec2 ec_step_size = tc_sample_direction * radius / float(
      steps);
```

We found that 6 steps is good enough in our case. This is similar to what the original authors found [Bavoil and Sainz 2008]. Note that we must maintain two step increments: One for texture coordinates and one for eye coordinates. Now the tangent must be reconstructed in eye coordinates. This is similar to how you would reconstruct a normal from position information.

We are now ready to begin the ray-marching. Note that we are never working directly with angle values α in radians but instead with $\tan(\alpha)$ and $\sin(\alpha)$ directly along with conversion functions tan_to_sin, etc. This saves many computations that would otherwise be spent evaluating expensive $\tan$ and $\sin$ functions. The ray marching is nested within the first for loop (over angles θ) as

```
float tan_tangent_angle = ec_tangent.z / length(ec_tangent.xy
      ) + tan(bias);
float tan_horizon_angle = tan_tangent_angle;
float sin_horizon_angle = tan_to_sin(tan_horizon_angle);

for (float j = 1.0; j <= float(steps); j += 1.0)
{
    vec2 tc_sample = vec2(tc_depths + tc_step_size * j);
    vec3 ec_horizon = vec3(ec_step_size * j, ec_depth(
          tc_sample) - ec_position_depth);
    float ec_horizon_length_squared = dot(ec_horizon,
          ec_horizon);
    float tan_sample = ec_horizon.z / length(ec_horizon.xy);

    ... // Evaluate sample
}
```

where tan_sample is the $\tan$ function applied to the horizon angle. Similar conventions apply to the other variables. Note how the ray is actually marched in both texture *and* eye coordinates. The latter is used to keep units consistent with the sample radius. The evaluation of the sample works as explained in section 3.2.4

```
if (radius_squared >= ec_horizon_length_squared && tan_sample
      > tan_horizon_angle)
{
    float sin_sample = tan_to_sin(tan_sample);
    float weight = 1.0 - ec_horizon_length_squared /
          radius_squared;

    ambient_occlusion.a += (sin_sample - sin_horizon_angle) *
          weight;

    tan_horizon_angle = tan_sample;
    sin_horizon_angle = sin_sample;
}
```

Each sample contributes by remembering the previous horizon_angle. Notice how the AO computation resembles that formula found in Equation 3.4.

5.2.4 [Szirmay-Kalos et al. 2009]

The sampling itself is similar to that of [Mittring 2007a] with the exception that each sample $\mathbf{s}_z = \mathbf{p}_z$ as seen in Figure 3.6. The new part is that the ratio visible to occluded line segment length must be found. First by finding the depth difference between $\mathbf{s}$ and $\mathbf{p}$

```
float ec_sample_1_depth = ec_depth(tc_sample_1);
float ec_sample_2_depth = ec_depth(tc_sample_2);

float depth_difference_1 = ec_position_depth -
      ec_sample_1_depth;
float depth_difference_2 = ec_position_depth -
      ec_sample_2_depth;
```

where ec_depth is a function that returns depths in eye coordinates; $ec\_position\_depth$ is $\mathbf{p}_d = depth(\mathbf{p}_{xy})$. Note that sampling is done in pairs symmetric around $\mathbf{p}$. There height of the sphere is found with trigonometry

```
float sphere_height( in vec2 position, in float radius )
{
    return sqrt(radius * radius - dot(position, position));
}
```

Now the algorithm can proceed to find the ratio of occluded to unoccluded line segment length

```
float volume_ratio_1 = (samples_sphere_height -
      depth_difference_1) * samples_sphere_depth_inverted;
float volume_ratio_2 = (samples_sphere_height -
      depth_difference_2) * samples_sphere_depth_inverted;
```

Note that the same sphere height is used for both samples. This is possible because of the symmetry. This is the algorithm in its basic form. However, we intend to provide the improved version that solves depth discontinuity problems with paired sampling [Ownby et al. 2010]. First we find which of the samples in the pair are valid (if any). This is simply checking whether volume ratios are within user-defined bounds (which depends on the scene)

```
bool sample_1_valid = lower_bound <= volume_ratio_1 &&
     upper_bound >= volume_ratio_1;
bool sample_2_valid = lower_bound <= volume_ratio_2 &&
     upper_bound >= volume_ratio_2;
```

If sample $\mathbf{s}_1$ is invalid, a flat surface is assumed and the inverted volume ratio of $\mathbf{s}_2$ is used instead. If both samples are invalid, we are short of options. We have opted to assume 50 % occlusion in this case as proposed by the authors of the method [Ownby et al. 2010].

```
// Should evaluate to a conditional assignment (no branching)
if (sample_1_valid || sample_2_valid)
{
    // If the sample is valid then use it. If not, then use
         the other one in the pair (inverted).
    ambient_occlusion.a += (sample_1_valid) ? volume_ratio_1
         : 1.0 - volume_ratio_2;
    ambient_occlusion.a += (sample_2_valid) ? volume_ratio_2
         : 1.0 - volume_ratio_1;
}
else
{
    // Not 0.5 but 1.0 because both samples were invalid
    ambient_occlusion.a += 1.0;
}
```

5.2.5 [McGuire et al. 2011]

As discussed before, the Alchemy AO method is somewhat of a hybrid in its implementation. It uses the sample distribution of [Loos and Sloan 2010] and then projects each sample onto the scene surface as in [Bavoil et al. 2008a] (See Figure 3.8)

```
vec3 tc_sample;
tc_sample.xy = tc_depths + sample_random_direction *
     projected_radius;
tc_sample.z = tc_depth(tc_sample.xy);
```

Samples are then backprojected from texture to world coordinates for the actual evaluation.

```
vec3 ndc_sample = tc_sample * 2.0 - 1.0;
vec4 temporary = inverse_view_projection_matrix * vec4(
     ndc_sample, 1.0);
vec3 wc_sample = temporary.xyz / temporary.w;
```

Now the AO contribution is evaluated according to Equation 3.10

```
vec3 v = wc_sample - wc_position;
ambient_occlusion.a += max(0.0, dot(v, wc_normal) - bias) / (
     dot(v, v) + epsilon);
```

Most of the surrounding details have already been covered in one form or another. One exception is the use of a distance-based sample count

```
int samples = max(int(base_samples / (1.0 + base_samples *
     ndc_linear_depth)), min_samples);
```

which can improve performance by sampling less at greater distances. However, this scheme conflicts with our later comparisons and have been disabled.

5.2.6 [Mittring 2012]

Similar to [Bavoil et al. 2008a] in theory but the implementations differ as we don't need to ray-march (See Figure 4.3f). Also, we use the sample pairs to conquer depth buffer discontinuities as proposed in [Ownby et al. 2010]. For both samples, the angle is found using vector mathematics

```
vec3 s = normalize(wc_sample - wc_position);
vec3 v = normalize(-vertex.wc_camera_ray_direction);

float vn = dot(v, wc_normal);
float vs = dot(v, s);
float sn = dot(s, wc_normal);

// Cap to tangent plane
vec3 tangent = normalize(s - sn * wc_normal);
float cos_angle = (0.0 <= sn) ? vs : dot(v, tangent);
```

where s is the vector $\mathbf{s} - \mathbf{p}$; v is the view vector and wc_normal is the normal. All variables are in world coordinates. In the end, we have to find the angle directly with the $acos$ function. However, this function is slow and we have opted to replace it with an approximation

```
float acos_approximation( float x )
{
   return (-0.69813170079773212 * x * x -
        0.87266462599716477) * x + 1.5707963267948966;
}
```

This was suggested on the blog AltDevBlogADay[2].

5.3 The Geometry-Aware Separable Blur

The separable blur is very straight-forward. It works as a normal blur filter but with both depth and normal differences weighing each sample's contribution

```
result = texture(source, tc);
float weightSum = 1.0;

for (int i = -1; i >= -samples_in_each_direction; --i)
{
    vec2 offset = vec2(float(i), 0).DIRECTION_SWIZZLE *
         inverted_source_size;

    float normalWeight = pow(dot(texture(wc_normals, tc +
         offset).xyz, texture(wc_normals, tc).xyz) * 0.5 +
         0.5, normalPower);
    float positionWeight = 1.0 / pow(1.0 + abs(ec_depth(tc) -
         ec_depth(tc + offset)), positionPower);
    float weight = normalWeight * positionWeight;

    result += texture(source, tc + offset) * weight;
    weightSum += weight;
}

for (int i = 1; i <= samples_in_each_direction; ++i)
{
    ... // Symmetrically
}

result /= weightSum;
```

where the constants $normal\_power$ and $depth\_power$ controls the weighing. They are scene dependent, and we have found that 10.0 and 0.5 works well for the sponza, respectively. Recall, that a separable blur filter first blurs in the horizontal and subsequently in the vertical direction. The $DIRECTION\_SWIZZLE$ is there because the shader source code is almost identical for both directions. It is toggled when compiling to for each pass as follows

```
#if defined HORIZONTAL
    #define DIRECTION_SWIZZLE xy
#elif defined VERTICAL
    #define DIRECTION_SWIZZLE yx
#else
```

[2] http://www.altdevblogaday.com/2012/10/12/angle-based-ssao/

