NTNU – Trondheim
Norwegian University of
Science and Technology

Image-space Ambient Obscurance in WebGL

Lorents Odin Gravås

Master of Science in Informatics
Submission date: August 2013
Supervisor: Theoharis Theoharis, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Image-space approaches to ambient obscurance have become the de-facto standard for realistic ambient lighting in real-time applications. This thesis investigates the potential applicability of such approaches for a WebGL-based implementation. As image-space ambient obscurance has been an active field of research in computer graphics the last few years, a lot of different techniques and enhancements have emerged. This thesis presents a systematic survey of the current state of the art techniques, along with an assessment of their potential for successful implementation using WebGL. Finally, I present a working WebGL-based prototype, yielding good performance and acceptable quality.

Sammendrag

Tilnærminger til teknikken “image-space ambient obscurance” har blitt en de-fakto standard for realistisk indirekt lsyssetting i sanntidsapplikasjoner. Denne oppgaven utforsker den poteniselle muligheten for å bruke slike teknikker i en WebGL-basert implementasjon. Denne oppgaven presenterer en systematisk gjennomgang av eksisterende teknikker, sammen med en vurdering av deres potensial for en implementasjon ved hjelp av WebGL. Avsluttningsvis presenterer jeg en fungerende WebGL-basert prototype, som demonstrerer god ytelse og akseptabel kvalitet.

Preface

The work presented in this thesis was performed as part of my masters study at the Norwegian University of Science and Technology (NTNU) in Trondheim. The timeframe for this project has been two semesters. I would like to thank my main advisor Theoharis Theoharis for help and support during the project, and for giving me the opportunity to work on such an interesting and hot topic.

I would also like to thank Jonny Ree for providing me with the animated model used in the prototype. The sibernik cathedral, also used in the prototype, was downloaded from http://graphics.cs.williams.edu/data/meshes.xml .

Contents

Chapter 1

Introduction

1.1 Motivation

GPUs are everywhere. Powerfull high-end graphics accelerators have become standard in about any desktop computer. At the same time, low-power embedded graphics processors have found its way into an even wider range of consumer devices, most notable smart phones and tablets.

With the emerging trend of "the cloud" being the next application platform, WebGL enters the scene as a standard for browser applications to directly harness the power of GPUs found in most clients today. However, because portability and avilability is the key selling point of browser based applications, the core WebGL-standard only relies on the limited feature set available on mobile devices.

The bleeding edge of research in computer graphics tend to focus on utilizing new features found only in the latest high-end desktop platforms, in order to push the limits of what is possible in real-time further. However, the recent adaption of WebGL by most main-stream browser vendors opens an interessting opportunity for exploration of advanced rendering techniques that are available to a much broader audience.

This thesis will focus on one such technique, namely ambient obscurance and occlusion, under the limitations set by the core WebGL-standard.

1.2 Structure of thesis

This thesis is structured as follows

Part 1 presents the motivation and background for this work.

Part 2 contains an overview of the field. The challenges of image-space ambient obscurance and occlusion techniques are presented, followed by a study of different approaches to solving these, and their theoretical applicability to a WebGL-based implementation.

Part 3 concerns implementation-specific considerations, and reasons for the choices made during implementation. Finally, an overview of the implemented prototype is provided.

Part 4 presents the results and conclusions from this work.

1.3 Nomenclature

Graphics applications often give rise to the definition of a handful of common vector spaces. To complicate things even further, different authors and APIs often have different names for the same space. For clarity, I will in this section briefly define the notation and vector spaces used throughout this thesis.

I will use a *right haneded* coordinate system . All coordinates and direction will be represented by *column vectors*, and transformation matrices are assumed to be *post-multiplied by vectors.*

1.3.1 Spaces and definitions

Tangent-space The local coordiante system of a surface. Positive Z along the normal, Positive X is along the tangent, Positive Y is the bi-tangent (also called bi-normal). In this thesis i define this space to be right-handed.

Object-space The local space unique to each object in the scene. This is the space in which the geometry is defined, without any transformations applied.

World-space The global absolute coordinate system, resulting from transforming each object in the scene by its world-transformation. In OpenGL this transformation is called the Model-transformation.

View-space The local coordinate system of the viewer. In some litterature this is called eye-space or camera-space. I defined this space similarly to OpenGL with the eye position (aka. camera or center of projection) in origo, the view direction as the negative Z unit-vector, the right hand direction as the positive X unit-vector, and the up direction as the positive Y unit-vector. This space is a just world space rotated and translated accorting to the View-matrix, with no projection or aspect scaling applied.

Clip-space Clip-coordinates are view-coordinates after multiplication by a projection matrix. In the OpenGL pipeline, this is the space of the output from the Vertex Shader. The set of coordinates visible on the screen is $x, y, z \in [-w, +w]$, $w \in \Re$.

Normalized Device Coordinates (NDC) Normalized device coordinates results from dividing the X, Y and Z clip-coordinates by the W-clip coordinate, also know as perspective-division. The set of coordinates visible on the screen is the "unit cube", i.e. $x, y, z \in [-1, +1]$. If homogenous coordinates are used, the W-component can also be assumed to be divided by itself and always be equal to 1 in this space.

Image-space In this thesis I will refer to image-space as the coordinate system of a texture storing normalized linear depth values. This definition is useful in the context of reconstructing view-coordinates from depth in section 3.1.3, but is otherwise not commonly used.

1.3.2 A note about screen-, image- and object-space

In the context of distinguishing between techniques operating in image-space/screen-space versus object-space, it is often arbitrary which coordinate system the technique actually operates in; rather it is used as a way of distinguishing between methods having a discrete approximation of the geometry based on a certain view compared to methods assuming full knowledge of the source geometry.

1.4 Tools

This section describes the tools used during the work on this thesis. It describes WebGL, which have served as target platform, and whose limitations have been defining for the scope of this work. Secondly, it describes Uno, a cross-compiling programming language used as a tool during prototyping.

1.4.1 WebGL

WebGL is a JavaScript API that enables hardware accelerated rendering in the browser. WebGL 1.0 is based upon the OpenGL ES 2.0 specification, which is a subset of desktop OpenGL aimed at embedded devices.

1.4.1.1 The WebGL graphics pipeline

WebGL can be described as a pipeline. The input to WebGL is a shader program along with its input in the form of vertex buffers, samplers and uniforms. The shader program defines the behavior of the programmable stages of the pipeline, and the internal state of WebGL defines the behavior of the fixed-function stages. Vertex buffers store per-vertex information such as position and per-vertex normals. Uniforms store global information such as transformation matrices and information about global lighting. Samplers store grids of data. While samplers typically store texture maps loaded from files, image-space ambient obscurance technques utilize this feature to gain random access to output from a previous rendering pass.

There are two programmable stages in the WebGL pipeline: The vertex shader and the fragment shader, whose respective domains are each vertex and each rasterized fragment.

The vertex shader reads the data for the current vertex from the vertex buffers, does some computation with the additional input of uniforms, and writes the output to *varyings*. Varyings are variables representing values to be passed further down the pipeline. Built-in varying are used to define where the resulting primitives are rasterized. The per-vertex values of the varyings are then interpolated across the rasterized primitives, and serve as input for the next programmable stage: The fragment shader.

The fragment shader uses the interpolated values passed as varyings, in addition to uniforms and samplers, as input to calculate the color of the shaded fragment. Finally, the color of the fragment is blended according to the blending state of WebGL to produce the color of the final pixel in the resulting framebuffer.

1.4.1.2 Support

According to statistics gathered from a range of sites by WebGLStats (2013), almost three quaters of desktop clients support WebGL as of the time of writing. If one include mobile devices the ratio is abut two thirds.

OpenGL ES 2.0 is allready supported by a wide range of mobile devices, which can be exploited by vendors to hardware accelerate WebGL implementations on these devices. On the desktop, WebGL is supported by a lot of browsers, most notable Google Chrome and Mozilla Firefox.

1.4.2 Uno

Prior to and during the writing this thesis I have been working at the NTNU-originated startup company Outracks Technologies AS, with the development of a new programming language called Uno. Uno makes writing applications that utilize hardware accelerated graphics less tedious by taking care of passing values between the cpu, the vertex shader and the fragment shader. Furthermore, Uno makes code in such applications more readable by using a single language across all stages. The Uno compiler can translate Uno source code to a multitude of different platforms, including Javascript and WebGL.

I have used Uno as a tool to speed up prototyping during the writing of this thesis. However, this has no effect on my results, as my findings are not related to the language of the source code but rather the limitations of the target platform.

In the appendices I have included both the Uno source code and the resulting generated GLSL shader code for relevant parts of the prototype.

1.5 Background

This section describes explains the technical, physical and mathematical background for image-space ambient obscurance and occlusion.

1.5.1 Illumination models

Creating a model for everything you see is a demanding task. Illumination models define how light interacts with matter in a virtual setting. The ultimate goal of an illumination model is to mimick the behavior of light in the real-world, but most are just crude approximations. In this section I will introduce some of the challenges motivating the definition of the particular illumination model known as Ambient obscurance or occlusion.

1.5.1.1 Physically-based rendering

The rendering equation, introduced by Kajiya (1986a), describes the radiance of light leaving a point in space in all directions, based on geometric optics. The full equation can be formulated as follows:

$$L_o(x,\omega_o,\lambda,t) = L_e(x,\omega_o,\lambda,t) + \int_\Omega f_r(x,\omega_i,\omega_o,\lambda,t)L_i(x,\omega_i,\lambda,t)(\omega_i \cdot n)d\omega_i \tag{1.5.1}$$

For simplicity, I rewrite this for a certain wavelength λ and at one point in time t:

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_\Omega f_r(x,\omega_i,\omega_o)L_i(x,\omega_i)(\omega_i \cdot n)d\omega_i$$

$L_o(x,\omega_o)$is the readiance outwards from x in direction ω. $L_i(x,\omega_i)$ is the radiance of light reaching x from direction ω_i, $L_e(x,\omega)$ is the emitted light radiance of light emitted from x in direction ω. $f_r(x)$ is the bidirectional reflection distribution function (BRDF) at point x, describing how much light is reflected from direction ω_i to ω_o at x. Ω is the directions in the hemisphere around x.

For this thesis I will define physically-based rendering as rendering techniques that are derived from close-to-physical models such as the rendering equation.

Physically-based rendering can create images that are difficult to distinguish from photographs, and can to some degree even predict observations in the real-world. Effects such as reflection, soft shadows and color bleeding, which otherwise would need to be computed explicitly, are inherit side effects of physically-based rendering.

With the rapid delvelopment in computer hardware, offline rendering solutions have started to include physically-based rendering techniques such as path tracing. We also see the emergance of real-time implementations of path tracing and photon mapping (van Schijndel, 2013). However, these implementations only run on high-end desktop workstations, and the quality is still greatly limitated by the number of rays affordable per pixel. Real-time physically-based rendering techniqus are in general plaged with severe noise artifacts, in cases where empirical approximations can give convincing results at several orders of magnitude higher frame rates.

All current successful implementations of real-time physically-based rendering depend on the avialablility GPGPU, which makes them unsuitable for a WebGL implementation. Fortunately, there have been an enormous effort put into the creation of empirical illumination models that, to various degrees of realism, approximate the effects of physically-based rendering, without the choking computational demands.

1.5.1.2 Local- and global illumination models

It is common to disguingish between *local-* and *global illumination models*. Global illumination models consider the geometric surface- and volumetric properties of the entire scene in the shading of each point, while local methods only consider the properties in the vicinity of - or at - the point to be shaded, under the assumption that other properties are globally constantant. This distinction is not very useful for physically-based illumination models as defined above, as such are nessecarely global.

While global illumination models can produce renderings that are superior in quality over local illumination models, they are often very expensive to compute. Acceleration structures are required to achieve real-time frame rates for any non-trivial dynamic scene. Unfortunately, such structures are often difficult to evaluate in a shader without hitting the limitations of shader length and complexity allowed by WebGL.

1.5.2 Ambient obscurance and occlusion

Ambient obscurance and occlusion, or AO for short, is a non-physically based local illumination model, that approximates some global illumination effects by considering geometry within a bounded distance of the shaded point. Ambient obscurance and occlusion specifically aims to approximate the amount of indirect light that reaches a point from all directions on the point's hemisphere, assuming no inter-reflections.

AO is an emprical model, motivated by the observation that in an environment under diffuse lighting, corners and other obscured areas appear darker than open areas (Zhukov et al., 1998). The locallity of the model bounds its computational demands, while still providing plausible global illumination effects. In fact, in the original paper by Zhukov et al. (1998), AO is described as a *locally-global* illumination model.

While traditionally a popular technique in film for reducing rendering time and ease creation of light environments (Christensen, 2003; Landis, 2002), the technique is now also commonly used in games and other real-time applications (Mittring, 2007, 2009; Filion and McNaughton, 2008; Scheer and Keutel, 2010; Smedberg and Wright, 2009; McGuire et al., 2011, 2012).

1.5.2.1 The secret behind its success

Ambient obscurance and occlusion is a subtle effect, but it covers an aspect of physical lighting that is important to human perception of scenery. Soft shadows caused by ambient obscurance and occlusion enhance perception of geometric detail, by darkening conave surfaces, cracks and corners. Contact shadows give important visual cues of the spatial relationship between objects, and the relative scale of soft shadows hint about the proximity of objects.

It is not surprising that AO is so commonly used in both real-time and offline rendering. While shadow mapping can account for shadows cast by direct light, AO accounts for shadows cast by indirect light, of which there tend to be most of in common settings. E.g. in indoor scenes, most light tend to be reflected off walls; in outdoor scenes, the sky cause ambient illumination.

In the absence of *any* ambient lighting, scenes look sharp, flat and unrealistic, and the areas not reached by direct light would be pitch black. With the common approximation of constant ambient lighting, shadows are no longer pitch black, but the scene will still look flat and unrealistic. Ambient obscurance and occlusion gives a softer look to the scene while providing a whole new level of realism.

Interestingly, even in scenarios where realism is not a key citeria, AO can ease the understanding of 3D geometric data. Hence, the technique has a broad range of applications in visualization outside games and film; including visualization of seismic data, computer aided design, molecular visualization and medical imaging, to mention a few.

1.5.2.2 Definition

In this thesis i will use the original definition of ambient obscurance from Zhukov et al. (1998), which can be formulated as follows:

$$A_P = \frac{2}{\pi} \int_{\Omega} p(V_P(\omega)) \cos \alpha d\omega \tag{1.5.2}$$

Informally, we integrate the visibility $p(V_P(\omega))$ of a point P over all direction ω in the hemisphere Ω; taking into account the angle α between the direction and the surface normal, similarely to the rendering equation. The calculated value A_p is 1 for unobscured points, and 0 for totally occluded points. Further, we define

$$V_P(\omega) = \begin{cases} \|P - C\| & \text{where } C \text{ is the first intersection point of the ray } P + \omega t \\ +\infty & \text{if no intersection occured} \end{cases}$$

and $p(r)$ as a monotonically increasing bounded function, which is 1 for $r > r_{max}$, where r_{max} is the maximum distance at which geometry can affect the obscurance of the point.

Taking the distance of the first intersection into account is motivated by the fact that surfaces not only occlude light, but also reflect it. *Ambient occlusion* is a special case of ambient obscurance that assumes the incident radiance from a blocked direction on the hemisphere to be zero. I.e:

$$p(r) = \begin{cases} 1 & \text{for } r > r_{max} \\ 0 & \text{otherwise} \end{cases}$$

In general, this assumption produce darker but still convincing results (Vardis et al., 2013), and is commonly used as a basis for implementations.

1.5.3 Image-space approximations

Ambient obscurance an occlusion is a purely geometric property, and hence not dependent on view- and light direction. For static scenes it can be precalced and stored in texture maps or as per-vertex information. However, for dynamic scenes this is not an option. In addition, this is unconvenient for WebGL applications as it will greatly increase the number and size of textures that needs to be downloaded. Hence approximating AO in real-time is of interest.

Calculating AO directly is still too computationally intensive to be done in real-time for complex geometry, as any point can be obscured from any direction by all geometry within a radius of r_{max}. In addition, this approach do not map well to the capabilities of the rendering pipeline.

