Università degli Studi di Salerno

Facoltà di Scienze Matematiche Fisiche e Naturali

Tesi di Laurea Specialistica in

Informatica

Solid Angle based Ambient Obscurance in Image Space

Relatori
Prof. Vittorio Scarano
Prof. Ugo Erra

Candidato
Dario Scarpa

Anno Accademico 2012-2013

Acknowledgements

First of all, I'd like to thank my family for supporting me during my years at the university - 'cause studying on the couch when someone else pays the bills is definitely easier.

Then, my friends and classmates, with a special mention for the ones I have spent many hours at the university, studying or working together on a number of interesting projects: Angelo Cafaro, Giannicola Scarpa, Raffaele Spinelli, Vincenzo Fariello, Carla Del Regno, Federica Sarro, Laura Maffei, Alessandro Barone, Carlo Maresca, Marco Gatto and the many others that don't come to my mind right now (sorry!).

Special thanks also to the friends and team-mates I rarely meet but whom I often have fun and collaborate online on a number of projects - Cristiano Caliendo, Davide Canavero - guys, if it wasn't for you, I would have graduated much sooner. Maybe.

Finally, thanks to my advisors and to everyone working in ISISLab, where high expertise and knowledge meet friendliness and easygoingness (yeah, that's an actual word, I checked), to go where no man has gone before - but never getting bored or lost along the way.

This thesis has been developed in ISISLab

Contents

Chapter 1

Introduction

We start by succinctly defining the main concepts and terminology needed to understand the scope of our work.
Then, we shortly discuss our objectives and results.
Finally, we present a short summary of the following chapters to show how the thesis is organized.

1.1 Basic Concepts

1.1.1 Rendering

Rendering is the process of converting a description of a three-dimensional scene into an image. A rendering system involves a number of algorithms and data structures that process and handle the scene description in extremely different ways according to the desired result.

A rough distinction between rendering systems is the one between off-line rendering and real-time rendering. **Off-line rendering** involves long processing times and aims to produce high quality images, often with a high degree of photorealism derived from an accurate, physically grounded simulation of light propagation.

On the opposite side, we have **real-time rendering**, suitable to produce interactive applications in which the user is able to freely navigate and modify the virtual environment. Three-dimensional videogames are the most popular (but not only) application heavily based on real-time rendering.

A real-time rendering system must produce an image in just a few **milliseconds**: the desired frequency is usually 60 frames per second (fps), that account for about 16 ms of processing time to generate each frame. In many circumstances 30 fps can be acceptable, and you can perceive some interactivity at 12-15 fps, but anyway we are in a totally different domain from off-line rendering, where rendering times are measured in **minutes** or even **hours**.

Of course, with such limited processing time available, real-time rendering abandons accurate physics simulation in favour of crude approximations that, anyway, try go give a visually pleasing and convincing result.

In this thesis, we work in the domain of real-time rendering.

1.1.2 Light, color, shading

The essence of the rendering process is setting the colors of pixels composing the output image. **Light** is electromagnetic energy that travels through space. **Color** is defined by the interaction between light and matter, so we need models that approximate the behaviour of light in the real world (or that define a non-realistic behaviour for artistic purposes). Such models take the name of **illumination models**. In computer graphics, **shading** is the process of altering color according to an illumination model or to achieve a number of special effects.

1.1.3 Local and Global Illumination

In very simple terms, light is emitted by a source (natural or artificial) and interacts with objects in the scene: when it hits a surface, part is absorbed, part is scattered and propagates in new directions. Finally, a tiny portion of the light travelling through the scene reaches a sensor that absorbs it (like the human eye).

The recursivity of the process, with light bouncing indefinitely around the scene, is what makes calculating illumination computationally heavy.

So, a reasonable simplification historically made in real-time rendering to achieve acceptable performance is taking account only of the **direct light** when shading surfaces. Direct light on a surface is the light that comes directly from a light source, and not from a bounce on surrounding geometry. This is called **local illumination**, from the fact that to be computed it only needs local information: the surface data at the visible point. Differently, the term **global illumination** refers to illumination models that also take account of indirect light and so need information about other objects that the one being shaded.

As can be seen in figure 1.1, global illumination algorithms give much better results, but are currently suitable only to off-line rendering.

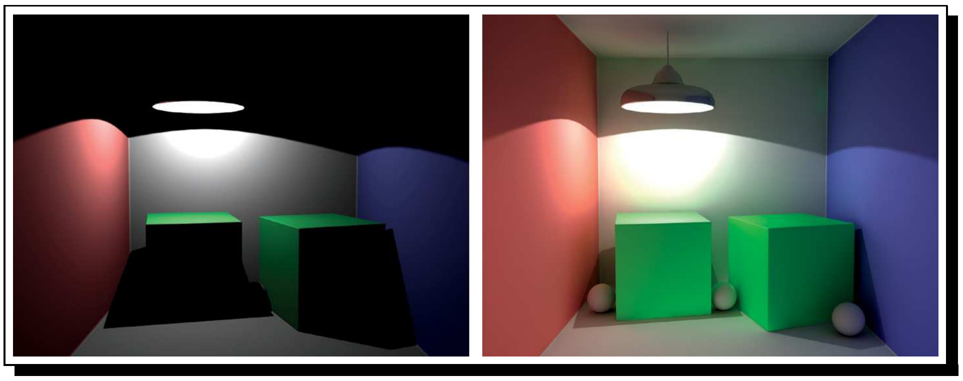


Figure 1.1: direct illumination VS global illumination

1.1.4 Ambient Light and Ambient Obscurance

With only direct illumination, all the surfaces not directly lit are totally black, as shown by the top image in figure 1.2.

A simple way to approximate the amount of indirect light scattered in the environment is adding to the calculated direct lighting of any surface an additional amount of constant radiance, called **ambient light** (figure 1.2, in the middle).

Being constant, ambient light doesn't vary depending on the characteristics of light sources nor the position and orientation of surfaces, and so, unfortunately, usually gives a "flat" look to rendered scenes, especially if not coupled with good direct lighting.

A substantial improvement to the ambient light approximation comes from **ambient occlusion**.

The essence of ambient occlusion is calculating a *visibility factor* for a point, by checking for intersections with the surrounding geometry that, intuitively, will stop some amount of indirect light. So, the visibility factor can be used to modulate the ambient light, resulting in occluded zones getting darker and giving perceptually pleasing results in the rendered images.

Figure 1.3 shows an example of the process, with the obscurance factor shown in the top image, the modulated ambient light in the middle one, and the final compositing, done by adding the direct light, in the bottom.

By depending not only on the point information, but also from surrounding geometry data, ambient occlusion is classified as related to global illumination, even if, of course, is just a crude approximation.

Ambient obscurance is a refinement of ambient occlusion that, in the calculation of the visibility factor, takes into account the distance of the occluding geometry, causing the occlusion to "fade out" with distance.

1.1.5 Indirect Lighting

While indirect lighting is not the primary focus of this thesis, there's a generic observation worth mentioning: usually, the visibility factor used in the ambient occlusion approximation is calculated by sampling the surrounding geometry to detect occlusors. If direct illumination has already been calculated for such occluding geometry, we can easily sample that too and treat the point belonging to the occlusor as radiance emitter to roughly approximate the first bounce of indirect light.

An interesting result shown in [TL04] is that the first bounce of indirect light is the most perceptually important: calculating the second bounce too gives a minimal improvement, and calculating the third bounce is close to useless.

We will exploit these observations to give some visually pleasing color bleeding effects related to the first bounce of indirect light.

1.1.6 Image-Space methods and Deferred Rendering

The rendering pipeline used in real-time rendering involves a number of pretty standardized stages, each one using a number of algorithms to accomplish different tasks. The last stage of the pipeline is **rasterization**, that is the crucial part where elements processed in the previous stages are finally mapped to coloured pixels in a buffer that, in a straightforward rendering setup, gets displayed on the screen. **Image-space methods** work on information gathered from the rasterization pass, doing additional processing on the produced image before finally displaying it. This brings some advantages (such as independence from scene complexity), but also has some drawbacks, namely the limited amount of information available.

Even if for some post-processing effects the information produced by default suffices, a wider range of possibilities is available when we use techniques related to **deferred rendering** to bring more information in custom *renderbuffers* and add rendering passes that process them. We still work with images (this, after all, originates the name "image-space methods"), but we encode in pixel colors arbitrary information about the rendered point.

More details in the following chapters will clarify the inner working and rationale of these techniques.

1.2 Objectives and results

We derive a new approximation of ambient obscurance focused on improving quality of the state of the art techniques used in real-time rendering.

We attempt to stay close to the original definition of ambient obscurance and bring into image-space, building up on the deferred rendering approach, information suitable to an accurate estimate of visibility that takes account of the position and orientation of near occluding geometry, allowing us to handle some situations that would result in artefacts or erroneous calculations with other techniques. At the best of our knowledge, the approximation of covered solid angle (used in our occlusion estimate) considering the area of surfaces, and the hemisphere partitioning that gives directional information about coverage, both done in image-space, are original contributions that could be further explored in future works.

To evaluate the quality of results, we compare screen-shots from our implementation with images rendered off-line in **Blender**, a popular open-source 3D graphics software that features a configurable ray-traced calculation of ambient occlusion and is easily instrumented through Python scripting.

For the comparison, we use the **Structural Similarity (SSIM) Index** [WBSS04], a metric that attempts to measure similarity between images in a way somewhat consistent with the human eye perception.

Our implementation achieves lower performances respect to some currently popular and widely adopted Screen-Space AO approximations, but still obtains real-time frame rates on the current hardware generation. Also, it offers many parameters that can be tuned to trade quality for efficiency.

1.2.1 Implementaton details

For learning purposes and to achieve total freedom in the implementation of the technique, we have built a rendering sandobox (named YARS, acronym of the unimaginative “Yet Another Rendering Sandbox”) from the ground up.

We based YARS on modern, shader based OpenGL 4.3, trying to follow best-practices and using features that should help achieve good performances on modern hardware.

Nonetheless, YARS is currently a basic and incomplete environment that lacks many features generically implemented in “real” 3D engines.

It would be interesting to implement our approach in a well-established rendering engine, such as G3D [EHF+], and see if better performances can be achieved.

1.3 Contents overview

- **chapter 2 - Theoretical Foundations**
  We explain the concept of solid angle and define ambient obscurance. Than, we briefly show the Blinn-Phong illumination model, explaining how ambient occlusion fits into it. The minimal information needed to understand the following topics will be provided.

- **chapter 3 - Related Works**
  We do a quick survey of the most influential and inspiring techniques related to image-space ambient occlusion/obscurance, and give references to additional sources to explore the field of interactive global illumination.

- **chapter 4 - Solid Angle based SSAO**
  We illustrate how our technique works from a theoretical perspective, defining its components and explaining how they fit together.

- **chapter 5 - SaSSAO implementation**
  First, we briefly describe the OpenGL rendering sandbox we built, YARS. Then, we give detailed information about the implementation of our ambient obscurance solution, focusing on the adopted rendering pipeline and on some particularly interesting details. Additionally, we quickly describe a modification of our technique that adds color-bleeding effects (resulting from the first bounce of indirect light).

- **chapter 6 - SaSSAO testing**
  We describe our simple testing architecture based on Blender and the SSIM index, and show some results. Some information about the relationship between parameter values and performances will be discussed.

- **chapter 7 - Conclusions**
  We recap and evaluate the achieved results, while giving some ideas about possible improvements or alternative roads we couldn't explore due to time constraints

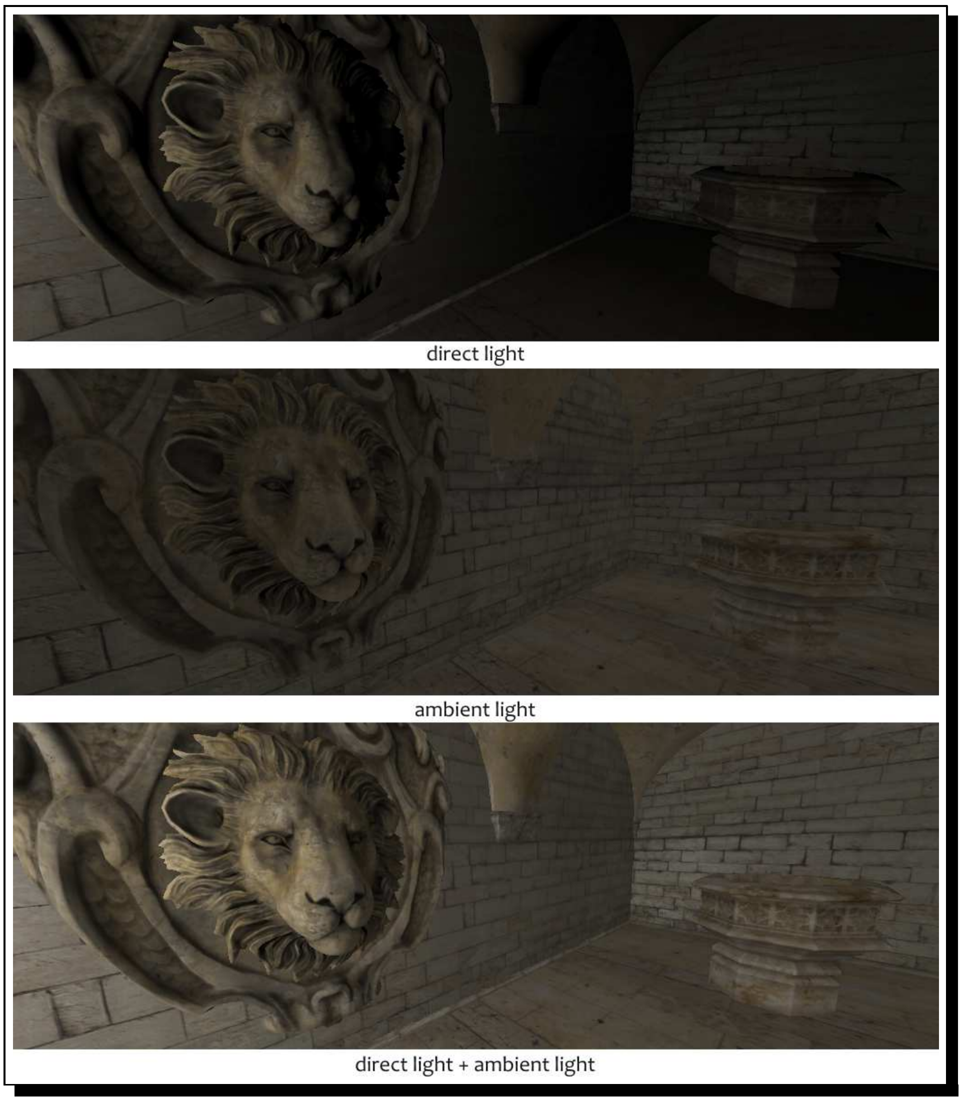


Figure 1.2: compositing direct and ambient light

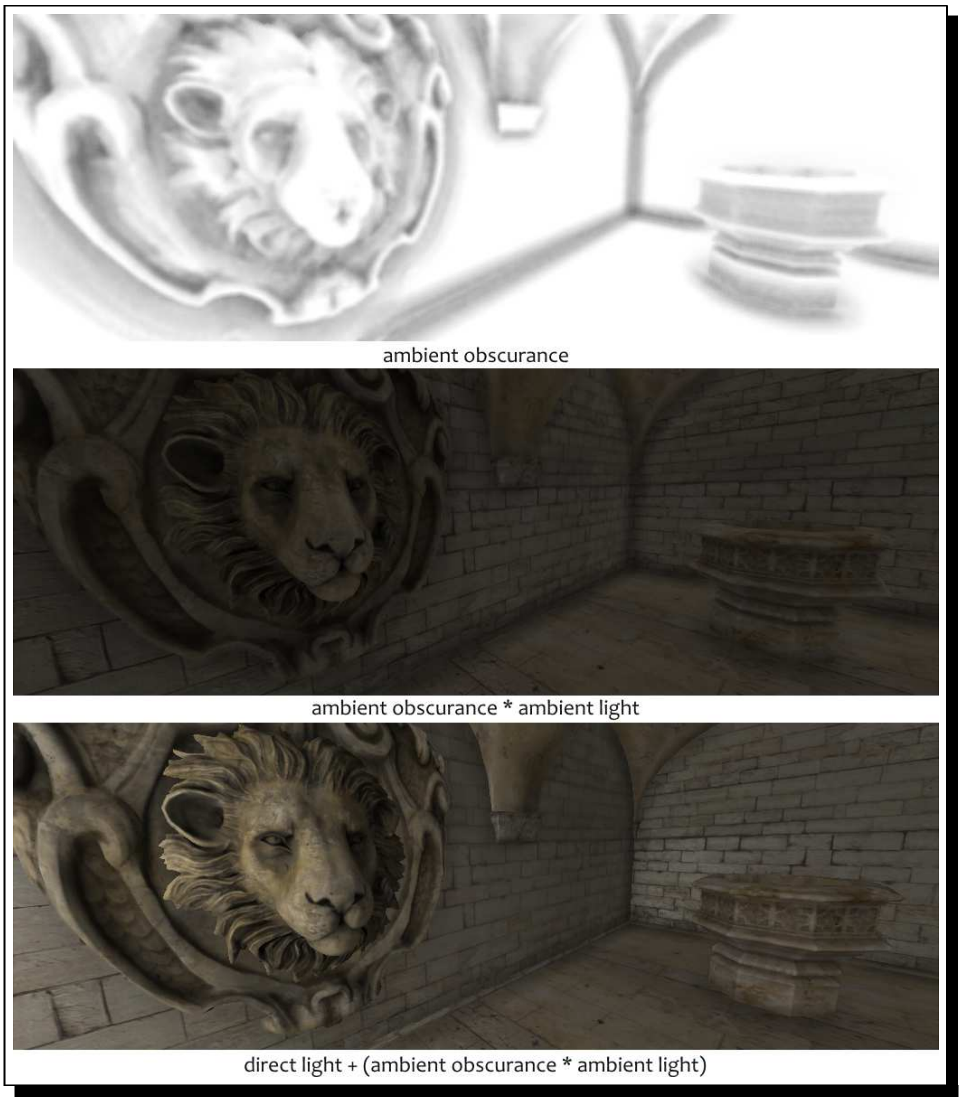

ambient obscurance
ambient obscurance * ambient light
direct light + (ambient obscurance * ambient light)

Figure 1.3: compositing direct light and ambient light modulated by ambient obscurance

Chapter 2

Theoretical foundations

We start by describing the concept of solid angle, often recurring in rendering formulas and particularly relevant to the evaluation of visibility, as we'll explain defining ambient occlusion and obscurance right afterwards. Then, we show the illumination model we adopted, Blinn-Phong, and how ambient obscurance fits into it.

2.1 Solid Angle

Solid angles are the 3D counterpart of planar angles, extending the 2D unit circle to a 3D unit sphere.

As you can measure the planar angle subtended by some object on a plane with respect to some position, you can measure the solid angle subtended by an arbitrary oriented surface at a point P.
On the plane, projecting an object onto the unit circle covers some length s. This is the angle subtended by the object and is measured in **radians**.

An arc of a circle with the same length as the radius of that circle corresponds to an angle of 1 radian. This implies, being the length of the entire circumference $2\pi r$, that a full circle corresponds to an angle of 2π radians.

The solid angle s subtended by an object c in three dimensions is computed by projecting c onto the unit sphere and measuring the area of its projection. Solid angles are measured in **steradians**.

Any area on a sphere that amounts to the square of the sphere radius

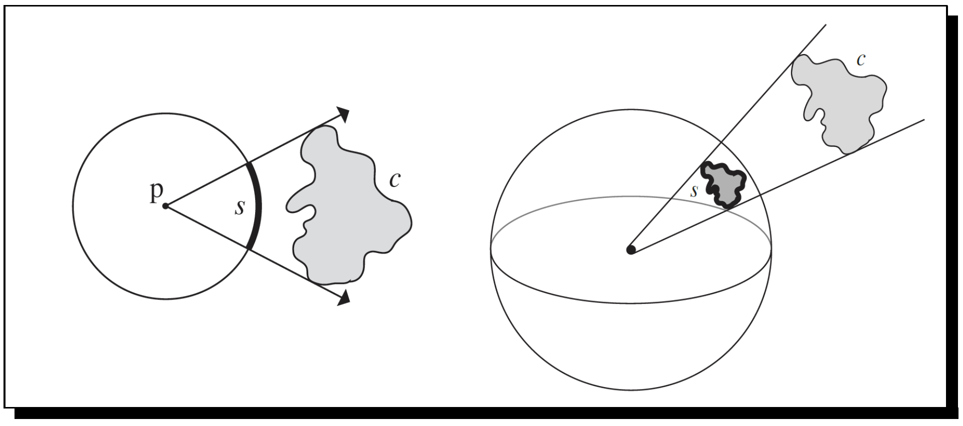


Figure 2.1: planar angle and solid angle (images taken from [PH10])

subtends precisely one steradian. A sphere surface area measures $4\pi r^2$, so the entire sphere subtends a solid angle of 4π steradians.

The solid angle for an arbitrary oriented surface subtended at a point can be calculated through a surface integral, but we will restrict the problem to a much simpler approximation suitable to our needs.

For our purposes, two things to remember about solid angles are:

- a hemisphere subtends 2π steradians (the full sphere, as we wrote, subtends 4π steradians)
- the set of points on the unit sphere surface centered at point P can be used to describe the direction vectors anchored at P (often these vectors are referred using the ω symbol)

2.2 Ambient Occlusion

As we briefly introduced in 1.1.4, shadowing of ambient light is referred to as ambient occlusion.

It has been shown in [LB00] that ambient occlusion offers a better perception of the 3D shape of the displayed objects, and a concrete proof of its effectiveness is the popularity the effect has gained in videogame engines.

The mathematical definition of ambient occlusion is related to the concept of solid angle that we just described.

In fact, the occlusion $A_{\bar{p}}$ at a point $\bar{p}$ on a surface with normal $\hat{n}$ can be computed by integrating the visibility function over the hemisphere Ω with respect to projected solid angle:

$$A_{\bar{p}} = \frac{1}{\pi} \int_{\Omega} V_{\bar{p},\hat{\omega}} (\hat{n} \cdot \hat{\omega}) \; \mathrm{d}\,\omega$$

where

- $V_{\bar{p},\hat{\omega}}$ is the visibility function at $\bar{p}$:
  - 0 if $\bar{p}$ is occluded in the direction $\hat{\omega}$
  - 1 otherwise
- $\mathrm{d}\,\omega$ is the infinitesimal solid angle step of the integration variable $\hat{\omega}$.

