# Stochastic-Depth Ambient Occlusion

Jop Vermeer

Technische Universiteit Delft

TU Delft

# Stochastic-Depth Ambient Occlusion

by

Jop Vermeer

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday September 24, 2020 at 14:00.

| | | |
|---|---|---|
| Student number: | 4462734 | |
| Project duration: | December 2, 2019 – September 24, 2020 | |
| Thesis committee: | Prof. dr. Elmar Eisemann, | TU Delft, supervisor |
| | Dr. ir. Rafael Bidarra, | TU Delft |
| | Dr. Julián Urbano, | TU Delft |
| | Dr. Leonardo Scandolo, | TU Delft |

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

# Preface

These past couple of months have been quite an adventure, while the whole world is going in lockdown due to the ongoing COVID-19 pandemic, I have been working steadily on this master thesis project. This work marks the end of my two-year journey as master student at the Delft University of Technology, during which I've learned a lot and worked on many interesting projects. While it was difficult at times, I'm glad that I have done it.

This master thesis project has been performed by Jop Vermeer at the Delft University of Technology, under the supervision of prof. dr. Elmar Eisemann and dr. Leonardo Scandolo. This project aims to improve upon traditional screen-space ambient occlusion techniques, which miss information about geometry occluded in screen space. Stochastic-depth ambient occlusion uses a stochastic transparency based approach to efficiently gather information about the geometry at multiple depth layers, allowing for more accurate results, while still providing real-time performance.

First of all, I would like to thank prof. dr. Elmar Eisemann, not only for his guidance and support during this project, but also for him inspiring me to do my thesis in the field of computer graphics. After following his course *3D Computer Graphics and Animation*, where, together with my good friends Shaad Alaka and Max Lopes Cunha, I created a 3D game engine/renderer in OpenGL, I knew I wanted to a thesis project in this field. I would also like to thank dr. Leonardo Scandolo for his excellent help and support throughout this project. He helped me get started with the MxEngine, provided valuable feedback on my work and helped me out whenever I faced any difficulties.

Furthermore, I would like to thank dr. ir. Rafael Bidarra and dr. Julián Urbano for being part of my thesis committee. I have worked together with and/or under the supervision of dr. ir. Rafael Bidarra many times throughout both my bachelor's and master's degree studies at the Delft University of Technology, and I would like to thank him for his support and the opportunities he gave me.

I would also like to thank my friends, in particular Shaad Alaka, Max Lopes Cunha and Nico Arjen Miedema. We have worked together many times during our studies and motivated each other to perform at our best. We inspired each other to put in just that little bit more effort to accomplish things that we can be proud of.

Last, but certainly not least, I would like to thank my parents and my family for their unconditional love and support. During these tough two years, they supported me however they could and motivated me to keep going.

*Jop Vermeer*
*Delft, September 2020*

# Abstract

Ambient occlusion is a popular rendering technique that creates a greater sense of depth and realism, by darkening places in the scene that are less exposed to ambient light (e.g., corners and creases). Ambient occlusion measures how geometrically occluded each point in the scene is and modulates the ambient light accordingly. In real-time applications, screen-space ambient occlusion approximations are used, due to their great performance and visual quality. However, these screen-space approximations only take the geometry currently visible on screen into account. This results in underestimated or missing ambient occlusion when geometry is hidden from view (e.g., hidden behind other geometry). Our proposal, stochastic-depth ambient occlusion, improves upon traditional screen-space ambient occlusion techniques by including information about geometry at multiple depth layers, using a stochastic transparency based approach. This allows us to efficiently gather the missing information in order to improve upon the accuracy and spatial stability of conventional screen-space approximations, while still maintaining the real-time performance. Our approach integrates well into existing rendering pipelines and we show how it can be generalized to work with different screen-space ambient occlusion techniques and extensions (such as bent normals and cones). We also demonstrate how multi-view stochastic-depth ambient occlusion can greatly improve upon the robustness of traditional multi-view ambient occlusion techniques. Furthermore, we provide an extensive analysis of the visual quality, robustness and performance of stochastic-depth ambient occlusion compared to conventional screen-space approaches.