This has lead to the emergance and wide adoption of *image-space ambient obscurance and occlusion*, also commonly refered to as Screen-space Ambient Occlusion (SSAO). Image-space effects are typically applied as a post-process, using results from earlier rendering passes as inputs, and can hence be considered a special case of deferred shading (Hargreaves and Harris, 2004). This is further discussed in section 3.1.1.

The core assumption of image-space AO is that a discretization of the currently visible surfaces within the view frustum provide a good enough approximation of the scene geometry to calculate ambient obscurance. In practice this is implemented by first rendering the scene geometry to a so-called G-buffer, a texture containing geometric information. The information stored in a G-buffer can for instance be positions, normals, texture coordinates, reflectivity, pixel depth, and so on.

The main benefit from the image-space approach is that the performance is independent of the scene complexity, as it works solely on a discretisation of the geometry. This makes it scale linearly with the number of visible pixels, instead of being dependent on the number and complexity of all objects.

Another attractive features of image-space AO is that no precomputation is needed, and there is no special cases for dynamic scenes. As an image-based effect, it is relatively easy to implement in a fullscreen fragment shader. This has lead the technique to become the de-facto standard for realistic ambient lighting in games.

While image-space approximations makes AO feasible for real-time applications, they have their own set of problems. Performance considerations cause undersampling, which can introduce noise or banding artifacts. Small details can be missed if a large radius is used in combination with few sample points. Texture cache trashing can result in huge performance swings. And as with all image-space approximations, view-dependency is a major issue. Often the artist is left tuning sub-intutive parameters such as epsilons and biases in order to hide artifacts caused by the implemented algorithm. These problems are further discussed in section §2.1.

In computer graphics, one is often left with a tradeoff between quality and performance. I would further propose that this tradeoff includes a third variable, how general - or flexible - the approach is. For very special cases, it is most certainly possible to get "perfect" ambient obscurance estimation while maintaining high performance. For instance, one can analytically solve the ambient obscurance integral for simple geometric shapes; or, as mentioned in the introduction of this section, for static scenes the obscurance factors can be pre-calced. In this sense, image-space approaches to ambient obscurance are very flexible and yield high perforance, at the cost of visual quality.

1.6 Related work

This section gives a brief review of ambient occlusion and obscurance techniques from a historical perspective. A more in-depth survey of the image-based approaches is given in section §2.2.

Miller (1994) defined *accessibility* as how easy it is for a spherical probe to reach a point on a surface, "accessibility shading" has an effect similar to what will later be known as Amient Obscurance. The definition of Ambient Obscurance used today was introduced by Zhukov et al. (1998). Landis (2002) describes the use of AO in a high-quality offline rendering environment. Pharr and Green (2004) adapts the technique to real-time for static scenes by storing a pre-calced ambient factor in texture maps, also known as "baking" AO.

Bunnell (2005) approximates the scene geometry with oriented discs that are used as occluders to calculate per-vertex occlusion values on the GPU. This technique suffers from problems handling dynamic scenes, and requires a relatively expensive preprocessing step. As a per-vertex technique it also requires a high tesseleation of the geometry. Kontkanen and Laine (2005) precomputes occlusion values realtive to each object and stores them in textures. The values from all objects in the scene is combined, in real-time, taking rigid transformations into account, hence partly solving the problem of dynamic scenes. However, this approach does not work for deformable objects, requires expensive preprocessing, and is limited in the number of allowed dynamic objects.

Ren et al. (2006) proposes a different definition of AO based on spherical harmonics and assuming spherical occluders. This is implemented on the GPU with support for dynamic scenes, only requireing a simple preprocessing step. However, this has approach suffers from performance problems with complex scenes.

Luft et al. (2006)is the first to note that an unsharp masking of the depth buffer resembles ambient obscurance. Shanmugam and Arikan (2007) and Mittring (2007) independently presented what can be considered the first image-space ambient obscurance techniques.

Shanmugam and Arikan (2007) creates an analytic representation of the visible geometry by assuming each pixel in a previous depth rendering to be the projection of a sphere. Their algorithm then uses the sphere cap projection onto the hemisphere of the shaded surface patch to estimate the occlusion due to spheres in its vicinity. Finally, a pre-calced approximation of the source geometry as larger spheres is used to accumulate occlusions from large and distant occluders.

Mittring (2007) introduces the term Screen-Space Ambient Occlusion (SSAO). Their technique simply projects random points within a sphere around the shaded point on the view plane and accumulates the attenuated difference between the points' depth and the corresponding depth found in the G-buffer. In effect, this results in a fuzzy open volume estimation in the vicinity of the point. Because it considers samples within a sphere rather than a hemisphere, this technique produces a non-photorealistic look.

While Shanmugam and Arikan (2007) use the image-based approach only for high-frequency occlusions, Mittring (2007) uses a much larger radius to capture occlusions from large and distant objects aswell. To compensate for undersampling caused by the larger radius, an edge-preserving blur is applied to the result.

Filion and McNaughton (2008) derives a more realistic variant of SSAO that do not consider occluders behind hemisphere of the shaded point, as opposed to the approach of Mittring (2007). Smedberg and Wright (2009) successfully applies a varity of performance optimizations to SSAO, in order to make it suitable for Xbox360-era consoles.

Szirmay-Kalos et al. (2010); Loos and Sloan (2010) improves upon open-volume based obscurance estimators by using line-samples instead of point samples.

Fox and Compton (2008) describes a technique that bears similarity to both Mittring (2007) and Shanmugam and Arikan (2007), but tries to approximate the rendering equation with random single-sample rays and Monte Carlo-integration. Dimitrov et al. (2008) and Bavoil et al. (2008) takes a more explicit approach, interpreting the depth buffer as a height field and tracing rays along a set of azimuthal directions.

McGuire et al. (2011) introduces the Alchemy screen-space ambient obscurance algorithm. An algorithm developed with the primary goal of estetics rather than realism, and under the performance limitations of the Xbox360. However, it *is* derived from the rendering equation, and bears some similarities to the approach of Fox and Compton (2008). This algorithm is improved to scale with better hardware and higher resolutions in McGuire et al. (2012).

In the last couple of years, there have been some GPGPU-based approaches yielding promising results. Most notable, Timonen (2013) recently introduced Line-Sweep Ambient Obscurance, the first algorithm for image-space ambient obscurance with linear time complexity.

A set of techniques that are orthogonal to the above research have also emerged since the first introduction of image-space ambient obscurance, further discussed in section §2.3:

- Rendering in a lower resolution (Smedberg and Wright (2009), Ritschel et al. (2009), Bavoil and Sainz (2009b), Hoang and Low (2010))

- Using a lower resolution depth G-buffer (McGuire et al. (2012))
- Using several depth G-buffers rendred from multiple points of view (Ritschel et al. (2009), Vardis et al. (2013))
- Using several depth G-buffers of different depth-peeled layers (Ritschel et al. (2009), Bavoil and Sainz (2009b))
- Utilizing temporal coherence (Smedberg and Wright (2009) , Mattausch et al. (2010), Mattausch et al. (2011))

Chapter 2

An overview of the field

This chapter gives an overview of the field of image-space approaches to ambient obscurance and occlusion, the challenges posed by such techniques, and the state of the art methods for overcoming these.

2.1 Challenges

The advantages of image-space approaches to ambient obscurance and occlusion are also the root cause of many of its problems. I would like to divide these problems into two categories: Approximation failures and implementation challenges.

2.1.1 Approximation failures

The core assumption of image-based approaches from section 1.5.3 fails for all but the most trivial scenes. Fortunately the eye can be forgiving, and this ends up not being the bane of such approaches. After all, it is still a lot better basis for approximation of ambient lighting than knowledge only of the currently shaded pixel (as in fully local models such as Phong (1975)).

Errors due to the missing information manifests itself as false occlusions, i.e. occlusion by geometry that is not present in the real geometry, and missing occlusion. These approximation failures are the cause of view-dependency artifacts seen in image-space ambient obscurance and occlusion.

2.1.1.1 False occlusions

If one assumes that all visible surfaces are the front of closed solids extending to infinity in the view direction, the apprixmated geometry will always contain a concave region where the object in front intersects the background, even though the intersecting objects in reality are disconnected. This results in what can be seen as a dark "halo" around the silhuette of objects disconnected from the

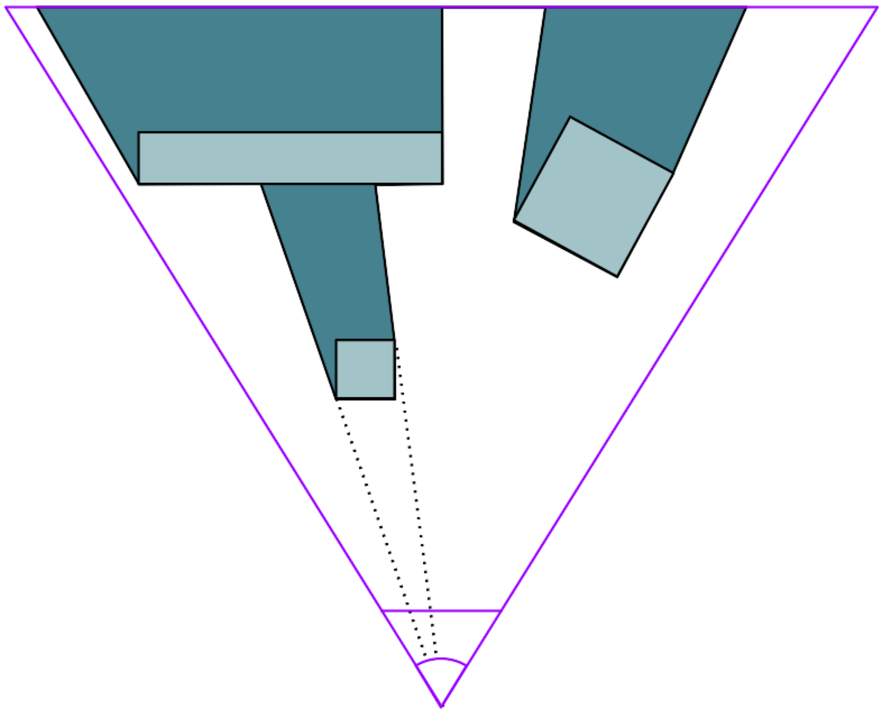


Figure 2.1.1: A cross section of a scene (light blue) and its image-space approximation based on a pre-pass rendering (dark blue). View frustum is shown in purple.

background. As the camera moves this effect gets more noticable, because the calculated occlusion behind the silhuette will follow the object in front. It is worth noting that the approximated geometry in this case will always have more closed regions than the true geometry, and hence the resulting ambient obscursion value will always be an overestimate of the correct occlusion.

2.1.1.2 Missing occlusion effect

Since false occlusions are generally more noticable than missing occlusions, implementers often makes assumptions about the geometry that reduce the former at the expense of introducing missing occlusions. A common approach is to ignore geometry outside of a given radius of the shaded point in view space. Another common approach is to assume the visible surface to be an empty shell, and hence underestimate the occlusion.

2.1.2 Implementation challenges

2.1.2.1 Under-sampling

One of the biggest performance killers for image-space approaches is the cost of sampling of the depth buffer many times per pixel. A perfect approximation given the available information would in worst case require sampling all depth values in the projected area of the hemisphere of the shaded point.

Noise and banding are common artifacts of under-sampling, depending on the sampling scheme used. If a large sampling radius is desired, fewer sampling points also means a higher chance of missing occlusions from high frequency changes in the geometry.

2.1.2.2 Performance swings

In order to keep the radius of the AO-effect consistent in world-space, shading surfaces close to the viewer will require a much larger sampling radius in image-space. This causes enormous performance swings as surfaces gets close to the viewer due to trashing of the texture cache. This is solvable by limiting the maximum allowed image-space radius, at the expence of inconsistensies as objects get closer to the viewer.

2.1.2.3 Self-occlusion

Another common artifact found in some implementations of image-space ambient obscurance and occlusion is *self-occlusion*. This happens when a surface is occluded by itself. The only good way to solve this problem is to take the surface normal into account. Some techniques try to solve this problem by requireing a minimum depth difference for a sample to contribute to the occlusion. However, the latter approach has the unfortunate side effect of "lifting" and offseting contact shadows.

A final problem, common not only to image-based approaches, is the fact that ambient obscurance and occlusion can reveal the tesselation of curved surfaces. This “artifact” is not really an artifact at all, but unfortunate due to the common practice of using low-poly models with smoothed normals to imitate smooth surfaces without the performance overhead.

2.2 Estimator functions

At the core of all implementations of image-space ambient obscurance is a function that estimates an obscurance factor for a point based on nearby samples. I call this the *estimator function.* There are two main approaches to estimator functions: *solid angle*-based estimators and *open volume*-based estimators. Solid angle-based approaches try to estimate the integral over directions in the hemisphere directly, and is closer to the definition in equation (1.5.1). Open volume-based approaches exploit the correlation between the “openess” of volume in the vicinity of the receiving point and its visibility.

In this section I will examine some of the most common estimator functions.

2.2.1 Open volume-based estimators

2.2.1.1 Point sampels of surrounding sphere

One of the first approaches to image-space ambient occlusion is that of Mittring (2007), developed for the computer game Crysis. Mittring (2007) estimates the “openness” of a sphere surrounding the shaded point based on sparse random sampling, and coins the term Screen-Space Ambient Occlusion (SSAO) for this approach.

First, I will introduce point-based sampling in general. Let $V(S)$ be a function dividing a volume into a visible part and an occluded part, so that:

$$V(S) = \begin{cases} 1 & \text{if S belongs to the visible part} \\ 0 & \text{otherwise} \end{cases}$$

A fuzzy estimate of the openness of the volume can now be achieved simply by averaging the result of random samples within the domain:

$$A = \frac{1}{N} \sum_{S \in Q} V(S) \tag{2.2.1}$$

This is the technique exploited by Mittring (2007), which specifically define Q to be a set of N random 3D sample positions within a sphere surrounding the shaded pixel, and let the visibility function be defined in terms of depth buffer, i.e:

$$V(S) = \begin{cases} 1 & \text{for } d_S > S_Z \\ 0 & \text{otherwise} \end{cases}$$

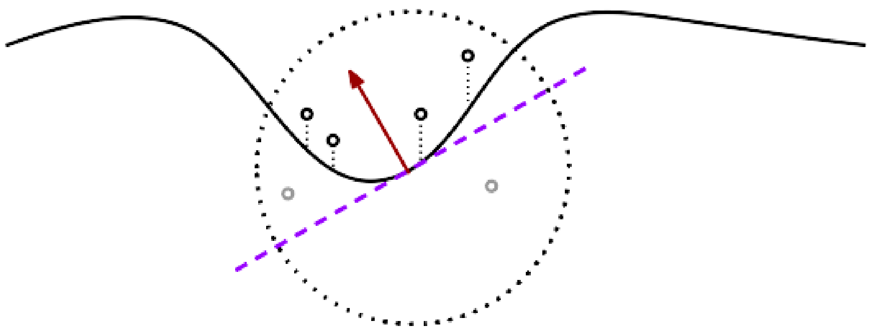


Figure 2.2.1: Point sampels of surrounding sphere. Note the gray dot behind the tanget plane (purple).

Where S is the view-space position of the random test sample, and d_S is the view-space depth found in the depth buffer at the projection of S.

Because the normal of the shaded surface is not taken into account, flat surfaces will occlude themselves. This means that non-occluded surfaces have an occlusion value of 0.5, resulting in the characteristic gray images with lighter concave regions found in scenes shaded using this model. According to Kajalin (2009), this was actually a design choice, as it enhanced geometric detail and they liked the non-photorealistic look it gave. This design choice was not repeated in later titles, though.