A simple way to approximate this integral in practice, in off-line rendering, is based on ray-tracing.

Rays are shot in uniform pattern across the hemisphere over point $\bar{p}$, and an occlusion value can be calculated as the number of rays that hits geometry divided by the total number of rays shot.

Rays can be restricted to a certain length, avoiding distant geometry to be taken into account while calculating the occlusion value. This is fundamental in closed environments, that would otherwise result in total occlusion at every point, and subsequently the complete removal of ambient light.

We will use ray-traced ambient occlusion, of which you can see an example in fig. 2.2, to evaluate the quality of our method in chapter 6.

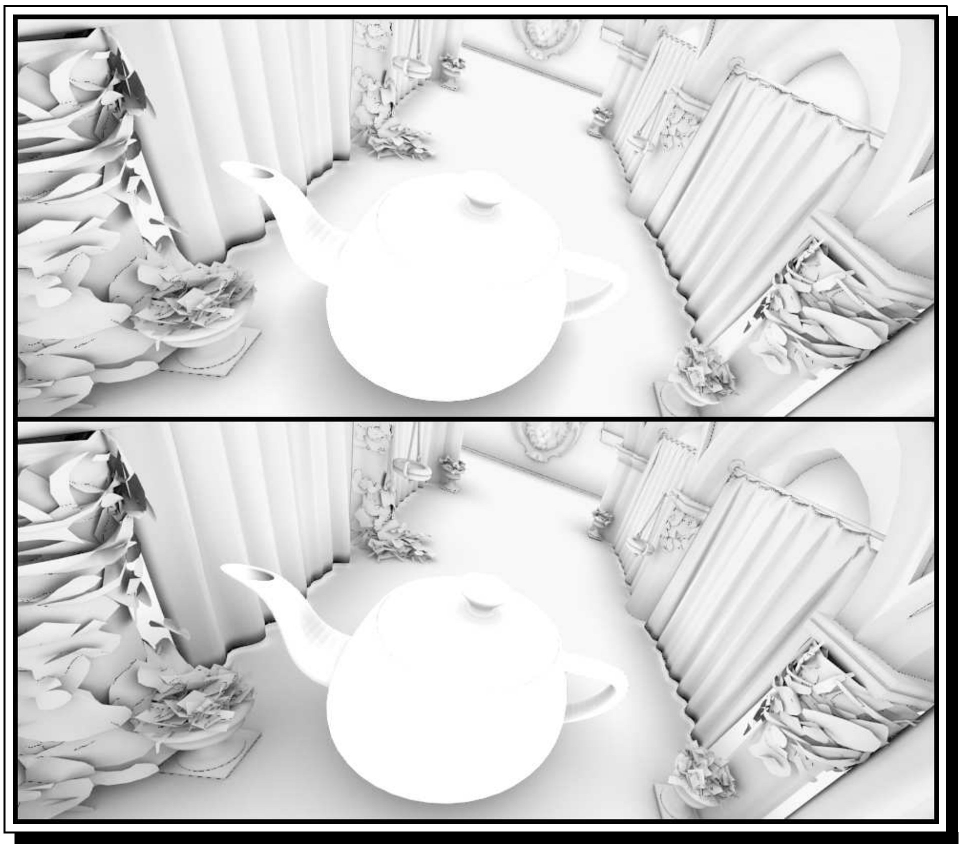


Figure 2.2: an example of ambient occlusion calculated through ray-tracing in Blender. The images are rendered using rays of two different lengths, 0.5 (first image) and 1.0 (second image). Some areas are dark in the second image but not in the first, meaning that the occlusion to those areas comes from geometry more distant that 0.5

2.3 Ambient Obscurance

Ambient Obscurance is an extension of ambient occlusion defined in [ZIK98]. A **falloff function** that reduces the influence of occlusion with distance is introduced in the formula:

$$AO_{\bar{p}} = \frac{1}{\pi} \int_{\Omega} \rho(D_{\bar{p},\hat{\omega}})(\hat{n} \cdot \hat{\omega}) \, \mathrm{d}\,\omega$$

If you compare this and the formula of ambient occlusion, you'll see that the difference is in the fact that in place of the binary visibility function $V_{\bar{p},\hat{\omega}}$, we have $\rho(D_{\bar{p},\hat{\omega}})$, where

- $D_{\bar{p},\hat{\omega}}$ is
  - the distance between p and the first intersection point along $\hat{\omega}$, if any
  - $+\infty$ if there are no intersections along $\hat{\omega}$
- ρ is the decreasing falloff function, with
  - $\rho(0) = 1$
  - $\rho(x) = 0$ if $x > r$, where r is the maximum distance at which any intersecting geometry is considered as bringing occlusion

This formulation also solves elegantly the problem of enclosed geometry that we discussed at the end of the previous paragraph.

In figure 2.3 you can see ambient obscurance calculated with the Alchemy screen-space ambient obscurance algorithm, one of the best AO approximation algorithms suitable for real-time rendering currently known.

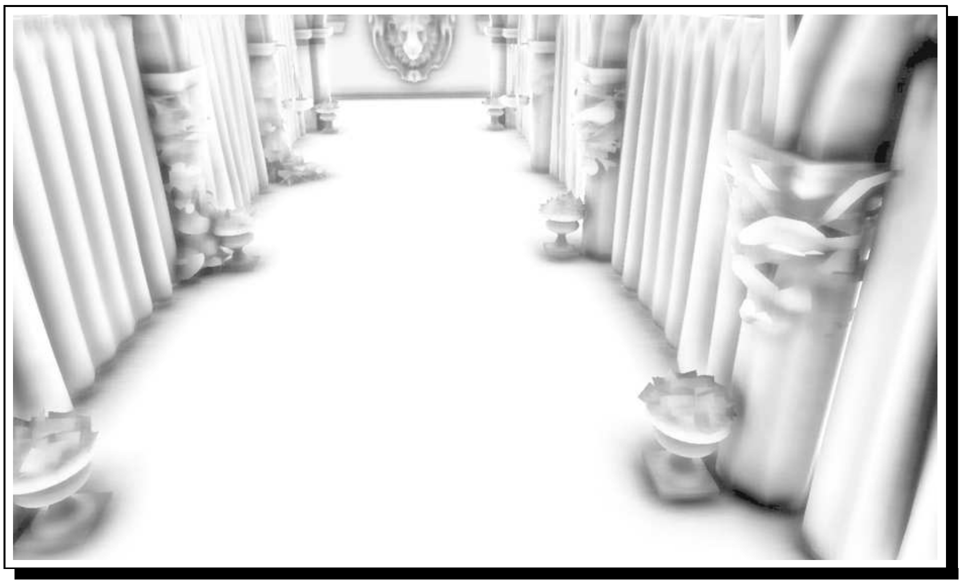


Figure 2.3: an example of ambient obscurance calculated with the Alchemy algorithm, from [MOBH11]

2.4 Blinn-Phong Reflection model

Blinn-Phong was the model implemented by default in the older, fixed-function pipeline GPUs. It was proposed in [Bli77] as a variation of the Phong model defined in [Pho75]

It is probably the most popular illumination model in real-time rendering, and many asset formats contain material descriptions consisting mainly of the coefficients used by this model. So, even if there's an on-going shifting to more physically-based models and materials, thanks to the freedom given by the programmable pipeline of current GPUs, we decide to use the Blinn-Phong model because it's easy to implement and understand. Additionally, inserting ambient occlusion in the model is straightforward.

The basic idea of this lighting model is adding a set of independently computed lighting components to get a total lighting effect for a particular spot on a material surface, as seen from a certain direction (the camera direction, also referred to as the "eye direction").

The three additive components are called **diffuse**, **specular** and **ambient**. On a higher level, the idea is that the illumination of a point P is

$$I = I_a + I_d + I_s$$

where I_d (diffuse component) and I_s (specular component) approximate direct lighting due to light sources, and I_a (ambient component) is a constant term that roughly approximates indirect light scattered into the environment, as introduced in 1.1.4. Figure 2.4 shows a visual example of the process.

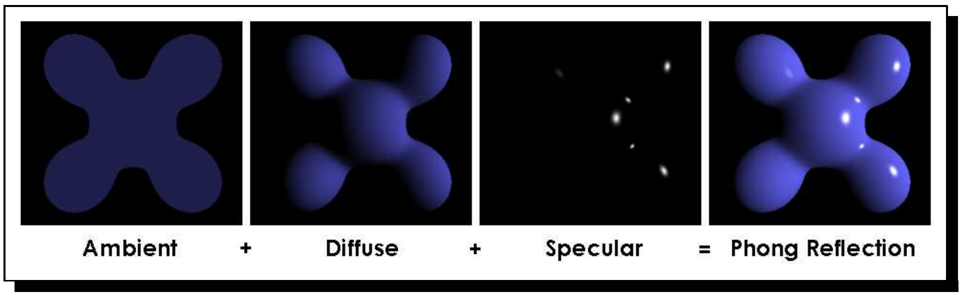


Figure 2.4: an example of how the three components are added, giving the final result

2.4.1 Surface Materials and Light Sources

As we told in 1.1.2, color is defined by the interaction of light and matter, so it not surprising that shading geometry requires information about the surface material and about the light sources.

A **material description** suitable for the model is composed of these attributes:

- emission: light produced by the material (if any)
- ambient: part of ambient light that is scattered
- diffuse: part of diffuse light that is scattered
- specular: part of specular light that is scattered
- shininess: level of sharpness of the specular reflection

All these attributes, except for the *shininess* that is a single floating point value, are RGB vectors.

A light source, other than other attributes we'll discuss later, also has an RGB vector that indicates the intensity for each color, ranging from darkness at $(0.0, 0.0, 0.0)$ to a maximum brightness white at $(1.0, 1.0, 1.0)$.

It is possible to have separate RGB attributes (ambient, specular and diffuse) even for light sources, and in that case, when the interaction between light and material is calculated, each of the ambient/specular/diffuse color of the material is multiplied by the respective ambient/specular/diffuse color of the light source. We stick to a simpler (and more common/desiderable) approach, that instead defines light sources as having a single *color* attribute, used in both the diffuse and specular calculations, and no specific ambient attribute.

If you are wondering what is the sense of an ambient attribute for a light source, as we have explained that ambient light is a global settings not related to light sources, it's worth knowing that in some cases it could make sense to calculate the global scene ambient light by adding to a single base value some contributions tied to the individual light sources. Specifically, it makes sense if you allow to turn off some light sources in the scene: when you turn off a light source, you remove its ambient light contribution, that represents the indirect light scattered into the environment due to that source.

The RGB values in material descriptions correspond to the proportion which the respective color is reflected: for example, if a material M has $M.diffuse = (1.0, 0.0, 0.0)$, it reflects all the red diffuse light it receives, and absorbs all the green and blue light. So, if you point a white light L (meaning that $L.color = (1.0, 1.0, 1.0)$) on a surface made of material M, it will look as red to the camera. If you point to it a totally blue light B ($B.color = (0.0, 0.0, 1.0)$) all the blue light will be absorbed and the surface will appear black. So, it is natural to model the interaction with between the color of the light and the color of the surface through **multiplication**.

The different material attributes are useful to approximate the behaviour of different kinds of surfaces. The main characteristic that affects how light is reflected is the level of roughness/smoothness that can be observed at the microscopic level:

- a **rough** surface (e.g. chalk) scatters light in all directions, because at the microscopic level, due to the differently oriented small facets that compose it, it will cause light to bounce around, as shown in the left image of figure 2.5.

- a **smooth** surface (e.g. plastic, metal) tends to reflect light in a specific direction, as in the right image of 2.5. A perfect mirror is the ideal smooth surface.

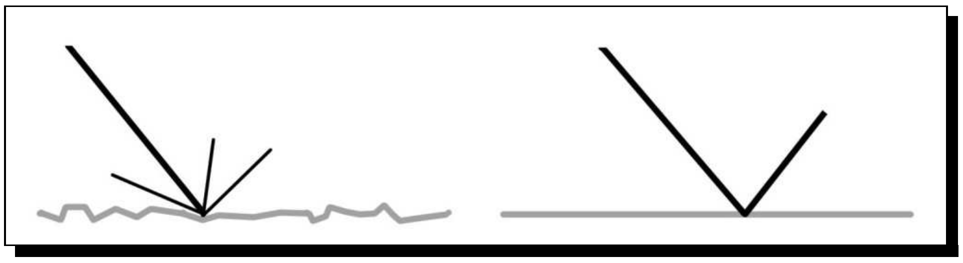


Figure 2.5: light bouncing on a rough and on a smooth surface

Giving different values to the $diffuse$, $specular$ and $shininess$ attributes of a material description can approximate the behaviour of a wide range of real-world materials. Figure 2.6 shows effectively examples of how a sphere of different materials (totally diffuse, totally specular, and with partial glossiness) would reflect light.

Let's now see how direct lighting (diffuse and specular) is actually computed, and then we'll discuss ambient lighting/obscurance.

2.4.2 Direct Lighting

In the actual calculation of the diffuse/specular components, a number of vectors, shown in figure 2.7, are used together with the material/light source info:

- N, the surface normal
- L, normalized vector in direction of light source
- R, normalized vector in direction of specular light reflection
- V, normalized vector in direction of viewer
- H, the "halfway" vector, a normalized vector in direction of $\frac{V+L}{2}$

The main point worth noting is that the diffuse component does only depend on the position and orientation of the light and of the surface (L and N vectors), while the specular component is different depending on the

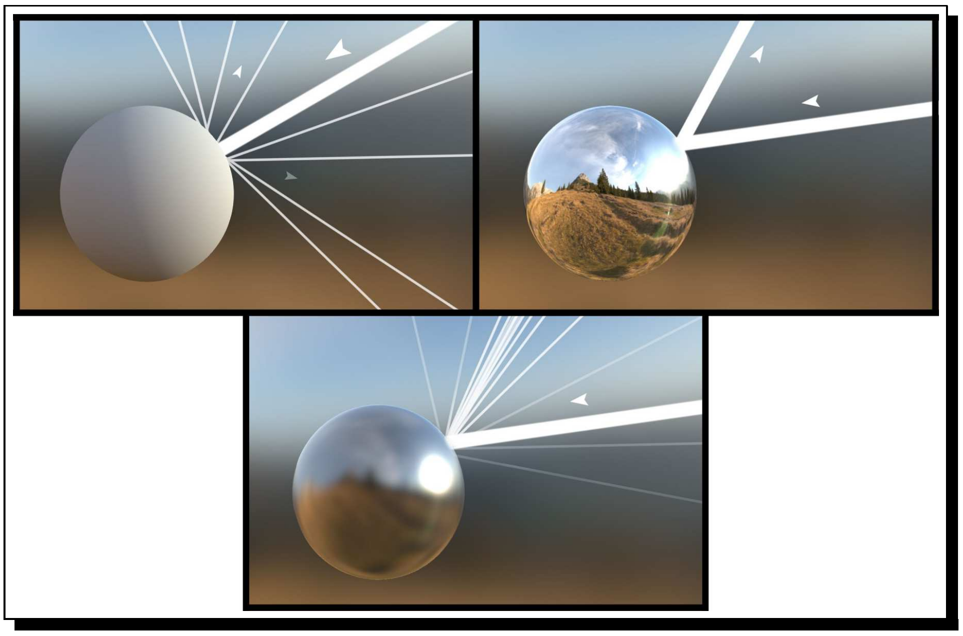


Figure 2.6: images showing a fully diffuse reflection on a rough surface, a fully specular reflection on a glossy surface, and something in between. Images taken from http://www.pernroth.nu/lightandmaterials/

position of the viewer. This happens because the specular highlights refer to how much the surface material acts like a mirror, and so vary according to how much the viewer direction V is close to the direction of specular light reflection R.

Diffuse Component

The $diffuse$ material attribute, together with the L and N vectors, is used to calculate the **diffuse reflection** component on the basis of Lambert's cosine law, defined in [Lam60].

The law says that the apparent brightness of an ideal diffusely reflecting surface is proportional to the cosine of the angle between the surface normal and the direction of the incident light.

According to our notation about involved vectors, and assuming m and

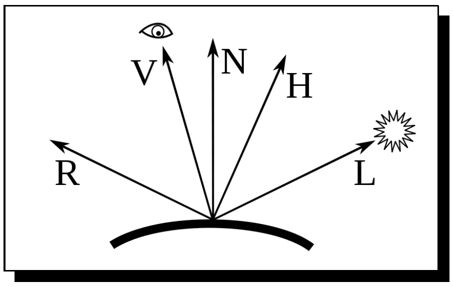


Figure 2.7: the vectors involved in the Blinn-Phong calculation

l are respectively a material description and a light source, the diffuse light component is calculated as:

$$I_d = m.diffuse \otimes l.color * max(N \cdot L, 0)$$

where the $\otimes$ symbol denotes piecewise vector multiply.

The $max(N \cdot L, 0)$ term is often called "clamped cosine factor" or "lambertian factor". The dot product between N and L essentially computes the cosine of the angle between the two vectors[1], and in case of negative values (that is, if the light source is in a position that can't irradiate the surface) the max function clamps the value to 0.

Specular Component

The *specular* and *shininess* material attributes, together with the L, N and V vectors, are used to calculate the **specular reflection** component.

This computation is slightly more complex than the one of diffuse light, because, as previously stated, depends on the direction of the viewer (V vector).

If we consider the microscopic structure of the surface, we can say that the specular reflectance comes from "tiny mirrors" oriented halfway between L and V. The **halfway vector** H, of unitary length, that represents this orientation, is consequently calculated like this:

$$H = \frac{L + V}{||L + V||}$$

[1]the dot product is defined as $||N||||L||cos\theta$, and N and L have unitary length so the dot product essentially computes the cosine of the angle θ formed by N and L

The angle between the N and H vectors, with a logic similar to the one of the Lambertian term of the diffuse component equation, is used to modulate the specular reflection, together with the *shininess* material attribute:

$$max(N \cdot H, 0)^{m.shininess}$$

Introducing the RGB values relative to material and light color, we have the final equation:

$$I_s = m.specular \otimes l.color * max(N \cdot H, 0)^{m.shininess}$$

Light attenuation

If, as in most cases, the light source has some attenuation characteristics, $()I_d + I_s)$ must also be multiplied by an $attenuationFactor$.

Different attenuation functions can be used, but a popular one is

$$attenuationFactor = \frac{1}{constantAtt + linearAtt * d + quadraticAtt * d^2}$$

where *constantAtt*, *linearAtt* and *quadraticAtt* are light source attributes and d is the distance between source and surface.

2.4.3 Ambient Lighting

I_a is generically derived by a global setting for the scene, a fixed amount that gets multiplied by the material ambient color and added to the direct light in the total illumination calculation.

Often, the *ambient* attribute is not specified in the material description, and the $diffuse$ attribute is used as fallback, representing in some way the "base color" of the surface. Keeping this in mind, we can write the ambient component equation as:

$$I_a = m.ambient \otimes l.color$$

2.4.4 Adding ambient obscurance to the model

After giving the details about how each of the three components I_a, I_d, I_s is calculated, let's get back to the big picture:

$$I = I_a + I_d + I_s$$

As we explained, ambient obscurance is related to ambient lighting, so in its more correct formulation the *AO* factor (representing the occlusion of the point we are shading) should modulate the ambient light (the I_a factor). Consequently, we have

$$I = AO * I_a + I_d + I_s$$

Sometimes (for example if the ambient occlusion is added as post-processing on the fully lightened scene, meaning that the separated components aren't available), the *AO* factor is used to scale the total lighting, without distinction between direct and ambient light. We stick to the first, more appropriate definition.

Chapter 3

Related works

Interactive global illumination is one of the “hot topics” in computer graphics research, and an impressive number of related works have been published. We suggest reading the 2012 survey [RDGK12] to get a generic overview on the field.

Here, we restrict the scope to techniques most closely related to our own: ambient occlusion/obscurance approximations suitable to real-time rendering that work in image space (also referred as screen space).

Even considering only this category of algorithms, a variety of approaches exist. Some sources try to correlate, compare and evaluate such techniques, and we suggest the interest reader to check [Aal] and [Gra13], two recent thesis that both agree on the **Alchemy** algorithm being the state of the art.

We'll briefly cover the more influential works in the following sections.

3.1 2005 - Dynamic Ambient Occlusion and Indirect Lighting

In his seminal work [Bun05], Micheal Bunnell of NVIDIA corporation describes a technique that approximates polygon meshes as a set of surface elements (discs) that can emit, transmit or reflect light and that can shadow each other. He defines an approximation of ambient occlusion on the basis of the calculated coverage between discs, and an approximation of indirect lighting by estimating the disc-to-disc radiance transfer.

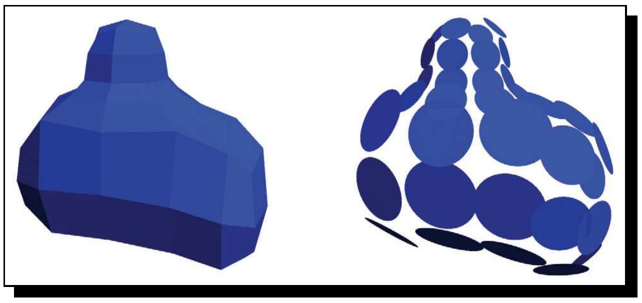


Figure 3.1: a polygonal mesh and its representation as disc-shaped elements, from the original paper

Major drawbacks of this algorithm are the dependence on scene complexity and the need to preprocess geometry, which has to be well tessellated to give good results. This also implies that the technique is not suitable to deformable objects.

It must be noted that this is not an image-space technique, but a geometry approximation one. Anyway, we mention it because it has some way inspired a number of subsequent works, including our own.

3.2 2007 - Hardware Accelerated Ambient Occlusion Techniques on GPUs

In [SA07] Shanmugam and Arikan approximate ambient occlusion through spherical proxies.

Their interesting idea is to approximately reconstruct the surface represented by a pixel using a sphere in world-space that roughly projects to that pixel on the screen.

By using deferred rendering, a ND-buffer storing normals and depths is created. From such information, each pixel can be mapped to a sample of some surface in world-space, that can be considered as an occlusor to other pixels.

The algorithm also uses a separate calculation (non screen-space) for low-frequency occlusion due to distant occluders, and then combines the results.

Even if presenting original and interesting ideas, this technique is maybe over-complicated and didn't have much luck, being subsequently surpassed in both quality and speed by simpler techniques.