***Keywords*** **–** Computer Graphics, Real-time Rendering, Screen-Space Ambient Occlusion, Stochastic Transparency

# Contents

# 1 Introduction

Light is a rather complex phenomenon. In 3D graphics, light is often modeled as rays, moving from a light source to the scene and eventually reaching the camera. When a light ray hits a surface, it reflects or scatters into new directions. As light bounces around in the scene, it illuminates objects not immediately visible from the light source. Simulating these light bounces is computationally very expensive, and is therefore often approximated via ambient light. Ambient light is an omni-directional light source with a fixed color and intensity, affecting all objects in the scene equally. It is light that is assumed to hit each point on a surface from all directions equally. In the real world however, light coming from certain directions may be blocked, due to occlusion from the surrounding geometry. In order to make this ambient lighting approximation more realistic, ambient occlusion (AO) is used. Ambient occlusion is a rendering technique used to determine how exposed each point in the scene is to ambient light. Ambient occlusion determines how geometrically occluded a point in the scene is and modulates the ambient lighting accordingly (typically darkening creases, holes, intersections etc.). This adds a greater sense of depth and realism to the image (Figure 1.1).

Ambient occlusion can be computed using ray-tracing, but this is often not feasible in real-time applications, due to the high runtime. Instead, very fast screen-space approximations are used, such as screen-space ambient occlusion (SSAO) or horizon-based ambient occlusion (HBAO). These screen-space techniques approximate the ambient occlusion using only the geometry shown on screen (i.e., the depth information stored in the depth buffer).These techniques are very popular, as they can deliver similar visual quality as their ray-traced counterpart at only a fraction of the rendering cost. While this screen-space approach greatly improves performance, these techniques lack information regarding geometry hidden from view, resulting in underestimated or missing ambient occlusion (Figure 1.2). The results of these screen-space techniques are also view-dependent, which can lead to spatial instability and flickering when moving the camera around.

Stochastic-depth ambient occlusion (SDAO) aims to improves upon these screen-space techniques by gathering and using information about the geometry hidden from view. It combines traditional screen-space ambient occlusion with transparency techniques, with the intuition that transparency techniques allow us to "see through" any obstructing geometry. Transparency techniques often build a representation of geometry at multiple depth layers that all contribute to a pixel's color. We use a stochastic transparency based approach to gather the depth information efficiently, using multi-sample anti-aliasing (MSAA). In this approach, we render the opaque scene as if it is transparent and store the depth information in a multi-sampled depth buffer.

We modify the traditional screen-space ambient occlusion algorithms to operate on this multi-sampled depth buffer, in order to more accurately compute the ambient occlusion. Stochastic-depth ambient occlusion improves the visual quality and spatial stability of conventional screen-space ambient occlusion techniques, while maintaining their real-time performance. Stochastic-depth


(a) Without ambient occlusion.


(b) With ambient occlusion (HBAO+).

(c) Only ambient occlusion.

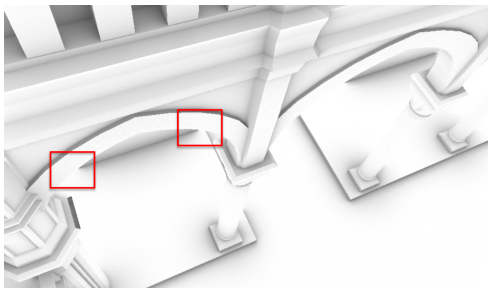
Figure 1.1: Screenshots of the game Assassin's Creed Syndicate with ambient occlusion on and off. The ambient light contribution for each pixel is multiplied with the corresponding ambient occlusion value (computed using the HBAO+ algorithm), greatly enhancing the sense of depth and realism in the image.

ambient occlusion is robust and works well in most scenes, and it integrates well into existing rendering pipelines. Furthermore, we show how extensions to screen-space ambient occlusion techniques, such as bent normals and cones, generalize to stochastic-depth ambient occlusion, with improved visual quality. Finally, we demonstrate how stochastic-depth ambient occlusion improves upon the robustness compared to traditional techniques when computing the ambient occlusion from multiple viewpoints (i.e., from multiple cameras), removing the need for manual, per-scene camera placement.