```
    "Define HORIZONTAL or VERTICAL before compiling this
        shader!";
#endif
```

6 Results and Findings

In this chapter we will present our the results of our comparisons and reflect at the findings.

6.1 Performance Configuration

We evaluate performance characteristics for two different views. The first view goes deep into the scene while the second is a close-up of a stone wall relief. The results are in Tables 2 and 3, respectively. We encourage readers of the PDF document to zoom in and view the images in detail. The **output** images are those rendered by the method with the settings of the **parameter** and **value** columns. The **reference** images are those ray-traced in Mental-Ray. The **raw** images are also by the method but without the blur pass and without the AO' post-processing function (as discussed in section 4.1.12). We aim to match the look of the reference by tweaking blur and AO' parameters. The raw images are provided to explain the workings of each method in further detail.

We have already outlined how we went about configuring the methods in the design section.

6.1.1 Precision

Let us first start by nothing that the performance timings are with a precision of ± 1 ms because of technical details that are hard to work out. We ran the application several times and found that the results varied (with ± 1 ms) between runs even though the settings were the same. We attribute this difference to technical details that are beyond our control. Nevertheless, we are still within a reasonable range of precision to make qualified statements.

6.1.2 Deep View Findings

Raw Results First and foremost, it is important to note how essential it is to tweak the raw AO computations to match the reference. All methods had to be adjusted even though some came naturally closer than others. This is not a surprising fact as SSAO methods are approximate in general. Some authors actually value the artistic control of functions such as AO' along with method specific parameters [McGuire et al. 2011]. Nevertheless, it is interesting to see which methods came naturally closest. The category of horizon-based methods seem to produce a lighter raw result. This under-occlusion might be attributed to the fact that they assume the horizon is completely clear. That is, they work under the continuous hight-field assumption which we have seen doesn't hold up every in Figure 4.1. The two sphere-sampling methods produce raw images that look very dark. However, as previously mentioned this is because half of the samples are known to be occluded. The effect is removed by setting $a = 0.5$. It is interesting to note how the concave geometry lights up in [Mittring 2007a] but not in [Loos and Sloan 2010]. This is because the latter method has been complemented with paired sampling—the lower bound removes the highlights as well.

Tweaked Output Halos are prominently present with [Mittring 2007a] as it does not deal with depth buffer discontinuity at all. The rest of the methods do so and resultingly, they have no halos. [Mittring 2007a] produces a result in general is akin to AO. However, it is more grainy because of the noise and blur and it seems somewhat overoccluded. However, that is most likely due to the halos. The [Filion and McNaughton 2008] method shares the grainy and blurred feel but solves the halo problem by introducing a falloff function.

The horizon-based methods fare fairly after they have been darked by a high c parameter. That is, their underoccluded raw images can be easily tweaked to give pleasing results. This fact may be attributed to the fact that they are based on a model for AO that is very close to the original definition. In fact, the only mathematical difference is due the continous hightfield assumption. Interestingly enough, [Bavoil et al. 2008a] is also the method that best captures the definitions between the leaves in the flower pots.

Alchemy AO also stands out as a good SSAO candidate. However, it does suffer from underocclusion in general too. We ponder that the problem may reside with the fact that the sampling scheme is designed to find occluders only. If many samples fail in regions the result is an undersampled AO approximation which in turn leads to too few occluders being found. The result is an underoccluded image. Again, the missing shadows from the topstory curtains are a good example of this.

Volumetric AO seems to suffer from the reverse problem: It is generally overoccluded. This may be attributed to the paired sampling scheme intended to fix undersampling. This scheme works under the assumption that surfaces are flat in general. When the assumption fails, the output will use erroneous sample values. Another reason may be due to the sampling approach. The method uses samples distributed in a disk projected into the scene and then assumes a sphere can be constructed around it. However, a disk projects to an ellipsis and not another disk. As an ellipsis can't be the base of a sphere, the method is biased. Despite of all this, Volumetric AO produces visually pleasing results.

The Numbers Maybe not surprisingly, the two oldest methods (the point sampling methods) are among the slowest while they simultenously produces the most grainy and blurred results. Some improvement can be seen when the hemisphere is used [Filion and McNaughton 2008]: Only 15 samples are needed as opposed to 24 for the full sphere. This is however not affected in the AO performance (3.75 ms vs 3.5) but that may be due to extra computations required to transform each sample from world to screen space. Also, please recall that the precision of the timings is ± 1 ms.

Unfortunately, the otherwise good-looking horizon-based approaches perform just as badly. However, note that there is a large difference in the number of samples needed between [Bavoil et al. 2008a] (8×6 samples) and [Mittring 2012] (6 samples). That holds even though the latter has somewhat appearance. This may not be surprising since it is 4 years older.

Standing out as the clear performance winners are Alchemy AO (2.75 ms) and Volumetric AO (1.85 ms). Simultaneously, these among the methods that require the smallest blur kernels. Volumetric AO beats Alchemy AO in terms of total performance because it requires only a single-pixel wide blur.

6.1.3 Close View

The same general observations as noted above also hold in the close-up of the lion-head. Alchemy AO and Volumetric AO are still the performance kings. In general, performance stays the same as in the deep view. In terms of quality however, things have changed for some methods.

The overocclusion noted previously with Volumetric AO has become much more visible in the details around the lion head. Even to an extent were they are visually unpleasing. One could argue that a single-pixel wide blur is not enough for this method if the viewer is also supposed to view objects close up. Another reason for this overocclusion is the underlying assumption of Volumetric AO: A ray traced in any direction from the center of the sample sphere only hits a single surface. With small ridges and reliefed detail seen in close-up like here, this assumption fails within the radius of the sample sphere.

The horizon-based methods still somewhat maintains the quality level. That being said, [Bavoil et al. 2008a] seem to highlight the underlying polygon surfaces. However, if we take a look at the ray-traced reference, this is actually the expected result. The [Mittring 2012] doesn't exhibit this property. Subjectively speaking, however, the result looks better without the polygon definition. [Mittring 2012] also has problems with strange halo-like artifacts. It seems like the flat surface assumption doens't hold up well for the round lion head.

Again, Alchemy AO looks great. Though this time around the resemblence with the ray-traced reference is not near the same. The symptomps are reversed of before and there is overocclusion were there shouldn't be. We suspect this may be the result of the specially-chosen ρ function. ρ in alchemy represents an inverse power function whereas the ray-traced reference uses linear attenuation. Consequently, the results will differ and especially for close distances as found in this setting.

6.2 Quality Configuration

We will now look into how each method fares when the aim is to produce highest quality results. See Table 4 for the results. We only use a single view here, as we have already discussed the implications of close-ups in the previous section. It should be noted that no post-process blur is used. The AO models stand on their own.

It is immediately noticable that something is wrong with [Bavoil et al. 2008a]. A gradient seems to have crept up along the floor and walls which shouldn't be there. We have not been able to identify the issue at hand but it must be within our implementation is no gradients have been reported in the literature. It should be noted that for the same amount of samples and a larger sample radius, the problem completely dissapears. It may be that we have run into an unfortunate combination of parameters. Anyhow, at 64 ms this is not really a candidate for the superior model!