3.3 2007 - CryEngine2 Screen Space Ambient Occlusion

In [Mit07] the denomination "screen space ambient occlusion" appears for the first time. Crytek implements in its CryEngine this technique, that works by sampling the surrounding of a pixel and, on the basis of the z-buffer, does depth comparisons.

Sample positions are distributed in a sphere around the pixel, and some randomness is introduced by reflecting position vectors on a random plane passing through the sphere origin.

The occlusion factor depends only on the depth difference between sampled points and current point. This, combined with the simple distribution of samples (around a sphere and not an hemisphere) causes some over-darkening: even flat, not occluded areas get some samples considered as occlusors (as you can see from the grey flat areas in figure 3.2), resulting in *self-occlusion.*

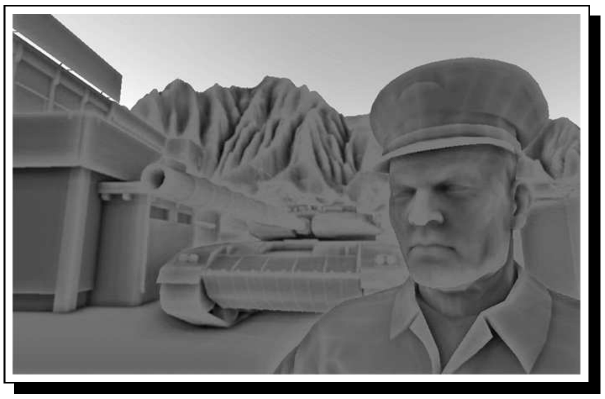


Figure 3.2: ambient occlusion as calculated by CryEngine 2

3.4 2008 - StarCraft 2 Ambient Occlusion

Some improvements over the CryEngine approach are shown in [FM08].

Samples are offset in 3D space from the current point being computed, and then projected back to screen space to sample the depth of the sample location. Normals (this algorithm is based on deferred rendering too) are used to flip the vectors that fall in the hemisphere below the current point, avoiding the *self-occlusion* exhibited by the Crytek algorithm.

An occlusion function maps the relationship between the depth delta ($sample.depth - projectedSample.depth$) and how much occlusion occurs.

A number of details are taken care of (sample randomization, filtering, down-sampled calculation) to improve performances and obtain a production ready solution.

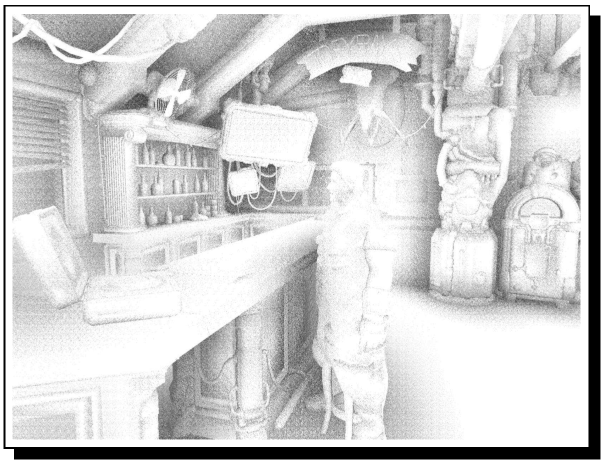


Figure 3.3: StarCraft II SSAO, before filtering

3.5 2008 - Image-space horizon-based ambient occlusion

The method shown in [BSD08] interprets the depth buffer as a height field and works by performing a form of *ray marching* in screen space. It considers the tallest occluder along each azimuthal direction, to determine the visible horizon on the hemisphere around the current point. This assumes a continuous depth buffer, so in cases like the one of figure 3.4 the occlusion is not calculated correctly.

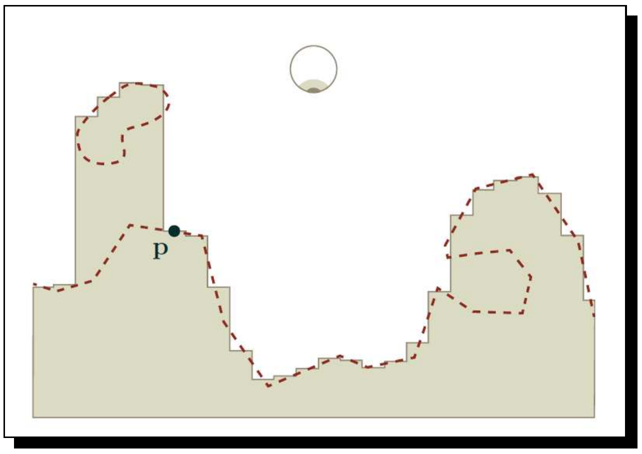


Figure 3.4: An example of discontinous depth buffer - image taken from [Aal]. In a case like this, HBAO wouldn't consider the unoccluded portion of emisphere on the left of p, below the floating occluder

Some refinements of the method were later published, but anyway it looks like the method is quite expensive compared to the quality it manages to obtain.

3.6 2011 - Alchemy Screen-Space Ambient Obscurance

Tha Alchemy SSAO algorithm, presented in [MOBH11], has been developed with the goal of artistic expressiveness rather than physically grounded realism.

The strength of Alchemy is in the way it derives its estimator: the chosen falloff function manages to cancel some expensive operations while staying meaningful. The obtained, highly efficient estimator is then applied to points sampled in the hemisphere, as in some previous methods.

Alchemy features a number of artist-tweakable parameters and generally gives very good quality results with high performances. Some improvements and modifications of the algorithm are discussed in [MML12].

Chapter 4

SaSSAO: Solid Angle based SSAO

In this chapter we are going to describe our technique from a theoretical perspective.

Unfortunately, many aspects presented are closely related to the implementation and to generic rendering theory. So, we advise to

- ignore some possibly obscure details specific to our technique, and revisit this chapter after reading the following one about the implementation
- refer to books such as [AMHH11] for the generic aspects we might have overlooked (or just mentioned briefly)

Anyway, it's worth briefly anticipating a few crucial things that can help understanding:

- we refer to "camera space" and "screen space" coordinates of a point to respectively identify
    - its the 3D coordinates in the scene, relative to the camera position
    - its 2D coordinates when it gets projected on the screen, in the rendered frame
- in deferred rendering, the G-Buffer (geometry buffer) is a set of buffers output by a "geometry pass" that saves information about the scene to be later processed

- the depth buffer (or z-buffer) stores a depth value for each pixel of the screen space: if a pair of screen-space coordinates x,y let you pin-point a pixel on the screen, the z-buffer value for that pixel tells you how far from the camera is the object that pixel belongs to
    - from screen space coordinates and depth, it is possible to reconstruct the camera space coordinates
- shaders are programs that are executed by the GPU and, depending on their type, run in a certain stage of the rendering pipeline, accessing different information and being able to do different operations. We use *vertex*, *fragment* and *geometry* shaders.

Moving on to the actual description, we can split it into two main parts:

- a scheme for hemisphere partitioning into "solid-angle buckets", and an algorithm that identifies to which bucket a certain direction vector belongs
- the obscurance estimation process, that essentially works by calculating the coverage of the hemisphere through sampling the G-Buffer and performing a number of computations involving the buckets

4.1 Hemisphere Partitioning

We bring to image-space a scheme to discretise the sphere into solid angles proposed in [KSMY07].

Let's consider a unit sphere centered on the origin of a Euclidean space.

The origin divides each of the axes into two halves: positive and negative semiaxis. Let's refer to the slice of sphere delimited by the three positive semiaxes, that is, the positive octant of the sphere (see fig. 4.1).

Let's consider the $x + y + z = 1$ plane and the equilateral triangle that lies on it with vertices $v_0 = (1, 0, 0)$, $v_1 = (0, 1, 0)$, $v_0 = (0, 0, 1)$.

We split each of the $\overrightarrow{v_0v_1}$ $\overrightarrow{v_0v_2}$ $\overrightarrow{v_1v_2}$ edges into n equal units. For edge $\overrightarrow{v_0v_1}$, we connect subdivisions between the other two edges with line segments parallel to $\overrightarrow{v_0v_1}$. Then, we repeat the same process for the remaining two edges, obtaining a tessellation of the original triangle into n^2 triangles. The process is illustrated by fig. 4.2

If we project onto the surface of the sphere the vertices of this tessellation, by normalization, we get a partition of the octant into spherical triangles.

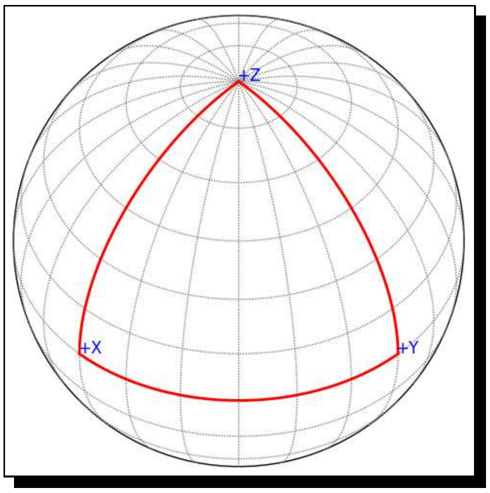


Figure 4.1: the surface above the positive octant of a sphere

Each of these spherical triangles represents a solid angle ω, associated with the direction Θ_ω passing through the triangle centroid.

Individual triangles are assigned unique identifiers, as shown in fig. 4.3.

Given an arbitrary direction Θ, we can identify the associated triangle in constant time. This is how it's done.

- we consider the unit vector along Θ and its intersection point p_Θ with the $x + y + z = 1$ plane
- we find the intervals of $\overrightarrow{v_0 v_1}$ and $\overrightarrow{v_1 v_2}$ that p_Θ lies into, restricting the region to a parallelogram defined by the segments parallel to $\overrightarrow{v_0 v_1}$ and $\overrightarrow{v_0 v_2}$
    - ○ $x = np_\Theta, z = np_\Theta$
        - ◇ this scales x and x from $[0...1]$ to $[0..n]$
    - ○ $x_i = \lfloor x \rfloor, z_i = \lfloor z \rfloor$
        - ◇ this identifies the x and z intervals p_Θ lies into
    - ○ $\xi_\Theta = z_i(2n - z_i) + 2x_i$
        - ◇ $z_i(2n - z_i)$ calculates the identifier associated to the first triangle of each row (for $n = 4$, it computes $0, 7, 12, 15$)
        - ◇ $2x_i$ adds the "horizontal" offset, so ξ_Θ locates the lower-left half of the parallelogram containing p_Θ

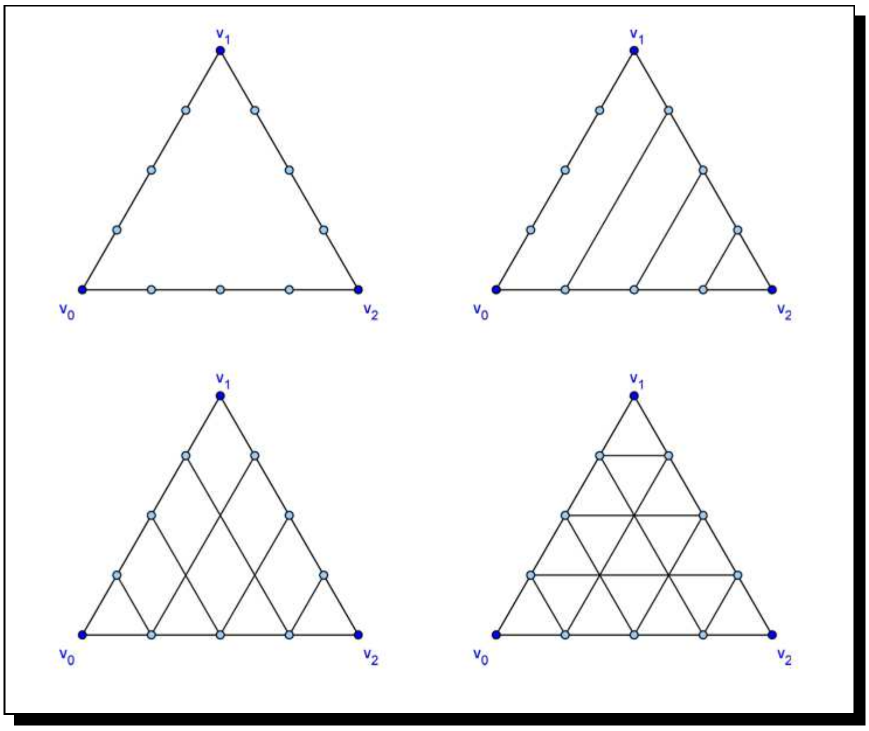


Figure 4.2: steps of triangle subdivision for $n = 4$

- we isolate the triangle testing if p_Θ lies above or below the diagonal of the located parallelogram[1]
    - $diag = (x - x_i) + (z - z_i)$
        - diag will be ≤ 1 if p_Θ lies above the diagonal, > 1 otherwise
    - $\xi'_\Theta = \xi_\Theta + \lfloor diag \rfloor$
        - if above, we are in the right half of the parallelogram, so we increment the triangle id[2] adding 1, otherwise we already have the triangle id as ξ_Θ ($\lfloor diag \rfloor$ will be 0)

This technique is more easily understood through a graphical example, like the one shown by figure 4.4.

[1]beware: on the version of [KSMY07] we consulted, the formula was erroneously reported as $diag = (x - x_i)(z - z_i)$

[2]on [KSMY07], this line is written in a code-like fashion as $if(diag > 1)\xi_\Theta + +$

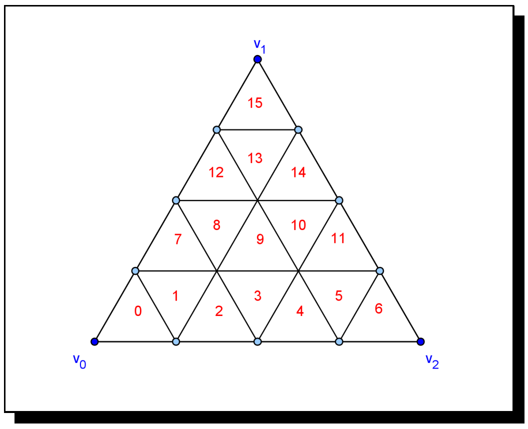


Figure 4.3: numeric identifiers for $n = 4$

4.1.1 Partitioning for the whole hemisphere

Of course, the same process can be applied for the other octants of the sphere.

For the ambient obscurance calculation, we are interested only in directions associated with the hemisphere "surrounding" the normal of the point, so we repeat the process for four octants, building a pyramid.

We've shown how for n subdivisions we get n^2 triangles, so for four octants we get $4\,n^2$ total triangles.

The triangle identifiers follow the pattern shown in fig. 4.3, but with an additional offset added, depending on the slice of hemisphere they are related: for example, the third slice will have triangle identifiers ranging between $2n$ and $3n - 1$.

The solid angle covering the full hemisphere is 2π, so the solid angle associated with every "bucket" is

$$\omega \approx \frac{2\pi}{4n^2}$$

The algorithm to find the bucket associated with any direction vector is easily adapted: the signs of the vector coordinates indicate in which of the four octants of the hemisphere we have to look.

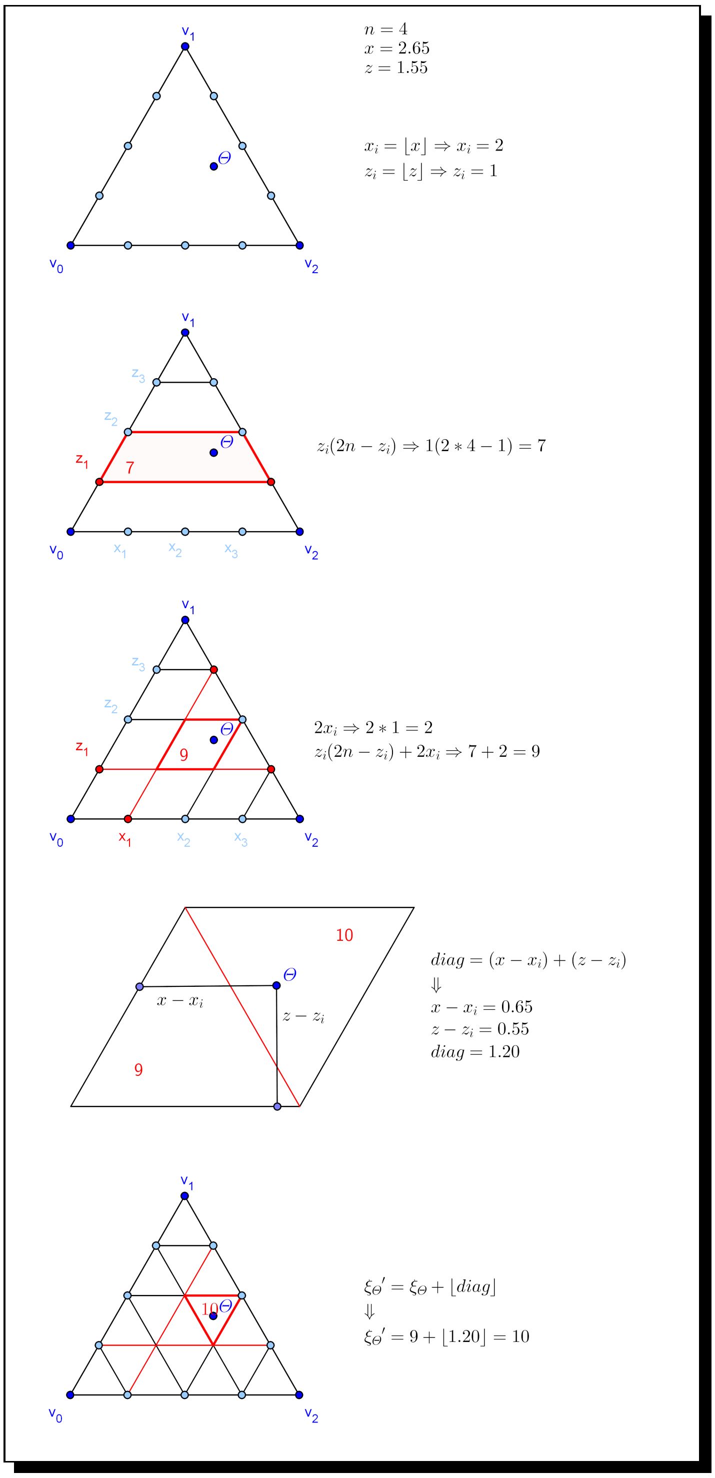

v_1
v_0
v_2
Θ
$n = 4$
$x = 2.65$
$z = 1.55$
$x_i = \lfloor x \rfloor \Rightarrow x_i = 2$
$z_i = \lfloor z \rfloor \Rightarrow z_i = 1$
z_3
z_2
z_1
7
x_1
x_2
x_3
$z_i(2n - z_i) \Rightarrow 1(2 * 4 - 1) = 7$
9
$2x_i \Rightarrow 2 * 1 = 2$
$z_i(2n - z_i) + 2x_i \Rightarrow 7 + 2 = 9$
10
$x - x_i$
$z - z_i$
$diag = (x - x_i) + (z - z_i)$
$\Downarrow$
$x - x_i = 0.65$
$z - z_i = 0.55$
$diag = 1.20$
$\xi_\Theta{}' = \xi_\Theta + \lfloor diag \rfloor$
$\Downarrow$
$\xi_\Theta{}' = 9 + \lfloor 1.20 \rfloor = 10$

Figure 4.4: steps of triangle identification by arbitrary direction

4.2 Ambient Obscurance calculation

On a higher level, our algorithm proceeds like this:

- in a first pass, the geometry shader computes an area value relative to each triangle processed, and forwards it to the fragment shader, that saves it into the G-Buffer together with normals and depths
- for each pixel, in the following pass, another fragment shader samples the G-Buffer to calculate the AO
- the AO-buffer gets filtered to lower the noise caused by the sampling and used to modulate the ambient factor in the final compositing of the rendered image

Now let's proceed with a detailed view of each phase.

4.2.1 Area Calculation

While the vertex shader operates on a per-vertex level, the geometry shader can access whole primitives (in our implementation, we only use triangles). So, for each triangle, the camera space position of its vertices is used as basis to compute an area that will be used in the AO calculation later.

Let p_0, p_1, p_2 be the camera space positions of the three vertices of each triangle, from which we calculate the length of its sides, a, b, c:

$$\begin{aligned} a &= length(p1 - p0) \\ b &= length(p2 - p1) \\ c &= length(p0 - p2) \end{aligned}$$

Then, by the **Heron formula**, in which s is the semiperimeter of the triangle, we can calculate the triangle area A_t:

$$\begin{aligned} s &= \frac{a+b+c}{2} \\ A_t &= \sqrt{s(s-a)(s-b)(s-c)} \end{aligned}$$

We can calculate additional quantities related to the triangle, such as the area of the circumscribed circle A_{cct} or the area of the inscribed circle A_{ict}.

$$radius_{cct} = \frac{abc}{4A_t} \Rightarrow A_{cct} = \pi radius_{cct}^2$$
$$radius_{ict} = \frac{2A_t}{(a+b+c)} \Rightarrow A_{ict} = \pi radius_{ict}^2$$

Which area calculation use, and an additional *areaMultiplier* parameter, will be configurable settings.

4.2.2 AO calculation

The second pass works in image space, accessing the G-Buffer created in the first pass. We'll talk about implementation details in the following chapter, for now suffices to say that we can retrieve some geometry related information from the G-Buffer according to a sampling pattern, and use it in our process of calculating the ambient obscurance.

G-Buffer information

Let's assume that we get values from the G-Buffer for a pixel P, at screen coordinates $P.x$ and $P.y$, by these functions:

- $P.depth()$ - returns the depth of the geometry at P, from the z-buffer, normalized to [0..1]
- $P.normal()$ - returns the normal of the geometry surface at P
- $P.csPos()$ - calculates the camera space position from $P.x$, $P.y$ and $P.depth()$
- $P.area()$ - returns the area related to the triangle pixel P belongs, calculated as described in 4.2.1

Sampling pattern

The selection of the screen space positions where to take samples for calculating the AO for point P is very important.

Basically, two categories of approaches are possible, both involving a radius around P, *samplingRadius*, often scaled by $P.depth()$, that limits the distance samples can be taken:

- "flat" sampling that locates points around P, in a circle of radius $samplingRadius$, considering the bidimensional screen space coordinates x, y of the pixel
- 3D sampling that consider the hemisphere around $P.normal()$ having radius $samplingRadius$ and take points in the screen space area delimited by such hemisphere

Perspective projection makes things a little complicated: when going back to camera space, pixels selected with both approaches can result in useless samples, related to points out of the area considered in the AO calculation.