In summary, our contributions include:

- An efficient and robust stochastic transparency based approach to gather and store information about multiple depth layers, that integrates well into a rasterization pipeline.
- Modified stochastic-depth versions of popular screen-space ambient occlusion techniques (i.e., stochastic-depth SSAO, HBAO and HBAO+).

(a) Ray-traced ambient occlusion.



(b) HBAO.

Figure 1.2: HBAO underestimates/misses the ambient occlusion compared to the ray-traced ground truth (in the marked regions), since it lacks information about the geometry hidden from view (i.e., the obstructed geometry). Images from [7].

- An implementation of bent normals and bent cones with our stochastic-depth ambient occlusion algorithms.
- A multi-view implementation of stochastic-depth ambient occlusion, that uses a secondary camera to further improve the visual quality and spatial stability.
- An extensive analysis of the visual quality, robustness and performance of the proposed methods compared to traditional techniques.

# 2

# Background Theory

This chapter explains the background theory necessary to discuss and reason about the various screen-space ambient occlusion algorithms and our proposed solution, stochastic-depth ambient occlusion. First, we discuss rasterization and virtual cameras, which form the core of real-time 3D graphics. After this we explain what ambient lighting and ambient occlusion is, and discuss the pitfalls of contemporary techniques, which we aim to solve with stochastic-depth ambient occlusion. Finally, since stochastic-depth ambient occlusion combines screen-space ambient occlusion with transparency techniques, we will discuss how transparency is handled in a rasterization pipeline.

## 2.1. Rasterization and Virtual Cameras

In 3D computer graphics, objects (often called models) are represented as a collection of triangles. These triangles are send to the *Graphics Processing Unit* (GPU), which projects the triangles onto the screen from the viewpoint of a virtual camera (Figure 2.1). After this, the GPU determines which pixels are covered by a triangle and colors these pixels according to a *shader* (a fully programmable step on modern GPUs). If multiple triangles cover the same pixel, it will be shaded based on the frontmost triangle (i.e., the triangle closest to the camera). The collection of data required to shade a pixel (e.g., the position, depth, interpolated vertex color, texture coordinates, etc.) is called a *fragment*. Each triangle and pixel can be projected and shaded independently. This process is called rasterization and allows modern GPUs to render high resolution images with many triangles in real time. Rasterization is the typical rendering technique, especially in real-time applications such as games.

The virtual cameras used in 3D graphics follow the pinhole camera model, where light from the scene passes through a very small hole (the *aperture*), showing a projection of the scene onto the image plane (Figure 2.2). The aperture is the center of the projection and represents the position of the virtual camera. In the real world, the image plane is behind the aperture and shows an inverted projection of the scene, but in the virtual world we can move this image plane in front of the aperture, such that the projected image is not inverted anymore. Furthermore, to represent depth in the infinite 3D space, we augment virtual cameras with a near and far clipping plane to form a *view frustum* (as seen in Figure 2.3). After the projections, depth ranges from the near plane to the to far plane (from $-1$ to 1) and only triangles inside this view frustum are drawn, the rest is discarded. With rasterization, we use the *projection matrix* to project triangles in this view frustum to the screen, in a process called *perspective projection*.