Kajalin (2009) gives a more in-depth explanation of the approach used in Crysis, and includes a compensation for "invalid ranges", i.e. samples where the difference in sample depth and depth found in the G-buffer is too large. Based on the source code listing from that paper, this attenuation coeficcient can be defined as the following:

$$F(d_P, d_s) = \min(\max(\frac{d_P - d_S}{d_S}, 0), 1)$$

$F((d_P, d_S)$ is then used to interpolate between the visiblity value $V(S)$ and the "neutral" obscurance value of 0.5. Formally, the complete equation finally becomes:

$$A = \frac{1}{N} \sum_{S \in Q} (V(S) + (0.5 - V(S))\, F(d_P, d_S)) \qquad (2.2.2)$$

To generate the 3D sample points Q, Mittring (2007) stores as set of randomly distributed 3D positions within a sphere in constant memory, and uses them as world-space offset vectors of the position.

2.2.1.2 Point samples of tangenting hemisphere

A change commonly made to the method above is to generate 3D sampling points within the point's tangenting hemisphere instead of a sphere centered

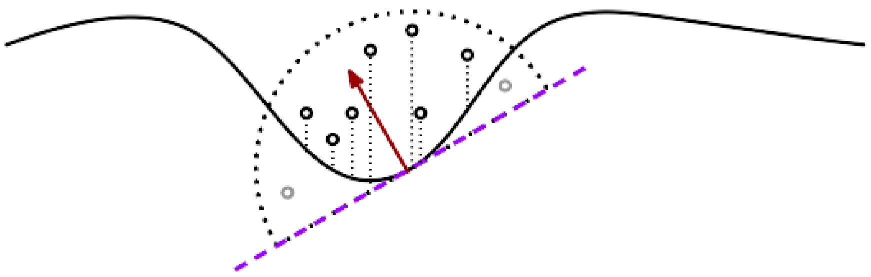


Figure 2.2.2: Point samples of tangenting hemisphere

around the point. This is more true to the rendering equation, which has a positive cosine falloff efficently ignoring rays coming from "behind" the point's tangent plane.

A naive approach is to store as random coordinates within a tangent hemisphere as tangent-space vectors in constant memory, and rotate the vectors based on the normal of the point. Filion and McNaughton (2008) suggests a much simpler approach, however: Store random vectors within a sphere, and reflect vectors ending up behind the tangent plane across the tangent plane, effectively only using points within the hemisphere, without the need for expensive rotation of offset vectors. This does however make the sampling pattern a bit more predictable.

Since a non-occluded surface with this approach returns an obscurance of 1 instead of 0.5, the need for a linear interpolation to fade out the effect of invalid ranges in equation (2.2.2) is removed, making the new equation:

$$A = \frac{1}{N} \sum_{S \in Q} V(S) \left(1 - F(d_P, d_S)\right) \tag{2.2.3}$$

2.2.1.3 Line samples of surrounding sphere

Loos and Sloan (2010) proposes to use line sampling instead of point sampling to estimate the open volume:

$$A = \frac{1}{N} \sum_{s \in Q} \max(\min(d_s, d_r^+(s)) + d_r^+(s), 0) \tag{2.2.4}$$

$$d_r^+(x, y) = \sqrt{1 - x^2 - y^2}$$

To simplify these equations, I define all values to be in view-space offset so the shaded pixel is in origo, i.e. sampling points are 2D offset vectors. (This is similar to the original paper, where it is called "the coordinate system of the

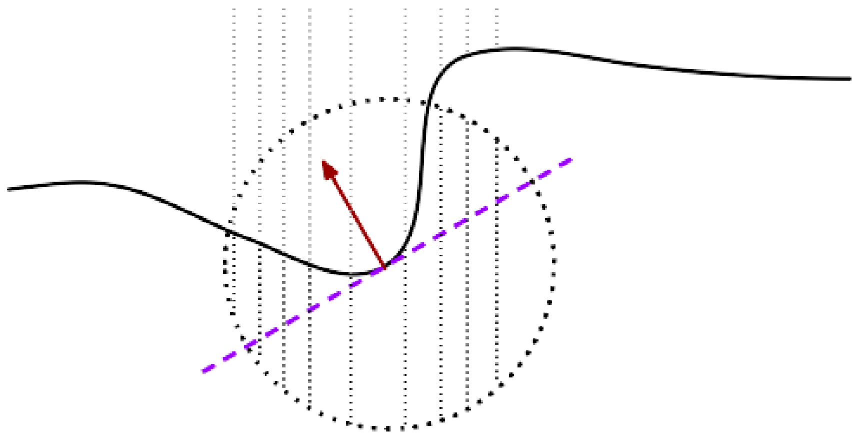


Figure 2.2.3: Line samples of surrounding sphere

sphere"). $d_r^+(x,y)$is the positive z value of the surface of sphere with radius 1 given x and y (i.e. one at origo and zero outside the unit circle in the xy plane). Since line samples have no z-component, Q is here a set of 2D positions on a disc in screen space. d_s is the depth found in the depth G-buffer at the sampled position. As we can see, the estimated volume of each line sampled is the length of the line bounded by the sphere of radius r and the surface defined by the depth buffer (see figure 2.2.3).

2.2.1.4 Line samples of tangenting hemisphere

Loos and Sloan (2010) also proposed a way to incorporate the normal into equation (2.2.4), making it an integral over line samples of the tangenting hemisphere instead of a sphere. This is done by clamping the integration domain to the depth of the tangent plane (see figure 2.2.4). The following is one possible formulation of this:

$$A = \frac{1}{N} \sum_{S \in Q} \max\left(\max\left(d_s, d_r^+(s)\right) - \min\left(d_t(s), -d_r^+(s)\right), 0\right) \tag{2.2.5}$$

$d_t(x,y)$ is the depth of the tangent plane at x,y. since the tangent plane is defined by the normal and going through the origo we have:

$$d_t(x,y) = -\frac{n_x x + n_y y}{n_z}$$

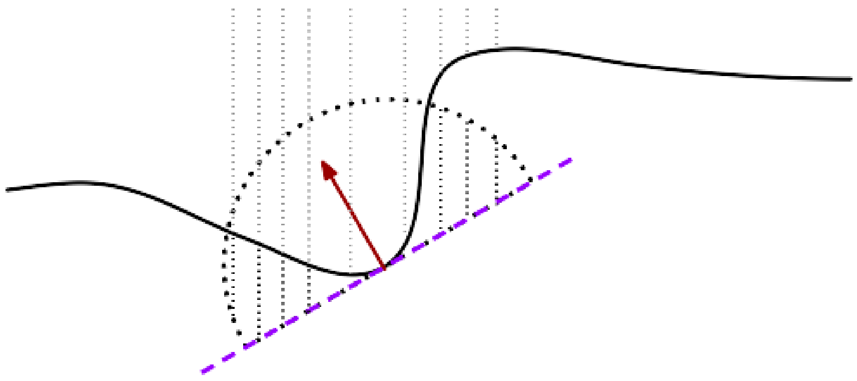


Figure 2.2.4: Line samples of tangenting hemisphere

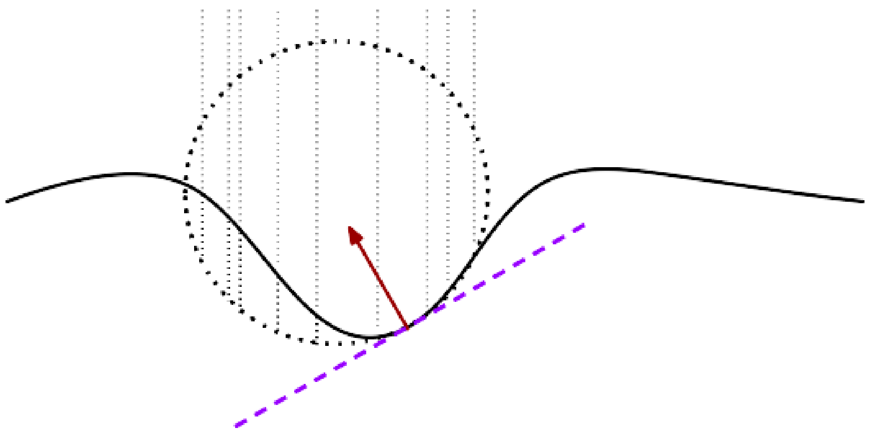


Figure 2.2.5: Line samples of tangenting sphere

2.2.1.5 Line samples of tangenting sphere

Szirmay-Kalos et al. (2010) proposed a novel ambient occlusion estimator based on estimating the open volume of a sphere tangenting the shaded point in the direction of the normal. This inheritly integrates the positive cosine angle falloff from the rendering equtaion (1.5.1) and intrinsicly takes the surface normal into account.

2.2.2 Solid angle-based

2.2.2.1 Monte Carlo integration based on point samples

The simplest solid angle-based estimator function is simply to approximate equation (1.5.2) by Monte Carlo integration and approximate rays to the visibility of a single point sample, as applied in Ritschel et al. (2009) and Mattausch et al. (2011), among others. This can be formulated as:

$$A = \frac{1}{N} \sum_{s \in Q} V(s)\, D(\|s - P\|) \max(v \cdot n, 0)$$

where $v_i = \frac{s-P}{\|s-P\|}$, Q are random sampling points in the vicinity of the shaded point. I categorize this as a solid angle-based estimator mainly because it uses the cosine angle between the normal and the vector to the sample point, otherwise it has more in common with the open volume-based approaches.

2.2.2.2 Horizon-based AO estimator

Bavoil et al. (2008) also approximates equation (1.5.2) by Monte Carlo integration, but interprets the depth buffer as a height field and does explicit ray marching. To make it feasible in real-time the approximation that it is only necessary to consider the tallest occluder along each azmuthal direction is utilized.

However, comparisons presented in McGuire et al. (2011) and other later papers suggests that this approach - while more true to the rendering equation - is rather expensive relative to the gained quality over fuzzy approaches. For this reason I choose not to focus on horizon-based AO in this thesis, even though it is most certainly an important piece of work in the field of image-space ambient obscurance.

2.2.2.3 Sphere caps

The pinoeering work of Shanmugam and Arikan (2007) interprets random neighbouring pixels as spheres that have a radius which project to a pixel, and sums the sphere cap projection of a set of samples on the hemisphere of the shaded point. This approach is, however, not of much interest as later solutions are both faster and yield better results. However, it is included here for consistency:

$$A = 2\pi \sum_{s \in Q} \left(1 - \cos\left(\sin^{-1}\left(\frac{r}{\|s - P\|}\right)\right)\right) \max(n \cdot v_s, 0)$$

In addition to being somewhat difficult to tweak, it uses arithmetic operations, which are ALU-heavy.

2.2.2.4 Alchemy AO estimator

McGuire et al. (2011) derives an AO estimator with the goal of artistic expressiveness rather than realism.

$$A = \max\left(0, 1 - \frac{2o}{N} \sum_{i=1}^{N} \frac{\max(0, v_i \cdot n + z_C \beta)}{v_i \cdot v_i + \epsilon}\right)^k \qquad (2.2.6)$$

Intuetively, the core of this estimaor is the cosine of the solid angle in the hemisphere with an inverse distance (not squared) attenuation. It features an artist-tweakable angle bias that increses with depth of the shaded point, aimed at getting rid of self occlusions due to percission issues and hiding tesselation of curved surfaces. The result is multiplied by a constant and raised to the power of another constant, to let artists tweak the brightness and contranst, respectively.

The original paper uses a 3D sampling scheme, but in the later improvement of the algorithm by McGuire et al. (2012), a similar 2D sampling scheme to that of Loos and Sloan (2010) was adopted. Because of the aritst-friendly parameters and simplicity, I have preferred this estimator.

2.2.3 On sample locator schemes

The choice of which neighbouring pixels to sample is very important for the quality of the AO. It also has a surprisingly large impact on the performance of a given technique, as shown in Part 4. This is likely due to the immense performance gained from texture cache hits and coalescing texture reads on modern GPUs. As mentioned in section 2.1.2.2, a large world-space radius can give bad performance when it projects to a large area in image-space. In addition, the pattern of consecutive texture reads should preferrably be predicted by the texture cache.

I would like to refer to Loos and Sloan (2010) for a study of the performance impact and quality of different sampling patterns in 2D.

2.2.3.1 Randomization

While the banding artifacts caused by few sampling points are very noticable, the eye is very forgiving for noise (see figure 2.2.6). We can trade banding for noise by randomizing the sampling points per pixel.

To randomize 3D sampling points within a sphere, Mittring (2007) proposes to flip the points around per-pixel random vectors. This has the advantage of not requireing ALU-heavy rotation of vectors. The same is also possible in 2D, however, if the sampling points are generated on the fly using a spiral-formula, as suggested by McGuire et al. (2012), the initial angle in the disc formula can simply be offset with a per-pixel random angle. This is the approach I have used in my implemenation.

2.3 Enhancements

There is a lot more to a sucessfull image-based AO technique than an estimator function and a sampling scheme. In this section I will give an overview of different techniques for minimizing the inherit problems of image-based AO outlined in section 2.1.2. Most of these techniques are orthogonal and not dependent on a specific estimator function, and hence the possible combinations of "tricks" with estimator functions and sampling schemes are in the hundreds.

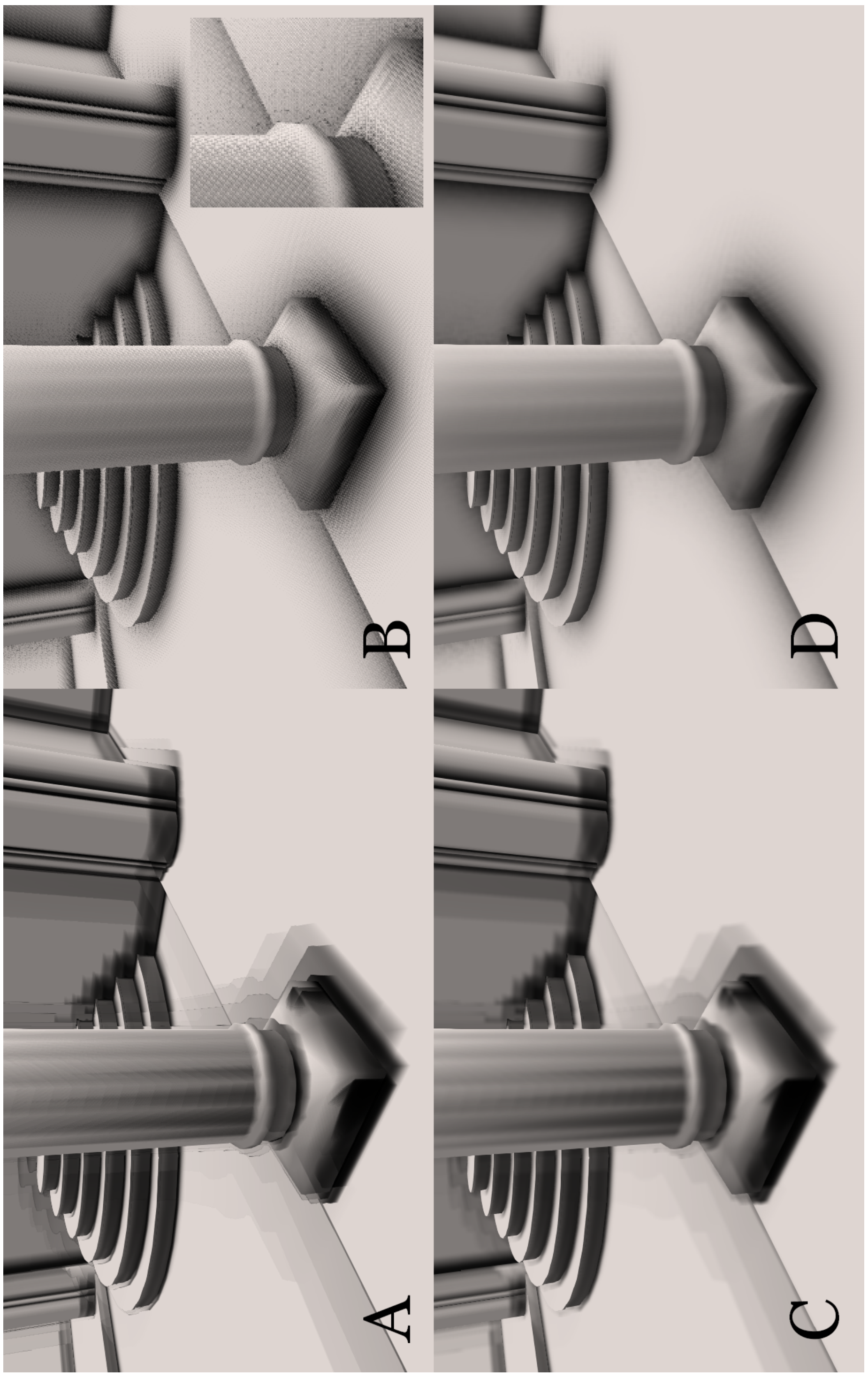


Figure 2.2.6: Banding artifacts without per-pixel randomization (A), noise-artifacts caused by per-pixel randomization (B). C, D: The respective results after two 7 tap directional bilateral box blur filters are applied. All images use nine AO samples per pixel.