Anyway, randomization is a crucial aspect of every sampling technique adopted. In fact, if we stick to a static, regular pattern, some banding artefacts will appear in the calculated AO. By applying some form of randomization, we avoid such artefacts, at the price of some high-frequency noise that can be handled with filtering, as will be shown in 4.2.3.

A simple way to introduce randomization is using some form of rotation dependent on a random value derived from the pixel coordinates:

- in flat sampling, a kernel of points randomly placeD around the center P is rotated around P
- in 3D sampling, a kernel of vectors reaching random points into the hemisphere is rotated using the normal of point P as rotation axis

We implemented both approaches (even with some variants) and left to the user the option to select and evaluate them.

Solid Angle estimator

In [Bun05], scene geometry is approximated with oriented discs considered as occluders to calculate per-vertex occlusion on the GPU, as briefly discussed in 3.1. We somewhat take inspiration from the technique but bring it to the image-space domain.
Let's consider two pixels, P and S, that belong to two different triangles.

We want to estimate what solid angle, at pixel P, is covered by the surface S belongs to.

Let d be the vector from camera space position of P to camera space position of S:

$$d = S.csPos() - P.csPos()$$

Let d' be its normalized version

$$d' = \frac{d}{||d||}$$

We can name θ_P the angle formed by d' and $P.normal()$ and θ_S the angle formed by d' and $S.normal()$.

So, the dot product

$$d' \cdot P.normal()$$

is basically the cosine of θ_P, and similarly

$$-d' \cdot S.normal()$$

is the cosine of θ_S.

Figure 4.5 visualizes the involved entities.

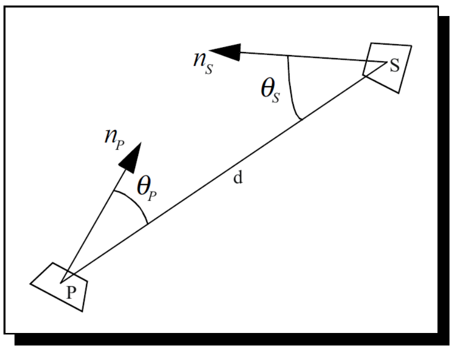


Figure 4.5: the angles and vectors related to the solid angle approximation

A possible solid angle approximation is

$$c = \frac{max(0, d' \cdot P.normal()) * S.area()}{d^2}$$

The key idea is that $max(0, d' \cdot P.normal())$ decreases the impact of occluders that only block incident light at shallow angles (which is radiometrically correct).

Conversely, multiplying by the area related to the occlusor surface modulates the contribution according to the dimensions of the surface.

If we want also consider the orientation of the occlusor, we must introduce θ_S into the equation. Ideally, if we consider a disc of area $S.area()$ and oriented according to $S.normal()$, its projected solid angle would be

$$solidAngle_S = \frac{S.area()\ cos\theta_S}{d^2}$$

This comes from the differential area being related to differential solid angle (as viewed from a point P) by

$$d\omega = \frac{dA\ cos\theta}{r^2}$$

where θ is the angle between the surface normal of dA and the vector to P, and r is the distance from P to dA, as shown by figure 4.6.

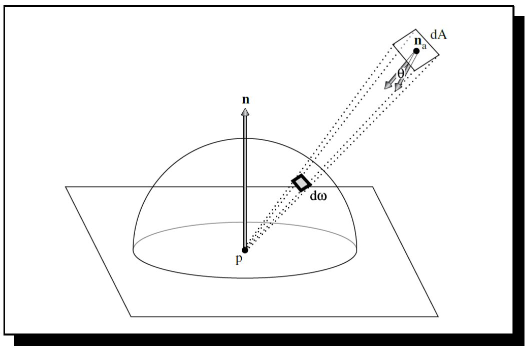


Figure 4.6: differential solid angle and and differential area - image taken from [PH10]

This formula can be understood intuitively:

- if dA is at distance 1 from P and it's aligned exactly so that it is perpendicular to $d\omega$, then $d\omega = dA$ and $cos\theta = 1$, and he equation holds

- as dA moves farther away from P, the r^2 term increases and so diving by it reduces accordingly $d\omega$
- as dA rotates so that it's not aligned with the direction of $d\omega$, the $cos\theta$ term decreases, reducing accordingly $d\omega$

Unfortunately, applying this formula with screen-space estimators results in horrible artefacts, because often the portion of an object visible to the camera is not the same that is oriented towards the surface we are calculating the occlusion for, as show by fig. 4.7.

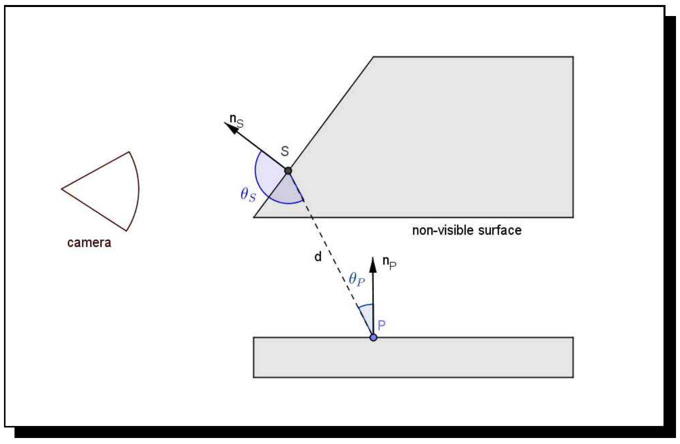


Figure 4.7: no occlusion would be calculated if applying the basic solid angle formula calculation

A possible approximate solution can be flipping the normal in such cases: what we are interested in, regarding solid angle coverage, is after all that there is some geometry occluding light, not if it is oriented towards the occluded surface or not.

In terms of calculations, instead of clamping to 0 the cosine value, we can take its absolute value: so, if the θ_S angle falls into the 90..180 degrees range, the negative cosine values relative to $S.normal()$ becomes a positive value for $-S.normal()$.

$$solidAngle'_S = S.area()\, abs(d' \cdot S.normal())$$

Modifying our approximation according to this factor, we get:

$$c' = \frac{max(0, d' \cdot P.normal()) * abs(d' \cdot S.normal()) * S.area()}{d^2}$$

Anyway, our implementation allows changing the solid angle approximation formula, even at runtime, to easily test different approaches.

Falloff function

A falloff function relative to the occluding geometry distance allows to smooth out the obscurance contribution with distance.

We adopt the falloff function proposed by the Alchemy SSAO algorithm in [MOBH11].

$$g(t) = u\, t\, max(u, t)^{-}2$$

where

- u is $samplingRadius * k$, with $k \approx 0.01$ (user modifiable)
- t is the distance between P and the occluding geometry

AO Computation

Let's call P the current pixel (that is, the pixel the fragment shader is running for).

$P.depth()$ is read and used to check if the pixel is part of the geometry or the background. If it's part of the background, there's no processing to be done. $P.normal()$ is retrieved and used, together with $P.depth()$, to calculate the screen space position of the samples to be taken.

Let k be the number of samples to take, and $S_i, i \in [0..k-1]$, the ith sample pixel position in screen space, located depending on the adopted sampling pattern, as discussed in 4.2.2.

Let $triangleDivs$ be the triangle subdivision factor, that defines how the hemisphere is discretized. From discretization of the hemisphere: from this factor (as explained in 4.1.1), we derive $hemisphereBuckets$, that is $4triangleDivs^2$ and $bucketSolidAngle$, that is $\frac{2\pi}{hemisphereBuckets}$.

triangleDivs	hemisphereBuckets	bucketSolidAngle
2	16	0,39270
3	36	0,01091
4	64	0,00017

So, for each S_i, the following steps are performed:

- $S_i.depth()$ is read. If the pixel is part of the background, the processing skips to the next sample.
- $S_i.csPos()$ is calculated, allowing to compute:
  - $sampleDir$, the vector that goes from $P.csPos()$ to $S_i.csPos()$
  - its length, $sampleDist$, that is the actual distance between $P.csPos()$ and $S_i.csPos()$
    - this distance, due to perspective projection, can be significantly greater than the screen space distance between P and S_i
- if $sampleDist$ is greater than the $maxDist$ parameter, the sampled point is too distant to be taken into account for the AO calculation, and so processing skips to the next sample
- otherwise, by the process illustrated in 4.1 (adapted to work for the four octants of the hemisphere as discussed in 4.1.1), the triangle id $sampleDir$ falls in is found. So, let t_i be the triangle id found for S_i.
- we check if the bucket t_i is already fully covered (meaning that we already know that the direction the sample belongs is already established as occluded)
  - if it is, we skip to the next sample
  - otherwise, we compute an estimate of the covered solid angle as described in 4.2.2, and we add it to the current coverage value for the bucket

After processing all the samples, we have an estimate of the visibility around the currently processing point in the form of the coverage values for all the buckets of our discretization. We know that the solid angle for the full hemisphere is 2π, so we sum the cover of all the buckets and divide it by 2π to get a global occlusion value.

Of course nothing can guarantee that by random sampling we get samples on every near-field occluder, but this is true for every SSAO technique.

The immediate advantage of our technique is that we avoid over-occlusion caused by multiple occluders covering each other but covering from the same direction.

4.2.3 AO filtering

Random sampling avoids banding issues, but introduces high frequency noise. This could be removed with a basic Gaussian blur:

$$I^{\text{filtered}}(x) = \sum_{x_i \in \Omega} I(x_i) g_s(\|x_i - x\|)$$

- I^{filtered} is the filtered image;
- I is the original input image to be filtered;
- x are the coordinates of the current pixel to be filtered;
- Ω is the "window" of pixels centered in x;
- g_s typically a Gaussian function, the spatial kernel for smoothing differences in coordinates

The problem with using Gaussian blur with AO is that it would also cause some shadow bleeding between surfaces at different depths or orientation.

We have normals and depths at our disposal, so a more intelligent filtering can be done.

A popular choice adopted by SSAO techniques is **bilateral filtering**. Bilateral filtering adds another function, the "range kernel", that weights the contribution of pixels by according to an additional criterion other than the distance between x and x_i.

$$I^{\text{filtered}}(x) = \frac{\sum_{x_i \in \Omega} I(x_i) f_r(\|I(x_i) - I(x)\|) g_s(\|x_i - x\|)}{\sum_{x_i \in \Omega} f_r(\|I(x_i) - I(x)\|) g_s(\|x_i - x\|)}$$

- f_r is the range kernel for smoothing differences in intensities

Anyway, we decided to use a filtering function defined in [Gra13], a box filter with bilateral weights based on normal and depth differences, and not Gaussian weights:

$$I^{\text{filtered}}(x) = \frac{\sum_{x_i \in \Omega} color(x_i) w(x, x_i)}{\sum_{x_i \in \Omega} w(x, x_i)}$$

where

$$w(x, x_i) = w_{normal}(x, x_i) w_{depth}(x, x_i)$$
$$w_{normal}(x, x_i) = \left(\frac{n_x \cdot n_{x_i} + 1}{2}\right)^{k_n}$$
$$w_{depth}(x, x_i) = \left(\frac{1}{1 + |d_x - d_{x_i}|}\right)^{k_d}$$

k_n and k_d are two constants that can be tuned to alter the contribution of the normal/depth discriminators in the weight calculation. Our implementation allows to change at runtime k_n, k_d and the size of Ω.

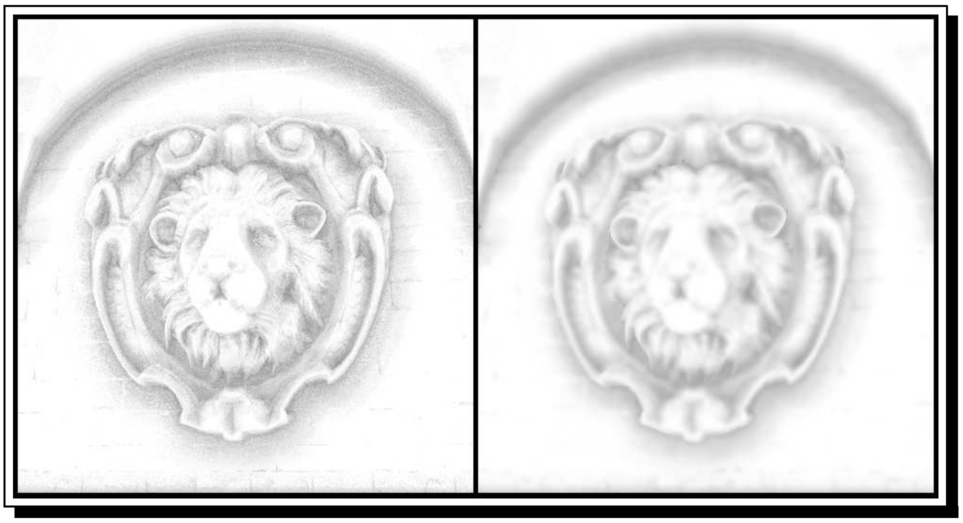


Figure 4.8: an example of ambient obscurance before and after filtering

Chapter 5

SaSSAO implementation

We use C++11 and modern OpenGL to implement our technique in a custom rendering sandbox named YARS (for "Yet Another Rendering Sandbox").

We briefly cover YARS features and then move to discuss the implementation of SaSSAO, giving some information about the relevant techniques, such as deferred rendering, and the needed OpenGL features.

We show the shader programs and framebuffers used in our pipeline and their interactions, building the Blinn-Phong/AO shading model by a geometry pass, an AO evaluation pass, and a filtering/compositing pass.

5.1 OpenGL and GLSL

OpenGL is not properly a software library, it is an API for which most graphics card drivers offer an implementation, allowing developers to exploit the GPU hardware capabilities in a standardized way.

Modern OpenGL works primarily through **GLSL shaders**, programs written into a shading language (GLSL stands for Open**GL** **S**hading **L**anguage). that get compiled by OpenGL implementation and run on the GPU, plugged in the rendering pipeline.

GLSL is C-like and "computer-graphics-friendly", implementing a number of operation very common in computer graphics, such as vector/matrix operations and trigonometric functions. GLSL shaders are compiled by

OpenGL into microcode suitable to be run on the GPU.

The OpenGL 4.3 pipeline is vastly programmable, allowing to plug-in into the rendering process many custom stages (implemented by means of GLSL shaders). Figure 5.1 shows the pipeline structure and positioning of the programmable stages. An OpenGL shader program must contain at least vertex and fragment shader.

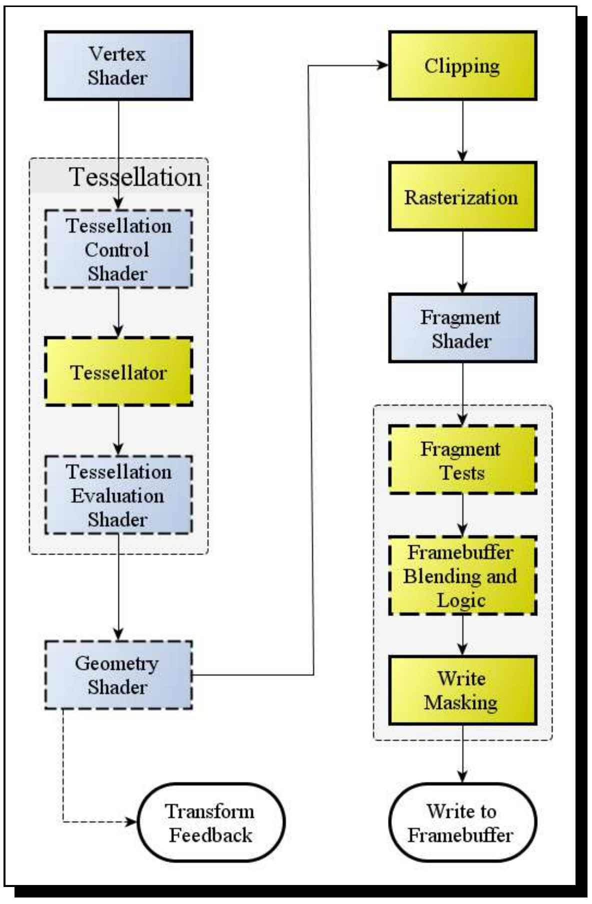


Figure 5.1: an overview of the OpenGL pipeline: blue boxes are programmable stages

We will only use **vertex**, **geometry** and **fragment** shaders.

In oversimplified terms, an OpenGL renderer:

- loads some resources on the GPU (vertex buffers, textures and texture coordinates...) and keeps some handles to such resources
- loads and compiles some shader programs, also identified by some handles
- iteratively, tells the GPU to "draw" *somewhere something* in *some way*, where
    - "somewhere" is the destination of the call (e.g. the default framebuffer shown on the screen, or an image buffer to be further processed)
    - "something" is related to some resource handles to process (e.g. an identifier that refers to the buffer containing the vertex coordinates for a cube, and another identifier that represents the texture that will be applied to the faces of such cube)
    - "some way" is related to the shader program to use, that will process the data, eventually rendering the textured cube (or doing something completely different: it's up to the shader)

5.2 YARS

YARS has been developed to provide a basic but effective OpenGL testing environment, taking care of essential features such as switchable renderers, scene management and a GUI to adjust parameters.

A complete description of YARS would bring the discussion away from the implementation of SaSSAO itself, so we only list its main characteristics and features, and give an idea about its architecture, avoiding unnecessary details even if they took a fair amount of programming work.

5.2.1 Supporting libraries

To build a rendering sandbox from the ground up, other than a graphics API such as OpenGL, a number of libraries are mandatory to shorten the development time and stay focused on the primary task.
We limit ourselves to listing the used libraries and their purposes:

- GLEW: OpenGL Extension Wrangler Library, exposes to the application the OpenGL features available on the system
- GLFW: portable windowing with OpenGL support, and basic input management
- GLM: GL Mathematics, providing a C++ implementation of most mathematical functions offered by GLSL, and additional useful math-related features
- DevIL: image loading and handling, useful to handle textures and for other image-related tasks, such as saving a screen-shot
- Assimp: Open Asset Import Library, to load 3D model formats
- AntTweakBar: simple and lightweight GUI for graphic applications

5.2.2 Architecture

The include dependency diagram in figure 5.2 can give a partial idea of the sandbox architecture. It is relatively simple but easily extendable by plugging new renderers and shaders, and maybe new abstractions for other OpenGL features.

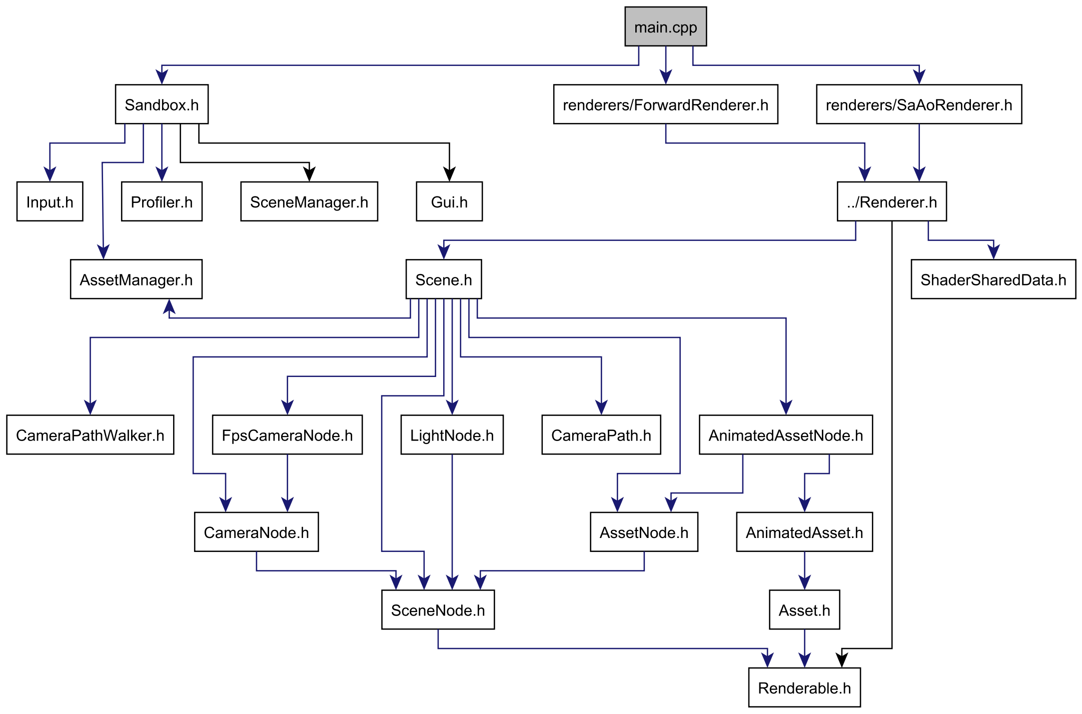


Figure 5.2: some of the files composing YARS and their include dependencies

Note that the input handling, scene management and actual rendering are somewhat decoupled: in the rendering loop, input is forwarded to the current scene, that adjusts its elements according to what is defined in its logic, and after that the current renderer actually renders the new state of the scene.
So, you can define new scene behaviours by inheriting from Scene and overriding the update() method, much likely you can implement a new renderer by inheriting from Renderer and providing a new render() method. Some classes related to shaders handling are also provided, and ShaderProgram can be used as it is or extended to provide additional helper functions to simplify shader communication, as we'll see in the following paragraphs.

5.2.3 Scene Management

A YARS scene essentially consists of a SceneNode tree. Multiple scenes can be loaded in memory at the same time and the GUI allows to select the current scene (that gets rendered to the screen).

SceneNodes are scene elements that, through the GUI, can be added, removed, transformed (position, scale, orientation) and assigned as child of another node.

Scenes can be saved to simple .json files describing the elements hierarchy and attributes, and then loaded at a later time.

There are three main types of SceneNodes:

- AssetNodes, that have some renderable asset attached to them (textured models, loaded from supported file formats)
- LightNodes, that represent light sources (color, type, falloff characteristics)
- CameraNodes, that model cameras (field of view, clip planes...)

A minimal support for animated AssetNodes and for camera flythroughs is provided. Mouse and keyboard input handles camera movement and nodes positioning and transformation.

5.2.4 Rendering

Multiple renderers can be initialized and switched at runtime, simplifying testing. The currently active renderer exposes its configurable parameters through the GUI.

Basically a renderer can access the current scene and instantiate one or more shader programs and buffers to finally build the framebuffer that will be shown on screen.