A virtual camera is described by a *view matrix* (which stores the position and orientation of the camera) and the projection matrix (which applies the perspective projection). These matrices are used to transform vertex coordinates to the different coordinate spaces (as shown in Figure 2.4). The

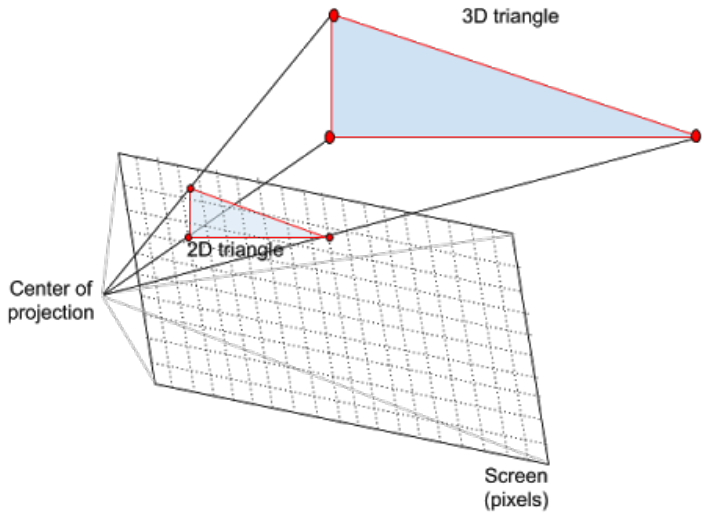
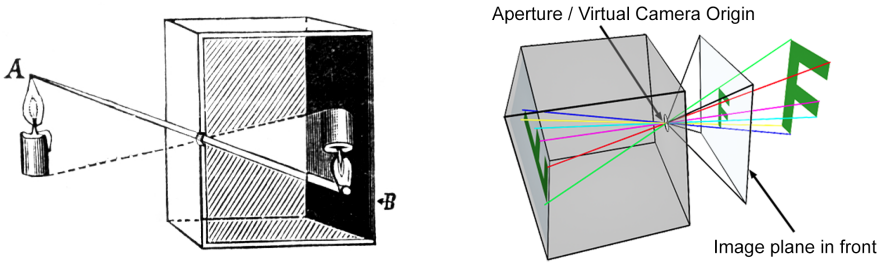



Figure 2.1: With rasterization, each triangle is projected onto the screen from the viewpoint of a virtual camera (the center of the projection). Then the color of each pixel covered by a triangle is determined using shading. Image from [49].




(a) Pinhole camera.

(b) Virtual camera.

Figure 2.2: A pinhole camera (or camera obscura) has a small hole (the aperture) where light passes through, projecting an inverse image of the scene onto its image plane. A virtual camera follows the same principle, but has its image plane in front of the aperture. This ensures that the projected image is not inverted. Images from [1].

*model matrix* describes how to translate, rotate and scale the individual models (all defined with their own local origin, i.e., in local space) to fit in the scene. We can transform these local space coordinates to *world space* (where the coordinates are all relative to some global origin) by multiplying them with the model matrix. The camera's view matrix is used to transform vertex coordinates from world space to *view space* (the scene from the camera's point of view, where the camera's position is the origin). Then using the projection matrix, we can transform these view space coordinates to *clip space* (which maps the view frustum to a cube, where the $X$, $Y$ and $Z$ coordinates range from $-1$ to 1). These clip space coordinates are transformed to *screen space* (the coordinates as actually shown on the screen), by dividing the homogeneous clip space coordinates by their $w$ component (converting them back to Cartesian coordinates) and scaling the $X$, $Y$ and $Z$ coordinates to range from 0 to 1. By applying the inverse operations we can also transform coordinates the other way around (e.g., using the inverse projection matrix, we can transform clip space coordinates back to view space). The algorithms described in this work, operate on screen-space coordinates and often transform these back to view space coordinates.

## 2.2. Ambient Lighting and Occlusion

Light is a complex phenomenon, often modeled as rays in 3D computer graphics. This means that light moves as rays from a light source into the scene, where it bounces around and eventually reaches

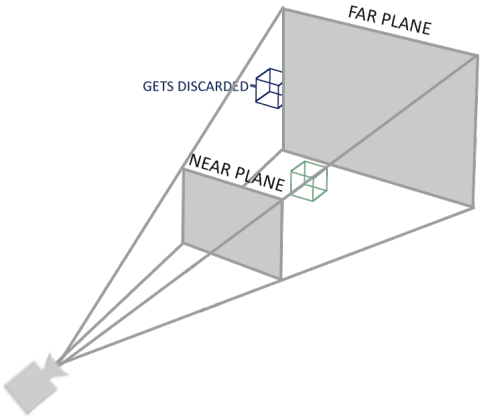



Figure 2.3: The view frustum with a near and far clipping plane. Triangles outside of this view frustum are discarded. Image from [14].