2.3.1 Utilizing the low frequency nature of ambient light

Ambinet light tends to be bounced from far away, and cast shadows of low frequency. A handfull of techniques utilize this for enhancing quality or improving performance of AO.

2.3.1.1 Bluring

Randomization sample positions per pixel removes banding issues, but introduce high frequency noise. Because AO is a low-freqency phenomena, a simple solution would be to just apply a low-pass filter to the resulting image. However, bluring with a regular gaussian kernel will cause shadows to bleed between surfaces at different depth and orientation. I.e. an object in the front might get a too dark outline and a white halo, or shadows from cracks might bleed onto the cracked surface. To solve this, one can use a *bilateral filter*.

A bilateral filter is defined as a convolution with a kernel varying across the filtered domain, and is often known as a content aware filter. My implementation of such a filter is thoroughly explained in section 3.2.4.

Bilateral blur is an enhancement used bymost real-life implementations of AO. The improved quality it offers can be seen in figure 2.2.6.

2.3.1.2 Rendering AO in a lower resolution

A trivial optimization of expensive rendering techniques is to render at a lower resolution, and sacrifice some quality. However, this sacrifice is smaller for ambient lighting models than full lighting models, as ambient phenomena tend to result in relatively low frequency images. To avoid the blockyness and shadow bleeding caused by composing low-resolution AO into the higher-resolution final image, most techniques using this approach utilize some form of bilateral upsampling.

Bavoil and Sainz (2009a) suggests to render the AO-pass in half resolution, and then upsample it during the edge-preserving blur. The quality of this approach is further improved by Bavoil and Sainz (2009b), which computes AO in full resolution for pixels where the low resolution would be most noticable, using the min-max range of low-resolution AO values in the vicinity of the shaded pixel as a heuristic. However, both approachs might fail to capture occlusions from objects too small to be rasterized in half-resolution.

Finally, Hoang and Low (2010) renders AO in a whole mip-chain of resolutions, starting with the coarsest resolution and upsamples it using an edge preserving filter while rendering the next (finer) resolution; in effect accumulating occlusions from many levels of detail. The radius used in this approach is constant in image-space, resulting in good texture cache utilization while still capturing occlusions from both large, distant occluders and finer details.

However, all approaches to rendering AO in low resolutions can cause *temporal flicker*; i.e. the ambient factor of a point dramatically changing from frame to frame, due to the changing discretization of the scene being amplified by the low resolution.

2.3.1.3 Reading from a lower resolution G-buffer

An alternative to rendering fewer pixels, is to minimize the cost per pixel. The main bottleneck in AO is bandwidth. Specifically, the large and often random image-space sampling pattern emplyed by AO makes the texture cache practically useless. If the read G-buffer texture have a lower resolution, image-space distances would naturally become smaller, and the chance of finding the read texel in the cache higher.

Filion and McNaughton (2008) observes that the low frequency nature of AO makes it uneccessary to sample the G-buffer at full resolution, and uses a down sampled version as input to the AO-pass. The need for rendering the G-buffer in full resolution in the first place is caused by the G-buffers being used by other passes in the presented pipeline. However, the cost of downsampling is paid back by the improved texture cache usage.

This approach is taken even further by McGuire et al. (2012), who creates a full mip-chain of the G-buffer. The projected distance from the shaded pixel to the sample point is used for deciding which mip-level to sample.

This approach is not plagued with the temporal flicker described in section 2.3.1.2. In addition, the desired radius of the AO no longer has an impact on the performance, and the desired quality/performance can be adjusted by a single variable, namely at which radius to switch to a coarser mip-level.

The mip-chain-based enhancement of McGuire et al. (2012) is not directly applicable to a WebGL, as GLSL ES lacks functinality for fetching values from an explicit mip-level of a mip-mapped texture (texelFetch). However, it might be possible to use one sampler unit per mip-level, and hardwire distant texture fetches to use the sampler units of the coarser mip-levels.

2.3.2 Utilizing temporal coherence

Most real time rendering applications tend to have a high degree of temporal coherence. Viewer position and orientation vary little from a frame to the next, and most dynamic geometry tend to change only by a small delta in position and orientation each frame. A world-space point visible in a given frame is hence likely to be visible in the previous frame, and shading caluclations can be reused if stored between frames.

This temporal coherance have been used previously to enhance shadow mapping in Scherzer et al. (2007). Nehab et al. (2007) formulates a general caching scheme for utilizing temporal coherence in a deferred rendering setting, and introduce the concept *reverse reprojection* for locating the currently shaded pixel in a previously rendred frame. (Note that this is equvivalent to what was utilized in Scherzer et al., 2007)

Let t_f be the post-projection world space position of the current pixel, the relation to the post-projection position in the previous frame t_{f-1} can be formulated as

$$t_{f-1} = P_{f-1}V_{f-1}V_f^{-1}P_f^{-1}t_f$$

where V_i and P_i is the view- and projection matrix for frame i, respectively. To aquire the pixel position within the previous G-buffer, simply perform perspective division on t_{f-1} and remap it to [0,1] range.

The matrix $P_{f-1}V_{f-1}V_f^{-1}P_f^{-1}$ is constant for the whole frame and can calculated once and passed as a uniform. This approach still requires one matrix mutiplication per pixel, but the performance gain and increased quality far outweights this cost in the case of AO. Mattausch et al. (2010, 2011) uses the reverse reprojection approach to refine AO estimation over time, interpolating between new contributions and previous contributions stored in a history buffer based a calculated confidence value.

As all caching schemes, utilizing temporal coherence can cause cache invalidation issues. In this case this is caused by dynamic scenes, Mattausch et al. (2011) discuss a possible solution to this problem by storing 3D optical flow in the G-buffer.

There is theoretically nothing preventing a WebGL-implementation of this approach. However, due to the inherent complexity of the approach I have focused on getting results from simpler enhancements.

2.3.3 Aquireing more information about the geometry

2.3.3.1 Guard banding

The intrinsic problem of missing information about the geometry out side of the field of view can be solved quite successfully by simply rendering depth information for a larger field of view. This technique is called guard banding. The downside of guard banding is a relatively large memory footprint that scales non-linearly with resolution, as discussed thoroughly in McGuire et al. (2012).

An alternative approach to hiding this artifact is to extrapolate based on the values at the edge of the framebuffer, by setting sampler wrap-mode clamp-to-edge (Mattausch et al., 2011). Alternatively, using texture border with the color of distant samples will fade out the effect of most estimator functions. In the prototype developed during this project I use the clamp-to-edge approach, allthough there is theoretically nothing making guard-banding unsuitable for a WebGL implementation. Texture borders are, however, not available in WebGL.

2.3.3.2 Using multiple layers

The typical approach to deferred shading exclusively use values from the front-most layer of the rendered geometry, ignoring all surfaces occluded by surfaces closer to the viewer. Depth peeling is an old technique for order-independent transparency, Everitt (2001) explains how the technique can be accelerated using the hardware depth/stencil buffer. The muliple “layers” achieved through depth peeling can be used to partly solve the problem of approximation errors due to missing information.

This is implemented by Ritschel et al. (2009) and Bavoil and Sainz (2009b) among others. Bavoil and Sainz (2009b) simply calculates the occlusion value of the shaded pixel based on each individual layer, and uses the maximum (darkest) of the calculated occlusion values. Ritschel et al. (2009) assumes closed meshes, and counts a point as inside a solid if it lie behind a front-facing layer and in front of a subsequent back-facing layer. This is a more conservative (and correct) approximation based on the available information, but the requirement of closed meshes can be problematic in cases such billboards commonly used for vegetation (Bavoil and Sainz, 2009b).

While using multiple layers typically results in a better approximation, it introduces overhead due to the requirement of rendering the full geometry for each layer. It also makes the AO-pass heavier as all layers have to be sampled for each AO-sample. A final consideration is that storing multiple layers of the G-buffers will introduce a bigger memory footprint. Clever buffer re-use could possibly minimize this effect.

2.3.3.3 Using multiple points of view

In cases where many surfaces overlap, a large number of depth-peeling layers may be required to hide all artifacts. Ritschel et al. (2009) suggests to instead use fewer layers rendered from different points of view. Vardis et al. (2013) demonstrates a general technique for combining information from different views, and specifically reuses information from shadow maps and other views that are readily available when calculating the obscurance, hence avoiding some of the negative performance implications of multi-view and multi-layer rendering.

Similarly to techniques utilizing temporal coherence (section 2.3.2), the technique of Vardis et al. (2013) uses reverse re-projection to locate the shaded point in different views rendered at the same point in time. In addtion, the technique utilize an *importance sampling* scheme to decide in which views to place the AO-samples. This is done by calculating a confidence value for the point in the different views based on relative orientation and proximity.

In addition to using existing views such as shadow maps, Vardis et al. (2013) suggest to create a “rig” of phantom cameras at a fixed position relative to the viewer.

2.3.3.4 Real-time volexization

An alternative to stroing the depth or position of the pixel in a G-buffer, is to interpret the pixel data as series of binary values, each represeting the occupacy of a voxel along the pixel-ray. This makes it possible to implement an object-space approach to ambient obsucrance in image-space (sic), as explained in Reinbothe et al. (2009).

Unfortunately, the pixel formats available in WebGL would limit this approach to 32 voxels along the z-axis, which would result in a prohibitively coarse discretization of the geometry. A last aspect making this approach difficult in a WebGL-implementation is the lack of binary operators in shader programs.

2.4 GPGPU-based approaches

Without the limitations of the standard graphics pipeline, it is possible to improve image-space ambient obscurance further. Bavoil (2011) shows that HBAO can be implemented using compute shaders. However, GPGPU does not only offer reudced overhead for existing algorithms, but also opens for new algorithms with asymptopically better time complexity.

This is done in Timonen (2013) by realizing that image-space AO is separable in azimuthal directions across the depth G-buffer. The presented algorithm spawns one thread per azimuthal direction, and performs an incremental line-sweep in each direction while maintaining a data structure that makes it possible to find the occluder casting the largest obscurance along the swept line in amortized constant time. While a regular implementation of image-space AO has a quadratic time complexity, the *Line-sweep Ambient Obscurance*-algorithm achieves linear time. It relies on the simplification that only the most significant occluder for a given point in each azimuthal direction needs to be considered, but the results are practically indistinguishable from approaches evaluating the full occlusion (Timonen, 2013).

As opposed to other enchancements listed in this thesis, line-sweep ambient obscurance is not strictly orthogonal to the estimator function used. Timonen (2013) demonstrates the technique using the horizon-based estimator of Bavoil et al. (2008), but it can be combined with any estimator function that allows evaluation as a combination of calculations done per azimuthal direction. This is likely to include (at least) most solid angle-based estimators.

However, while there exists a draft for WebCL, a browser-based API for GPGPU, the specification is incomplete and largely unimplemented by most browser vendors at the time of writing. This ultimately makes GPGPU-based approaches less relevant for this thesis.

Chapter 3

Implementing ambient obscurance in WebGL

In this part I will shed light on some of the considerations faced when implementing ambient obscurance in WebGL, and present the prototype developed during the work on this thesis.

Integrating image-space techniques in an application requires background knowledge of deferred rendering techniques. In my experience, it is easy to get minor things like the depth G-buffer wrong, resulting in strange artifacts. For this reason I hope this part may double as an implementation guide for future implementers.

3.1 Implementation considerations

Even though Image-based effects are simple in theory, one is often faced with challenges during implementation. Either due to limitations of the target platform or because of its many pitfalls combined with the difficulty of debugging shaders. While the major issues making some approaches unsuitable for WebGL have been accounted for in the previous sections, I here examine the more low-level considerations and challenges.

3.1.1 Deferred- versus Forward rendering

Deferred shading is a concept that was introduced by Hargreaves and Harris (2004), in order to decuple shading cost from geometry complexity. The technique can be summarized as the following: Instead rendering the geometry with the final shading, render the surface properties to different textures called G-buffers, which acts as input to later passes that are combined to create the final shading.

With reguar forward rendering, a pixel may be shaded multiple times due to overdraw, and hence the worst case complexity of rendering is number of objects

$\times$ visible pixels $\times$ number of effects and lightsources. With deferred shading, all effects and lights are applied at most once per visible pixel, and the worst case complexity is reduced to number of objects + visible pixels $\times$ number of effects and lightsources. However, this is not the key selling point of deferred shading, as this can also be achieved using an "early-z pass", i.e. a pass rendering the scene only to the depth buffer, prior to rendering the fully shaded geometry with a less-or-equal depth comparison function.

A more compelling feature of deferred shading is that it allows a very large number of light sources with great ease. This is done by calculating the lighting contribution for a given light source on the screen-space projection of the geometry representing its light volume, and additively blend together the results. While I am (in this thesis) not interested in an infinite number of point light sources, the introduction of deferred shading was partly motivated by the shader length limitation found in the early programmable GPUs. A limitation that is once again relevant today when targeting WebGL.

In addition to performance gain and increased scalability in number of materials and light sources, deferred shading provides a big architectural benefit. The shader programming is made simpler as the the G-buffers act as a common "interface" between effects and geometry/surface information. In forward-rendering, each shader rendering geometry would have to be combined with every effect it could possible be used with, known as the shader combination-problem. An alternative approach partly solving this problem is to use a so-called "übershader": A single big shader with lots of preprocessor directives. However, this approach results in an exponential number shaders that have to be compiled load-time when using GLSL.

3.1.1.1 Bandwidth and G-buffer rendering overhead

On mobile platforms, the greatest bottleneck is bandwidth (Lassen, 2010). This greatly limits how much information can be read from G-buffers in a deferred shader, and it becomes important to store as much information in as few bytes as possible.

An additional consideration, is that while most modern desktop graphics hardware support rendering to multiple render targets in one pass in hardware, this is not common in current mobile hardware, and hence not possible in WebGL. This poses a great challenge for deferred techniques, as the full scene geometry will have to be rendered once per G-buffer.

3.1.1.2 Antialiasing

WebGL lacks multisampled renderbuffers, but many implementations provide full-screen anti-aliasing of the backbuffer. The rendering to textures that deferred shading implies means that we will have to sacrifice the anti-aliasing we get "for free" in a regular forward rendering.

However, rendering the ambient occlusion directly to the backbuffer also means that we are unable blur the result, which would cause a severe *drop* in quality, which is just what we are trying to avoid. During prototyping I have experimented with multiple approaches: A full forward approach, a fully deferred approach, and a hybrid approach multiplying deferred AO on top of a regular forward rendering of the scene.

3.1.1.3 Experimental results

In terms of quality, I got best results with the hybrid approach, which combines the benefits of bilateral blur with only visible aliasing artifacts between surfaces of very different ambient lighting (e.g. small cracks in a flat surface). In terms of performance, it is difficult to beat the deferred approach as long as we only need the G-buffers already rendered for the AO. However, if texture mapping or more advanced shading is desired, the deferred approach is likely to require rendering of more G-buffers, and ultimately end up being both worse looking *and slower*. The benefits of deferred shading in WebGL compared to the rather large geometry rendering overhead per G-buffer could be explored in a future study. A supersampling-based approach for anti-aliasing could also be of interest in that case.

To summarize, one should try to pack as much information as possible in as few (and narrow) G-buffers as possible. However, it turns out that the pixel format limitations of WebGL poses yet another challenge here, as I will discuss in the next section.

3.1.2 Pixel format limitations

While most modern desktop computers support a variety of different pixel formats, including floating point pixel formats, the core WebGL specification only provides pixel formats up to 32 bits per pixel. Further, WebGL only provides a maximum of 8 bit per component, and up to four components (Khronos, 2013).