The basic implementation objective for the sandbox was fully supporting the Sponza model by Crytek[1], one of the most popularly used in global illumination demos. This guided the rendering development into supporting some features and characteristics, such as:

- texture mapping (material diffuse color taken from image maps)
- specular mapping (material specular color taken from image maps)

[1]at the time of writing, such model is available for download at http://www.crytek.com/cryengine/cryengine3/downloads

- normal mapping (surface normals taken from image maps)
- alpha masking (image maps define the transparency of some geometry)

Then, the three conventional OpenGL types of light sources, as modeled by the LightNodes (directional, point and spot), are supported by specific shading functions.

A basic forward renderer that processes all the scene assets as seen from the current camera, and calculates scene lighting by iterating through the light sources, is the base upon more complicated renderers can be built, as we'll explain.

5.2.5 Shaders

GLSL shaders are handled through three YARS classes:

- Shader: an abstraction on a sigle shader, specifying its type (vertex, geometry, fragment) and source code, and handling its compilation
- ShaderSubroutine: an abstraction on the shader subroutines OpenGL feature, allowing to switch shader functions implementation at runtime (a function pointer like mechanism)
- ShaderProgram: an abstraction on a full shader program, composed of more Shader objects linked together and exposing functions to access the uniforms, UBO and texture bindings of the shader program

Even if using instances of the generic ShaderProgram class is possible, the most straightforward way to handle a shader program is defining a new class that inherits from ShaderProgram and builds upon it.

One of the ShaderProgram constructors allows to just specify the text files containing the shader code for the desired stages (for example: geometryPass.vert, geometryPass.geom, and geometryPass.frag, for vertex, geometry and fragment shader). Such constructor reads, compiles and links together the shaders, building ready-to-use shader program.

Each ShaderProgram-inherited class retrieves and stores the handles to uniforms and UBOs and binds the texture samplers to some texture units: in practice, it does all the necessary operations to later allow straightforward and efficient communication between the host application and shader program running on the GPU.

We also implemented in YARS a couple of functionalities on top of the standard OpenGL features. Detailing each one would take too much space, but a briefly mention is necessary to the reader to understand the provided shader code and the sandbox behaviour:

- include mechanism: to better factorize the shader code, factorizing some shared functions in separate files was needed. A simple "#include", C-like mechanism was implemented in the text file reading routine called by the ShaderProgram constructor
- editable compile-time constants: it might be desired to change some values from the host application that will stay constant after shader initialization, and so do not need to be declared as uniform. We expose such values to the host application an trigger shader recompilation on change
- shader subroutine handling/emulation: OpenGL 4.3 offers a mechanism to easily select different implementation of a shader function - we wrap it to allow easy implementation selection from the host application and we implement a switch-based replacement, mostly to evaluate the performances of switch vs subroutine selection

Uniforms and Uniform Buffer Objects

Efficient communication between host application and shader programs is a key point of OpenGL programming.

The basic way the host application communicates with shader program is by setting **Uniform Variables**. Every shader can define such variables, and the host application can query for their locations and manipulate their values. Then, the shader program can access (read-only) such values. We use Uniform Variables for shader-specific parameters.

Additionally, we use **Uniform Buffer Objects** (UBOs). They allow access from multiple shaders and offer a more efficient way to set large chunks of data with a single operation. They also allow to quickly change between sets of uniforms. We use UBOs to set into shaders the model/view/projection matrices (and some more), to keep material descriptions on the GPU (switchable sets of uniforms), and to efficiently update the light sources definitions. In YARS, the class SharedShaderData handles the UBOs.

It's imperative to understand how data flows between the shader stages and what data is accessible by each stage. Also, is important to do computations in the right place.

For example, let's consider lighting. We do lighting in camera space. This means that light sources position must be transformed from world space to camera space, essentially multiplying the LightNodes world space information by the view matrix.
We could upload world space light coordinates as they are, and then multiply them in the fragment shader, because it's there that we calculate lighting and we need them. Unfortunately, that would mean that such multiplication would be performed for each fragment, thousand of times - something that is absolutely useless: you can multiply once in the application and upload the already transformed coordinates to the UBO. Anyway, it must also be noted that even if this principle is generally valid, it's sometimes possible, due to hardware optimizations, that doing some operations multiple times on the GPU is still faster than doing them just once on the CPU side.

5.3 Deferred Shading

SaSSAO implementation, as it's common for SSAO techniques, is based on **deferred shading**.

Deferred shading gets its name from the fact that the first rendering pass (often called *geometry pass*) doesn't perform shading, that is instead "deferred" to a second pass.
The first pass renders to a **G-buffer** (for "geometry buffer"), that consists of a number of textures that will store the information needed for later processing.

An modern GPU feature that is commonly used to build the G-buffer is **MRT rendering**. MRT stands for *Multiple Render Target*, and allows having multiple outputs in the fragment shader: in a single geometry pass, all the textures of the G-Buffer can be filled with color data representing the information designated to be stored. In the past, that would have required multiple passes, causing inefficiency.

After rendering to the G-buffer textures, the common practice is switching to another shader program and rendering as geometry a *full-screen quad*: in such shaders, using texture sampling on the G-buffer, the information needed for lighting or other processing is accessed.

Deferred rendering techniques have both advantages and disadvantages: the main advantage is the ability to decouple the geometry processing from the shading calculation, that usually is the most computationally heavy part of the process. Some disadvantages are the difficulties in handling semi-transparent objects, or using different shading functions for different materials. More low-level problems are the lack of hardware anti-aliasing and the high memory bandwidth requirements. Anyway, these in many cases are a reasonable price to pay for the many scree-space processing possibilities made possible by the G-buffer. Hybrid approaches are also possible, in which for example semi-transparent geometry is rendered on top with a separated forward pass.

We ignore all these "advanced" issues and focus on flexibility and some degree of performance.

The MrtBuffer and FullScreenQuad classes, in YARS, support these techniques.

5.4 SaSSAO rendering pipeline

The SaSSAO rendering pipeline is implemented by the SaAoRenderer class. The include dependecy graph shown by figure 5.3 gives a glimpse about the structure of the renderer implementation, excluding the GLSL shaders.

Let's recap the essential implementation information presented until here, before moving to showing the actual pipeline elements and organization.

- we have loaded resources (geometry descriptions, textures, materials) on the GPU
- we have handles to such resources in AssetNodes, organized in a hierarchy, that is the SceneNode tree of the currently active Scene
- we have an active CameraNode that defines the view that will be rendered by the current Renderer

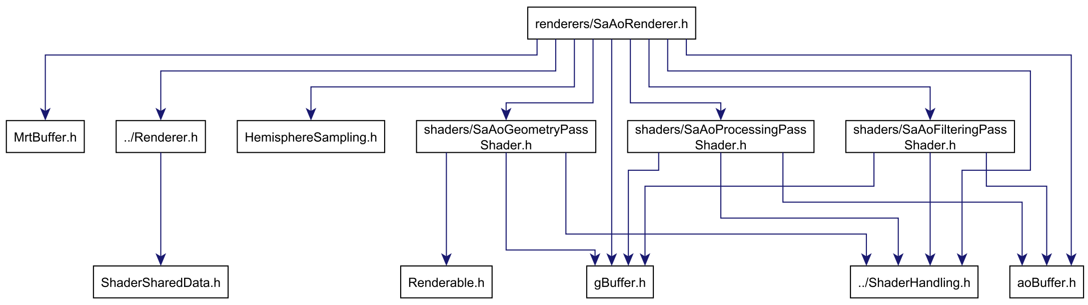


Figure 5.3: some of the files composing YARS and their include dependencies

- we update the active Scene according to user input
- we send the needed information to shader programs through uniforms and UBOs
    - something is updated only occasionally, something every frame

The SaAoRenderer pipeline elements are three instances of classes inheriting from ShaderProgram, and two instances of MrtBuffer:

- programs::
    - SaAoGeometryPassShader geometryPassShader
    - SaAoProcessingPassShader processingPassShader
    - SaAoFilteringPassShader filteringPassShader
- buffers:
    - MrtBuffer gBuffer
    - MrtBuffer aoBuffer

All these elements are initialized by SaAoRenderer. The three Shader-Programs are connected to the SharedShaderData class handing the UBOs.

The pipeline involves three passes working on these building blocks, essentially with this logic, that can better be understood by also looking at the diagram in figure 5.4:

- first pass (geometry pass)

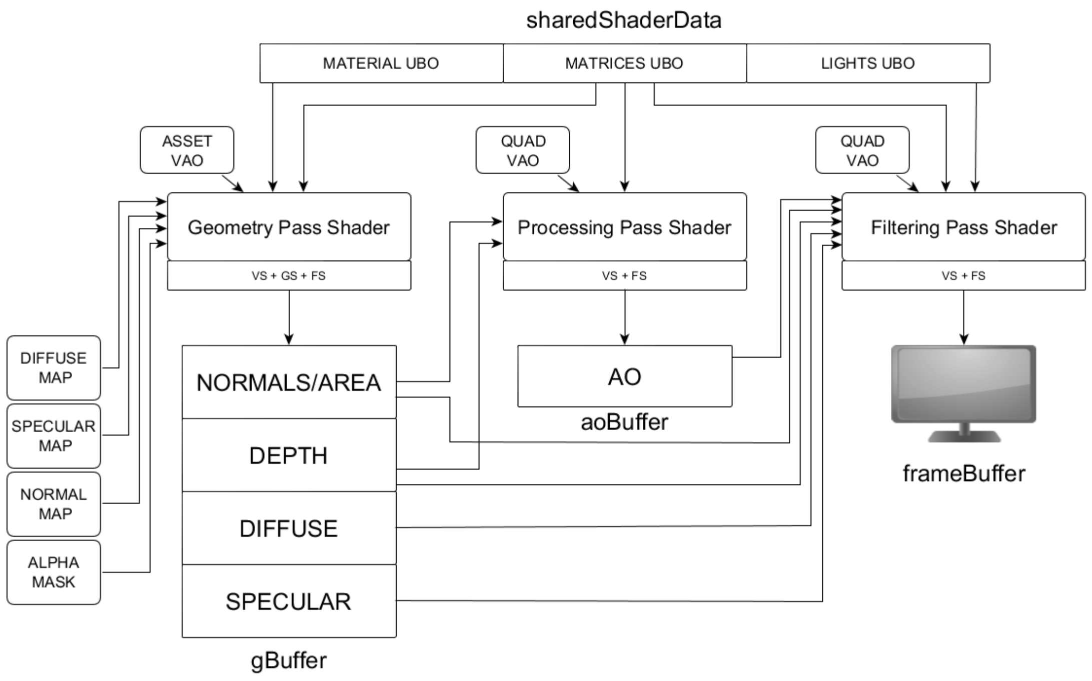


Figure 5.4: an overview of our rendering pipeline

- ◦ gBuffer is set as current rendering target
- ◦ geometryPassShader is set as the active shader
- ◦ all the current Scene geometry is rendered by issuing *draw calls* for the AssetNodes
  - ◇ this essentially builds the G-buffer, preparing color/normal/depth textures for the following stages

- second pass (processing pass)
  - ◦ gBuffer textures are bound to be used as input data
  - ◦ aoBuffer is set as current rendering target
  - ◦ processingPassShader is set as the active shader
  - ◦ a full-screen quad is rendered
    - ◇ this causes the ambient obscurance calculation to be estimated (by sampling gBuffer) and written to aoBuffer

- third pass (filtering/compositing pass)
    - both gBuffer and aoBuffer textures are bound to be used as input data
    - the default framebuffer (shown on-screen) is set as current rendering target
    - filteringPassShader is set as the active shader
    - a full-screen quad is rendered
        - the aoBuffer gets filtered to remove noise
        - the filtered aoBuffer value modulates the ambient light factor
        - the modulated ambient light factor gets added to the direct lighting of the scene, calculated from gBuffer, compositing the final image

Let's give some details about the renderer initialization and each phase of the process.

5.4.1 Initialization

The renderer initialization consists in the setup of the two MrtBuffer instances, and of the three ShaderPrograms.

Thanks to our infrastructure, most of the OpenGL work is abstracted by the MrtBuffer and ShaderProgram classes.

The MrtBuffer constructor takes the number and type of the desired render targets, and the init() method actually allocates appropriate textures into the GPU memory and binds them to a *framebuffer object*, an OpenGL abstraction that specifies a buffer that can be target of drawing operations.

Such buffers can be bound for reading or writing by the current shader program, and so can be used to transfer data between different rendering passes.

YARS automatically re-initializes the MrtBuffer in two cases:

- the window size changes, and so new textures of appropriate size must replace the old ones
- the user changes the texture storage type (this is allowed through the GUI), something that can be useful to do at runtime while testing and profiling

  - "wider" data types use more memory (and so, more bandwidth) but allow more quality: for example, you can have configure the depth buffer to use 16bits or 32bits for storing the depth of each pixel, on the basis of the depth range and precision you need to handle

Textures typically store four values for each pixel, referred interchangeably as RGBA or XYZW according to the usage. To store depth information, specific types must be used.

The gBuffer instance has three color targets (diffuse, specular, normal) and a depth target, and is structured like this:

- diffuse
  - RGB: the diffuse color RGB values, coming from a texture or material description
  - A: not used
- specular
  - RGB: the specular color RGB values
  - A: shininess
- normal
  - normal.XYZ: the normal vector at the pixel (considering normal-mapping)
  - W: the area related to the triangle the pixel belongs to, calculated in the geometry shader
- depth: storing the depth of each pixel, and handled automatically by OpenGL

The aoBuffer is much simpler, only containing a single target of a special type of texture with a single component (the RED component)

- ao.R: the ambient obscurance level for the pixel (ranging between 0 - totally occluded - and 1 - unoccluded)

Regarding the ShaderPrograms, they are so structured in terms of stages and main files loaded (we skip the details about the subdivision of shader code into different teext files that get "included"):

- SaAoGeometryPassShader geometryPassShader
  - vertex: deferred.vert
  - geometry: saao_geometryPass.geom
  - fragment: saao_geometryPass.frag
- SaAoProcessingPassShader processingPassShader
  - vertex: deferred_fullscreenquad.vert
  - fragment: saao_processingPass.frag
- SaAoFilteringPassShader filteringPassShader
  - vertex: deferred_fullscreenquad.vert
  - fragment: saao_filteringPass.frag

5.4.2 Geometry Pass

On the CPU side, the geometry pass is performed by this chunk of code:

```
gBuffer.bindForWriting();
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glEnable(GL_DEPTH_TEST);

geometryPassShader->use();
renderSceneGeometry(scene);
gBuffer.unbind();
```

Let's repeat the pass overview adding some detail for each step:

- gBuffer is set as current rendering target
  - gBuffer.bindForWriting() sets gBuffer as target of the following draw calls. It's cleared to black, and the depth test (an OpenGL feature needed to be able to draw geometry in arbitrary order without risking of drawing a far object in front of a near one), is enabled
- geometryPassShader is set as the active shader
  - geometryPassShader->use() activates the geometryPassShader, meaning that the following draw calls will be processed by such program on the GPU

- all the current Scene geometry is rendered by issuing *draw calls* for the AssetNodes
    - renderSceneGeometry(scene) traverses the scene graph and does render calls for every AssetNode in the scene, binding the appropriate textures and materials. Also, it uploads to the shader UBOs the shared data, including the transformation matrices and light sources description

On the GPU side, the draw calls will cause rendering to the G-Buffer, handling normal mapping, transparent geometry, texture mapping and alpha masking. [2]

Figure 5.5 shows an example of the output of this pass.

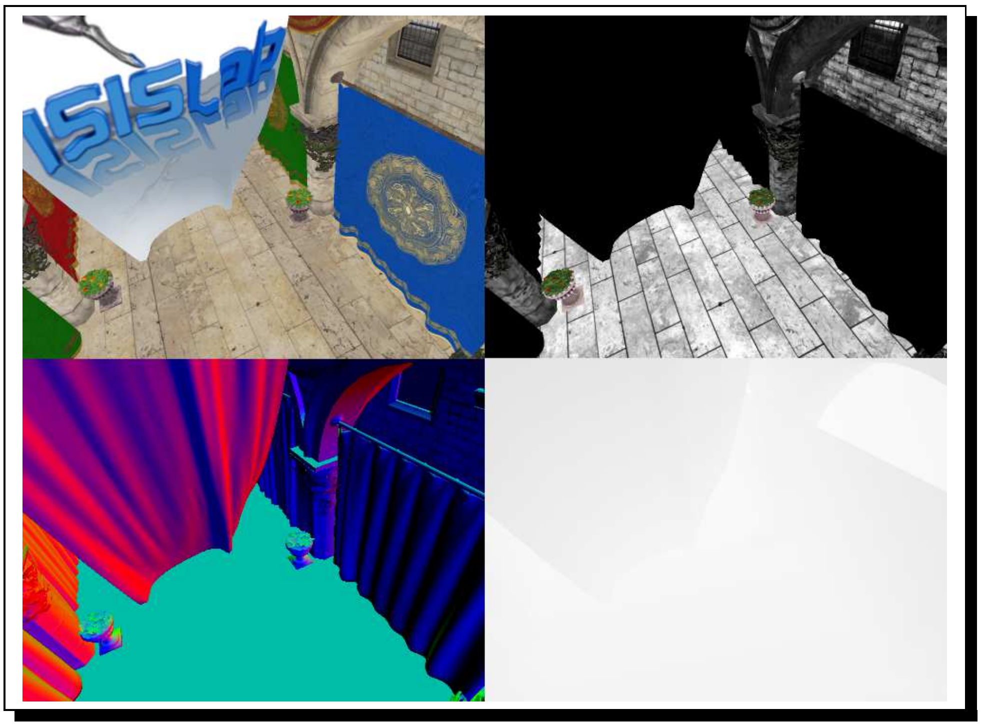


Figure 5.5: the G-Buffer after the geometry pass

[2]these operations are out of the scope of our specific techniques and so we don't discuss them

5.4.3 Obscurance Processing Pass

This is the most complex and performance critical pass.

```
glDisable(GL_DEPTH_TEST);

gBuffer.bindForReadingAsTextures();
aoBuffer.bindForWriting();

glClearColor(0.0, 0.0, 0.0, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);

processingPassShader->use();

quad->draw();
aoBuffer.unbind();
```

In the host application, the code is straightforward: after disabling the depth test, (who doesn't make any sense when drawing a single full screen quad as we are in the second and third pass), we bind gBuffer for reading, meaning that we can sample its textures from the active shader. Then, we set aoBuffer as rendering target, enable the processingPassShader shader, and draw a quad.

When drawing a quad, we basically just use the fragment shader to do any kind of full screen processing. In this case, we want to fill the aoBuffer, that will contain our obscurance estimate.

On the GPU side, the fragment shader implements all the steps of the technique described in chapter 4, and the interested reader can easily understand the shader code by referring to such chapter.

Figure 5.6 shows an example of aoBuffer after the execution of the pass.

5.4.4 Filtering/Compositing Pass

This pass is simple but important: as we discussed previously, high frequency noise is the best we can expect in aoBuffer (the worse would be banding artefacts or other artefacts related to regular sampling) without using a very high number of samples.

A filtering pass on aoBuffer can definitely improve the quality of the result, removing or at least reducing such noise and giving ambient obscurance a smoother look. Also, in this step we have to finally render the framebuffer,

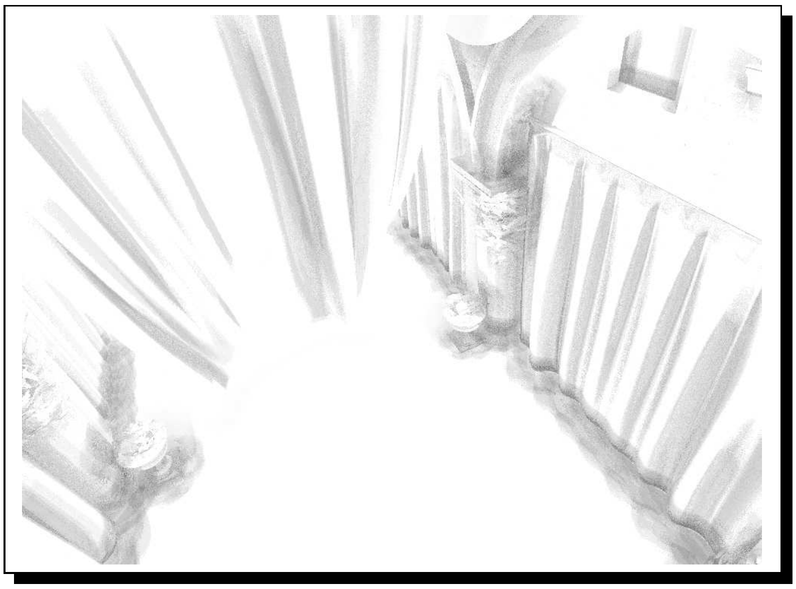


Figure 5.6: the AO-Buffer after the processing pass

computing the direct lighting and adding the ambient lighting (modulated by the obscurance).

```
const glm::vec3& bg = scene.getBackgroundColor();
setBackgroundColor(bg);
glClearColor(bg.r, bg.g, bg.b, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);

aoBuffer.bindForReadingAsTextures();
filteringPassShader->use();
quad->draw();
```

The framebuffer is cleared with the scene background color. In this pass we need to read both MrtBuffers: gBuffer is already bound from the previous pass, and we bind aoBuffer too.

Then, the filteringPassShader is activated and a full screen quad is drawn, as in the previous pass.

This time, the fragment shader is relatively simple, if we skip the details about standard direct lighting.

- direct lighting is calculated, by fetching from gBuffer diffuse color, specular color/shininess and normal value. The *eye* vector is derived from the camera space position, that itself is derived by the pixel coordinates and depth

- the filtered ambient obscurance value for the pixel is calculated by applying the bilateral filtering process discussed in 4.2.3 to aoBuffer and then used to modulate the ambient light
- direct light and ambient light are combined together and written to the framebuffer

So, this fragment shader essentially implements the modified Blinn-Phong shading equation discussed in 2.4.4:

$$I = AO * I_a + I_d + I_s$$

Figure 5.7 shows the final compositing of the example rendering used in the previous paragraphs. You might also want to revisit figure 1.3, shown in the introduction, to better appreciate how the different components are blended together.