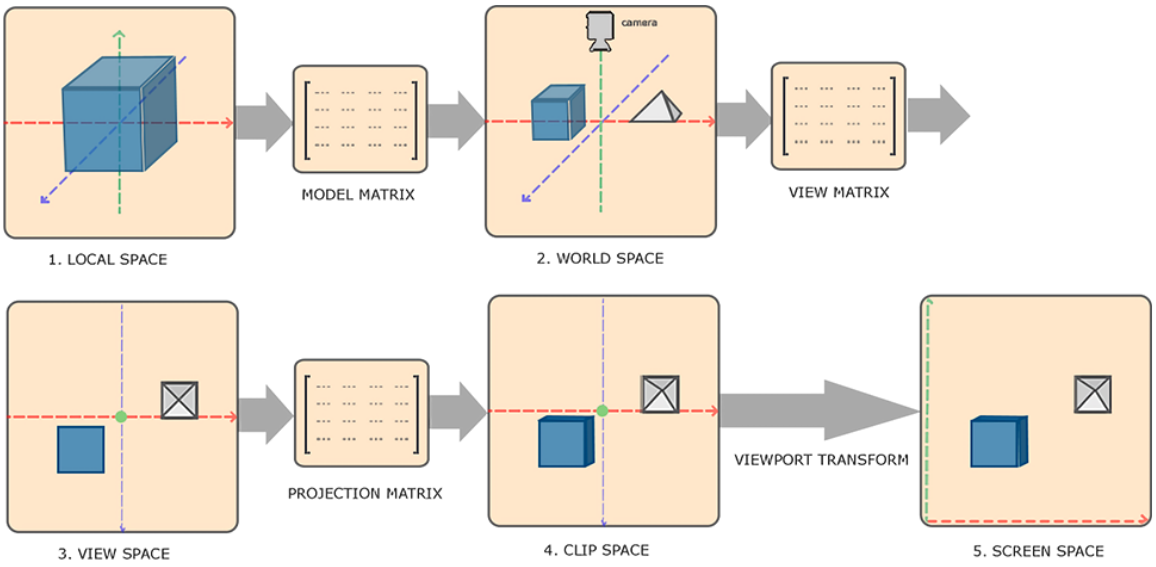
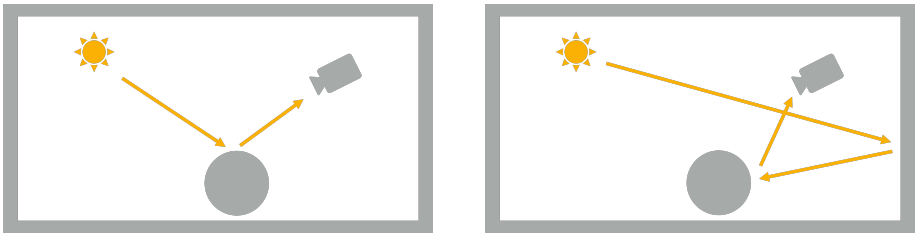



Figure 2.4: The different coordinate spaces used in 3D graphics and how to transform between them. Image from [14].