Anyhow, the other methods perform remarkably well. One artifact common to all the methods in this configuration is overocclusion (see the shadow between the curtain and the wall). We attribute this to the factors already listed for overocclusion in the sections above. Alchemy AO is the only method which exhibits the least amount of overocclusion. This is not surprising as we found it to be underoccluding before. We theorize that the extra amount of samples diminishes the effect of undersampling due to rejected samples. Consequently, more occluders are found and the result is darker.

Most notable overall is the performance characteristics. Volumetric AO stands out by only using 6 ms in its best configuration. Alchemy AO (12 ms) and and the [Mittring 2012] horizon-method (16) ms are also good performers. It can be directly related to the fact the all 3 methods require the least extra samples over the performance configuration counterparts. [Mittring 2007a] performs the worst and uses an astonishing 200 samples. By sampling there hemisphere instead, [Filion and McNaughton 2008] accomplishes similar results with only 80 samples.

6.3 Scalability

Last comes the issue of scalability. That is, how the methods reacts to increasing the sample radius. To test this, we have used the performace configuration as that is the one likely to be used in a production setting. We double the sample radius to 20.0 world coordinate units. The results are in Table 5.

Performance-wise, all the methods take a slight hit but nothing major. Certainly out of the ± 1 ms precision range. We theorize that the computational overhead of space transformations and the like simply outweighs the effect of increasing the sample radius.

Quality, however, is the more concerning factor. In general, all the methods suffer a bit by deviating more from the reference image. With that being said, Alchemy AO stands out as being closest to the reference.

6.4 The Superior Method

We only went wthing the bounds of the 2 ms industry average render budget for one method: Alchemy AO. Following the discussion above, there is no doubt that this method is superior in terms of both performance and quality. We attribute the quality to the strong mathematical foundation that it builds upon which never deviates from the original definition of AO (Equation 2.5). It never makes any further assumptions that may break when it is applicable. The only real concern is the underocclusion which occurs as an effect of under-sampling. As an added benefit, it also scales superiorly compared to the competition. The horizon-based methods have a similarly sound mathematical foundation that doesn't deviate from the orignal AO definition. However, they suffer in terms of performance—they strategy simply doesn't compete with Alchemy AO. The early point sampling methods are completely out of the line in terms of both performance and scalability. They were interesting to include in the comparison to how much the field of SSAO really has improved over time. Volumetric AO takes the second-place. It has problems with over-occlusion but strong performance characteristics.

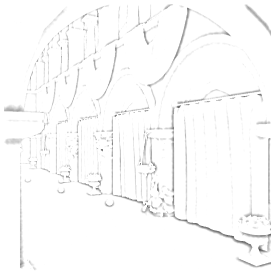
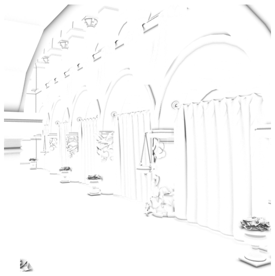
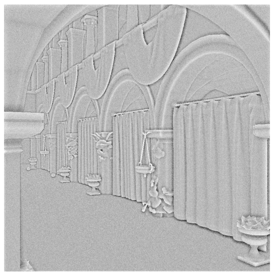
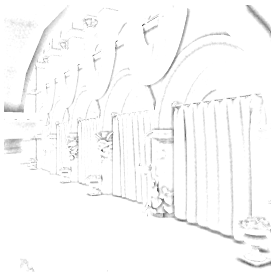
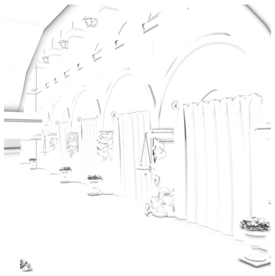
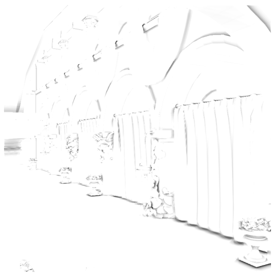
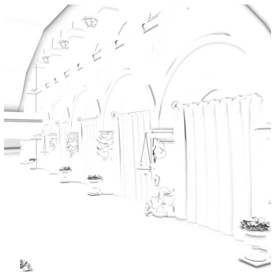
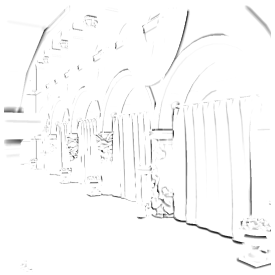
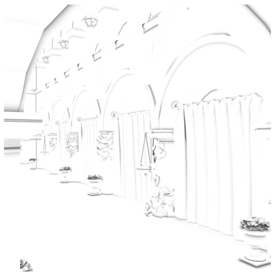
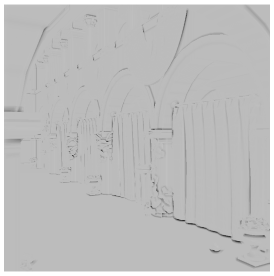
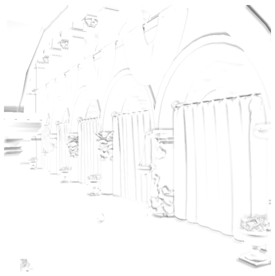
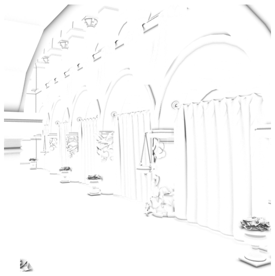
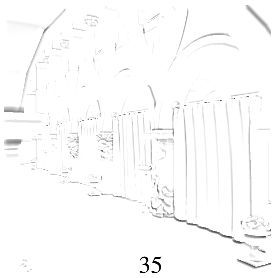
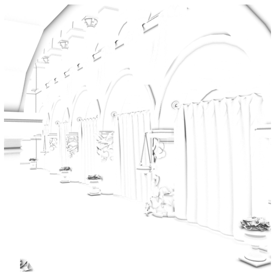
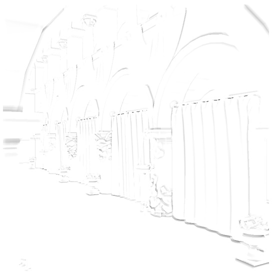
Parameter	Value	Output	Reference	Raw
Method	[Mittring 2007a]			
Sampling Radius	10.0			
Samples	24			
AO′ Parameters	$a = 0.57, b = 1.0, c = 5.0$			
Blur size	3			
Performance	7.5 ms (AO 3.5, Blur 4)			
Method	[Filion and McNaughton 2008]			
Sampling Radius	10.0			
Samples	15			
AO′ Parameters	$a = 0.0, b = 1.12, c = 2.5$			
Blur size	2			
Performance	6.5 ms (AO 3.75, Blur 2.75)			
Method	[Bavoil et al. 2008a]			
Sampling Radius	10.0			
Samples	8 (6 ray-march steps)			
AO′ Parameters	$a = 0.0, b = 1.03, c = 4.0$			
Blur size	1			
Performance	8.5 ms (AO 6.75, Blur 1.75)			
Method	[Loos and Sloan 2010]			
Sampling Radius	10.0			
Samples	24			
AO′ Parameters	$a = 0.5, b = 1.01, c = 8.0$			
Blur size	1			
Performance	4.1 ms (AO 2.75, Blur 1.35)			
Method	[McGuire et al. 2011]			
Sampling Radius	10.0			
Samples	6			
AO′ Parameters	$a = 0.0, b = 1.005, c = 5.0$			
Blur size	2			
Performance	4.75 ms (AO 1.85, Blur 2.9)			
Method	[Mittring 2012]			
Sampling Radius	10.0			
Samples	6			
AO′ Parameters	$a = 0.0, b = 1.0, c = 3.0$			
Blur size	2			
Performance	6.1 ms (AO 3.1, Blur 3.0)			

Table 2: *Performance configurations, deep view.*

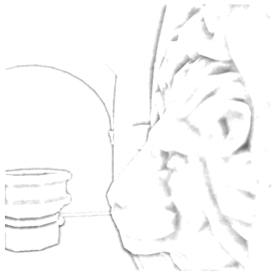
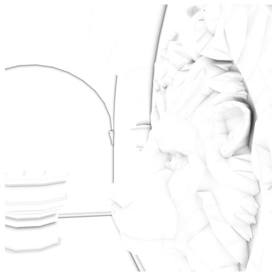
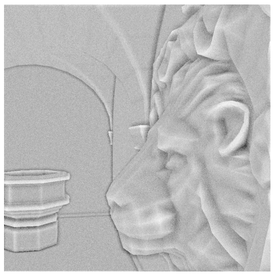
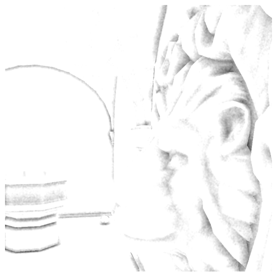
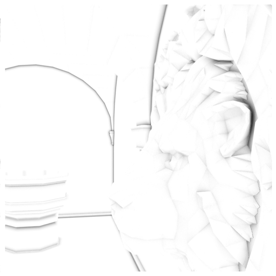
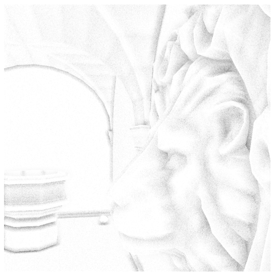
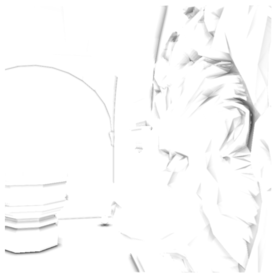
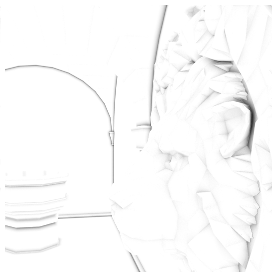
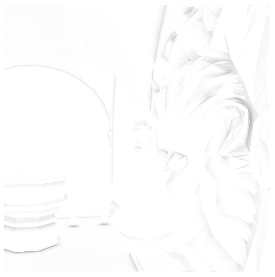
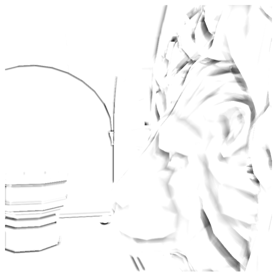
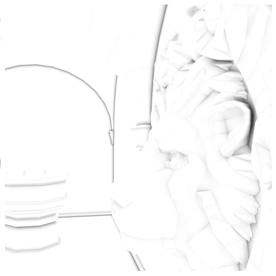
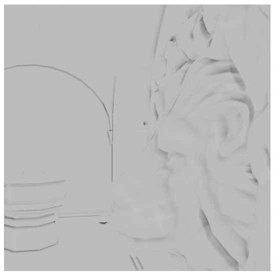
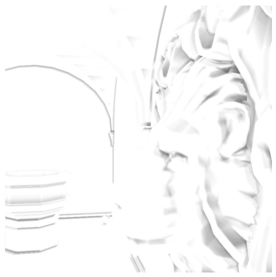
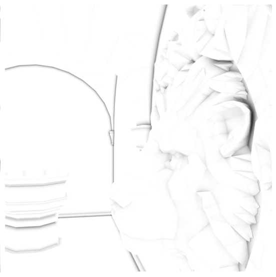
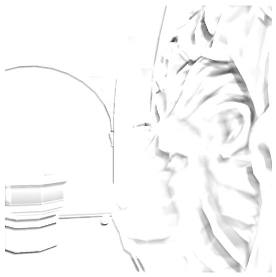
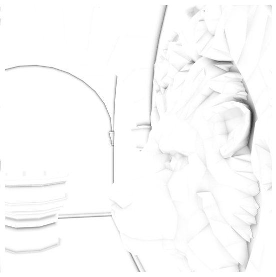
Parameter	**Value**	**Output**	**Reference**	**Raw**
Method	[Mittring 2007a]			
Sampling Radius	10.0			
Samples	24			
AO' Parameters	$a = 0.57, b = 1.0, c = 5.0$			
Blur size	3			
Performance	7.5 ms (AO 3.5, Blur 4)			
Method	[Filion and McNaughton 2008]			
Sampling Radius	10.0			
Samples	15			
AO' Parameters	$a = 0.0, b = 1.12, c = 2.5$			
Blur size	2			
Performance	7.0 ms (AO 4.0, Blur 3.0)			
Method	[Bavoil et al. 2008a]			
Sampling Radius	10.0			
Samples	8 (6 ray-march steps)			
AO' Parameters	$a = 0.0, b = 1.03, c = 4.0$			
Blur size	1			
Performance	8.2 ms (AO 6.6, Blur 1.6)			
Method	[Loos and Sloan 2010]			
Sampling Radius	10.0			
Samples	24			
AO' Parameters	$a = 0.5, b = 1.01, c = 8.0$			
Blur size	1			
Performance	4.4 ms (AO 2.9, Blur 1.5)			
Method	[McGuire et al. 2011]			
Sampling Radius	10.0			
Samples	6			
AO' Parameters	$a = 0.0, b = 1.005, c = 5.0$			
Blur size	2			
Performance	4.65 ms (AO 1.75, Blur 2.9)			
Method	[Mittring 2012]			
Sampling Radius	10.0			
Samples	6			
AO' Parameters	$a = 0.0, b = 1.0, c = 3.0$			
Blur size	2			
Performance	6.0 ms (AO 3.0, Blur 3.0)			

Table 3: *Performance configurations, close view.*

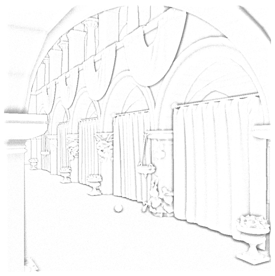
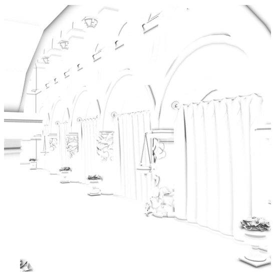
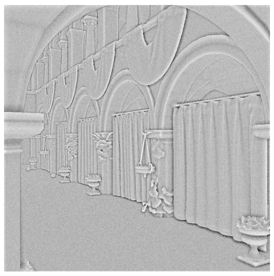
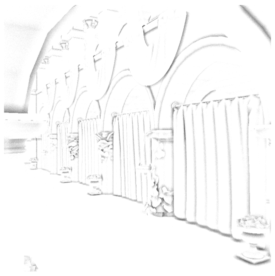
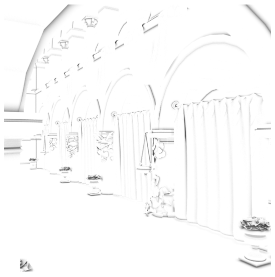
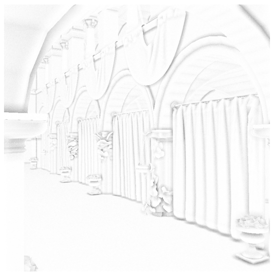
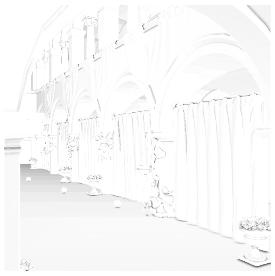
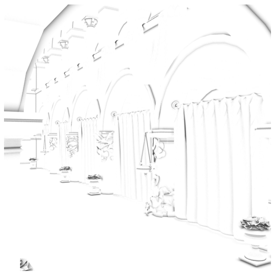
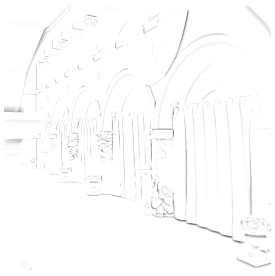
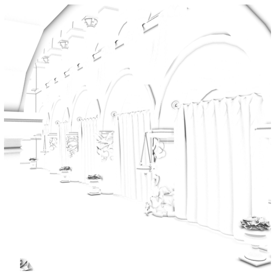
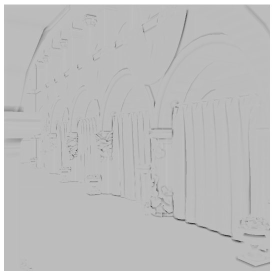
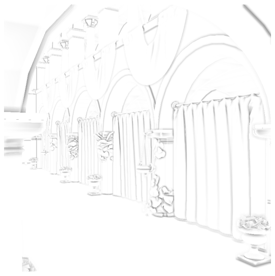
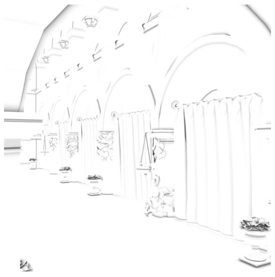
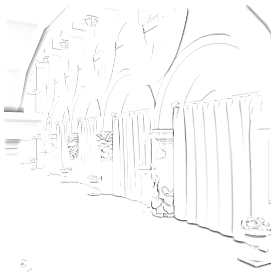
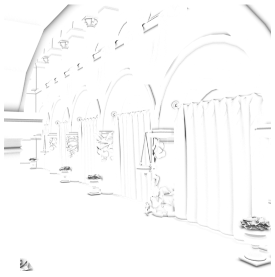
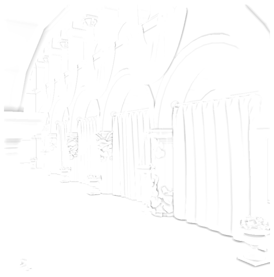
Parameter	**Value**	**Output**	**Reference**	**Raw**
Method *Sampling Radius* *Samples* *AO′ Parameters* *Blur size* *Performance*	[Mittring 2007a] 10.0 200 $a = 0.55, b = 1.0, c = 4.0$ N/A 25 ms			
Method *Sampling Radius* *Samples* *AO′ Parameters* *Blur size* *Performance*	[Filion and McNaughton 2008] 10.0 80 $a = 0.0, b = 1.1, c = 2$ N/A 19 ms			
Method *Sampling Radius* *Samples* *AO′ Parameters* *Blur size* *Performance*	[Bavoil et al. 2008a] 10.0 80 (6 ray-march steps) $a = 0.0, b = 1.0, c = 3.0$ N/A 64 ms			
Method *Sampling Radius* *Samples* *AO′ Parameters* *Blur size* *Performance*	[Loos and Sloan 2010] 10.0 40 $a = 0.5, b = 1.0, c = 3.0$ N/A 6 ms			
Method *Sampling Radius* *Samples* *AO′ Parameters* *Blur size* *Performance*	[McGuire et al. 2011] 10.0 64 $a = 0.0, b = 1.0, c = 5.0$ N/A 12 ms			
Method *Sampling Radius* *Samples* *AO′ Parameters* *Blur size* *Performance*	[Mittring 2012] 10.0 32 $a = 0.0, b = 1.0, c = 3.0$ N/A 16 ms			

Table 4: *Quality configurations, deep view.*

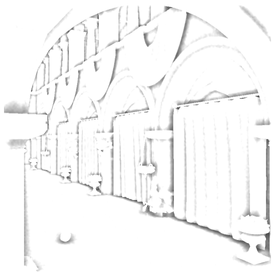
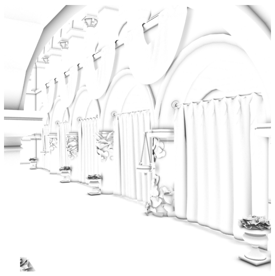
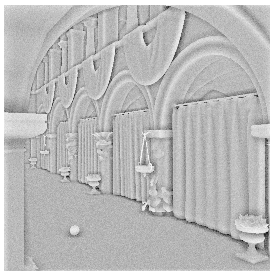
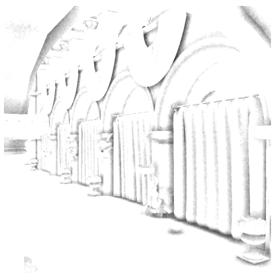
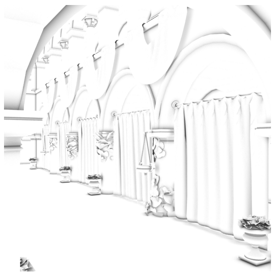
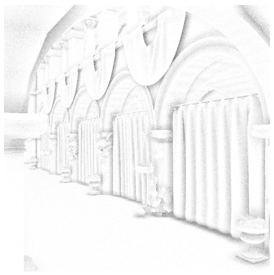
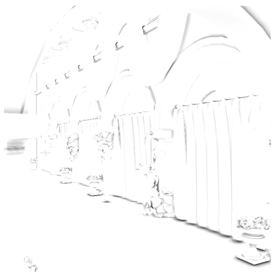
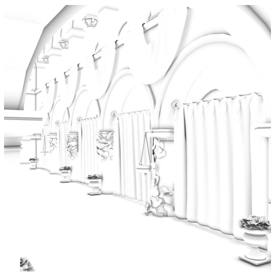
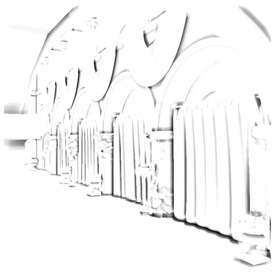
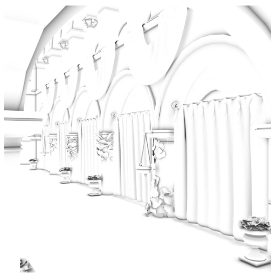
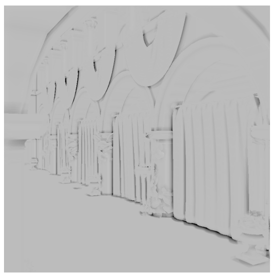
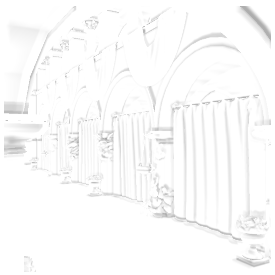
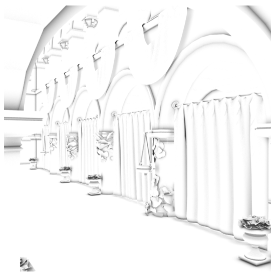
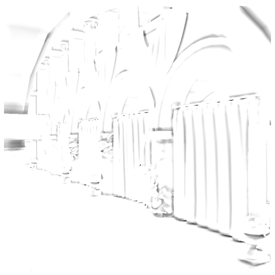
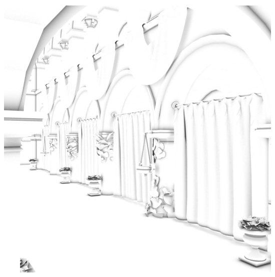
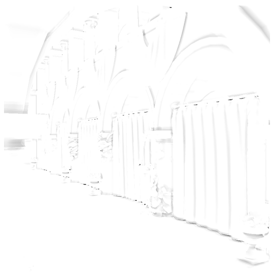
Parameter	Value	Output	Reference	Raw
Method	[Mittring 2007a]			
Sampling Radius	20.0			
Samples	24			
AO′ Parameters	$a = 0.57, b = 1.0, c = 5.0$			
Blur size	3			
Performance	7.5 ms (AO 3.5, Blur 4)			
Method	[Filion and McNaughton 2008]			