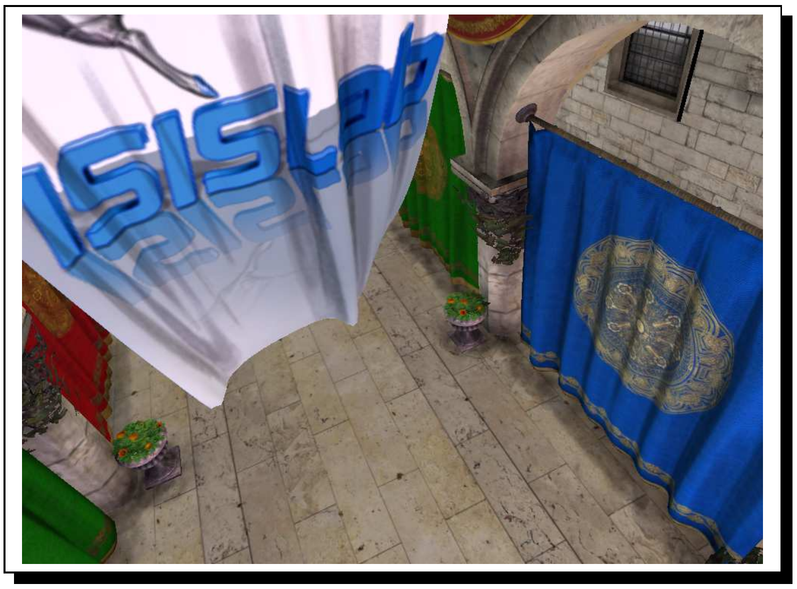


Figure 5.7: the framebuffer, after the compositing pass. The scene has low direct lighting and high ambient light to better show the ambient obscurance effect

5.5 Parameters and performance

The quality and performance of the technique vary sensibly according to a number of parameters (that complicate testing and comparing the technique to other approaches). All these parameters are configurable through the YARS GUI to have instant visual feedback.

5.5.1 aoMultiplier

The $aoMultiplier$ is a multiplicative factor applied to the filtered obscurance value, before modulating the ambient light. So, adjusting the value can attenuate or strengthen the ambient obscurance effect.

5.5.2 Number of Samples

The $samplesNo$ parameter adjusts the number of samples taken in the sampling area. Of course, more samples mean more quality, but also more calculations and texture fetches, so this parameter has a dramatic impact on how the technique behaves in terms of quality and performance. A high number of texture fetches can quickly degrade performances.

You can think about $samplesNo$ as the image-space counterpart to the number of rays shot in ray-traced ambient occlusion.

5.5.3 Number of triangle buckets

The $triangleDivs$ parameter, that can be set to 2, 3 or 4, defines the level of precision we can have when classifying occluders directions.

We have discussed the details about the hemisphere partitioning scheme in 4.1, but as a quick reminder, you get the hemisphere partitioned in $4 * triangleDivs^2$ buckets.

More buckets generally allow more precision in classifying occluders direction, but another factor must be taken into account: we cap the possible occlusion brought by a sample to the solid angle of a bucket, so having more buckets also means that each sample can bring less coverage. This implies that the number of buckets should be somewhat related to the number of samples: taking few samples with many buckets won't work well, catching very low coverage and so very low obscurance.

Even if the computational complexity of finding the bucket is the same independently on *triangleDivs*, an higher number of buckets implies larger arrays to keep the values (and so, less cache coherence) and longer loops, so generally a higher *triangleDivs* impacts negatively performances.

5.5.4 Sampling Pattern

As discussed in 4.2.2, we implemented two samples positioning schemes,

- flat: samples in a circle around the current pixel
- hemisphere: samples in a hemisphere around the current pixel, oriented according to its normal

In both case, some randomization is needed to avoid artefacts. We tried different approaches, and ended up generating a random value according to a popular GLSL one-liner that works well in practice:

```
float rand(const in vec2 co){
    return fract(sin(dot(co.xy, vec2(12.9898,78.233))) *
        43758.5453);
}
```

According to the pixel position on screen space, a random value is generated. Such value is used to rotate a pre-calculated set of displacement vectors that also get scaled according to the sampling radius.

The *hemisphere* sampling is more computationally heavy than the *flat*, because sampling vectors must be transformed to the tangent space of the current point, according to its position and normal.

5.5.5 Sampling Radius

This is the radius of the circle (for flat sampling) or sphere (for hemisphere sampling) into which sample points are taken (in camera space). So, positioning of samples varies according to the depth of the pixel, as it makes sense to cover the same actual distance in the scene.

Adjusting *samplingRadius* (and the *maxDistance* we'll discuss in the next paragraph) changes the range in which we take account for occluding geometry. So, we'd like a reasonably large radius to get samples on more distant geometry: a very short radius would result in very localized darkened areas and not in soft diffuse shadows.

Unfortunately the *samplingRadius* has a significant impact on performances, because the texture fetches are related to cache efficiency.

As illustrated in [Dog12], the texture cache exploits the heavy reuse between neighboring pixels, so sampling a close fragment will often result in a cache hit, while fetching values for a distant fragment could end up being a much more expensive operation.

Note: the samples aren't taken at exactly *samplingRadius* distance (that would cause artefacts), but at random distances between ϵ and *samplingRadius*.

5.5.6 Max Distance

The sampling scheme does the best that can in placing the sample points "near" the current point, according to the *samplingRadius* but, due to perspective projection, geometry much more distant than expected can be found when actually sampling the point. Unfortunately, this can't be know before sampling at least the depth of the sample pixel, from which the camera space position is reconstructed.

After reconstructing such position, its distance with the camera position of the current point is evaluated.

If such distance is greater than *maxDistance*, the sample is no further processed, because is assumed to be too distant to bring occlusion.

Setting a high *maxDistance* can lower performance, because more distant samples are taken into account and calculated, even if they result in giving no contribution to the obscurance. On the other hand, it should always be greater or equal than the *samplingRadius* parameter, or we would generate sample positions that would be discarded in any case, and that doesn't make sense. A rule of thumb could be setting *maxDistance* to twice the value of *samplingRadius*.

5.5.7 Angle Bias

The *angleBias* parameter, that often appears in SSAO techniques, is used to limit the self-occlusion and the artefacts due to geometry almost co-planar with the one of the current point. If the cosine of the angle between the current point normal and the direction to the occluder is less than *angleBias*, the sample is ignored.

5.6 Adding indirect illumination

We didn't have the time to adequately explore screen space indirect lighting, but we did a quick experiment by modifying our pipeline to achieve some visually pleasing color bleeding effects.

The core idea (also exploited in, among others, [Bun05], [SHR10], [RGS09]) is that occluding geometry is also a source of diffuse indirect light, and that knowing its position, orientation and direct diffuse lighting, we can estimate a form of radiance transfer from the occluding geometry to the current point.
Of course, the scope of the effect is limited to light coming from geometry in the *samplingRadius* range.

Our approximation, based on the differential element to differential element form factor formula, and using the notation and elements already presented in chapter 4.2.2 by fig. 4.5, shown again for reader convenience here as fig. 5.8, is

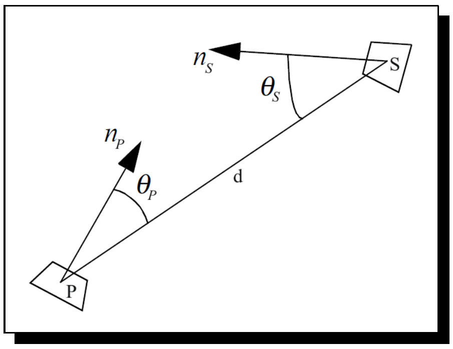


Figure 5.8: the angles and vectors related to the form factor approximation

$$ff_{S->P} = \frac{cos\theta_P \; cos\theta_S}{\pi r^2} A_S$$

5.6.1 Pipeline modification

The diagram in figure 5.9 shows the modified pipeline.

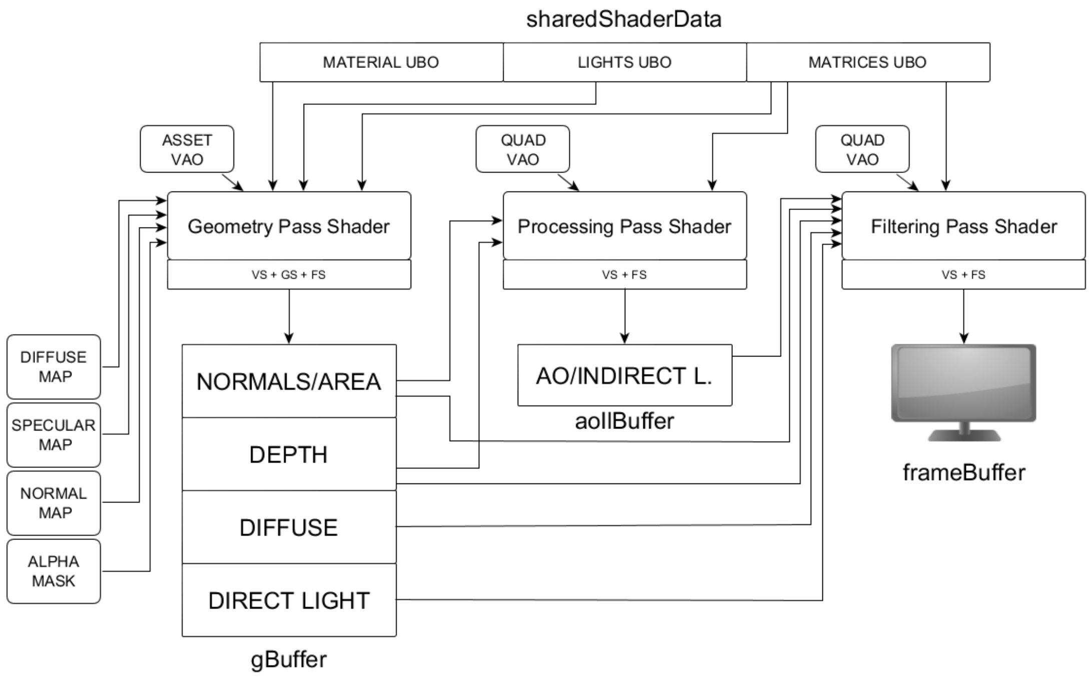


Figure 5.9: an overview of our rendering pipeline modified to also approximate some indirect lighting

We don't defer direct lighting: we calculate it in the geometry pass, in a forward-rendering fashion, but save it in gBuffer, substituting the specular data (that is not needed any more in later passes).

Having such data in gBuffer allows us to sample the direct (unoccluded) lighting at the same time when we do sampling to calculate the ambient obscurance, so the performance impact of the additional calculations is very limited.

The aoBuffer, renamed aoIIBuffer, doesn't store only the occlusion value in a single-color texture. A full RGBA texture is used, with the indirect light color in the RGB components and the obscurance values in the A component.

In the filtering pass, the indirect lights gets filtered together with the obscurance value, resulting in smooth color bleeding, again without significant performance loss.

The final compositing changes slightly

- the direct lighting is already available in the gBuffer, calculated in the geometry pass
- the filtered indirect light value must also be added to the final result

$$I = indirectLight + AO * I_a + I_d + I_s$$

We show in 5.10 fig a comparison of an image rendered without ambient obscurance, with ambient obscurance only, and with ambient obscurance and indirect light. Both effects are modulated with a high "strength" parameter to better show the kind of effects achievable.

No testing was done, for time constraints and to not diverge too much from the subject of our work, about the actual correspondence with ray-traced single bounce indirect light calculation.

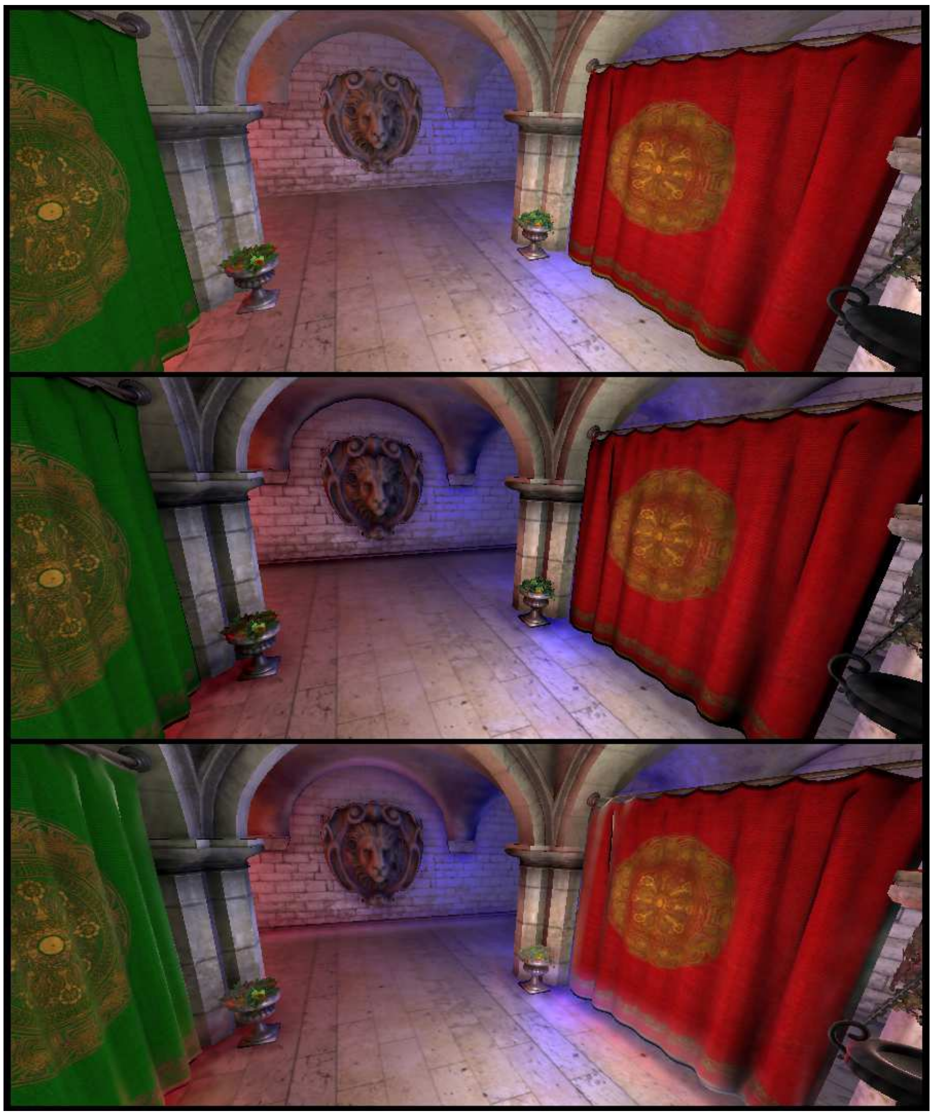


Figure 5.10: no ambient obscurance, ambient obscurance only, ambient obscurance and indirect light

Chapter 6

SaSSAO testing

Testing real-time rendering algorithms can be complicated: they aim to provide visually pleasing results to the human eye.

Anyway, an objective testing methodology can be helpful in evaluating the validity of a rendering technique and in comparing it with other algorithms.

In terms of quality, it makes sense to use an off-line rendered image, calculated through ray tracing, as reference image: the more we can get close tu such image with a real time renderer, the better. We use **Blender** to render ray-traced ambient occlusion.

Of course a reference image is only the first step: we also need a metric to define the similarity between two images. We adopt the **Structural Similarity (SSIM) Index**, a metric that attempts to measure similarity between images in a way somewhat consistent with the human eye perception.

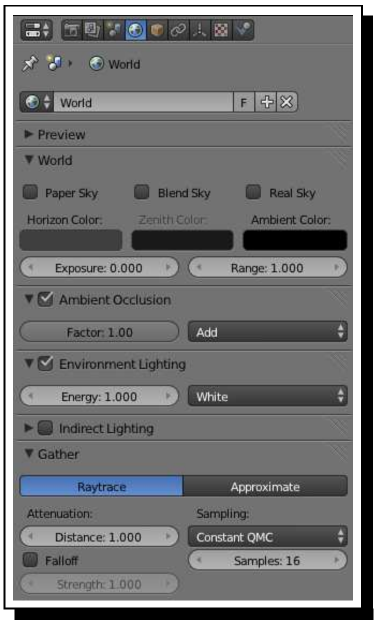
We build a simple but effective testing infrastructure that allows as to get the two rendered images (from our technique and from Blender) and their SSIM evaluation with a couple of key-presses.

6.1 Blender Ambient Occlusion

Using Blender to render ambient occlusion images, for the purpose of quality comparison with a real-time technique, is relatively straightforward.

As we introduced in 2.2, ray-traced ambient occlusion is calculated by casting rays from each visible point and calculating the percentage of them obstructed by geometry.

There are a few parameters inherent the calculation of ambient occlusion that we must discuss, accessible from the GUI (as shown by fig 6.1) but also through Python scripting.



World

World F

Preview

World

Paper Sky Blend Sky Real Sky

Horizon Color: Zenith Color: Ambient Color:

Exposure: 0.000 Range: 1.000

✓ Ambient Occlusion

Factor: 1.00 Add

✓ Environment Lighting

Energy: 1.000 White

Indirect Lighting

Gather

Raytrace Approximate

Attenuation: Sampling:

Distance: 1.000 Constant QMC

Falloff Samples: 16

Strength: 1.000

Figure 6.1: the blender panel showing most ambient occlusion related settings

Rays are shot at the hemisphere according to a random pattern controlled by the *Sampling* parameter, that features three options:

- Constant QMC: quasi-Monte Carlo, evenly and randomly distributed rays
- Adaptive QMC: an improved version of QMC, that tries to determine

when the sample rate can be lowered or the sample skipped

- Constant Jittered: samples are distributed according to a fixed grid

Constant QMC gives the best quality, and so it's our choice for this parameter.

Then, we can configure the number of rays used, controlled by the *samples* parameter. Of course more rays imply more accurate results and slower render times.

The *distance* parameter defines how far the occluded geometry is considered as bringing occlusion. This somewhat maps to the *maxDistance* parameter of our technique implementation. A shorter distance can speed-up rendering because it implies that the renderer has to search for intersections in a narrower area.

We set this parameter on the basis of the *maxDistance* parameter currently set in YARS.

Moreover, there's the *attenuation* setting. If enabled, the distance to the occluding objects will influence the calculation: higher values of attenuation *strength* will cause quicker falloff (that will result in shorter shadows).

The valid range for the attenuation *strength* parameter is 0...10.

The falloff function used by Blender is not defined in the documentation, but a quick analysis of the source code shows that it's an exponential with negative exponent.

```
// from rayshade.c

if (RE_rayobject_raycast(R.raytree, &isec)) {
    if (R.wrld.aomode & WO_AODIST)
        fac+= expf(-isec.dist*R.wrld.aodistfac);
    else
        fac+= 1.0f;
}
```

The `exp(x)` function returns the exponential value of its parameter, that is e^x. So, we have:

$$e^{-intersectionDistance*falloffStrength}$$

Being the falloff calculation different than our own (as the occlusion estimate, anyway), we decide to disable the falloff in the Blender test renders. This is left as an interesting note about where to look in the Blender C source code in case someone wants to use it for more advanced comparisons or alternative occlusion computation.

6.2 SSIM index

The Structural SIMilarity (SSIM) index, introduced in [WBSS04], is a method for measuring the similarity between two images.

For applications in which images are ultimately to be viewed by human beings, subjective evaluation is the ultimate method to quantify quality. Of course, it is also inconvenient, expensive and error-prone, so research in the field of objective image quality assessment tries to develop quantitative measures that can automatically predict perceived image quality.

SSIM has been designed to be more consistent with the human eye perception than other traditional methods of image comparison. It evaluates change in structural information, assuming that pixels spatially close have strong inter-dependencies, and that such dependencies carry important information about the structure of the objects in the image.

Going through the inner workings of SSIM is out of the scope of our work, and is not needed to implement our testing system because there are ready-to-use binaries[1] implementing the comparison on image files.

6.3 Testing infrastructure

First, we have recreated our test scene in Blender, by loading the assets and placing them appropriately. This could have been automatized too, by writing a script to parse our .json scene description files, but wasn't done because, for time constraints, we limited testing to a single scene, implying that automatic scene setup in Blender wasn't needed.

Then, we have written a Python script that moves and rotates the camera and adjusts a number of parameters related to the ambient occlusion

[1]we use a C++ implementation by Mehdi Rabah, based on OpenCV and downloadable from `http://mehdi.rabah.free.fr/SSIM/`

rendering. In this script, we have used a number of place-holder constants that will be substituted later at runtime (by YARS) with actual values for the current test.

In YARS, we bind to a key-press a simple function that reads the script file and writes a copy of it in which the place-holder constants are substituted with the current camera settings (and a few other parameters, such as the distance considered in the occlusion calculation). Also, a screen-shot of the YARS rendered image is saved.

Simply executing the generated script in Blender produces the ray-traced equivalent of the YARS saved-screenshot, and executes the SSIM utility that compares the two images. The output of such utility is saved to a text file.

Figure 6.2 summarizes the process.
The file names of the text file and the images include the current test values, avoiding name clashes.

Such testing environment is very simple but quite effective. With a little more effort, YARS could be connected to Blender and continuously update the camera and parameters. Then, a single command from YARS could save the screenshot, launch the Blender render and, when finished, perform the SSIM comparison and finally show/save the result. Another possible improvement would be supporting a .json description of test cases that could be automatically repeated, maybe trying different ranges of values for the different parameters, finding which ones give better results in the SSIM comparison.