To circumvent the limitation of the maximum allowed bits per component, we need to pack values across multiple components. A typical implementation would just bit shift and mask the binary representation, however, WebGL does not support bitwise operators, so it needs to be done with arithmetic instead. The following equation packs a value between 0 and 1 into two 8 bit components, resulting in 16 bit precission:

$$v_{packed} = v_f - \begin{bmatrix} \frac{(v_f)_y}{255} \\ 0 \end{bmatrix} \tag{3.1.1}$$

$$v_f = v_s - \lfloor v_s \rfloor$$

$$v_s = \begin{bmatrix} 1 \\ 255 \end{bmatrix} d$$

Intuetively, this stores the 1/255ths of the value in the Y-component and the rest in the X-component. Unpacking the value requires just a dot product with a scaling vector:

$$d = \begin{bmatrix} 1 \\ \frac{1}{255} \end{bmatrix} v_{packed} \tag{3.1.2}$$

3.1.3 Reconstructing view-space position in image-space

A requirement for most image-space ambient obsucrance and occlusion techniques is to read the view- or world-space position of visible surface patches in the pixel-neighbourhood. As explained in section 1.5.3, this is done by in an earlier G-buffer pass storing information about the geometry. The G-buffer can then be accessed randomly by the AO shader.

A naive approach is to store the world-space position of the pixels directly in a G-buffer. This is, however, a very bad idea in WebGL. The lacking of floating point textures would limit the possible size of the scene to a unit cube, and with the pixel format limitations of WebGL you would have only 8 bit percission per dimension, which is useless for most real use cases. Storing a (scaled) view-space position would help on the former problem, but even more percission would be lost as there would be a lot of range spent on space outside of the of the visible frustum.

Storing the full position is actually redundant. Intuetively, we have already limited two degrees of freedom by knowing the pixel position, we just need to know the depth to be able to reconstruct an exact view-space position.

3.1.3.1 Using inverse projection directly

The straight-forward approach is to first reconstruct the NDC position, and then use the inverse projection matrix to reconstruct view-position from NDC.

The x- and y-components of the NDC position are easily reconstructed from the pixel position, simply by mapping it to $[-1, +1]$ range. The NDC z-component can be stored in a G-buffer and remapped to $[-1, +1]$ if needed. Finally, the w-component is, by definition, always 1 for NDC. Multiplying the NDC-position by the inverse projection matrix we get a 4D homogenous view-space position. To aquire the regular cartesian view-space position we need to divide the vector by its w-component. Careful readers might notice that we just skipped the intermediate clip-space representation, however, this apporach is valid due to the nature of projective spaces.

While an intuitive approach, this requires a 4x4 vector-matrix multiplication and one division per pixel (in addition to some remapping). With knowledge of how the projection matrix is contructed one could simplify this operation, but we will see in the next section that there are more convenient ways to go.

3.1.3.2 Storing linear depth

It is tempting that the former approach seemingly do not require storing any additional information other than what is readily available in the hardware z-buffer. However, a lot of hardware and APIs, WebGL included, will not give the user direct read access to the z-buffer. Additionally, storing non-linear z-over-w values skews the precission heavily towards the near part of the view frustum, and makes reconstructing the view-space position more complicated than it needs to be.

Hargreaves and Harris (2004) sugested to instead store the distance from the camera in a G-buffer, and use it to scale the a unit view- or world-space ray going through the pixel. While simple and intuitive, this is not the most convenient approach in WebGL as the range of each component is still limited to [0,1]. Instead, we introduce a quantity commonly referred to as *linear depth*:

$$d_p = -\frac{p_z^{view} - z_{near}}{z_{far} - z_{near}}$$

The linear depth is simply the relative z-location of the position inside of the view-frustum, i.e. $d_p = 0$ for points on the near plane, $d_p = 1$ for points on the far-plane, and $d_p = \frac{1}{2}$ for points right in the middle of the two.

3.1.3.3 Interpolating the frustum corners

On the same note as above, I define image-space to be the full relative position inside the frustum:

$$c^{image} = \begin{bmatrix} \frac{c_x^{clip}+1}{2} \\ \frac{c_y^{clip}+1}{2} \\ d_p \end{bmatrix}, \ x, y, z \in [0, 1]$$

This image-space vector can be used to do a plain old trilinear interpolation of the frustum corners:

$$p^{view} = Lerp(p_{far}^{view}, p_{near}^{view}, c_z^{image})$$

$$p_{far}^{view} = Lerp(Lerp(f_{+++}^{view}, f_{-++}^{view}, c_x^{image}), Lerp(f_{+-+}^{view}, f_{--+}^{view}, c_x^{image}), c_x^{image})$$

$$p_{near}^{view} = Lerp(Lerp(f_{++-}^{view}, f_{-+-}^{view}, c_x^{image}), Lerp(f_{+--}^{view}, f_{---}^{view}, c_x^{image}), c_y^{image})$$

Each corner of the frustum are here notated as f^{view} subscripted with the sign of components of the coresponding NDC unit cube corner.

Further, we know the far plane and near plane are parallell to our coordinate system's xy-plane, and that the x- and y-axis of all coordinate systems are aligned, so we can simplify finding the position on the near and far plane:

$$p_{far}^{view} = f_{+++}^{view} + \begin{bmatrix} (d_{far}^{view})_x \\ 0 \\ 0 \end{bmatrix} c_x^{image} + \begin{bmatrix} 0 \\ (d_{far}^{view})_y \\ 0 \end{bmatrix} c_y^{image}$$

$$p^{view}_{near} = f^{view}_{++-} + \begin{bmatrix}(d^{view}_{near})_x \\ 0 \\ 0\end{bmatrix} c^{image}_x + \begin{bmatrix}0 \\ (d^{view}_{near})_y \\ 0\end{bmatrix} c^{image}_y$$

where

$$d^{view}_{far} = f^{view}_{--+} - f^{view}_{+++}$$

$$d^{view}_{near} = f^{view}_{---} - f^{view}_{++-}$$

This approach maps well to a few of vector multiply-adds on modern GPUs.

3.1.3.4 Interpolating the far plane vector

It turns out we can do even better, by observing that the position on the near plane in view-space is always on the line through origio that intersects the point on the far plane. Just interpolating the ray from origo to the point on the far plane by the depth stored in the linear depth G-buffer will not be correct though, as this would assume the near clip plane to be located at z=0. To solve this we simply scale and offset the linear depth value so the range [0, 1] maps to the range$[\frac{z_{near}}{z_{far}}, 1]$, and the formula for the view-space position becomes:

$$p^{view} = \left(\left(1 - \frac{z_{near}}{z_{far}}\right) c^{image}_z + \frac{z_{near}}{z_{far}}\right) p^{view}_{far} \tag{3.1.3}$$

where

$$p^{view}_{far} = f^{view}_{+++} + \begin{bmatrix}(d^{view}_{+})_x \\ 0 \\ 0\end{bmatrix} c^{image}_x + \begin{bmatrix}0 \\ (d^{view}_{+})_y \\ 0\end{bmatrix} c^{image}_y$$

This approach should require only about two or three multiply-add instructions, which is pretty good. Storing the remapped linear depth value in the G-buffer directly would make this even cheaper, but at the cost of losing some depth percission. Because of the problems with precision already posed by WebGL's pixel format limitations, I stay with the normalized linear depth approach in equation (3.1.3) for my implementation.

3.1.3.5 Special cases for the currently shaded pixel

Aquireing the view-space position of the currently shaded pixel can be further optimized. If ambient occlusion is applied in a regular forward pass, getting this value is trivial; simply provide the pixel shader with the view position as a varying. In full deferred rendering this is not possible, as the shader is just applied to a screen aligned quad, not the scene geometry itself. However, we can get the vector from origo to the pixel projected on the far plane "for free" by interpolating it in a varying: In the vertex shader, simply output the view space position of the current vertex projected on the far plane and let hardware interpolation do the job. A similar approach is used to reconstruct world-space position in Wenzel (2006) and Filion and McNaughton (2008).

3.1.4 Reconstructing normal from depth

One technique to avoid a sperate full geometry pass to generate a normal G-buffer is to reconstruct it from the depth information. On desktop OpenGL, the partial derivative operators dFdx and dFdy can be applied to a reconstructed view position to aquite view-space tangent-vectors, and the cross product of the normalized tangent-vectors would be the view-space normal. This technique is used in e.g. Bavoil and Sainz (2009a).

Unfortunately, partial derivative operators are not supported by the current version of WebGL. However, the same approach can be applied by explicitly reconstructing the view space position of neighbouring pixels in the x- and y-direction. A shader program implementing this approach is provided in appendix appendix D.

An artifact by this approach is visible 1-pixel errors at large discontinunities in the depth G-buffer. McGuire et al. (2012) shows that the artifacts cused by this error can be completely hidden by the bilateral blur pass. For obvious reasons, any technique reconstructing normals from depth information will reconstruct face normals. For most ambient occlusion usages, this is a good thing, as interpolated normals can cause self-occlusion. However, if the same normals are used for direct light calculations, tesselation of smooth surfaces are likely to be revealed.

A realization I made during implementation is that this technique is very sensitive to the resolution of the depth G-buffer. Ultimately, this made me decomission this approach, as G-buffer precission is problematic in WebGL.

3.1.5 Storing compressed normals

When using two 8-bit components for depth as explained in section 3.1.2, we have two 8-bit components available for other data. If we could fit the view-space normal into these two components we can avoid having to render the geometry twice, while at the same time not relying on reconstructing it from depth information as explained in section 3.1.4.

Intuitively, we know that 3D normals have unit length and hence only has two degrees of freedom, after all, they can be represented as spherical coordinates. Storing normals as spherical coordinates, however, is not very attractive in this case; the required trigometric operations are very ALU-heavy unless a lookup table is used (Pranckevicius, 2010).

3.1.5.1 Storing only two components

Valient (2007) propses to store only the x- and y-components of the view-space

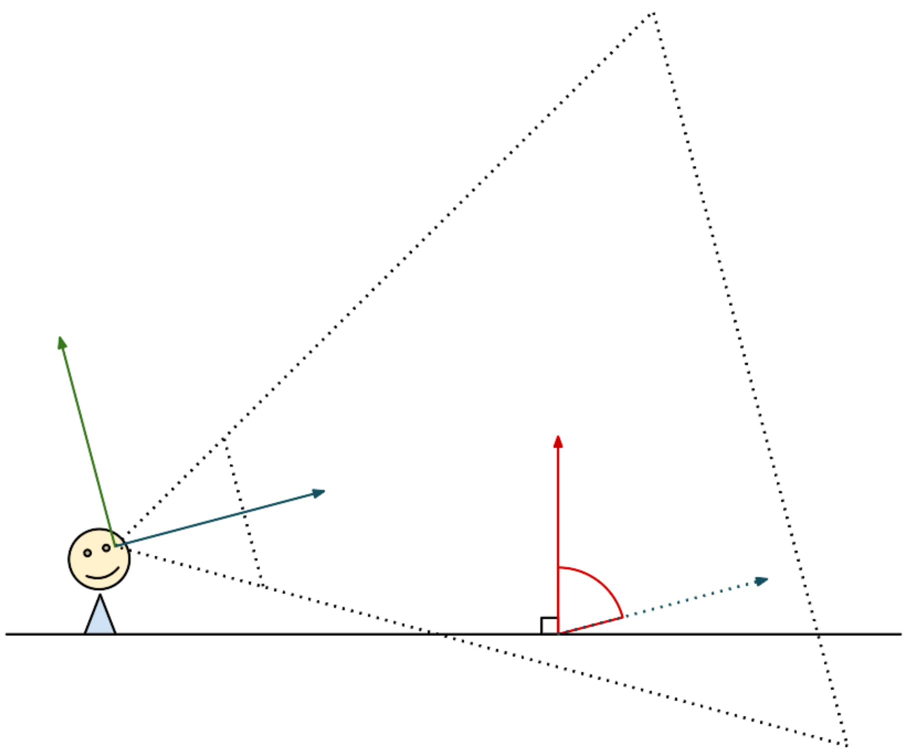


Figure 3.1.1: A case where cosine angle between the view-space normal (red) and the view direction (blue) is positive

normal, and then reconstruct the last component assuming it is positive using the following equation:

$$n_z = \sqrt{1 - n_x^2 - n_y^2} \tag{3.1.4}$$

The assumption of the z-component of the view-space normal being positive tends to hold for most visible points due to backface culling. However, due to the nature of the perspective projection, this is not true for all cases. A quite common case of the latter is a viewer standing on the ground, looking slightly above the horizon, with the ground is still inside his field of view (see figure 3.1.1). This will cause the cosine angle between the ground normal and the view direction to become greater than zero (i.e. “facing away”), and hence the assumption of the z-component always having the same sign fails.

I found that, when using this approach in combination with solid-angle based estimators, changing the orientation of the camera would result in a sudden darkening of surfaces just before the surface left the field of view. This artifact is very disturbing, possibly amplified by the human eye’s sensitivity to rapid change in the outer field of view.

Note that we still need to store the x and y components in normalized range and map them back to [-1,+1] before we can reconstruct the last component using equation (3.1.4).

3.1.5.2 Sphere projections

Rather than making assumptions about some of the componenents, we can use a more traditional unwrapping approach. Mapping positions on a sphere to two dimensions happens to be a well studied problem, possibly due to the fact that our planet resembles one. As such, there exists a multitude of different map projection methods. For the problem at hand, however, almost any method that is simple to compute will suffice.

Mittring (2009) proposes the following projection:

$$v = \sqrt{\frac{n_z + 1}{2}} \frac{n_{xy}}{\|n_{xy}\|} \tag{3.1.5}$$

And the inverse:

$$n_{xy} = \sqrt{1 - n_z^2} \frac{v}{\|v\|}$$

$$n_z = 2v \cdot v - 1$$

Where n is the view-space normal and v its stored two-component representation. In effect, this is similar to the projection used for spherical environment maps. Both techniques are in fact similar to a traditional map projection technique called Lambert Ezimuthal Equal-Area Projection.

For a comparison of performance and accuracy of different techniques, I refer to Pranckevicius (2010).

3.2 Implemented solution

This section gives an overview of the prototype implemented during the writing of this thesis.

3.2.1 Renderer overview

The geometry is rendered to a texture with a combined depth and normal G-buffer shader (discussed in section 3.2.3). This texture is used as input to the AO pass. The AO pass is rendered to a texture that is used as input to two subsequent 7-tap directional bilateral box blur filters (explained in section 3.2.4). In additional to the result of the previous pass, the bilateral filters need the normal+depth G-buffer as input to use as discriminator. The result of the last blur filter is then used to modulate the result of a regular forward lighting pass using blending. For more advanced shading, the AO texture should be used as input to the lighting pass, not simply modulated on top (shown as the dashed black line in figure 3.2.1). Note that *all* geometry passes perform GPU skinning individually in the vertex shader for skinned models. However, for clarity this is sown as only one box in the diagram.

During benchmarking, I rendred the final result to a texture which was blitted on the backbuffer instead of rendering directly to the backbuffer, in order to make switching between resolutions easy. In effect this resulted in supersampling when rendering to resolutions higher than the resolution of the backbuffer. This looked simply stunning, but was also correspondingly slow.

The point of this hybrid forward / deferred approach is to utilize FSAA of the backbuffer combined with blurring of the AO result, as discussed in section 3.1.1.3. A handfull other approaches were implemented and tested during prototyping, these are shown with colored dashed lines in figure 3.2.1, and will be explained in the next sections.

3.2.2 Ambient obscurance estimator

The earliest implementation used a point-based sampling method similar to that of Mittring (2007), but I failed to achieve good results using this approach with a sensible number of samples. I ended up prefering the Alchemy estimator due to its simplicity and ease of tweaking. The choice of implementing the this estimator, over for instance a horizon-based approach, was partly due to the promising performance and quality demonstrated in McGuire et al. (2011). The fact that it was made for the Xbox 360 was also a big motivational factor behind the choice, as the Xbox 360 has realtively low-end graphics hardware with a limited feature set, similar to WebGL. Other estimators can be a relevant subject of future studies.