Sampling Radius	20.0			
Samples	15			
AO′ Parameters	$a = 0.0, b = 1.12, c = 2.5$			
Blur size	2			
Performance	6.7 ms (AO 4.0, Blur 2.7)			
Method	[Bavoil et al. 2008a]			
Sampling Radius	20.0			
Samples	8 (6 ray-march steps)			
AO′ Parameters	$a = 0.0, b = 1.03, c = 4.0$			
Blur size	1			
Performance	8.4 ms (AO 6.8, Blur 1.6)			
Method	[Loos and Sloan 2010]			
Sampling Radius	20.0			
Samples	24			
AO′ Parameters	$a = 0.5, b = 1.01, c = 8.0$			
Blur size	1			
Performance	4.65 ms (AO 3.1, Blur 1.55)			
Method	[McGuire et al. 2011]			
Sampling Radius	20.0			
Samples	6			
AO′ Parameters	$a = 0.0, b = 1.005, c = 5.0$			
Blur size	2			
Performance	4.85 ms (AO 1.95, Blur 2.9)			
Method	[Mittring 2012]			
Sampling Radius	20.0			
Samples	6			
AO′ Parameters	$a = 0.0, b = 1.0, c = 3.0$			
Blur size	2			
Performance	6.5 ms (AO 3.5, Blur 3.0)			

Table 5: *Scalability using the performance configuration, deep view.*

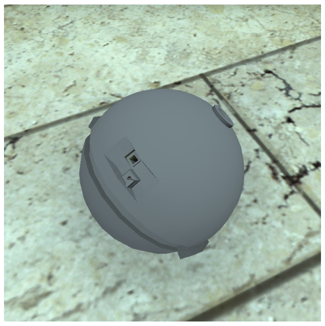


Figure 7.1: *Animated spheres used to test for temporal coherence. This topic was left out of the report.*

7 Discussion

7.1 Extensions to SSAO

In the discussion we outlined som extensions applicable to many SSAO methods. One was the use of directional occlusion (SSDO), another was the use of bent normals, and a third was to go multi-resolution. Some methods integrate better bent normals, SSDO, and multi-resolution. than others. We have not touched upon how such extensions integrate with the presented SSAO methods. This could be an interesting topic for a follow-up report.

Temporal coherence is also an extension that has gotten a lot of academic interest and production use. In this report, we have focussed on static scenes (though you may occassionally have noticed an animated sphere in the screenshots, see Figure 7.1). However, real virtual worlds are dynamic and the SSAO methods must also work in these environments.

8 Conclusion

We have presented a large array of SSAO methods and implemented six of them for comparison. Along the way we have uncovered many vastly different approaches to AO and analyzed the strenghts and weaknesses of each. Based on a thorough comparison, we found that Alchemy AO was the superior method overall. However, as the field of SSAO has only existed for 6 years, many methods remain to be discovered. As such, there should hopefully be work for another comparison in the future. As of now, we will recommend the use of Alchemy AO untill a superior method surfaces. We mentioned in the discussion that competing global illumination methods are on the rise. It would be an equally interesting topic for a future study to see how these methods approach AO. We can only hope of the day that we won't need the trickery that is SSAO and instead use real-time ray-tracing to visualize our virtual worlds.

References

AKSOY, V., AND PHANEUF, G. 2011. Character shading in ea sports mma^{TM} using projected poisson disk based ambient occlusion. In *ACM SIGGRAPH 2011 Posters*, ACM, New York, NY, USA, SIGGRAPH '11, 17:1–17:1.

BAVOIL, L., AND SAINZ, M. 2008. Screen space ambient occlusion.

BAVOIL, L., AND SAINZ, M. 2009. Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH 2009: Talks*, ACM, New York, NY, USA, SIGGRAPH '09, 45:1–45:1.

BAVOIL, L., SAINZ, M., AND DIMITROV, R. 2008. Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 talks*, ACM, New York, NY, USA, SIGGRAPH '08, 22:1–22:1.

BAVOIL, L., SAINZ, M., AND DIMITROV, R., 2008. Image-space horizon-based ambient occlusion.

BUKOWSKI, M., HENNESSY, P., MCGUIRE, M., AND OSMAN, B. 2012. Scalable high-quality motion blur and ambient occlusion.

COOK, R. L., AND TORRANCE, K. E. 1982. A reflectance model for computer graphics. *ACM Trans. Graph. 1*, 1 (Jan.), 7–24.

DACHSBACHER, C., AND KAUTZ, J. 2009. Real-time global illumination for dynamic scenes. In *ACM SIGGRAPH 2009 Courses*, ACM, New York, NY, USA, SIGGRAPH '09, 19:1–19:217.

DIMITROV, R., BAVOIL, L., AND SAINZ, M. 2008. Horizon-split ambient occlusion. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '08, 5:1–5:1.

DONZALLAZ, P.-Y., AND SOUSA, T., 2011. Lighting in crysis 2.

FILION, D., AND MCNAUGHTON, R. 2008. Effects & techniques. In *ACM SIGGRAPH 2008 Games*, ACM, New York, NY, USA, SIGGRAPH '08, 133–164.

HATTORI, T., KUBO, H., AND MORISHIMA, S. 2010. Curvature depended local illumination approximation of ambient occlusion. In *ACM SIGGRAPH 2010 Posters*, ACM, New York, NY, USA, SIGGRAPH '10, 122:1–122:1.

HATTORI, T., KUBO, H., AND MORISHIMA, S. 2011. Real time ambient occlusion by curvature dependent occlusion function. In *SIGGRAPH Asia 2011 Posters*, ACM, New York, NY, USA, SA '11, 48:1–48:1.

HOANG, T.-D., AND LOW, K.-L. 2010. Multi-resolution screen-space ambient occlusion. In *Proceedings of the 17th ACM Symposium on Virtual Reality Software and Technology*, ACM, New York, NY, USA, VRST '10, 101–102.

HOANG, T.-D., AND LOW, K.-L. 2012. Efficient screen-space approach to high-quality multi-scale ambient occlusion. In *The Visual Computer, 28(3): 289-304, 2012*.

HUANG, J., BOUBEKEUR, T., RITSCHEL, T., HOLLÄNDER, M., AND EISEMANN, E. 2011. Separable approximation of ambient occlusion. In *Short paper at Eurographics*.

KAJIYA, J. T. 1986. The rendering equation. *SIGGRAPH Comput. Graph. 20*, 4 (Aug.), 143–150.

KAPLANYAN, A., 2010. Cryengine 3: Reaching the speed of light.

KASYAN, N., SCHULZ, N., AND SOUSA, T., 2011. Secrets of cryengine 3 graphics technology.

KLEHM, O., RITSCHEL, T., EISEMANN, E., AND SEIDEL, H.-P. 2011. Bent normals and cones in screen-space. In *Vision, Modeling and Visualization Workshop*.

LANDIS, H. 2002. Production-Ready Global Illumination. In *Siggraph Course Notes*, vol. 16.

LANGER, M. S., AND BÜLTHOFF, H. H. 2000. Depth discrimination from shading under diffuse lighting. *Perception 29*, 6, 649–660.

LANGER, M. S., AND ZUCKER, S. W. 1994. Shape-from-shading on a cloudy day. *Journal of the Optical Society of America A 11*, 2, 467.

LOOS, B. J., AND SLOAN, P.-P. 2010. Volumetric obscurance. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '10, 151–156.

LUFT, T., COLDITZ, C., AND DEUSSEN, O. 2006. Image enhancement by unsharp masking the depth buffer. *ACM Transactions on Graphics 25*, 3 (jul), 1206–1213.

MATTAUSCH, O., SCHERZER, D., AND WIMMER, M., 2010. High-quality screen-space ambient occlusion using temporal coherence, Dec.

MCGUIRE, M., OSMAN, B., BUKOWSKI, M., AND HENNESSY, P. 2011. The alchemy screen-space ambient obscurance algorithm. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ACM, New York, NY, USA, HPG '11, 25–32.

MCGUIRE, M., MARA, M., AND LUEBKE, D. 2012. Scalable ambient obscurance. In *High-Performance Graphics 2012*.

MITTRING, M. 2007. Finding next gen: Cryengine 2 (course notes). In *ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, SIGGRAPH '07, 97–121.

MITTRING, M. 2007. Finding next gen: Cryengine 2 (slides). In *ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, SIGGRAPH '07, 97–121.

MITTRING, M., 2012. The technology behind the unreal engine 4 elemental demo.

MÖLLER, T., HAINES, E., AND HOFFMAN, N. 2008. Real-time rendering. 425–430.

NGUYEN, H. 2007. *Gpu gems 3*, first ed. Addison-Wesley Professional.

OWNBY, J.-P., HALL, C., AND HALL, R., 2010. Toy story 3: The video game rendering techniques.

PHARR, M., AND FERNANDO, R. 2005. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional.

RITSCHEL, T., GROSCH, T., AND SEIDEL, H.-P. 2009. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '09, 75–82.

RITSCHEL, T., DACHSBACHER, C., GROSCH, T., AND KAUTZ, J. 2012. The state of the art in interactive global illumination. *Comput. Graph. Forum 31*, 1 (Feb.), 160–188.

SAINZ, M. 2008. Real-time depth buffer based ambient occlusion.

SCHERZER, D., YANG, L., AND MATTAUSCH, O. 2010. Exploiting temporal coherence in real-time rendering. In *ACM SIGGRAPH ASIA 2010 Courses*, ACM, New York, NY, USA, SA '10, 24:1–24:26.

SHANMUGAM, P., AND ARIKAN, O. 2007. Hardware accelerated ambient occlusion techniques on gpus. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '07, 73–80.

SMEDBERG, N., AND WEIGHT, D., 2009. Rendering techniques in gears of war 2.

SONG, Y., LIU, F., AND XU, J. 2010. Horizon-based screen-space ambient occlusion using mixture sampling. In *ACM SIGGRAPH ASIA 2010 Posters*, ACM, New York, NY, USA, SA '10, 50:1–50:1.

SOURIMANT, G., GAUTRON, P., AND MARVIE, J.-E. 2011. Poisson disk ray-marched ambient occlusion. In *Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '11, 210–210.

SZIRMAY-KALOS, L., UMENHOFFER, T., TÓTH, B., SZÉCSI, L., AND SBERT, M., 2009. Volumetric ambient occlusion.

WHITE, J., AND BARRÉ-BRISEBOIS, C., 2011. More performance! five rendering ideas from battlefield 3 and need for speed: The run.

ZHUKOV, S., INOES, A., AND KRONIN, G. 1998. An Ambient Light Illumination Model. In *Rendering Techniques '98*, Springer-Verlag Wien New York, G. Drettakis and N. Max, Eds., Eurographics, 45–56.

9 Appendix

9.1 Shaders

We have listed the full fragment shader implementations below. The accompanying vertex shaders are simple and have been omitted here. Please clone the git repository at `https://github.com/frederikaalund/sfj` for further reference.

Listing 1: *CryEngine2 SSAO*