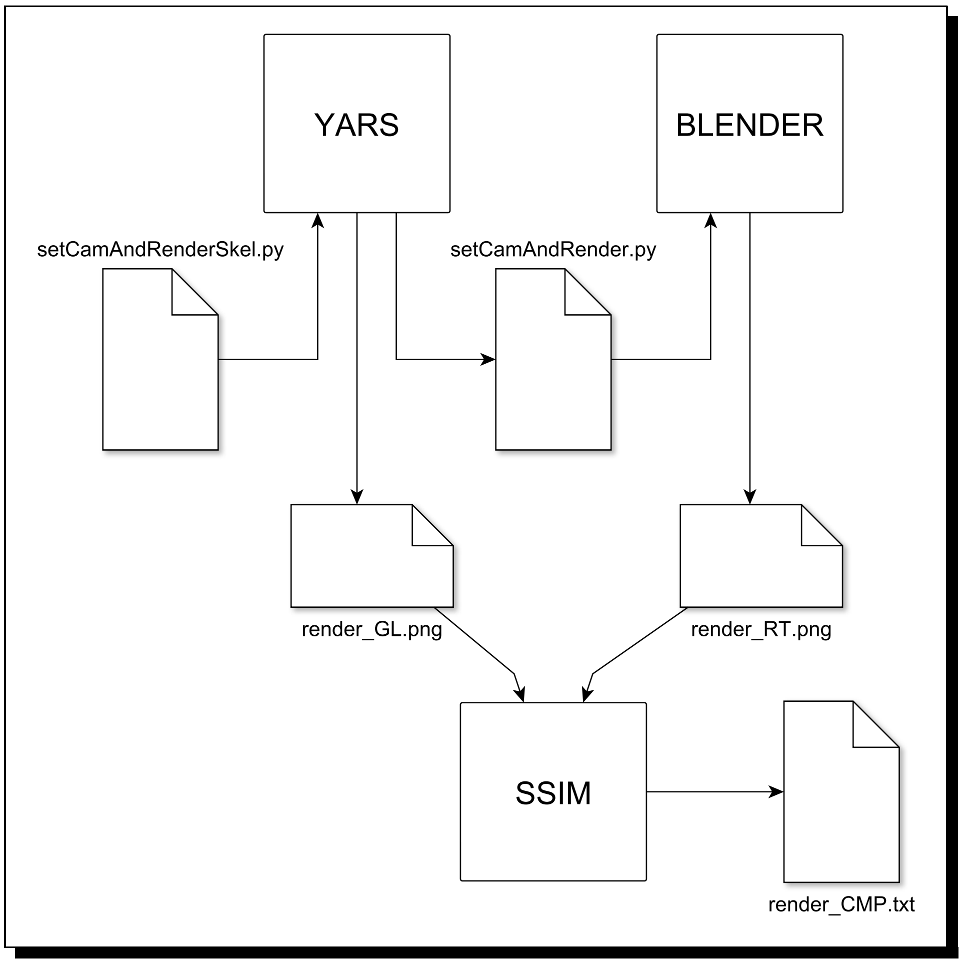


Figure 6.2: A diagram of our simple test infrastructure

6.4 Test Results

Our testing system is equipped with an Intel Core i7-3820 CPU 3.60 Ghz, 16 GB of RAM and a GeForce GTX TITAN GPU.

We select a number of camera positions and capture screenshots while varying some parameters. With the previously described infrastructure, we launch off-line renderings in Blender from the same camera settings and do batched SSIM index comparisons. The Blender renderings are done setting 64 as number of samples, a very high number that gives excellent quality images (that take a few minutes to render). The resolution of all the images is 1280x720 pixels.

Additionally, we have implemented the Alchemy algorithm, to be able to evaluate SaSSAO against an already established technique. We implemented Alchemy doing the minimal changes needed to our already active pipeline, so we can share many parameters between the two techniques, and have more meaningful comparisons. Only the shader function that creates the AO-Buffer, in SaAoProcessingPassShader, was modified. Of course, this also means that our implementation could be far from optimal, so results are merely indicative.

Some results are encouraging, with our algorithm often giving results of comparable quality to Alchemy (sometimes even slightly superior). Alchemy does much less calculations for each sample, so with the same number of sample is generally faster, but the interesting aspect is that with less samples and more calculations we sometimes get approximately the same quality at similar frame rate. This could be useful in bandwidth-limited situations.

The ultimate truth, anyway, seems to be that adjusting properly the parameters is what can totally change the results. Also, we feel like an higher SSIM index image is not always matching what would be picked by a human observer as the best result, so we're not completely confident that such index is a particularly effective metric for ambient obscurance comparisons. For example, sometimes, smoother images taken with more samples receive a lower SSIM index, as you can see from the following tables.

Unfortunately, due to the high number of parameters combinations, we can't be exhaustive. Interested readers are encouraged to download our source code and do their own tests.

Tables are sorted by descending SSIM index, and for each technique the rows with higher SSIM index and with the highest frame rate are highlighted. For each scene, we show an image with the ray-traced render and the best quality screenshot, according to SSIM, for each of the two techniques.

For saSSAO, we use the area of circumscribed circle as area approximation, and hemisphere sampling. For Alchemy, we use flat sampling because it looks working definitely better. Other parameters are adjusted by hand trying to have good results or to show some particular behaviour.

Here's a list of the test scenes, followed by data tables and images.

- Lion head close up
  - max distance 0.5, angle bias 0.3: table 6.1, figure 6.3
  - max distance 1.0, angle bias 0.3: table 6.2, figure 6.4
- Lion head and drapes
  - max distance 0.25, angle bias 0.6: table 6.3, figure 6.5
  - max distance 1.5, angle bias 0.4: table 6.4, figure 6.6
- Atrium (from top)
  - max distance 0.8, angle bias 0.3: table 6.5, figure 6.7
- Atrium
  - max distance 0.8, angle bias 0.3: table 6.6, figure 6.8

Table 6.1: Lion head close up - max distance 0.5, angle bias 0.3

			shared parameters			saSSAO	Alchemy		
SSIM	tech	fps	radius	samples	aoMul	tDivs	k	\rho	u
92,80%	**S**	**35,90**	**0,4**	**16**	**1**	**4**			
92,71%	A	35,75	0,4	16	1	3			
92,71%	**A**	**22,26**	**0,5**	**64**	**1**		**1**	**0,20**	**0,01**
92,69%	S	21,79	0,4	32	1	4			
92,66%	A	34,08	0,5	32	1		1	0,20	0,01
92,35%	S	12,65	0,4	64	1	4			
92,19%	A	38,71	0,4	32	1		1	0,20	0,01
92,17%	A	23,17	0,4	64	1		1	0,20	0,01
92,15%	S	22,57	0,4	32	1	3			
91,78%	A	60,45	0,5	16	1		1	0,20	0,01
91,77%	***A***	***62,55***	***0,4***	***16***	***1***		***1***	***0,20***	***0,01***
91,13%	S	13,54	0,4	64	1	3			
90,59%	***S***	***51,44***	***0,4***	***16***	***1***	***2***			
88,68%	S	25,04	0,4	32	1	2			
85,94%	S	14,69	0,4	64	1	2			
84,58%	A	24,76	0,4	64	1		1	0,05	0,01
84,07%	A	42,06	0,4	32	1		1	0,05	0,01
82,77%	A	55,37	0,4	16	1		1	0,05	0,01

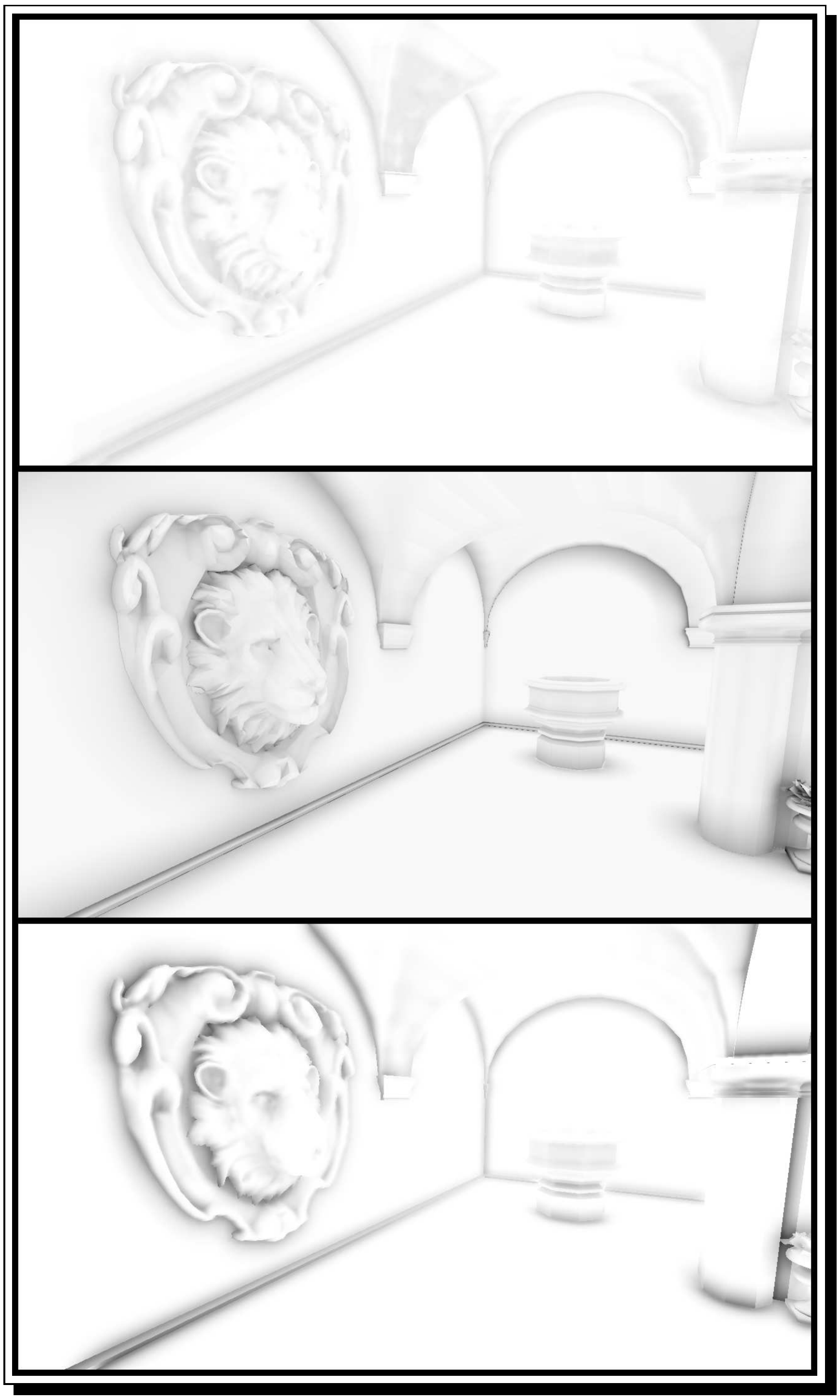


Figure 6.3: Lion head close up - max distance 0.5, angle bias 0.3. Top: saSSAO, SSIM 92,80%, 35,90 fps - bottom: Alchemy, SSIM 92,71%, 22.26 fps

Table 6.2: Lion head close up - max distance 1.0, angle bias 0.3

			shared parameters			saSSAO	Alchemy		
SSIM	tech	fps	radius	samples	aoMul	tDivs	k	rho	u
92,95%	**A**	**22,96**	**1**	**64**	**1**		**1**	**0,20**	**0,01**
92,47%	A	41,33	1	32	1		1	0,20	0,01
91,87%	**S**	**22,86**	**1**	**32**	**1**	**4**			
91,70%	***A***	***85,76***	***1***	***16***	***1***		***1***	***0,20***	***0,01***
91,70%	S	12,25	1	64	1	4			
91,62%	S	46,00	1	16	1	3			
91,54%	S	25,82	1	32	1	3			
91,24%	S	37,24	1	16	1	4			
89,71%	S	12,76	1	64	1	3			
89,57%	***S***	***84,70***	***1***	***8***	***1***	***2***			
88,97%	S	43,27	1	16	1	2			
87,60%	S	24,91	1	32	1	2			
85,43%	S	13,76	1	64	1	2			

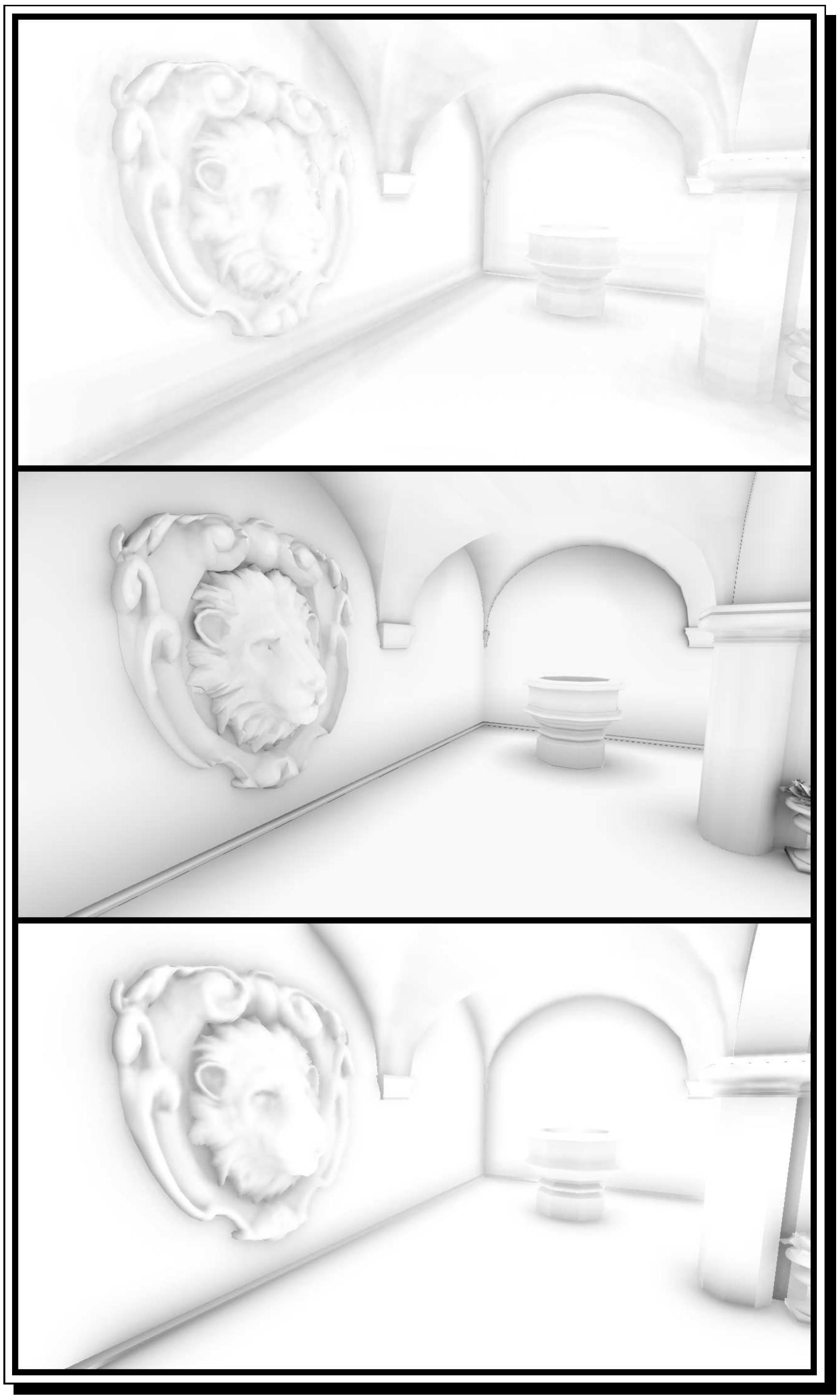


Figure 6.4: Lion head close up - max distance 1.0, angle bias 0.3. Top: saSSAO, SSIM 91,87%, 22,86 fps - bottom: Alchemy, SSIM 92,95%, 22.26 fps

Table 6.3: Lion head and drapes - max distance 0.25, angle bias 0.6

			shared parameters			saSSAO	Alchemy		
SSIM	tech	fps	radius	samples	aoMul	tDivs	k	rho	u
92,64%	**S**	**14,32**	**0,2**	**64**	**1**	**3**			
92,50%	S	13,36	0,2	64	1	4			
92,33%	S	30,63	0,2	32	1	2			
92,31%	S	48,45	0,2	16	1	2			
92,29%	S	29,50	0,2	32	1	3			
92,09%	S	25,06	0,2	32	1	4			
91,90%	S	47,19	0,2	16	1	3			
91,85%	S	25,25	0,2	32	2	4			
91,69%	S	43,23	0,2	16	2	4			
91,29%	S	27,97	0,2	32	2	3			
91,28%	S	37,43	0,2	16	1	4			
91,26%	S	13,81	0,2	64	2	4			
91,07%	***S***	***55,41***	***0,2***	***16***	***2***	***2***			
90,70%	S	15,26	0,2	64	2	3			
90,31%	S	49,61	0,2	16	2	3			
90,28%	S	16,09	0,2	64	1	2			
88,32%	S	30,21	0,2	32	2	2			
87,92%	***A***	***86,35***	***0,2***	***16***	***1***		***1***	***0,2***	***0,001***
87,91%	A	24,75	0,2	64	1		1	0,2	0,001
87,89%	A	45,45	0,2	32	1		1	0,2	0,001
83,50%	S	15,54	0,2	64	2	2			

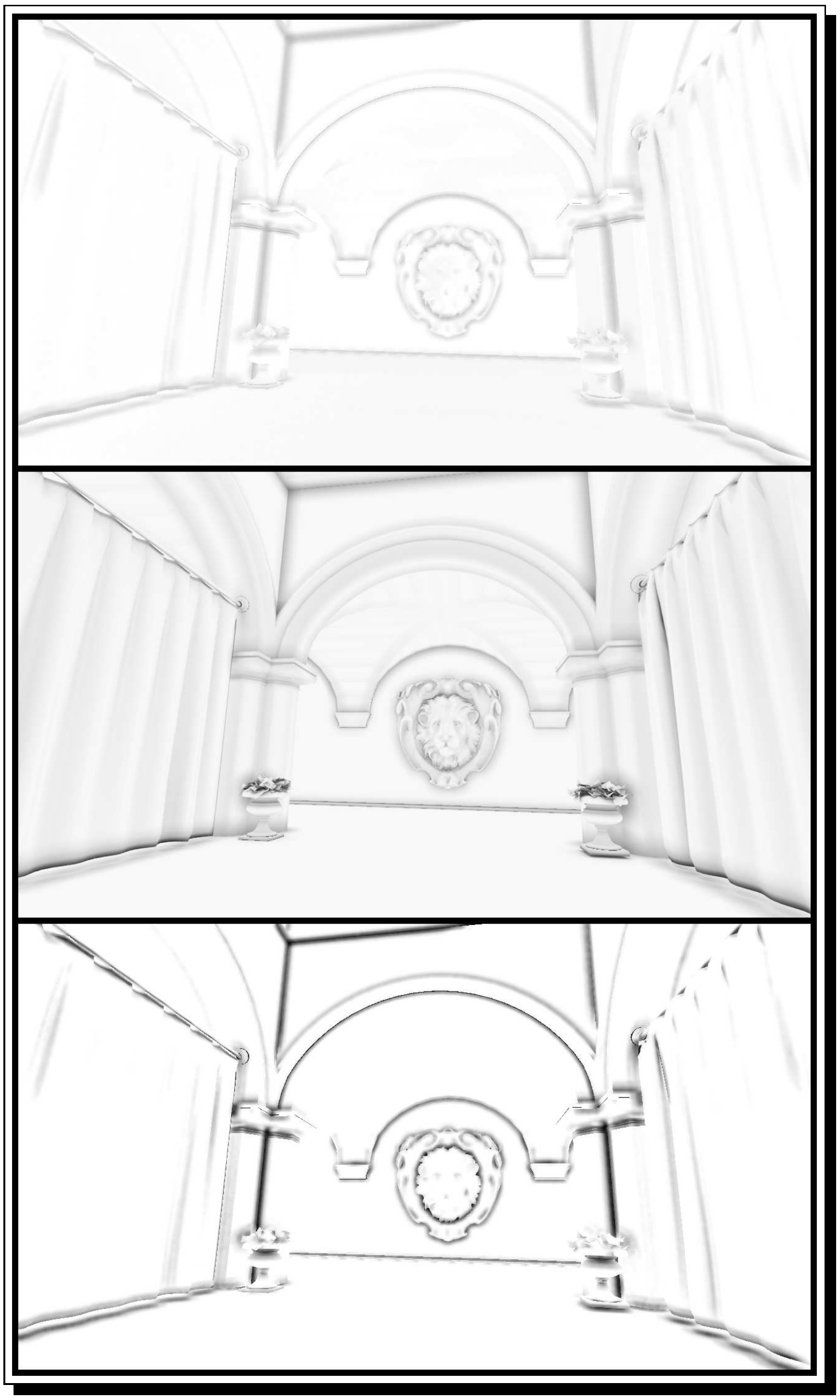


Figure 6.5: Lion head and drapes - max distance 0.25, angle bias 0.6. Top: saSSAO, SSIM 92,64%, 14,32 fps - bottom: Alchemy, SSIM 91,07%, 55.41 fps

Table 6.4: Lion head and drapes - max distance 1.5, angle bias 0.4

			shared parameters			saSSAO	Alchemy		
SSIM	tech	fps	radius	samples	aoMul	tDivs	k	rho	u
90,46%	**S**	**11,53**	**1,5**	**64**	**1**	**4**			
89,95%	S	37,97	1,5	16	1	3			
89,90%	S	20,26	1,5	32	1	4			
89,77%	S	12,61	1,5	64	1	3			
89,76%	S	24,42	1,5	32	1	3			
89,75%	S	21,55	1,5	32	1,5	4			
89,73%	S	38,41	1,5	16	1,5	4			
89,72%	S	11,39	1,5	64	1,5	4			
89,68%	S	36,76	1,5	16	1	4			
89,49%	S	38,26	1,5	16	2	4			
89,01%	***S***	***43,25***	***1,5***	***16***	***1***	***2***			
86,87%	S	24,16	1,5	32	1	2			
86,75%	S	12,93	1,5	64	1	2			
84,81%	***A***	***83,19***	***1,5***	***16***	***3***		***1***	***0,2***	***0,001***
79,01%	A	38,76	1,5	32	3		1	0,2	0,001
77,59%	A	21,53	1,5	64	3		1	0,2	0,001

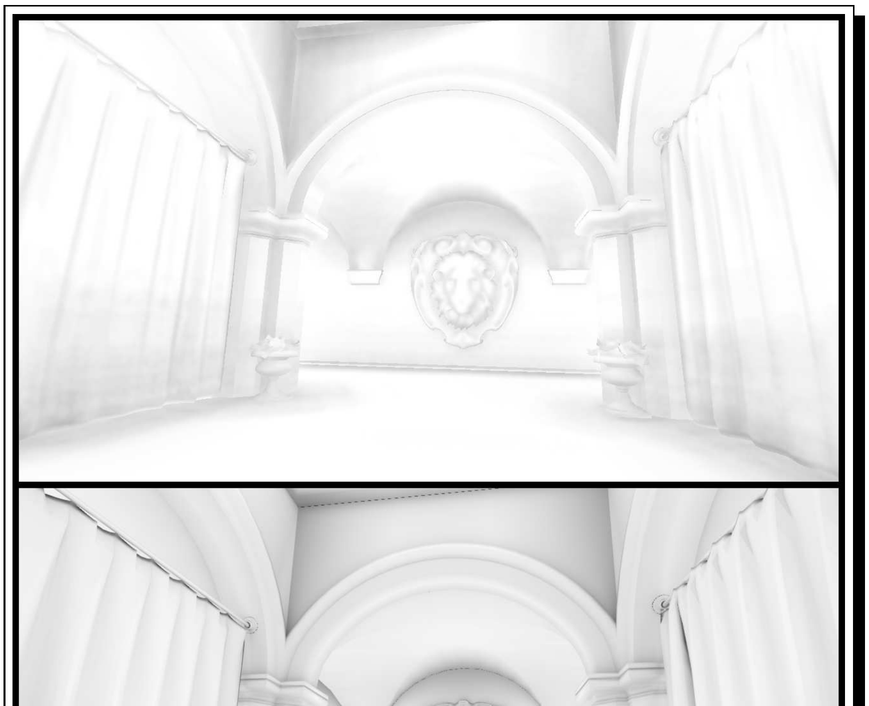


Figure 6.6: Lion head and drapes - max distance 1.5, angle bias 0.4. Top: saSSAO, SSIM 90,46%, 11,53 fps - bottom: Alchemy, SSIM 84,81%, 83.19 fps