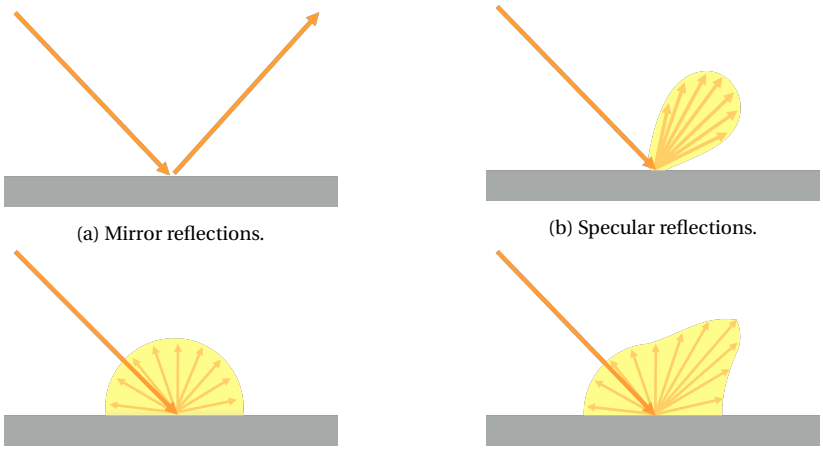
the camera. When a light ray hits a surface, it scatters and bounces in many different directions, allowing the light to reach places not immediately visible from the light source. The light that hits a surface directly from the light source is called *direct illumination* (Figure 2.5a), while the light reflected from other surfaces is called *indirect* or *global illumination* (Figure 2.5b). Light reflecting of smooth surfaces such as metal leads to *specular* reflections (Figure 2.6b), while rougher surfaces such as cloth reflect/scatter light in all directions leading to *diffuse* reflections (Figure 2.6c). In practice, depending on the material properties of a surface, part of the light is reflected specularly and part of the light is diffusely scattered in all directions (Figure 2.6d).

Direct illumination can be computed efficiently in a rasterization pipeline, as it only depends on the light source and fragment itself. With indirect illumination, each time a light ray hits a surface, it reflects/scatters into new directions. In a sense, these surfaces become a light source themselves, casting light on other fragments. This makes indirect illumination an expensive problem, due to its recursive nature which adds dependence between the triangles/fragments. There are some techniques that aim to compute or approximate the indirect lighting in real-time [15, 31, 46, 50], but



(a) Direct illumination. (b) Indirect illumination.

Figure 2.5: With direct illumination light rays bounce directly from the object into the camera, while with indirect illumination the light rays are first reflected by other surfaces.



(a) Mirror reflections. (b) Specular reflections.

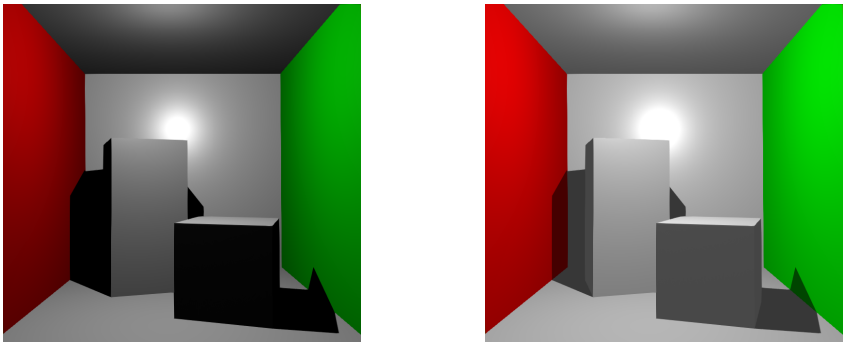
(c) Diffuse reflections. (d) Diffuse + specular reflections.

Figure 2.6: Smooth surfaces lead to mirror or specular reflections, where the incoming light rays are reflected into a single directions or a very narrow lobe. Rougher surfaces create diffuse reflections, where incoming light is reflected in all directions. In most cases, materials have both a diffuse and specular component.

these still remain expensive and/or have large limitations (e.g., only computing reflections for objects currently seen on screen). For this reason the diffuse indirect illumination is often approximated with a simple *ambient lighting term* (Figure 2.7). This ambient lighting term is simply a multiplication of the object's color and the ambient lighting intensity. The intensity is constant and independent of the position in the scene. This can be extended using an *environment map* (as seen in Figure 2.8), which is used to generate/pre-compute an *irradiance texture*, i.e., a texture storing for each direction the average color of the incoming light from the far-away environment. This irradiance texture can be queried with the surface normal, creating a more realistic approximation of the indirect light compared to just using the ambient lighting term.