```
uniform sampler2D depths;
uniform sampler2D random_texture;

uniform vec3 wc_camera_eye_position;
uniform float z_far;

uniform vec2 tc_window;

uniform mat4 projection_matrix;




struct vertex_data
{
        vec3 wc_camera_ray_direction;
};
noperspective in vertex_data vertex;

out vec4 ambient_occlusion;



float ec_depth( in vec2 tc )
{
        float buffer_z = texture(depths, tc).x;
        return projection_matrix[3][2] / (-2.0 * buffer_z + 1.0 - projection_matrix[2][2]);
}



void main()
{
        vec2 tc_depths = gl_FragCoord.xy / tc_window;
        float ec_depth_negated = -ec_depth(tc_depths);
        vec3 wc_position = wc_camera_eye_position + vertex.wc_camera_ray_direction *
            ec_depth_negated / z_far;

        ambient_occlusion.a = 0.0f;
        const float radius = 10.0;
        const int samples = 200;

        float projection_scale_xy = 1.0 / ec_depth_negated;
        float projection_scale_z = 100.0 / z_far * projection_scale_xy;

        float scene_depth = texture(depths, tc_depths).x;

        vec2 inverted_random_texture_size = 1.0 / vec2(textureSize(random_texture, 0));
        vec2 tc_random_texture = gl_FragCoord.xy * inverted_random_texture_size;

        vec3 random_direction = texture(random_texture, tc_random_texture).xyz;
        random_direction = normalize(random_direction * 2.0 - 1.0);

        for (int i = 0; i < samples; ++i)
        {
                vec3 sample_random_direction = texture(random_texture, vec2(float(i) *
                    inverted_random_texture_size.x, float(i / textureSize(random_texture, 0).x) *
                    inverted_random_texture_size.y)).xyz;
                sample_random_direction = sample_random_direction * 2.0 - 1.0;
                sample_random_direction = reflect(sample_random_direction, random_direction);

                vec3 tc_sample_pos = vec3(tc_depths.xy, scene_depth)
                        + vec3(sample_random_direction.xy * projection_scale_xy,
                            sample_random_direction.z * scene_depth * projection_scale_z) * radius;

                float sample_depth = texture(depths, tc_sample_pos.xy).x;

                ambient_occlusion.a += float(sample_depth > tc_sample_pos.z);
        }

        ambient_occlusion.a /= float(samples);
}
```

Listing 2: *Starcraft II SSAO*

```
uniform sampler2D depths;
uniform sampler2D wc_normals;
uniform sampler2D random_texture;

uniform vec3 wc_camera_eye_position;
uniform float z_far;

uniform vec2 tc_window;

uniform mat4 view_matrix;
uniform mat4 projection_matrix;
uniform mat4 view_projection_matrix;



struct vertex_data
{
        vec3 wc_camera_ray_direction;
};
noperspective in vertex_data vertex;

out vec4 ambient_occlusion;



float ec_depth( in vec2 tc )
{
        float buffer_z = texture(depths, tc).x;
        return projection_matrix[3][2] / (-2.0 * buffer_z + 1.0 - projection_matrix[2][2]);
}



void main()
{
        vec2 tc_depths = gl_FragCoord.xy / tc_window;
        vec3 wc_normal = texture(wc_normals, tc_depths).xyz;
        vec3 wc_position = wc_camera_eye_position + vertex.wc_camera_ray_direction * -ec_depth(
            tc_depths) / z_far;

        ambient_occlusion.a = 0.0;
        //vec3 bent_normal = vec3(0.0);
        const float radius = 10.0;
        const int samples = 80;

        float scene_depth = texture(depths, tc_depths).x;

        vec2 inverted_random_texture_size = 1.0 / vec2(textureSize(random_texture, 0));
        vec2 tc_random_texture = gl_FragCoord.xy * inverted_random_texture_size;

        vec3 random_direction = texture(random_texture, tc_random_texture).xyz;
        random_direction = normalize(random_direction * 2.0 - 1.0);

        for (int i = 0; i < samples; ++i)
        {
                vec3 sample_random_direction = texture(random_texture, vec2(float(i) *
                    inverted_random_texture_size.x, float(i / textureSize(random_texture, 0).x) *
                    inverted_random_texture_size.y)).xyz;
                sample_random_direction = sample_random_direction * 2.0 - 1.0;
                sample_random_direction = reflect(sample_random_direction, random_direction);
                sample_random_direction = faceforward(sample_random_direction,
                    sample_random_direction, -wc_normal);

                vec3 wc_sample = wc_position + sample_random_direction * radius;
                vec3 ec_sample = (view_matrix * vec4(wc_sample, 1.0)).xyz;
                vec4 cc_sample = view_projection_matrix * vec4(wc_sample, 1.0);
                vec3 ndc_sample = cc_sample.xyz / cc_sample.w;
                vec2 tc_sample = (ndc_sample.xy + vec2(1.0)) * 0.5;

                float scene_depth = ec_depth(tc_sample);
                float sample_depth = ec_sample.z;

                float depth_difference = scene_depth - sample_depth;