The source of the implemented solution is provided in appendix B.

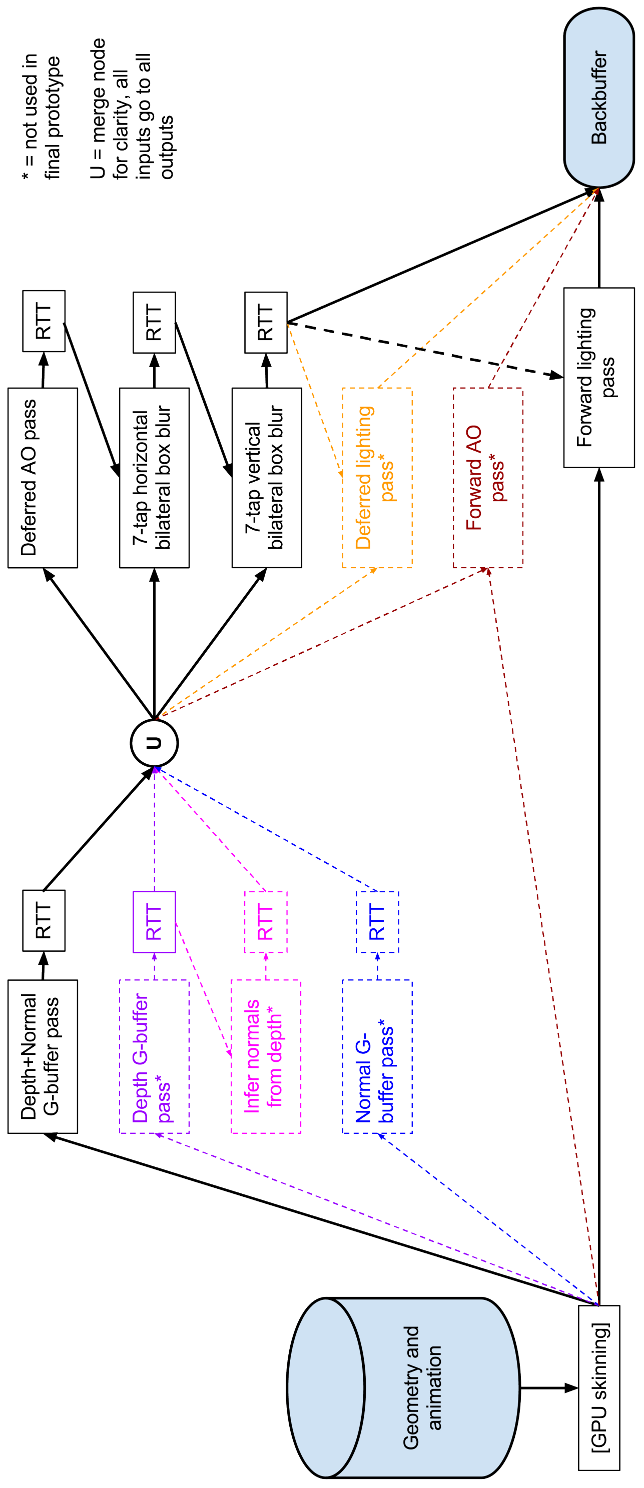
3.2.3 G-buffer layout

The Alchemy ambient obscurance pass requires information enough to reconstruct both the view-position of random samples and and the normal of the currently shaded pixel.

My initial approach was rather trivial, using a full 32-bit RGBA texture for both, and packing linear depth values using four components similarly to the approach in section 3.1.2. This is illustrated with blue and purple dashed lines in figure 3.2.1.

A problem with this approach is that it requires two full geometry rendering passes, as explained in section 3.1.1.1, and that it is wastful in terms of bandwith. A temporary solution to the rendering overhead problem was implemented using the technique described in section 3.1.4. This temporary solution is illustrated with pink and purple dashed lines infigure 3.2.1.

However, I soon found that linear depth values packed in only *two* 8-bit components are good enough for AO, even for a view distances of over 300m. This fact



* = not used in final prototype
U = merge node for clarity, all inputs go to all outputs
Geometry and animation
[GPU skinning]
Depth+Normal G-buffer pass
RTT
Depth G-buffer pass*
RTT
Infer normals from depth*
RTT
Normal G-buffer pass*
RTT
U
Deferred AO pass
RTT
7-tap horizontal bilateral box blur
RTT
7-tap vertical bilateral box blur
RTT
Deferred lighting pass*
Forward AO pass*
Forward lighting pass
Backbuffer

Figure 3.2.1: Overview of rendrer

was combined with the normal compression scheme outlined in section 3.1.5.2 to create the following G-buffer layout:

R	G	B	A
Normal	Normal	Depth (High)	Depth (Low)

Because all required data now fits in a single 32-bit RGBA texture, it only requires a single geometry pass, and uses little bandwidth. The source of the G-buffer rendering pass is provided in appendix A.

3.2.4 Bilateral blur

Because of the immense enhanced quality offered by blurring the result of the AO-pass (see figure 2.2.6), I implemented a bilateral blur pass in the prototype. Some experimentation lead me to the definition of the following filter function, which is a box filter with bilateral weights based on normal and depth differences:

$$c'_p = \frac{\sum_{v_i \in R} c_{p+v_i} w(p+v_i)}{\sum_{v_i \in R} w(p+v_i)} \tag{3.2.1}$$

Where p is the coordinate of the current pixel, c is the values in the color buffer, and R is the set of sampling offsets for the taps. To improve cache coherency and performance, I separate the filter into a horizontal and a versatical pass. While not correct, it gives good results. Each pass uses $R = \{v_{dir}t \,|\, t \in [-3, +3]\}$ to get a 7-tap directional blur in direction v_{dir}. $w(s)$ is my discriminator function, taking both normal and depth differences into consideration:

$$w(s) = w_{normal}(s) * w_{depth}(s)$$

$$w_{normal}(s) = \left(\frac{n_c \cdot n_s + 1}{2}\right)^{k_n} \tag{3.2.2}$$

$$w_{depth}(s) = \left(\frac{1}{1 + |d_c - d_s|}\right)^{k_d} \tag{3.2.3}$$

It is important that the depth discrimnator uses a quantity in world- or view-space. If normalized values are used directly it will be dependent on the near and far plane of the view frustum, which is not a desirable feature. In equation (3.2.3) I use a function of the absolute difference in view-space euclidian distance to the point from the camera. My experience with using a function of only the distance or only the normal did not yield satisfying results.

In theory, equation (3.2.3) would have a problem with surfaces at steep angles with the direction of the blur pass. However, this is not a visible problem, and chances are good for that a second orthogonal blur pass will not to experience the same problem.

For performance reasons I do *not* use a gaussian bilateral kernel, as the falloff caused by the gaussian wheights would decrease the blur effect per pass and

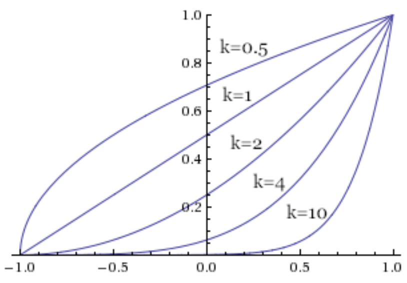


Figure 3.2.2: Plots of the normal discriminator function (3.2.2) for different values of k_n. The horizontal axis is the cosine angle $n_c \cdot n_s$, and the vertical axis is the discriminator value w_{normal}. The normal discriminator is the most important discriminator and least likely to cause artifacts, so I use a high value for k_n (e.g. 10).

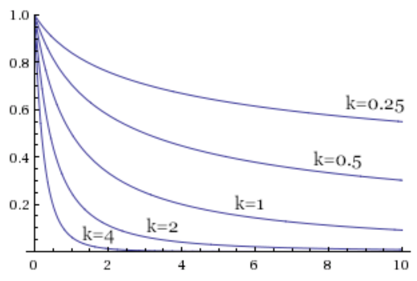


Figure 3.2.3: Plots of depth discrimnator function (3.2.3) for different values of k_d. The horizontal axis is absolute depth difference $|d_c - d_s|$ in decimeters, and the vertical axis is the discriminator value w_{depth}. I found that $k_d = 2$ works well for my test scene.

hence require multiple passes or a bigger kernel to achieve the same strength. A future improvement should however consider this for increased quality.

For the final source code of the implemented bilateral blur filter, see appendix C.

3.2.5 Considered enhancements

I did start some experimentation with using multiple views, but ultimately this was put on hold due to concerns abut the overhead it would introduce when rendering larger scenes, and the requirement of pixel shader branching for an optimal solution. I might have put this away too early though, and it should be considered in a future implementation.

Even though the technique used in Mattausch et al. (2010) is very promising in terms of performance and quality, I chose to focus on simpler techniques due to the inherent complexity of the approach. I would very much like to see an implementation of this technique running in a browser in the future.

Chapter 4

Results and discussion

4.1 Method

Throughout this project I have prototyped multiple approaches to Image-space Ambient Obscurance and Occlusion in WebGL. While a working prototype is a testament to the applicability of a given technique in itself, quality and performance have been the key criterias driving this work further.

4.1.1 Benchmarcing of performance

While a tool such as Nvidia Shaderperf can be (and have been) used to assess the complexity of a given shader, it is unable to predict the actual performance when texture sampling is involved. This is probably due to texture cache usage being an unpredictable phenomena during static analysis. This is one of the reasons I have emphasised "real-life" performance benchmarking of the prototype, running it in the browser, with a fully animated scene of sufficient complexity.

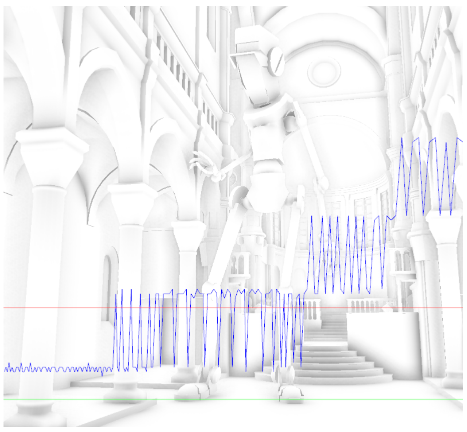


Figure 4.1.1: Plotting the observed difference in time from frame to frame

4.1.1.1 Sources of error

However, there are some gotchas when doing benchmarking in the browser. Many impelemntations of WebGL (Chrome included) uses the Angle, which translates WebGL shaders and commands to DirectX9 commands.

This include unrolling all loops and potentially changing the shaders in other unpredictable (and unobservable) ways. I observed a lot better pixel throughput when running the same fragment shaders on a native OpenGL implementation, which can be a sign that something fishy is going on, and a further analysis of how Angle transforms shaders could perhaps shed some light on this.

VSync poses a minor problem when doing benchmarking in the browser, as it is impossible to measure higher framerates than the refresh rate, typically 60 frames per second (FPS). In addition, I observe that the measured frame interval for a single frame is "snapped" to fractions of the refresh interval (See figure 4.1.1). To overcome this problem, I measure the number of rendered frames over a period of about 8 seconds, I also measure the exact start and end time to avoid errors caused by the fact that t=8 seconds might be in the middle of a frame. An alternative approach would be to measure the time it took to render a fixed number of frames, however, this will introduce unbounded benchmarking times. The choice of 8 seconds is simply due to the animations in the scene looping with an 8 second interval, and the performance being affected by the proximity of visible surfaces.

For timing the internal JavaScript getTime() function was used.

4.1.1.2 Test setup

Most image-space effects scale lineary with the resolution of the rendered image. It has also been of interest to see how the perfomance and quality of image-space ambient obscurance scales with the number of sampling points. Because of texture caching in modern graphics hardware, the pattern of sampling points also plays an important role in the performance and quality of a technique, which is reflected in the benchmark results.

The test setup have been an Intel Core i7-3610QM (2.3 GHz) with the mid-range dedicated graphics accelerator GeForce GT 650M (348 Cuda Cores), running Chrome v. 28.0.1500.95m in Windows 7 Professional (SP1).

I tested resolutions ranging from 256^2 to 1792^2, using 3, 9 and 32 samples per pixel, and for each configuration measured the timings without AO (baseline), with only AO, and with AO and bilateral blur.

4.1.2 Assessment of quality

As ambient occlusion is not a physically-based model, but rather an empirical model seeking to produce visual cues that are easy to pick up by the human visual system, testing the difference between photographs and images synthesized by my implementation is not very interesting. Testing against high-quality rendred images could be of use when implementing realistic estimator functions such as Bavoil et al. (2008), however, my choice of the esthetics-oriented Alchemy estimator calls for a more subjective assessment of quality.

4.2 Results

The benchmarking results are provided in a digital attachment. The main observations I would like to draw from the testing and development can be summarized as follows:

Performance is dependent on the number of shaded pixels This comes as no surprise, as ambient obscurance is an image-space effect.

Performance is dependent on radius I found that the radius of the AO has a great impact on the performance, especially at higher resolutions. This is likely due to the radius being defined in world-space, which projects to increasingly longer distances in image-space with higher resolutions.

Per-pixel randomization With per-pixel randomization of sampling points, as little as 3 samples per pixel can give results recognizable as ambient obscurance. Without per-pixel randomization two samples per pixel typically just looks like a couple of incorrect hard shadows with a strange deformation.

Unfortunately, per-pixel randomized sampling points combined with samples being far apart in image-space is a worst-case access pattern scenario for the texture cache. However, the reduced number of required sampling points outwheights the increased per-sample cost, as evidenced by the attached results.

Bilateral blur has a neglectable overhead Blurring the result of the AO-pass with the two 7-tap directional blur filters presented in this thesis has a neglectable overhead compared to cost of the AO-pass itself. According to Nivida shaderperf, the blur shader is actually about as complex as the AO shader in terms of cycles, but it has almost no performance implications due to high cache utilization.

Number of samples It comes as no surpsise that the number of samples affect the overall performance, as randomly sampling the G-buffer is bandwidth heavy due to the poor texture cache utilization.

However, I found that using per-pixel randomziation and bilateral blur, as few as 3 AO-samples per pixel gives results recognizable as ambient obscurance for a small radius (0.5m). In this case the perfomrance is well within real-time for all tested resolutions.

A good tradeoff between performance and quality for radii up to 1m (which is what I personally prefer) is 9 samples per pixel. This results in real-time frame rates up to a resolution of 1280^2, at which it drops to 21.83 FPS. At higher resolution I do not achieve acceptable performance. With a radius of 2, no higher resolution than 1024^2 yield acceptable performance.

I also included test results for 32 samples, however, this is just for comparison, and not a recommended setting, as the quality gained by multiple AO-samples seems to decrease drasticly when passing 16 samples. This is somewhat surprising, considering that 16 samples is still a quite sparse kernel. However, this is likely to be caused by the bilateral blur filter being applied to the result. The performance of 32 samples at 1m is acceptable up to a resolution of 1024^2.

Performance depends on the proximity of objects The observations from my implementation is consistent with previous work in the field pointing out this effect, whch is caused by the radius being defined in view-space. Clamping the radius to a maximum distance in screen-space is effecient for preventing performance swings due to this, but the effect can be noticable.

4.3 Conclusion and further work

I have done a survey of current techniques in the field of image-space ambient obscurance, and successfully implemented a working prototype in WebGL.

This thesis only touched the surface of the many possibilities that WebGL offers for portable real-time rendering applications. A natural extension would be to look more advanced global illumination approximations, for instance the screen-space directional occlusion algorithm presented in Ritschel et al. (2009). As mentioned in previously, a more thorough study of the performance implications of deferred rendering in WebGL would be useful for future implementers.

Allthough a lot of work has been done since 2007, there are yet some opportunities for more research into the field of image-space ambient obscurance and occlusion. The pieces resulting from the dissection of current implementations in chapter 3 can quite easily be combined in a variety of ways, possibly with intereseting results. Trying out the hiearchical depth buffer approach by McGuire et al. (2012) in WebGL could be an interesting task in itself, as the retrofitting needed to make it suitable for WebGL is likely to affect its performance and applicability. Further, I believe combing this approach with other estimator functions can yield interesting results. If this task is undertaken, the novel estimator function of Szirmay-Kalos et al. (2010) should be considered as a subject, to shed some light on how the technique interacts with volume-based estimators.