Table 6.5: Atrium (from top) - distance 0.8, angle bias 0.3

			shared parameters			saSSAO	Alchemy		
SSIM	tech	fps	radius	samples	aoMul	tDivs	k	rho	u
85,51%	**S**	**35,59**	**0,8**	**16**	**3**	**4**			
84,62%	S	22,97	0,8	32	3	4			
82,29%	***S***	***44,78***	***0,8***	***16***	***3***	***3***			
81,16%	***A***	***84,10***	***0,8***	***16***	***3***		***1***	***0,15***	***0,01***
81,15%	S	25,49	0,8	32	3	3			
81,04%	S	12,10	0,8	64	3	4			
78,33%	A	43,35	0,8	32	3		1	0,15	0,01
77,09%	A	22,91	0,8	64	3		1	0,15	0,01
76,88%	S	42,18	0,8	16	3	2			
75,86%	S	12,95	0,8	64	3	3			
70,89%	S	26,20	0,8	32	3	2			
66,34%	S	13,24	0,8	64	3	2			
62,37%	A	83,90	0,8	16	3		1	0,5	0,01
59,09%	A	43,46	0,8	32	3		1	0,5	0,01
58,65%	A	23,78	0,8	64	3		1	0,5	0,01

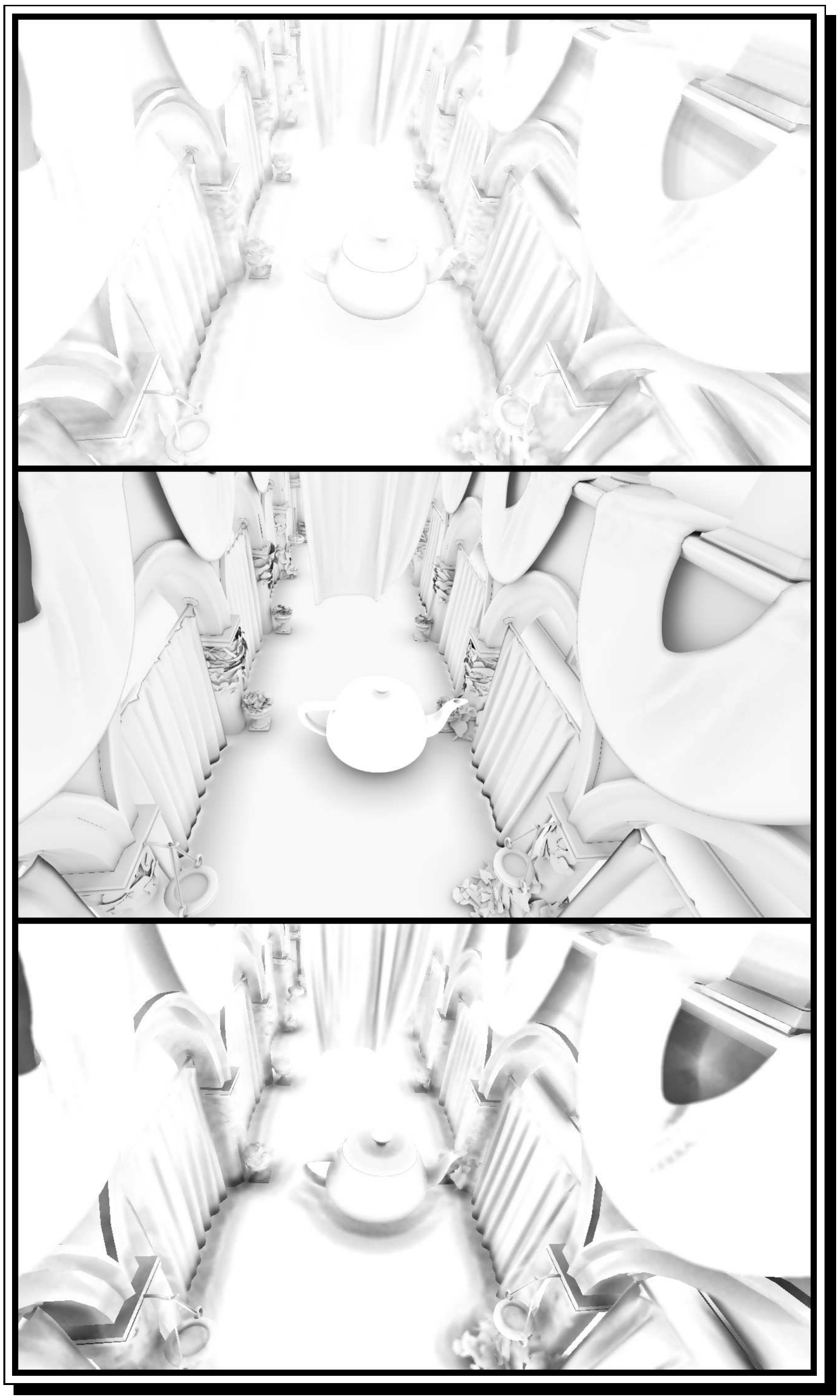


Figure 6.7: Atrium (from top) - max distance 0.8, angle bias 0.3. Top: saSSAO, SSIM 85,51%, 35,59 fps - bottom: Alchemy, SSIM 81,16%, 84.10 fps

Table 6.6: Atrium - distance 0.8, angle bias 0.3

			shared parameters			saSSAO	Alchemy		
SSIM	tech	fps	radius	samples	aoMul	tDivs	k	rho	u
87,77%	**S**	**13,28**	**0,5**	**64**	**1**	**4**			
87,73%	S	23,94	0,5	32	1	4			
87,49%	S	27,04	0,5	32	1	3			
87,42%	S	41,97	0,5	16	1	3			
86,91%	***A***	***84,52***	***0,5***	***16***	***1***		***1***	***0,2***	***0,01***
86,88%	S	13,65	0,5	64	1	3			
86,82%	A	44,08	0,5	32	1		1	0,2	0,01
86,79%	A	24,31	0,5	64	1		1	0,2	0,01
86,72%	S	38,81	0,5	16	1	4			
86,50%	***S***	***52,20***	***0,5***	***16***	***1***	***2***			
85,49%	S	34,43	0,5	16	3	4			
85,49%	S	39,91	0,5	16	3	4			
85,45%	S	29,51	0,5	32	1	2			
84,66%	S	23,98	0,5	32	3	4			
83,96%	S	14,46	0,5	64	1	2			
81,07%	S	12,54	0,5	64	3	4			
81,07%	S	13,38	0,5	64	3	4			
77,17%	A	79,22	0,5	16	1		1	0,5	0,001
75,39%	A	39,77	0,5	32	1		1	0,5	0,001
74,92%	A	24,09	0,5	64	1		1	0,5	0,001

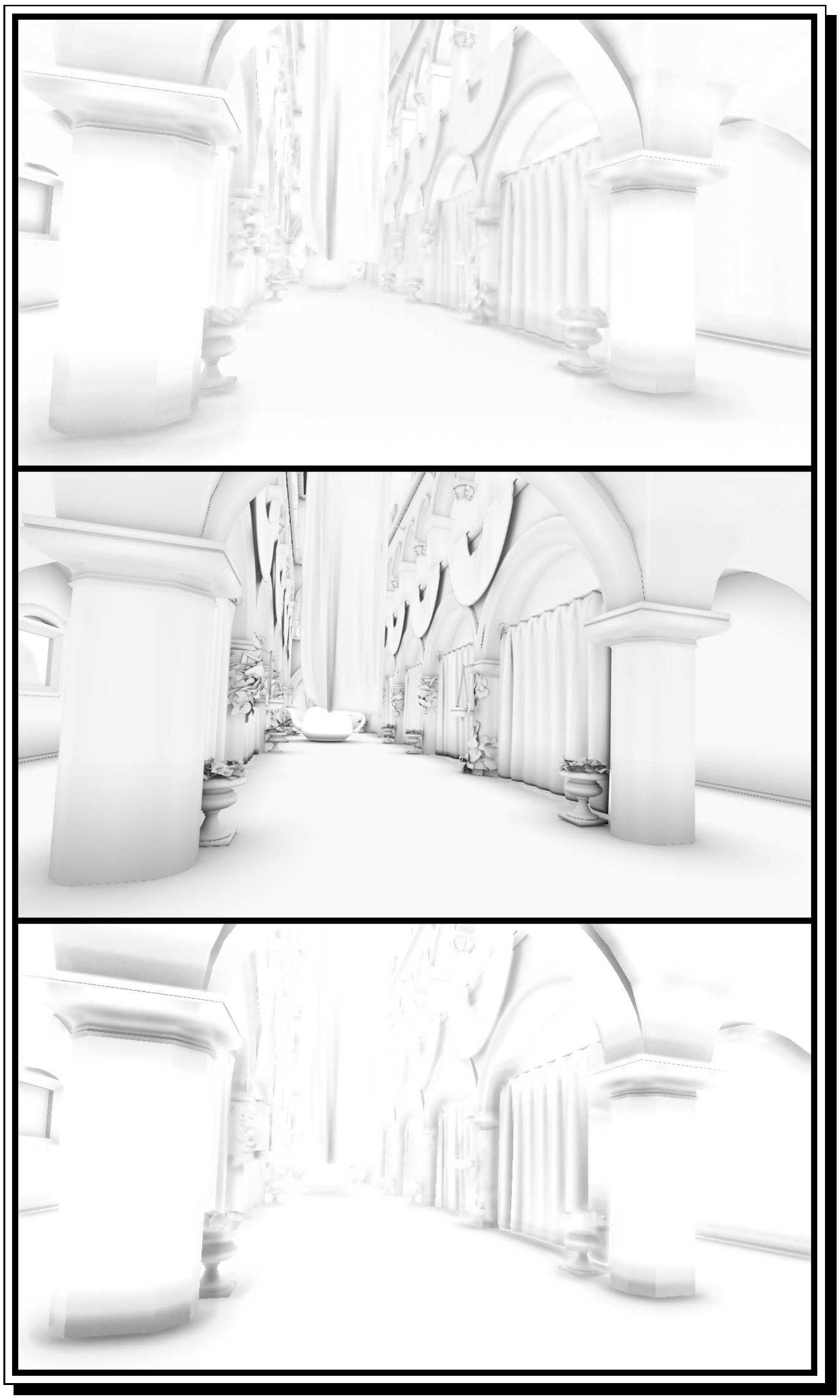


Figure 6.8: Atrium - max distance 0.8, angle bias 0.3. Top: saSSAO, SSIM 87,77%, 13,28 fps - bottom: Alchemy, SSIM 86,91%, 84.52 fps

Chapter 7

Conclusions

We have developed a new technique in the field of Image-Space Ambient Obscurance.

The general approach, based on deferred rendering and G-buffer sampling, is common in the literature and shared by a number of algorithms in the field.

Our original contribution consists in the use of a geometry shader to approximate the area of occlusors, and in the adoption of a hemisphere discretization technique aimed at classifying the occlusors according to their position. To estimate coverage, we use a solid-angle approximation derived from our experiments with other algorithms and from some observations related to the lack of data inherent to image-based algorithms.

This kind of approach, where we evaluate the level of occlusion considering the direction from which coverage comes from, storing such result in our “triangle buckets”, allows to avoid over-occlusion and should generally give an improved quality result thanks to the implicit weighting of samples contributions.

Quality is in fact our primary concern, and we evaluate our results by comparing the structural similarity with off-line rendered images calculated through ray-tracing in Blender.

The results look encouraging, even if accurately evaluating the validity and efficiency of the technique against others is quite complicated. Many parameters are involved, and even if the source code of some other techniques is available, comparisons are all but straightforward: different algorithms, in many cases, don't use the same parameters, and just minor adjustments can dramatically change the quality of results and performance. Also, there can be some scene dependency, causing some technique to perform better than others only in some scenes.

We worked hard on having the fundamental parts of the implementation easily customizable and swappable, even at runtime, to rapidly test different formulas, rendering pipelines or sampling techniques. Many attempts that didn't bring to particularly good results have been omitted from this thesis for brevity and time constraints.

We spent a substantial amount of time on the implementation, mainly because we had decided to build a rendering sandbox from the ground up. This was interesting and gave us total freedom, but in terms of comparing techniques against each other, it would be nice if a single rendering infrastructure would become popular in the computer graphics community, allowing everyone working in the field to fairly compare techniques or quickly extends other algorithms when source code is provided. The G3D engine looks like a promising platform in this perspective, and in hindsight using it as a basis for our implementation could have been a wiser choice. It would be interesting to port the technique to such engine and see if it gives better performances.

7.1 Future work

Ambient obscurance and, generically, global illumination approximations (for real-time rendering) are constantly improving, following the evolution of hardware, APIs and literature.

We share some ideas that we couldn't explore due to time constraints, and give some references about other techniques approaching real-time global illumination from other perspectives.

7.1.1 Directional occlusion, bent normals

Our pyramid of "triangle buckets" doesn't only tell how much a point is covered, but also from where (with customizable level of precision). This could maybe exploited to achieve other kinds of results, such as "directional occlusion" [RGS09], and could be used to calculate direct lighting using "bent normals", normals adjusted to consider the direction from which more incoming light will potentially reach the surface (that is, where there are no occlusors).

7.1.2 Indirect lighting

Regarding indirect lighting calculation, we just scratched the surface of the problem. Anyway, good far-field approximation of indirect light are impossible to achieve with a basic image-space approach, for the lack of crucial information due to non-visible areas.

Some work to address this kind of limitation has been recently shown in [MML], where a "deep G-buffer" is used to keep a second layer of geometry-related information, and used as a basis for a screen-space radiosity approximation.

Alternatively, interesting results come from a drastically different approaches that could maybe be complemented by image-space techniques, as is for example suggested in [RGS09]. We list a few techniques we think are particularly interesting:

- **Cascaded Light Propagation Volumes** [Kap09]: implemented in CryEngine3, works by calculating indirect low frequency light using a 3D grid and spherical harmonics
- **Voxel Cone Tracing** [CNS+11]: uses a real-time mesh voxalization and octree building technique to obtain a discretization of the scene suitable to indirect light calculation
- **Imperfect Shadow Maps** [RGK+08] and their proposed refinements: they are themselves an improvement on the idea of **Reflective Shadow Maps** [DS05], that build upon on the shadow mapping technique, rendering the scene from the light sources positions and storing lighting information in the resulting buffers (instead of the depth used in shadow mapping)

7.1.3 Compute shaders

Compute shaders, added in OpenGL 4.3, add another level of freedom in the utilization of the GPU (in the direction of other general purpose APIs that exploit GPUs and other parallel hardware, such as OpenCL and CUDA).

Compute shaders operate differently from other shader stages: for example, they have no well-defined set of input values and no frequency of execution specified by the nature of the stage (once per vertex, once per fragment...).

The added level of flexibility also adds, as always, a degree of complexity, but that could be well worth facing if significant performance improvements can be achieved. The first thing that comes to mind is a more efficient sampling: our technique does a number of texture fetch operations (particularly relevant to the efficiency of the technique) for each fragment. We get data and can use it to calculate the obscurance value for the current fragment, that is the only output that we can write to... then, maybe, for a close fragment, we fetch again the same data. What if we could cache them in some way, for example using the highly efficient GPU shared memory?

Very interesting results in the SSAO field, based on CUDA implementations, recently appeared in [Tim] and [Tim13].

7.1.4 Testing methodology and temporal coherence

A problem of many SSAO techniques, often overlooked, is their behaviour with a moving camera: with the constantly changing scene information that becomes available in image-space (due to the different viewing positions), the obscurance calculation for the same area can vary depending on where we are looking from. Areas suddenly getting darker or brighter with no reason while moving into a scene can definitely ruin the immersion, especially in the absence of a good direct lighting.

Our testing infrastructure could easily be extended to produce video sequences: the camera fly-through feature could save the camera position and orientation for each rendered frame. Such data could then used to generate sequences of both YARS and Blender rendered images to be automatically transformed into video files. Of course the SSIM index could be evaluated for each couple of images and some global statistics could be derived, but more interestingly, some measure of the obscurance change artefacts could be derived from the comparison with a window of previous frames. That could be a basis to design an algorithm to detect and avoid such artefacts when calculating the obscurance, exploiting information from previous frames kept available (similarly to what is done in motion blur implementations).

Bibliography

[Aal] Frederik P. Aalund. A comparative study of screen-space ambient occlusion methods. 3, 3.4

[AMHH11] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. Real-time rendering. 2011. 4

[Bli77] James F Blinn. Models of light reflection for computer synthesized pictures. In *ACM SIGGRAPH Computer Graphics*, volume 11, pages 192–198. ACM, 1977. 2.4

[BSD08] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 talks*, page 22. ACM, 2008. 3.5

[Bun05] Michael Bunnell. Dynamic ambient occlusion and indirect lighting. *Gpu gems*, 2(2):223–233, 2005. 3.1, 4.2.2, 5.6

[CNS$^+$11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, volume 30, pages 1921–1930. Blackwell Publishing Ltd, 2011. 7.1.2

[Dog12] Michael Doggett. Texture caches. *Micro, IEEE*, 32(3):136–141, 2012. 5.5.5

[DS05] Carsten Dachsbacher and Marc Stamminger. Reflective shadow maps. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 203–231. ACM, 2005. 7.1.2

[EHF$^+$] Kevin Egan, Daniel J Hilferty, US Air Force, Daniel F Keefe, Morgan McGuire, Casey O'Donnell, Peter G Sibley, Corey Taylor, Electronic Arts, and Tom Wardill. The g3d engine as platform for research and education. 1.2.1

[FM08] Dominic Filion and Rob McNaughton. Effects & techniques. In *ACM SIGGRAPH 2008 Games*, pages 133–164. ACM, 2008. 3.4

[Gra13] Lorents Odin Gravås. Image-space ambient obscurance in webgl. 2013. 3, 4.2.3

[Kap09] Anton Kaplanyan. Light propagation volumes in cryengine 3. *ACM SIGGRAPH Courses*, 2009. 7.1.2

[KSMY07] Pankaj Khanna, Mel Slater, Jesper Mortensen, and Insu Yu. A non-parametric guide for radiance sampling in global illumination. *Computer Graphics, Imaging and Visualisation, 2007. CGIV'07*, pages 41–48, 2007. 4.1, 1, 2

[Lam60] Jean-Henri Lambert. *JH Lambert,... Photometria, sive de Mensura et gradibus luminis, colorum et umbrae.* sumptibus viduae E. Klett, 1760. 2.4.2

[LB00] Michael S Langer and Heinrich H Bülthoff. Depth discrimination from shading under diffuse lighting. *Perception*, 29(6):649–660, 2000. 2.2

[Mit07] Martin Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*, pages 97–121. ACM, 2007. 3.3

[MML] Michael Mara, Morgan McGuire, and David Luebke. Lighting deep g-buffers: Single-pass, layered depth images with minimum separation applied to indirect illumination. 7.1.2

[MML12] Morgan McGuire, Michael Mara, and David Luebke. Scalable ambient obscurance. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 97–103. Eurographics Association, 2012. 3.6

[MOBH11] Morgan McGuire, Brian Osman, Michael Bukowski, and Padraic Hennessy. The alchemy screen-space ambient obscurance algorithm. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 25–32. ACM, 2011. 2.3, 3.6, 4.2.2

[PH10] Matt Pharr and Greg Humphreys. Physically based rendering: From theory to implementation. 2010. 2.1, 4.6

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975. 2.4

[RDGK12] Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz. The state of the art in interactive global illumination. In *Computer Graphics Forum*, volume 31, pages 160–188. Blackwell Publishing Ltd, 2012. 3

[RGK$^+$08] Tobias Ritschel, Thorsten Grosch, Min H Kim, H-P Seidel, Carsten Dachsbacher, and Jan Kautz. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Transactions on Graphics (TOG)*, 27(5):129, 2008. 7.1.2

[RGS09] Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 75–82. ACM, 2009. 5.6, 7.1.1, 7.1.2

[SA07] Perumaal Shanmugam and Okan Arikan. Hardware accelerated ambient occlusion techniques on gpus. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 73–80. ACM, 2007. 3.2

[SHR10] Cyril Soler, Olivier Hoel, and Frank Rochet. A deferred shading pipeline for real-time indirect illumination. In *ACM SIGGRAPH 2010 Talks*, page 18. ACM, 2010. 5.6

[Tim] Ville Timonen. Screen-space far-field ambient obscurance. 7.1.3

[Tim13] Ville Timonen. Line-sweep ambient obscurance. In *Computer Graphics Forum*, volume 32, pages 97–105. Blackwell Publishing Ltd, 2013. 7.1.3

[TL04] Eric Tabellion and Arnauld Lamorlette. An approximate global illumination system for computer generated films. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 469–476. ACM, 2004. 1.1.5

[WBSS04] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: From error visibility to structural similarity. *Image Processing, IEEE Transactions on*, 13(4):600–612, 2004. 1.2, 6.2

[ZIK98] Sergey Zhukov, Andrei Iones, and Grigorij Kronin. An ambient light illumination model. In *Rendering Techniques' 98*, pages 45–55. Springer, 1998. 2.3

Appendix A

Appendix

A.1 Additional material and code listings

Currently, as measured with sloccount, the YARS source code including the technique implementation is around 12k lines of C++, not counting the GLSL shaders. As we are more interested in saving trees than making the printed version of the thesis look thicker, we decided to not include the full source code here (that, anyway, would benefit some refactoring and clean-up before publishing).

A digital version of this thesis, related slides, full source code of the implementation and a prebuilt demo can be obtained browsing these URLs:

- `http://isis.dia.unisa.it/wiki/index.php?title=User:Darsca`
- `http://www.duskzone.it/works/unisa/masterThesis`
- `http://github.com/darioscarpa/YARS`