                // CryEngine2 rho
                //float rho = (depth_difference <= 0.0 || depth_difference > radius) ? 1.0 : 0.0;

                float rho = clamp((depth_difference - radius) / depth_difference, 0.0, 1.0);
                ambient_occlusion.a += rho;
                //bent_normal += normalize(sample_random_direction) * rho;
        }

        //bent_normal = normalize(bent_normal) * 0.5 + 0.5;
        //ambient_occlusion.rgb = bent_normal;
        ambient_occlusion.a /= float(samples);
}
```

Listing 3: *HBAO*

```
#define USE_RANDOM_DIRECTION 0

uniform sampler2D depths;
uniform sampler2D wc_normals;
uniform sampler2D random_texture;

uniform vec3 wc_camera_eye_position;
uniform float z_far;

uniform vec2 tc_window;

uniform mat4 view_matrix;
uniform mat4 projection_matrix;
uniform mat4 view_projection_matrix;
uniform mat4 inverse_view_projection_matrix;



struct vertex_data
{
        vec3 wc_camera_ray_direction;
};
noperspective in vertex_data vertex;

out vec4 ambient_occlusion;



float tc_depth( in vec2 tc )
```

```
{
        return texture(depths, tc).x;
}

float ec_depth( in vec2 tc )
{
        float buffer_z = texture(depths, tc).x;
        return projection_matrix[3][2] / (-2.0 * buffer_z + 1.0 - projection_matrix[2][2]);
}

float tan_to_sin( in float x )
{
    return x * pow(x * x + 1.0, -0.5);
}

vec3 tc_to_ec( in vec2 tc )
{
        vec3 tc_sample;
        tc_sample.xy = tc;
        tc_sample.z = tc_depth(tc_sample.xy);
        vec3 ndc_sample = tc_sample * 2.0 - 1.0;
        vec4 temporary = inverse_view_projection_matrix * vec4(ndc_sample, 1.0);
        vec3 wc_sample = temporary.xyz / temporary.w;
        vec3 ec_sample = (view_matrix * vec4(wc_sample, 1.0)).xyz;
        return ec_sample;
}

vec3 minimum_difference( in vec3 p, in vec3 p_right, in vec3 p_left )
{
    vec3 v1 = p_right - p;
    vec3 v2 = p - p_left;
    return (dot(v1, v1) < dot(v2, v2)) ? v1 : v2;
}

void main()
{
        vec2 depths_size = textureSize(depths, 0);
        vec2 depths_size_inversed = vec2(1.0) / depths_size;
        vec2 tc_depths = gl_FragCoord.xy / tc_window;
        vec3 wc_normal = texture(wc_normals, tc_depths).xyz;
        float ndc_linear_depth = -ec_depth(tc_depths) / z_far;
        vec3 wc_position = wc_camera_eye_position + vertex.wc_camera_ray_direction *
            ndc_linear_depth;

        vec3 ec_position = (view_matrix * vec4(wc_position, 1.0)).xyz;
        float ec_position_depth = ec_position.z;

        ambient_occlusion.a = 0.0;

        const int base_samples = 0;
        const int min_samples = 8;
        const float radius = 20.0;
        const float radius_squared = radius * radius;
        const float bias = 0.3;

        int samples = max(int(base_samples / (1.0 + base_samples * ndc_linear_depth)), min_samples);

        float projected_radius = radius / -ec_depth(tc_depths);

        vec2 inverted_random_texture_size = 1.0 / vec2(textureSize(random_texture, 0));
        vec2 tc_random_texture = gl_FragCoord.xy * inverted_random_texture_size;

        vec3 random_direction = texture(random_texture, tc_random_texture).xyz;
        random_direction = normalize(random_direction * 2.0 - 1.0);

        float angle_step = 2.0 * PI / float(samples);
        for (int i = 0; i < samples; ++i)
        {
```

```
#if USE_RANDOM_DIRECTION
                vec2 sample_random_direction = texture(random_texture, vec2(float(i) *
                    inverted_random_texture_size.x, float(i / textureSize(random_texture, 0).x) *
                    inverted_random_texture_size.y)).xy;
                sample_random_direction = sample_random_direction * 2.0 - 1.0;
                vec2 tc_sample_direction = sample_random_direction;
#else
                vec2 tc_sample_direction = vec2(cos(float(i) * angle_step), sin(float(i) *
                    angle_step));
#endif


            // Tangent vector
            vec3 p_right, p_left, p_top, p_bottom;
            p_right = tc_to_ec(tc_depths + vec2(depths_size_inversed.x, 0.0));
            p_left = tc_to_ec(tc_depths + vec2(-depths_size_inversed.x, 0.0));
            p_top = tc_to_ec(tc_depths + vec2(0.0, depths_size_inversed.y));
            p_bottom = tc_to_ec(tc_depths + vec2(0.0, -depths_size_inversed.y));
            vec3 dp_du = minimum_difference(ec_position, p_right, p_left);
            vec3 dp_dv = minimum_difference(ec_position, p_top, p_bottom) * (depths_size.y *
                depths_size_inversed.x);
        vec3 ec_tangent = tc_sample_direction.x * dp_du + tc_sample_direction.y * dp_dv;


                const int steps = 6;
                vec2 tc_step_size = tc_sample_direction * projected_radius / float(steps);
                vec2 ec_step_size = tc_sample_direction * radius / float (steps);

                float tan_tangent_angle = ec_tangent.z / length(ec_tangent.xy) + tan(bias);
                float tan_horizon_angle = tan_tangent_angle;
                float sin_horizon_angle = tan_to_sin(tan_horizon_angle);

                for (float j = 1.0; j <= float(steps); j += 1.0)
                {
                        vec2 tc_sample = vec2(tc_depths + tc_step_size * j);
                        vec3 ec_horizon = vec3(ec_step_size * j, ec_depth(tc_sample) -
                            ec_position_depth);
                        float ec_horizon_length_squared = dot(ec_horizon, ec_horizon);
                        float tan_sample = ec_horizon.z / length(ec_horizon.xy);

                        if (radius_squared >= ec_horizon_length_squared && tan_sample >
                            tan_horizon_angle)
                        {
                                float sin_sample = tan_to_sin(tan_sample);
                                float weight = 1.0 - ec_horizon_length_squared / radius_squared;
                                ambient_occlusion.a += (sin_sample - sin_horizon_angle) * weight;
                                tan_horizon_angle = tan_sample;
                                sin_horizon_angle = sin_sample;
                        }
                }
        }

        ambient_occlusion.a /= samples;
        ambient_occlusion.a = 1.0 - ambient_occlusion.a;
}
```

Listing 4: *Volumetric Obscurance*

```
uniform sampler2D depths;
uniform sampler2D wc_normals;
uniform sampler2D random_texture;

uniform vec3 wc_camera_eye_position;
uniform float z_far;

uniform vec2 tc_window;

uniform mat4 view_matrix;
uniform mat4 projection_matrix;
uniform mat4 view_projection_matrix;
uniform mat4 inverse_view_projection_matrix;


struct vertex_data
{
        vec3 wc_camera_ray_direction;
};
noperspective in vertex_data vertex;

out vec4 ambient_occlusion;



float tc_depth( in vec2 tc )
{
        return texture(depths, tc).x;
}

float ec_depth( in vec2 tc )
{
        float buffer_z = texture(depths, tc).x;
        return projection_matrix[3][2] / (-2.0 * buffer_z + 1.0 - projection_matrix[2][2]);
}

float sphere_height ( in vec2 position, in float radius )
{
        return sqrt(radius * radius - dot(position, position));
}

void main()
{
        vec2 tc_depths = gl_FragCoord.xy / tc_window;
        vec3 wc_normal = texture(wc_normals, tc_depths).xyz;
        float ndc_linear_depth = -ec_depth(tc_depths) / z_far;
        vec3 wc_position = wc_camera_eye_position + vertex.wc_camera_ray_direction *
            ndc_linear_depth;

        vec3 ec_position = (view_matrix * vec4(wc_position, 1.0)).xyz;
        float ec_position_depth = ec_position.z;

        ambient_occlusion.a = 0.5;

        const int base_samples = 0;
        const int min_samples = 32;
        const float radius = 10.0;
        const float lower_bound = 0.35;
        const float upper_bound = 1.0;

        int samples = max(int(base_samples / (1.0 + base_samples * ndc_linear_depth)), min_samples);

        mat4 inverse_view_projection_matrix = inverse(view_projection_matrix);
        float projected_radius = radius / -ec_depth(tc_depths);

        vec2 inverted_random_texture_size = 1.0 / vec2(textureSize(random_texture, 0));
        vec2 tc_random_texture = gl_FragCoord.xy * inverted_random_texture_size;

        vec3 random_direction = texture(random_texture, tc_random_texture).xyz;
        random_direction = normalize(random_direction * 2.0 - 1.0);

        for (int i = 0; i < samples; ++i)
        {
                vec2 sample_random_direction = texture(random_texture, vec2(float(i) *
                    inverted_random_texture_size.x, float(i / textureSize(random_texture, 0).x) *
                    inverted_random_texture_size.y)).xy;
                sample_random_direction = sample_random_direction * 2.0 - 1.0;
                vec2 sample_random_direction_negated = -sample_random_direction;

                vec2 tc_sample_1 = tc_depths + sample_random_direction * projected_radius;
                vec2 tc_sample_2 = tc_depths + sample_random_direction_negated * projected_radius;

                float ec_sample_1_depth = ec_depth(tc_sample_1);
                float ec_sample_2_depth = ec_depth(tc_sample_2);
                float depth_difference_1 = ec_position_depth - ec_sample_1_depth;
                float depth_difference_2 = ec_position_depth - ec_sample_2_depth;
                float samples_sphere_height = sphere_height(tc_sample_1, radius);
                float samples_sphere_depth_inverted = 1.0 / (2.0 * samples_sphere_height);

                float volume_ratio_1 = (samples_sphere_height - depth_difference_1) *
                    samples_sphere_depth_inverted;
                float volume_ratio_2 = (samples_sphere_height - depth_difference_2) *
                    samples_sphere_depth_inverted;

                bool sample_1_valid = lower_bound <= volume_ratio_1 && upper_bound >= volume_ratio_1
                    ;
                bool sample_2_valid = lower_bound <= volume_ratio_2 && upper_bound >= volume_ratio_2
                    ;