4.4 A step back

There is hope for global illumination methods and physically based rendering such as path tracing, but as mentioned in the introduction we are not quite there *yet*. In the meanwhile, we can have a lot of fun with our triangles and image-space approaches. Maybe, as one today might marvel in amusement at the arcane tricks of the software rendering-era, we might in the future look back similarly at all the crazyness todays industry have gone through to achieve realism on top of the *great hack* that is rasterized graphics.

Bibliography

Bavoil, L. (2011). Horizon-based ambient occlusion using compute shaders. *NVIDIA Graphics SDK 11 Direct3D*, 2.

Bavoil, L. and Sainz, M. (2008). Screen space ambient occlusion. *NVIDIA developer information: http://developers. nvidia. com*, 6.

Bavoil, L. and Sainz, M. (2009a). Image-space horizon-based ambient occlusion. *Shader X*, 7:425–444.

Bavoil, L. and Sainz, M. (2009b). Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH 2009: Talks*, SIGGRAPH '09, pages 45:1–45:1, New York, NY, USA. ACM.

Bavoil, L., Sainz, M., and Dimitrov, R. (2008). Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 talks*, SIGGRAPH '08, pages 22:1–22:1, New York, NY, USA. ACM.

Bunnell, M. (2005). Dynamic ambient occlusion and indirect lighting. *Gpu gems*, 2(2):223–233.

Christensen, P. H. (2003). Global illumination and all that. *SIGGRAPH 2003 course notes*, 9:31–72.

Cook, R. L. and Torrance, K. E. (1982). A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24.

Dimitrov, R., Bavoil, L., and Sainz, M. (2008). Horizon-split ambient occlusion. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, pages 5:1–5:1, New York, NY, USA. ACM.

Evans, A. (2006). Fast approximations for global illumination on dynamic scenes. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, pages 153–171, New York, NY, USA. ACM.

Everitt, C. (2001). Interactive order-independent transparency. *White paper, nVIDIA*, 2(6):7.

Filion, D. and McNaughton, R. (2008). Effects & techniques. In *ACM SIGGRAPH 2008 Games*, SIGGRAPH '08, pages 133–164, New York, NY, USA. ACM.

Fox, M. and Compton, S. (2008). Ambient occlusive crease shading. *Game Developer Magazine*, pages 19–23.

Hargreaves, S. and Harris, M. (2004). Deferred shading. In *Game Developers Conference*, volume 2.

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*, VRST '10, pages 101–102, New York, NY, USA. ACM.

Hoberock, J. and Jia, Y. (2007). High-quality ambient occlusion. *GPU gems*, 3:257–274.

Kajalin, V. (2009). Screen space ambient occlusion. *Shader X*, 7:413–24.

Kajiya, J. T. (1986a). The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150.

Kajiya, J. T. (1986b). The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA. ACM.

Khronos (2013). Webgl specification version 1.0.2. https://www.khronos.org/registry/webgl/specs/1.0.2/.

Kontkanen, J. and Laine, S. (2005). Ambient occlusion fields. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, I3D '05, pages 41–48, New York, NY, USA. ACM.

Landis, H. (2002). Production-ready global illumination. *Siggraph course notes*, 16(2002):11.

Lassen, A. K. (2010). High performance mobile game graphics with mali. Presentation for ARM Norway.

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

Luft, T., Colditz, C., and Deussen, O. (2006). Image enhancement by unsharp masking the depth buffer. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 1206–1213, New York, NY, USA. ACM.

Mattausch, O., Scherzer, D., and Wimmer, M. (2010). High-quality screen-space ambient occlusion using temporal coherence. *Computer Graphics Forum*, 29(8):2492–2503.

Mattausch, O., Scherzer, D., and Wimmer, M. (2011). Temporal screen-space ambient occlusion. *Gpu Pro 2*, 2:123.

McGuire, M. (2010). Ambient occlusion volumes. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 47–56, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

McGuire, M., Mara, M., and Luebke, D. (2012). Scalable ambient obscurance. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, EGGH-HPG'12, pages 97–103, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

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*, HPG '11, pages 25–32, New York, NY, USA. ACM.

Miller, G. (1994). Efficient algorithms for local and global accessibility shading. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 319–326, New York, NY, USA. ACM.

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

Mittring, M. (2009). A bit more deferred–cryengine 3. In *Triangle Game Conference*, volume 4.

Nehab, D., Sander, P. V., Lawrence, J., Tatarchuk, N., and Isidoro, J. R. (2007). Accelerating real-time shading with reverse reprojection caching. In *Graphics Hardware*, pages 25–35.

Outracks (2013). Uno - a new gpu-powered front-end programming language. http://www.outracks.com/#services.

Pharr, M. and Green, S. (2004). Ambient occlusion. *GPU Gems*, 1:279–292.

Phong, B. T. (1975). Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317.

Pranckevicius, A. (2010). Compact normal storage for small g-buffers. *http://aras-p.info/texts/CompactNormalStorage.html*.

Reinbothe, C. K., Boubekeur, T., and Alexa, M. (2009). Hybrid ambient occlusion. In *Eurographics 2009-Areas Papers*, pages 51–57. The Eurographics Association.

Ren, Z., Wang, R., Snyder, J., Zhou, K., Liu, X., Sun, B., Sloan, P.-P., Bao, H., Peng, Q., and Guo, B. (2006). Real-time soft shadows in dynamic scenes using spherical harmonic exponentiation. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 977–986, New York, NY, USA. ACM.

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*, I3D '09, pages 75–82, New York, NY, USA. ACM.

Rosado, G. (2007). Motion blur as a post-processing effect. *GPU gems*, 3:575–581.

Scheer, F. and Keutel, M. (2010). Screen space ambient occlusion for virtual and mixed reality factory planning.

Scherzer, D., Jeschke, S., and Wimmer, M. (2007). Pixel-correct shadow maps with temporal reprojection and shadow test confidence. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, EGSR'07, pages 45–50, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

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*, I3D '07, pages 73–80, New York, NY, USA. ACM.

Sloan, P.-P., Govindaraju, N., Nowrouzezahrai, D., and Snyder, J. (2007). Image-based proxy accumulation for real-time soft global illumination. In *Computer Graphics and Applications, 2007. PG '07. 15th Pacific Conference on*, pages 97–105.

Smedberg, N. and Wright, D. (2009). Rendering techniques in gears of war 2. In *Game Developers Conference.*

Szirmay-Kalos, L., Umenhoffer, T., Toth, B., Szecsi, L., and Sbert, M. (2010). Volumetric ambient occlusion for real-time rendering and games. *Computer Graphics and Applications, IEEE*, 30(1):70–79.

Timonen, V. (2013). Line-sweep ambient obscurance. *Computer Graphics Forum*, 32(4):97–105.

Timonen, V. and Westerholm, J. (2010). Scalable height field self-shadowing. *Computer Graphics Forum*, 29(2):723–731.

Valient, M. (2007). Deferred rendering in killzone 2. In *The Develop Conference and Expo.*

van Schijndel, J. (2013). The brigade renderer: A path tracer for real-time games. *International Journal of Computer Games Technology*, 2013.

Vardis, K., Papaioannou, G., and Gaitatzes, A. (2013). Multi-view ambient occlusion with importance sampling. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '13, pages 111–118, New York, NY, USA. ACM.

WebGLStats (2013). Webgl stats. http://webglstats.com/.

Wenzel, C. (2006). Real-time atmospheric effects in games. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, pages 113–128, New York, NY, USA. ACM.

Zhukov, S., Iones, A., and Kronin, G. (1998). An ambient light illumination model. In Drettakis, G. and Max, N., editors, *Rendering Techniques 98*, Eurographics, pages 45–55. Springer Vienna.

Appendix A

G-buffer packing and unpacking source code

```
using Uno;
using Uno.Collections;
using Uno.Graphics;
using Uno.Scenes;
using Uno.Scenes.Primitives;
using Uno.Scenes.Designer;
using Uno.Content;
using Uno.Content.Models;

using static SSAO3.BufferData;
using static SSAO3.Sampling;
using static Math;
using static Vector;

namespace SSAO3
{
  public static class BufferData
  {
    public static float3 ReadViewPos(float4 packedData,
        float2 texCoord, float4x4 near, float4x4 far)
    {
      var linearDepth = BufferPacker.UnpackFloat16(
          packedData.ZW);
      return ToViewSpace(float3(texCoord,linearDepth),
          near, far);
    }

    public static float3 ReadViewNormal(float4 packedData
        )
    {
      return BufferPacker.UnpackNormal(packedData.XY);
  }

  public static float3 ToViewSpace(float3 imagePosition
      , float4x4 near, float4x4 far)
  {
    float3 farTL = far[0].XYZ; float3 farTR = far[1].
        XYZ; float3 farBL = far[2].XYZ; float3 farBR =
        far[3].XYZ;
    float3 nearTL = near[0].XYZ; float3 nearTR = near
        [1].XYZ; float3 nearBL = near[2].XYZ; float3
        nearBR = near[3].XYZ;

    float farZ = farTL.Z;
    float nearZ = nearTL.Z;
    float ratio = nearZ/farZ;

    var linearDepth = imagePosition.Z;
    var linearDist = (1 - ratio) * linearDepth + ratio;

    var farDiag = farTR.XY - farBL.XY;
    var farPos = farBL + float3(farDiag * imagePosition
        .XY,0);

    return farPos * linearDist;
  }
}

public static class BufferPacker
{
  public static float2 PackNormal(float3 n)
  {
    var f = Sqrt(8 * n.Z + 8);
    return n.XY / f + 0.5f;
  }

  public static float3 UnpackNormal(float2 enc)
  {
    var fenc = enc * 4 - 2;
    var f = Dot(fenc,fenc);
    var g = Sqrt(1 - f/4);
    return float3(fenc * g, 1 - f/2);
  }

  public static float UnpackFloat16(float2 rg_depth)
  {
    return rg_depth.X + rg_depth.Y * 0.00392156862f;
  }

  public static float2 PackFloat16(float depth)
  {
      var enc = float2(1.0f, 255.0f) * depth;
      enc = float2(Frac(enc.X), Frac(enc.Y));
      enc -= float2(enc.Y * 0.00392156862f, 0);
      return enc;
    }

    public static float4 Frac(float4 f)
    {
      return float4(Frac(f.X),Frac(f.Y),Frac(f.Z),Frac(f.
         W));
    }

    public static float Frac(float f)
    {
      return f - Floor(f);
    }
  }
}
```

A.1 Generated GLSL shader program for G-buffer pass

```
uniform mat3 ViewNormal_4_50_4;
uniform float LinearDepth_4_24_8, LinearDepth_4_24_9;
uniform mat4 WorldViewProjection_4_41_12, World_19_9_13,
    WorldView_4_40_14;

attribute vec3 model_VertexPosition_13_2_0;
attribute vec4 model_VertexNormal_13_3_2;

varying vec3 ViewNormal_4_50_15;
varying float LinearDepth_4_24_16;

void main()
{
    vec4 ClipPosition_4_77_1 =
        WorldViewProjection_4_41_12 * vec4(
        model_VertexPosition_13_2_0, 1.0);
    ViewNormal_4_50_15 = normalize(ViewNormal_4_50_4 *
        normalize(mat3(World_19_9_13[0].xyz, World_19_9_13
        [1].xyz, World_19_9_13[2].xyz) *
        model_VertexNormal_13_3_2.xyz));
    LinearDepth_4_24_16 = (-(WorldView_4_40_14 * vec4(
        model_VertexPosition_13_2_0, 1.0)).xyz.z -
        LinearDepth_4_24_8) / LinearDepth_4_24_9;
    gl_Position = ClipPosition_4_77_1;
}
```

Listing A.1: Vertex shader

```
varying vec3 ViewNormal_4_50_15;
varying float LinearDepth_4_24_16;

vec2 PackNormal_0(vec3 n){
    float f = sqrt((8.0 * n.z) + 8.0);
    return (n.xy / f) + 0.5;
}

float Frac_2(float f){
    return f - floor(f);
}

vec2 PackFloat16_1(float depth){
    vec2 enc = vec2(1.0, 255.0) * depth;
    enc = vec2(Frac_2(enc.x), Frac_2(enc.y));
    enc = enc - vec2(enc.y * 0.00392156839, 0.0);
    return enc;
}
```

```
void main()
{
    vec2 normal_22_1_6 = PackNormal_0(ViewNormal_4_50_15)
        ;
    vec2 depth_22_0_11 = PackFloat16_1(
        LinearDepth_4_24_16);
    gl_FragColor = vec4(normal_22_1_6, depth_22_0_11);
}
```

Listing A.2: Fragment shader

Appendix B

Ambient obscurance source code

```
using Uno;
using Uno.Collections;
using Uno.Graphics;
using Uno.Scenes;
using Uno.Scenes.Primitives;
using Uno.Scenes.Designer;
using Uno.Content;
using Uno.Content.Models;

using static SSAO3.BufferData;
using static SSAO3.Sampling;
using static Math;
using static Vector;

namespace SSAO3
{
  public class AO : Node
  {
    /// Input to pass

    public IProvideTexture GBuffer { get; set; }
    public float4x4 FarCorners { get; set; }
    public float4x4 NearCorners { get; set; }

    /// tweaking parameters

    [Range(0,10)]
    public float Multiplier { get; set; }

    [Range(0,100)]
    public float Exponent { get; set; }
```

```
[Range(0,0.1f)]
public float Bias { get; set; }

[Range(0.0f,1.0f)]
public float Eps { get; set; }

[Range(0,1000)]
public float Radius { get; set; }

[Range(0,2f)]
public float MaxScreenRadius { get; set; }

/// drawable defintion

apply Quad; // makes us a fullscreen quad and
    provides us TexCoord, among other things

float2 ScreenCoord : float2(TexCoord.X, 1-TexCoord.Y)
    ;
float2 ScreenSize : float2(Uno.Application.Viewport.
    Size.X, Uno.Application.Viewport.Size.Y);

float4 Data : sample(GBuffer.Texture, ScreenCoord);
float3 ViewPosition : ReadViewPos(Data, ScreenCoord,
    NearCorners, FarCorners);
float3 ViewNormal : ReadViewNormal(Data);

float ScreenRadius : Min(MaxScreenRadius, Radius *
    0.01f / -ViewPosition.Z);

float AmbientFactor :
{
  const int s = 9;
  float oneOverSampleCount = (1.0f / (float)s);
  float z = ViewPosition.Z;
  float ao = 0.0f;
  for (int i = 0; i<s; i++)
  {
    float angularOffset = ScreenCoord.X * ScreenSize.
        X * 2.3734372915f // experimental values for
        less banding
              + ScreenCoord.Y * ScreenSize.Y *
                 3.58540627421f;

    float2 sampleCoord = Spiral(ScreenCoord,
        ScreenRadius, i, oneOverSampleCount,
        angularOffset);
    float4 data = sample(GBuffer.Texture, sampleCoord
        );
        float3 viewSample = ReadViewPos(data, sampleCoord
            , NearCorners, FarCorners);
        float3 v_i = (viewSample - ViewPosition);
        float3 n = ViewNormal;
        ao += Max(0, Dot((v_i), n) + ViewPosition.Z *
            Bias)
            / ((Dot(v_i, v_i) + Eps));
      }
      ao *= 2.0f * Multiplier / (float)s;
      return Pow(Max(0, 1.0f - ao), Exponent);
    };

    PixelColor : float4(float3(AmbientFactor), 1);

    DepthTestEnabled : false;
    WriteDepth : false;

    public override void Draw(DrawContext dc)
    {
      if (GBuffer == null) return;
      GBuffer.Draw(dc);
      draw;
    }
  }

  public static class Sampling
  {
    public static float2 Spiral(float2 xy, float radius,
        int sampleNo, float oneOverSampleCount, float
        rotationOffset)
    {
      float alpha = oneOverSampleCount  * ((float)
          sampleNo + 0.5f);
      float theta = 2.0f * Math.PIf * alpha +
          rotationOffset;
      var dir = float2(Cos(theta), Sin(theta));
      return xy + dir * radius * alpha;
    }
  }
}
```