                // Should evaluate to a conditional assignment (no branching )
                if (sample_1_valid || sample_2_valid)
                {
                        // If the sample is     valid then use it. If not, then use the other one in
                             the pair (inverted).
                        ambient_occlusion.a += (sample_1_valid) ? volume_ratio_1 : 1.0 -
                            volume_ratio_2;
                        ambient_occlusion.a += (sample_2_valid) ? volume_ratio_2 : 1.0 -
                            volume_ratio_1;
                }
                else
                {
                        // Not 0.5 but 1.0 because both samples were invalid.
                        ambient_occlusion.a += 1.0;
                }
        }

        ambient_occlusion.a /= float(samples * 2.0 + 1.0);
        ambient_occlusion.a = 1.0 - ambient_occlusion.a;
}
```

Listing 5: *Alchemy AO*

```
uniform sampler2D depths;
uniform sampler2D wc_normals;
uniform sampler2D random_texture;

uniform vec3 wc_camera_eye_position;
uniform float z_far;

uniform vec2 tc_window;

uniform mat4 projection_matrix;
uniform mat4 view_projection_matrix;
uniform mat4 inverse_view_projection_matrix;


struct vertex_data
{
        vec3 wc_camera_ray_direction;
};
noperspective in vertex_data vertex;

out vec4 ambient_occlusion;

```

```


float tc_depth( in vec2 tc )
{
        return texture(depths, tc).x;
}

float ec_depth( in vec2 tc )
{
        float buffer_z = texture(depths, tc).x;
        return projection_matrix[3][2] / (-2.0 * buffer_z + 1.0 - projection_matrix[2][2]);
}



void main()
{
        vec2 tc_depths = gl_FragCoord.xy / tc_window;
        vec3 wc_normal = texture(wc_normals, tc_depths).xyz;
        float ndc_linear_depth = -ec_depth(tc_depths) / z_far;
        vec3 wc_position = wc_camera_eye_position + vertex.wc_camera_ray_direction *
            ndc_linear_depth;

        ambient_occlusion.a = 0.0;

        const int base_samples = 0;
        const int min_samples = 64;
        const float radius = 10.0;
        const float projection_factor = 0.75;
        const float bias = 1.0;
        const float sigma = 2.0;
        const float epsilon = 0.00001;

        int samples = max(int(base_samples / (1.0 + base_samples * ndc_linear_depth)), min_samples);

        mat4 inverse_view_projection_matrix = inverse(view_projection_matrix);
        float projected_radius = radius * projection_factor / -ec_depth(tc_depths);

        vec2 inverted_random_texture_size = 1.0 / vec2(textureSize(random_texture, 0));
        vec2 tc_random_texture = gl_FragCoord.xy * inverted_random_texture_size;

        vec3 random_direction = texture(random_texture, tc_random_texture).xyz;
        random_direction = normalize(random_direction * 2.0 - 1.0);

        for (int i = 0; i < samples; ++i)
        {
                vec2 sample_random_direction = texture(random_texture, vec2(float(i) *
                    inverted_random_texture_size.x, float(i / textureSize(random_texture, 0).x) *
                    inverted_random_texture_size.y)).xy;
                sample_random_direction = sample_random_direction * 2.0 - 1.0;

                vec3 tc_sample;
                tc_sample.xy = tc_depths + sample_random_direction * projected_radius;
                tc_sample.z = tc_depth(tc_sample.xy);
                vec3 ndc_sample = tc_sample * 2.0 - 1.0;
                vec4 temporary = inverse_view_projection_matrix * vec4(ndc_sample, 1.0);
                vec3 wc_sample = temporary.xyz / temporary.w;

                vec3 v = wc_sample - wc_position;

                ambient_occlusion.a += max(0.0, dot(v, wc_normal) - bias) / (dot(v, v) + epsilon);
        }

        ambient_occlusion.a = max(0.0, 1.0 - 2.0 * sigma / float(samples) * ambient_occlusion.a);
}
```

Listing 6: *UnrealEngine4 SSAO*

```
uniform sampler2D depths;
uniform sampler2D wc_normals;
uniform sampler2D random_texture;

uniform vec3 wc_camera_eye_position;
uniform float z_far;

uniform vec2 tc_window;

uniform mat4 view_matrix;
uniform mat4 projection_matrix;
uniform mat4 view_projection_matrix;
uniform mat4 inverse_view_projection_matrix;



struct vertex_data
{
        vec3 wc_camera_ray_direction;
};
noperspective in vertex_data vertex;

out vec4 ambient_occlusion;



float tc_depth( in vec2 tc )
{
        return texture(depths, tc).x;
}

float ec_depth( in vec2 tc )
{
        float buffer_z = texture(depths, tc).x;
        return projection_matrix[3][2] / (-2.0 * buffer_z + 1.0 - projection_matrix[2][2]);
}

// Reference: http://stackoverflow.com/a/3380723/554283
float acos_approximation( float x )
{
   return (-0.69813170079773212 * x * x - 0.87266462599716477) * x + 1.5707963267948966;
}

float calculate_angle(
        in vec2 direction,
        in vec2 tc_depths,
        in float projected_radius,
        in vec3 wc_position,
        in vec3 wc_normal,
        in float bias,
        inout float pair_weight )
{
        vec3 tc_sample;
        tc_sample.xy = tc_depths + direction * projected_radius;
        tc_sample.z = tc_depth(tc_sample.xy);
        vec3 ndc_sample = tc_sample * 2.0 - 1.0;
        vec4 temporary = inverse_view_projection_matrix * vec4(ndc_sample, 1.0);
        vec3 wc_sample = temporary.xyz / temporary.w;

        vec3 s = normalize(wc_sample - wc_position);
        vec3 v = normalize(-vertex.wc_camera_ray_direction);

        float vn = dot(v, wc_normal);
        float vs = dot(v, s);
        float sn = dot(s, wc_normal);

        // Cap to tangent plane
```

```
	vec3 tangent = normalize(s - sn * wc_normal);
	float cos_angle = (0.0 <= sn) ? vs : dot(v, tangent);

	// Invalid samples are approximated by looking at the partner in the pair of samples
	vec3 ec_position = (view_matrix * vec4(wc_position, 1.0)).xyz;
	float depth_difference = ec_depth(tc_sample.xy) - ec_position.z;
	if (20.0 < depth_difference)
	{
		pair_weight -= 0.5;
		cos_angle = max(dot(v, -s), 0.0);
	}

	return max(acos_approximation(cos_angle - bias), 0.0);
}



void main()
{
	vec2 tc_depths = gl_FragCoord.xy / tc_window;
	vec3 wc_normal = texture(wc_normals, tc_depths).xyz;
	float ndc_linear_depth = -ec_depth(tc_depths) / z_far;
	vec3 wc_position = wc_camera_eye_position + vertex.wc_camera_ray_direction *
	    ndc_linear_depth;

	ambient_occlusion.a = 0.0;

	const int base_samples = 0;
	const int min_samples = 32;
	const float radius = 10.0;
	const float bias = 0.08;
	const float projection_factor = 0.75;

	int samples = max(int(base_samples / (1.0 + base_samples * ndc_linear_depth)), min_samples);

	float projected_radius = radius * projection_factor / -ec_depth(tc_depths);

	vec2 inverted_random_texture_size = 1.0 / vec2(textureSize(random_texture, 0));
	vec2 tc_random_texture = gl_FragCoord.xy * inverted_random_texture_size;

	vec3 random_direction = texture(random_texture, tc_random_texture).xyz;
	random_direction = normalize(random_direction * 2.0 - 1.0);

	float weight_sum = 0.0001;
	for (int i = 0; i < samples; ++i)
	{
		vec2 sample_random_direction = texture(random_texture, vec2(float(i) *
		    inverted_random_texture_size.x, float(i / textureSize(random_texture, 0).x) *
		    inverted_random_texture_size.y)).xy;
		sample_random_direction = sample_random_direction * 2.0 - 1.0;
		vec2 sample_random_direction_negated = -sample_random_direction;

		float pair_weight = 1.0;

		float angle_sum =
			calculate_angle(sample_random_direction, tc_depths, projected_radius,
			    wc_position, wc_normal, bias, pair_weight)
			+ calculate_angle(sample_random_direction_negated, tc_depths,
			    projected_radius, wc_position, wc_normal, bias, pair_weight);

		ambient_occlusion.a += angle_sum * pair_weight;
		weight_sum += pair_weight;
	}

	ambient_occlusion.a /= weight_sum * PI;
}
```

Listing 7: *Geometry-aware Blur*

```
uniform sampler2D depths;
uniform sampler2D wc_normals;
uniform sampler2D source;

uniform mat4 projection_matrix;



out vec4 result;



float ec_depth( in vec2 tc )
{
	float buffer_z = texture(depths, tc).x;
	return projection_matrix[3][2] / (-2.0 * buffer_z + 1.0 - projection_matrix[2][2]);
}



void main()
{
	vec2 inverted_source_size = 1.0 / vec2(textureSize(source, 0));
	vec2 tc = gl_FragCoord.xy * inverted_source_size;

#if defined HORIZONTAL
#define DIRECTION_SWIZZLE xy
#elif defined VERTICAL
#define DIRECTION_SWIZZLE yx
#else
	"Define␣HORIZONTAL␣or␣VERTICAL␣before␣compiling␣this␣shader!";
#endif



	result = texture(source, tc);



	const float normal_power = 10.0;
	const float depth_power = 0.5;
	const int samples_in_each_direction = 1;
	float weightSum = 1.0;

	for (int i = -1; i >= -samples_in_each_direction; --i)
	{
		vec2 offset = vec2(float(i), 0).DIRECTION_SWIZZLE * inverted_source_size;

		float normalWeight = pow(dot(texture(wc_normals, tc + offset).xyz, texture(
			wc_normals, tc).xyz) * 0.5 + 0.5, normal_power);
		float positionWeight = 1.0 / pow(1.0 + abs(ec_depth(tc) - ec_depth(tc + offset)),
			depth_power);
		float weight = normalWeight * positionWeight;

		result += texture(source, tc + offset) * weight;
		weightSum += weight;
	}

	for (int i = 1; i <= samples_in_each_direction; ++i)
	{
		vec2 offset = vec2(float(i), 0).DIRECTION_SWIZZLE * inverted_source_size;

		float normalWeight = pow(dot(texture(wc_normals, tc + offset).xyz, texture(
			wc_normals, tc).xyz) * 0.5 + 0.5, normal_power);
		float positionWeight = 1.0 / pow(1.0 + abs(ec_depth(tc) - ec_depth(tc + offset)),
			depth_power);
```

```
                float weight = normalWeight * positionWeight;

                result += texture(source, tc + offset) * weight;
                weightSum += weight;
        }

        result /= weightSum;
}
```

Listing 8: *Lighting and Shading*

```
uniform sampler2D depths;
uniform sampler2D wc_normals;
uniform sampler2D albedos;
uniform sampler2D random_texture;
uniform sampler2D ambient_occlusion_texture;
uniform sampler2DShadow shadow_map;

uniform samplerBuffer lights;
#ifdef USE_TILED_SHADING
uniform isamplerBuffer light_grid;
uniform isamplerBuffer light_index_list;
uniform int tile_size;
#else
uniform int lights_size;
#endif

uniform ivec2 window_dimensions;
uniform ivec2 grid_dimensions;

uniform vec3 wc_camera_eye_position;
uniform float z_far;

uniform mat4 view_matrix;
uniform mat4 projection_matrix;
uniform mat4 view_projection_matrix;
uniform mat4 light_matrix;



struct vertex_data
{
        vec3 wc_camera_ray_direction;
};
noperspective in vertex_data vertex;

out vec4 color;
out vec4 overbright;



float eye_z( in vec2 tc )
{
        float buffer_z = texture(depths, tc).x;
        return projection_matrix[3][2] / (-2.0 * buffer_z + 1.0 - projection_matrix[2][2]);
}



vec4 calculate_direct_light(
        in vec3 wc_position,
        in vec3 wc_normal,
        in vec3 albedo,
        in float specular_exponent )
{
        vec4 result = vec4(0.0, 0.0, 0.0, 1.0);

```

```
#ifdef USE_TILED_SHADING
        ivec2 grid_index = ivec2(gl_FragCoord.xy) / tile_size;
        ivec2 grid_data = ivec2(texelFetch(light_grid, grid_dimensions.x * grid_index.y + grid_index
            .x).xy);

        int offset = grid_data.x;
        int count = grid_data.y;
#else
        int count = lights_size;
#endif

        for (int l = 0; l < count; ++l)
        {
#ifdef USE_TILED_SHADING
                int light_id = int(texelFetch(light_index_list, offset + l).x);
#else
                int light_id = l;
#endif
#define LIGHT_STRUCT_SIZE 9

                vec3 position = vec3(texelFetch(lights, light_id * LIGHT_STRUCT_SIZE).x, texelFetch(
                    lights, light_id * LIGHT_STRUCT_SIZE + 1).x, texelFetch(lights, light_id *
                    LIGHT_STRUCT_SIZE + 2).x);
                vec3 light_color = vec3(texelFetch(lights, light_id * LIGHT_STRUCT_SIZE + 3).x,
                    texelFetch(lights, light_id * LIGHT_STRUCT_SIZE + 4).x, texelFetch(lights,
                    light_id * LIGHT_STRUCT_SIZE + 5).x);
                float constant_attenuation = texelFetch(lights, light_id * LIGHT_STRUCT_SIZE + 6).x;
                float linear_attenuation = texelFetch(lights, light_id * LIGHT_STRUCT_SIZE + 7).x;
                float cubic_attenuation = texelFetch(lights, light_id * LIGHT_STRUCT_SIZE + 8).x;

                vec3 light_direction = position - wc_position;
                float light_distance = length(light_direction);
                light_direction /= light_distance;

                float falloff = 1.0 / (constant_attenuation + light_distance * (linear_attenuation +
                     cubic_attenuation * light_distance));

                // Diffuse
                vec3 light_contribution = albedo * light_color * max(dot(wc_normal, light_direction)
                     , 0.0);
                // Specular
                light_contribution += specular_exponent * max(pow(dot(wc_normal, -normalize(vertex.
                    wc_camera_ray_direction)), 2.0), 0.0);

                // TODO: Remove the following line. I'm just testing HDR!
                light_contribution *= 4.0;

                // Falloff
                result.rgb += light_contribution * falloff;
        }

        return result;
}



float shadow_map_lookup_with_offset( in sampler2DShadow shadow_map, in vec2 inverted_shadow_map_size
     , in vec4 v, in vec2 offset )
{
        return textureProj(shadow_map, vec4(v.st + offset * inverted_shadow_map_size * v.w, v.zw));
}

float calculate_shadow_coefficient( in vec3 wc_position )
{
        const float bias = 1.0;
        vec4 cc_position_from_lights_view = light_matrix * vec4(wc_position, 1.0);
        cc_position_from_lights_view.z -= bias;
        /*
```

```
    float shadow_coefficient = textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(-2, 2));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(-1, 2));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(0, 2));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(1, 2));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(2, 2));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(-2, 1));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(-1, 1));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(0, 1));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(1, 1));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(2, 1));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(-2, 0));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(-1, 0));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(0, 0));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(1, 0));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(2, 0));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(-2, -1));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(-1, -1));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(0, -1));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(1, -1));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(2, -1));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(-2, -2));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(-1, -2));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(0, -2));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(1, -2));
    shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(2, -2));
    */

    // The following works just as well on NVidia hardware.
    // Note that the pragma works globally! I.e., all loops will be unrolled.
    float shadow_coefficient = 0.0;
    #pragma optionNV(unroll all)
    for (int t = 5; t >= -5; --t)
        for (int s = -5; s <= 5; ++s)
            shadow_coefficient += textureProjOffset(shadow_map, cc_position_from_lights_view, ivec2(s, t));

    return shadow_coefficient / (11.0 * 11.0);
/*
    vec2 inverted_shadow_map_size = 1.0 / vec2(textureSize(shadow_map, 0));
    vec2 inverted_random_texture_size = 1.0 / vec2(textureSize(random_texture, 0));
    vec2 tc_random_texture = gl_FragCoord.xy * inverted_random_texture_size;
    float shadow_coefficient = 0.0;

```

```


        vec2 random_direction = texture(random_texture, tc_random_texture).xy;

        mat2 orient = mat2(random_direction, vec2(-random_direction.y, random_direction.x));

        for (float t = 3.5; t >= -3.5; t -= 1.0)
        {
                for (float s = -3.5; s <= 3.5; s += 1.0)
                {

                        shadow_coefficient += shadow_map_lookup_with_offset(shadow_map,
                            inverted_shadow_map_size, cc_position_from_lights_view, vec2(s, t)) +
                            orient[0][0] * 1.0e-32;

                        //tc_random_texture.x += s * inverted_random_texture_size.x;
                }
                //tc_random_texture.x += t * inverted_random_texture_size.y;
        }

        shadow_coefficient *= 1.0 / (64.0);
        return shadow_coefficient;
        */
}




vec4 calculate_direct_light2(
        in vec3 wc_position,
        in vec3 wc_normal,
        in vec3 albedo,
        in vec3 bent_normal,
        in float specular_exponent )
{
        vec4 result = vec4(0.0, 0.0, 0.0, 1.0);

#ifdef USE_TILED_SHADING
        ivec2 grid_index = ivec2(gl_FragCoord.xy) / tile_size;
        ivec2 grid_data = ivec2(texelFetch(light_grid, grid_dimensions.x * grid_index.y + grid_index
            .x).xy);

        int offset = grid_data.x;
        int count = grid_data.y;
#else
        int count = lights_size;
#endif

        for (int l = 0; l < count; ++l)
        {
#ifdef USE_TILED_SHADING
                int light_id = int(texelFetch(light_index_list, offset + l).x);
#else
                int light_id = l;
#endif
#define LIGHT_STRUCT_SIZE 9

                vec3 position = vec3(texelFetch(lights, light_id * LIGHT_STRUCT_SIZE).x, texelFetch(
                    lights, light_id * LIGHT_STRUCT_SIZE + 1).x, texelFetch(lights, light_id *
                    LIGHT_STRUCT_SIZE + 2).x);
                vec3 light_color = vec3(texelFetch(lights, light_id * LIGHT_STRUCT_SIZE + 3).x,
                    texelFetch(lights, light_id * LIGHT_STRUCT_SIZE + 4).x, texelFetch(lights,
                    light_id * LIGHT_STRUCT_SIZE + 5).x);
                float constant_attenuation = texelFetch(lights, light_id * LIGHT_STRUCT_SIZE + 6).x;
                float linear_attenuation = texelFetch(lights, light_id * LIGHT_STRUCT_SIZE + 7).x;
                float cubic_attenuation = texelFetch(lights, light_id * LIGHT_STRUCT_SIZE + 8).x;

                vec3 light_direction = position - wc_position;
                float light_distance = length(light_direction);
                light_direction /= light_distance;

                float falloff = 1.0 / (constant_attenuation + light_distance * (linear_attenuation +
                    cubic_attenuation * light_distance));

                // Diffuse
                vec3 light_contribution = albedo * light_color * max(dot(bent_normal,
                    light_direction), 0.0);

                // Specular
                light_contribution += specular_exponent * max(pow(dot(wc_normal, -normalize(vertex.
                    wc_camera_ray_direction)), 2.0), 0.0);

                // Falloff
                result.rgb += light_contribution * falloff;
        }

        return result;
}




float ec_depth( in vec2 tc )
{
        float buffer_z = texture(depths, tc).x;
        return projection_matrix[3][2] / (-2.0 * buffer_z + 1.0 - projection_matrix[2][2]);
}






void main()
{
        vec2 tc_window = gl_FragCoord.xy / window_dimensions;
        vec3 wc_position = wc_camera_eye_position + vertex.wc_camera_ray_direction * -eye_z(
            tc_window) / z_far;
        vec3 wc_normal = texture(wc_normals, tc_window).xyz;
        vec3 albedo = texture(albedos, tc_window).xyz;
        float specular_exponent = texture(albedos, tc_window).w;
        vec3 bent_normal = normalize(texture(ambient_occlusion_texture, tc_window).rgb * 2.0 - 1.0);
        float ambient_occlusion = texture(ambient_occlusion_texture, tc_window).a;
        const float a = 0.5;
        const float b = 1.0;
        const float c = 1.0;
        ambient_occlusion = pow(b * (ambient_occlusion + a), c);

        // Direct light
        color = 0.5880 * calculate_direct_light2(wc_position, wc_normal, albedo, bent_normal,
            specular_exponent);

        // Shadow Mapping
        color.rgb *= calculate_shadow_coefficient(wc_position);

        // Indirect light
        color.rgb += 0.15880 * albedo * ambient_occlusion;

        // Overrides
        //color.rgb *= 1.0e-32;
        //color.rgb += vec3(-eye_z(tc_window) / z_far);
        //color.rgb += vec3(ambient_occlusion);
        //color.g += texture(ambient_occlusion_texture, tc_window).x;
        //color.rgb += texture(random_texture, tc_window).rgb;
        //color.rgb += bent_normal;
        //color.rgb += wc_normal;
        //color.rgb += albedo;
        //color.rgb += 0.5880 * calculate_direct_light2(wc_position, wc_normal, albedo, bent_normal,
            specular_exponent).rgb;

        // Overbright
    const float bloom_limit = 1.0;
    vec3 bright_color = max(color.rgb - vec3(bloom_limit), vec3(0.0));
    float brightness = dot(bright_color, vec3(1.0));
    brightness = smoothstep(0.0, 0.5, brightness);
    overbright.rgb = mix(vec3(0.0), color.rgb, brightness);
}
```