B.1 Generated GLSL fragment shader for AO pass

```
uniform mat4 ViewPosition_3_15_6, ViewPosition_3_15_7;
uniform float ScreenRadius_3_17_9, ScreenRadius_3_17_10,
    AmbientFactor_3_18_22, AmbientFactor_3_18_24,
    AmbientFactor_3_18_26, AmbientFactor_3_18_28;
uniform vec2 AmbientFactor_3_18_16;

uniform sampler2D Data_3_14_2;

varying vec2 ScreenCoord_3_12_31;

float UnpackFloat16_1(vec2 rg_depth){
    return rg_depth.x + (rg_depth.y * 0.00392156839);
}

vec3 ToViewSpace_2(vec3 imagePosition, mat4 near, mat4
    far){
    vec3 farTL = far[0].xyz;
    vec3 farTR = far[1].xyz;
    vec3 farBL = far[2].xyz;
    vec3 farBR = far[3].xyz;
    vec3 nearTL = near[0].xyz;
    vec3 nearTR = near[1].xyz;
    vec3 nearBL = near[2].xyz;
    vec3 nearBR = near[3].xyz;
    float farZ = farTL.z;
    float nearZ = nearTL.z;
    float ratio = nearZ / farZ;
    float linearDepth = imagePosition.z;
    float linearDist = ((1.0 - ratio) * linearDepth) +
        ratio;
    vec2 farDiag = farTR.xy - farBL.xy;
    vec3 farPos = farBL + vec3(farDiag * imagePosition.xy
        , 0.0);
    return farPos * linearDist;
}

vec3 ReadViewPos_0(vec4 packedData, vec2 texCoord, mat4
    near, mat4 far){
    float linearDepth = UnpackFloat16_1(packedData.zw);
    return ToViewSpace_2(vec3(texCoord, linearDepth),
        near, far);
}

vec3 UnpackNormal_4(vec2 enc){
    vec2 fenc = (enc * 4.0) - 2.0;
    float f = dot(fenc, fenc);
    float g = sqrt(1.0 - (f / 4.0));
    return vec3(fenc * g, 1.0 - (f / 2.0));
}

vec3 ReadViewNormal_3(vec4 packedData){
    return UnpackNormal_4(packedData.xy);
}

vec2 Spiral_6(vec2 xy, float radius, int sampleNo, float
    oneOverSampleCount, float rotationOffset){
    float alpha = oneOverSampleCount * (float(sampleNo) +
        0.5);
    float theta = (6.28318548 * alpha) + rotationOffset;
    vec2 dir = vec2(cos(theta), sin(theta));
    return xy + ((dir * radius) * alpha);
}

float Draw_AmbientFactor_90316d99_3_18_4_5(vec3
    ViewPosition_3_18_13, vec2 ScreenCoord_3_18_14, vec2
    AmbientFactor_3_18_15, float ScreenRadius_3_18_17,
    mat4 AmbientFactor_3_18_18, mat4 AmbientFactor_3_18_19
    , vec3 ViewNormal_3_18_20, float AmbientFactor_3_18_21
    , float AmbientFactor_3_18_23, float
    AmbientFactor_3_18_25, float AmbientFactor_3_18_27){
    int s = 9;
    float oneOverSampleCount = 0.111111112;
    float z = ViewPosition_3_18_13.z;
    float ao = 0.0;

    for (int i = 0; i < 9; i++)
    {
        float angularOffset = ((ScreenCoord_3_18_14.x *
            AmbientFactor_3_18_15.x) * 2.3734374) + ((
            ScreenCoord_3_18_14.y * AmbientFactor_3_18_15.
            y) * 3.5854063);
        vec2 sampleCoord = Spiral_6(ScreenCoord_3_18_14,
            ScreenRadius_3_18_17, i, oneOverSampleCount,
            angularOffset);
        vec4 data = texture2D(Data_3_14_2, sampleCoord);
        vec3 viewSample = ReadViewPos_0(data, sampleCoord
            , AmbientFactor_3_18_18, AmbientFactor_3_18_19
            );
        vec3 v_i = viewSample - ViewPosition_3_18_13;
        vec3 n = ViewNormal_3_18_20;
        ao = ao + (max(0.0, dot(v_i, n) + (
            ViewPosition_3_18_13.z * AmbientFactor_3_18_21
            )) / (dot(v_i, v_i) + AmbientFactor_3_18_23));
    }

    ao = ao * ((2.0 * AmbientFactor_3_18_25) / 9.0);
    return pow(max(0.0, 1.0 - ao), AmbientFactor_3_18_27)
        ;
}

void main()
{
    vec4 Data_3_14_5 = texture2D(Data_3_14_2,
        ScreenCoord_3_12_31);
    vec3 ViewPosition_3_15_8 = ReadViewPos_0(Data_3_14_5,
         ScreenCoord_3_12_31, ViewPosition_3_15_6,
        ViewPosition_3_15_7);
    float AmbientFactor_3_18_30 =
        Draw_AmbientFactor_90316d99_3_18_4_5(
        ViewPosition_3_15_8, ScreenCoord_3_12_31,
        AmbientFactor_3_18_16, min(ScreenRadius_3_17_10,
        ScreenRadius_3_17_9 / -ViewPosition_3_15_8.z),
        ViewPosition_3_15_6, ViewPosition_3_15_7,
        ReadViewNormal_3(Data_3_14_5),
        AmbientFactor_3_18_22, AmbientFactor_3_18_24,
        AmbientFactor_3_18_26, AmbientFactor_3_18_28);
    gl_FragColor = vec4(vec3(AmbientFactor_3_18_30), 1.0)
        ;
}
```

Appendix C

Bilateral blur source code

```
using Uno;
using Uno.Collections;
using Uno.Graphics;
using Uno.Scenes;
using Uno.Content;
using Uno.Content.Models;

using static Uno.Math;
using static Uno.Vector;

namespace SSAO3
{
  public class BilateralBlur2 : Node
  {
    public IProvideTexture Input { get; set; }
    public IProvideTexture GBuffer { get; set; }

    [Uno.Scenes.Designer.Range(0,10)]
    public float NormalEffect { get; set; }

    [Uno.Scenes.Designer.Range(0,10)]
    public float DepthEffect { get; set; }

    public float2 Direction { get; set; }
    public override void Draw(DrawContext dc)
    {
      base.Draw(dc);
      if (Input != null && Input.Texture != null &&
          GBuffer != null)
      {
        var size = Application.Viewport.Size;
        var colorBuffert = Input.Texture;
        var gBuffert = GBuffer.Texture;
        var dex = DepthEffect;
        var nex = NormalEffect;
        var near = (float4x4)dc["NearCorners"];
        var far = (float4x4)dc["FarCorners"];

        draw Uno.Scenes.Primitives.Quad
        {
          TexCoord : float2(prev.X, 1-prev.Y);

          float2 lol : float2(1.0f) / float2(size.X,size.
             Y);
          float2 DirectionOverSize : Direction * lol;
          PixelColor :
          {
            var data = sample(gBuffert, TexCoord);
            var depth = Vector.Length(BufferData.
               ReadViewPos(data, TexCoord, near, far));
            var normal = BufferData.ReadViewNormal(data);

            var color = float4(0);
            var weightSum = 0.0f;

            for (int i=-3; i <= 3; i++)
            {
              var tc = TexCoord + DirectionOverSize * i;
              var sampleColor = sample(colorBuffert, tc);
              var sampleData = sample(gBuffert, tc);
              var sampleDepth = Vector.Length(BufferData.
                 ReadViewPos(sampleData, tc, near, far));
              var sampleNormal = BufferData.
                 ReadViewNormal(sampleData);
              var weight = (Pow(1.0f/(1+Math.Abs(depth -
                 sampleDepth)), dex))
                     * (Pow(Dot(normal, sampleNormal) *
                        0.5f + 0.5f, nex));
              color += sampleColor * weight;
              weightSum += weight;
            }
            color /= weightSum;
            return color;
          };
        };
      }
    }
  }
```

C.1 Generated GLSL fragment shader for bilateral blur pass

```
uniform mat4 PixelColor_6_4_8, PixelColor_6_4_10;
uniform vec2 PixelColor_6_4_12;
uniform float PixelColor_6_4_14, PixelColor_6_4_16;

uniform sampler2D PixelColor_6_4_2, PixelColor_6_4_5;

varying vec2 TexCoord_6_1_19;

float UnpackFloat16_2(vec2 rg_depth){
    return rg_depth.x + (rg_depth.y * 0.00392156839);
}

vec3 ToViewSpace_3(vec3 imagePosition, mat4 near, mat4
   far){
    vec3 farTL = far[0].xyz;
    vec3 farTR = far[1].xyz;
    vec3 farBL = far[2].xyz;
    vec3 farBR = far[3].xyz;
    vec3 nearTL = near[0].xyz;
    vec3 nearTR = near[1].xyz;
    vec3 nearBL = near[2].xyz;
    vec3 nearBR = near[3].xyz;
    float farZ = farTL.z;
    float nearZ = nearTL.z;
    float ratio = nearZ / farZ;
    float linearDepth = imagePosition.z;
    float linearDist = ((1.0 - ratio) * linearDepth) +
        ratio;
    vec2 farDiag = farTR.xy - farBL.xy;
    vec3 farPos = farBL + vec3(farDiag * imagePosition.xy
        , 0.0);
    return farPos * linearDist;
}

vec3 ReadViewPos_1(vec4 packedData, vec2 texCoord, mat4
   near, mat4 far){
    float linearDepth = UnpackFloat16_2(packedData.zw);
    return ToViewSpace_3(vec3(texCoord, linearDepth),
        near, far);
}

vec3 UnpackNormal_5(vec2 enc){
    vec2 fenc = (enc * 4.0) - 2.0;
    float f = dot(fenc, fenc);
    float g = sqrt(1.0 - (f / 4.0));
    return vec3(fenc * g, 1.0 - (f / 2.0));
```

```
}

vec3 ReadViewNormal_4(vec4 packedData){
    return UnpackNormal_5(packedData.xy);
}

vec4 Draw_PixelColor_86984bb1_6_4_4_0(vec2 TexCoord_6_4_6
    , mat4 PixelColor_6_4_7, mat4 PixelColor_6_4_9, vec2
    PixelColor_6_4_11, float PixelColor_6_4_13, float
    PixelColor_6_4_15){
    vec4 data = texture2D(PixelColor_6_4_2,
        TexCoord_6_4_6);
    float depth = length(ReadViewPos_1(data,
        TexCoord_6_4_6, PixelColor_6_4_7, PixelColor_6_4_9
        ));
    vec3 normal = ReadViewNormal_4(data);
    vec4 color = vec4(0.0);
    float weightSum = 0.0;

    for (int i = -3; i <= 3; i++)
    {
        vec2 tc = TexCoord_6_4_6 + (PixelColor_6_4_11 *
            float(i));
        vec4 sampleColor = texture2D(PixelColor_6_4_5, tc
            );
        vec4 sampleData = texture2D(PixelColor_6_4_2, tc)
            ;
        float sampleDepth = length(ReadViewPos_1(
            sampleData, tc, PixelColor_6_4_7,
            PixelColor_6_4_9));
        vec3 sampleNormal = ReadViewNormal_4(sampleData);
        float weight = pow(1.0 / (1.0 + abs(depth -
            sampleDepth)), PixelColor_6_4_13) * pow((dot(
            normal, sampleNormal) * 0.5) + 0.5,
            PixelColor_6_4_15);
        color = color + (sampleColor * weight);
        weightSum = weightSum + weight;
    }

    color = color / weightSum;
    return color;
}

void main()
{
    vec4 PixelColor_6_4_18 =
        Draw_PixelColor_86984bb1_6_4_4_0(TexCoord_6_1_19,
        PixelColor_6_4_8, PixelColor_6_4_10,
        PixelColor_6_4_12, PixelColor_6_4_14,
        PixelColor_6_4_16);
    gl_FragColor = PixelColor_6_4_18;
}
```

Appendix D

Normal reconstruction source code

(not used in final prototype)

```
using Uno;
using Uno.Collections;
using Uno.Graphics;
using Uno.Audio;
using Uno.Scenes;
using Uno.Content;
using Uno.Content.Models;

namespace SSAO3
{
  public class NormalFromDepth : RenderToTexture
  {
    public IProvideTexture PackedLinearDepth { get; set;
       }

    public override void OnDraw(DrawContext dc)
    {
      base.OnDraw(dc);

      if (PackedLinearDepth == null) return;
      var depthBuffer = PackedLinearDepth.Texture;
      if (depthBuffer == null) return;

      var nearCorners = (float4x4)dc["NearCorners"];
      var farCorners = (float4x4)dc["FarCorners"];
      draw Uno.Scenes.Primitives.Quad
      {
        sampler2D depthSampler : pixel_sampler(
            depthBuffer);
        TexCoord : float2(prev.X, 1 - prev.Y);

        float2 OneOverSize : float2(1.0f / dc.Rect.Size.X
            , 1.0f / dc.Rect.Size.Y);

        float3 c : BufferData.ReadViewPos(depthSampler,
            TexCoord, nearCorners, farCorners);
        float3 dx : Vector.Normalize(BufferData.
            ReadViewPos(depthSampler, TexCoord + float2(
            OneOverSize.X, 0), nearCorners, farCorners) -
            c);
        float3 dy : Vector.Normalize(BufferData.
            ReadViewPos(depthSampler, TexCoord + float2(0,
             OneOverSize.Y), nearCorners, farCorners) - c)
            ;
        float3 normal : Vector.Cross(dx,dy);

        PixelColor : float4((normal * 0.5f + 0.5f),1);
      };
    }
  }
}
```

D.1 Generated GLSL fragment shader for normal reconstruction

```
uniform mat4 dx_7_5_8, dx_7_5_9;

uniform sampler2D depthSampler_7_1_2;

varying vec2 dx_7_5_7, dy_7_6_13, TexCoord_7_2_17;

float UnpackFloat_2(vec4 rgba_depth){
    return rgba_depth.x + (rgba_depth.y * 0.00392156839);
}

float ReadLinearDepth_1(sampler2D depthBuffer, vec2
   texCoord){
    vec4 PackedDepth = texture2D(depthBuffer, texCoord);
    return UnpackFloat_2(PackedDepth);
}

vec3 ToViewSpace_3(vec3 imagePosition, mat4 near, mat4
   far){
    vec3 farTL = far[0].xyz;
    vec3 farTR = far[1].xyz;
    vec3 farBL = far[2].xyz;
    vec3 farBR = far[3].xyz;
    vec3 nearTL = near[0].xyz;
    vec3 nearTR = near[1].xyz;
    vec3 nearBL = near[2].xyz;
    vec3 nearBR = near[3].xyz;
    float farZ = farTL.z;
    float nearZ = nearTL.z;
    float ratio = nearZ / farZ;
    float linearDepth = imagePosition.z;
    float linearDist = ((1.0 - ratio) * linearDepth) +
        ratio;
    vec2 farDiag = farTR.xy - farBL.xy;
    vec3 farPos = farBL + vec3(farDiag * imagePosition.xy
        , 0.0);
    return farPos * linearDist;
}

vec3 ReadViewPos_0(sampler2D depthBuffer, vec2 texCoord,
   mat4 near, mat4 far){
    float linearDepth = ReadLinearDepth_1(depthBuffer,
        texCoord);
    return ToViewSpace_3(vec3(texCoord, linearDepth),
        near, far);
}
```

```
void main()
{
    vec3 c_7_4_10 = ReadViewPos_0(depthSampler_7_1_2,
        TexCoord_7_2_17, dx_7_5_8, dx_7_5_9);
    vec4 PixelColor_7_8_16 = vec4((cross(normalize(
        ReadViewPos_0(depthSampler_7_1_2, dx_7_5_7,
        dx_7_5_8, dx_7_5_9) - c_7_4_10), normalize(
        ReadViewPos_0(depthSampler_7_1_2, dy_7_6_13,
        dx_7_5_8, dx_7_5_9) - c_7_4_10)) * 0.5) + 0.5,
        1.0);
    gl_FragColor = PixelColor_7_8_16;
}
```