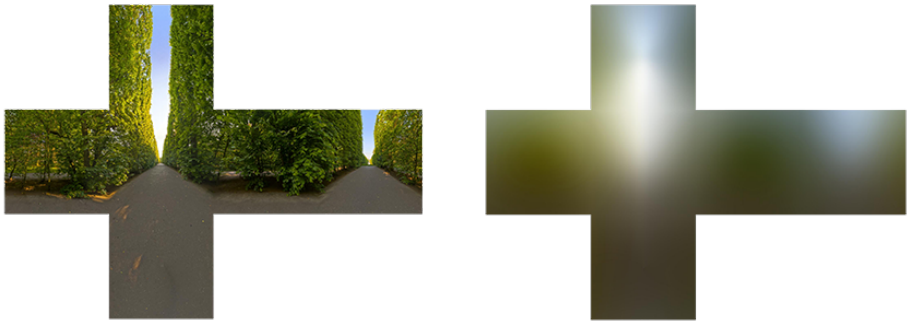
Of course, the indirect light should not be able reach and illuminate all places equally. In the real world, less light is be able reach corners or creases, since these places are obstructed by the geometry around it, blocking incoming light. This is where *ambient occlusion* (AO) comes in, which is a measure of how geometrically occluded a point in the scene is. Ambient occlusion is be used to modulate the ambient lighting term, based on how exposed each point in the scene is to indirect lighting. Ambient lighting and ambient occlusion are not physically accurate, instead they are clever tricks without a real physical basis, used to approximate the indirect lighting. But even though they are not physically accurate, they add a greater sense of depth and realism to the final image.

The ambient occlusion $A_p$ at point $p$ in the scene with normal $n$ is computed by casting a number of rays around a normal-oriented hemisphere $\Omega$ from $p$ [30] (see Figure 2.9). When a ray ends up behind the geometry it is considered occluded, otherwise it is considered visible. Then the ambient occlusion $A_p$ is simply the fraction of unoccluded rays. We only look for occluders in a normal-



(a) Without ambient lighting. (b) With ambient lighting.

Figure 2.7: Without ambient lighting, regions in shadow are pitch black. Adding ambient lighting gives a more natural look, approximating the light bouncing around the scene that illuminates the regions obstructed from the light source.



(a) Environment map. (b) Irradiance map.

Figure 2.8: An environment map can be used to describe the distant environment surrounding the scene. This environment map can be used to pre-compute an irradiance map, which stores the irradiance for each direction (i.e., the average incoming light for each direction). Using the surface normal, we can lookup the color of the incoming light to shade the surface more accurately compared to only using an ambient term. Images from [14].

oriented hemisphere, as geometry behind $p$ is not able to occlude $p$. This approach computes the ambient occlusion $A_p$ by integrating the visibility function over the normal-oriented hemisphere $\Omega$ [57]:

$$A_p = \frac{1}{\pi} \int_{\Omega} (n \cdot \omega) V_{p,\omega} d\omega$$

where $V_{p,\omega}$ is the visibility function, which returns 0 if $p$ is occluded in direction $\omega$ and 1 otherwise.

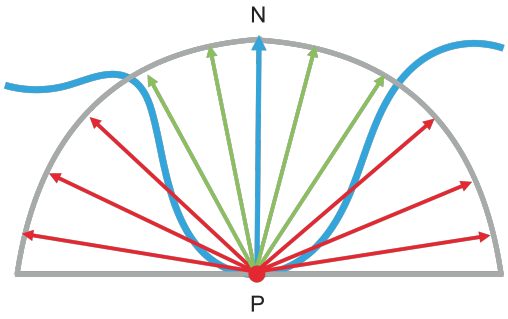
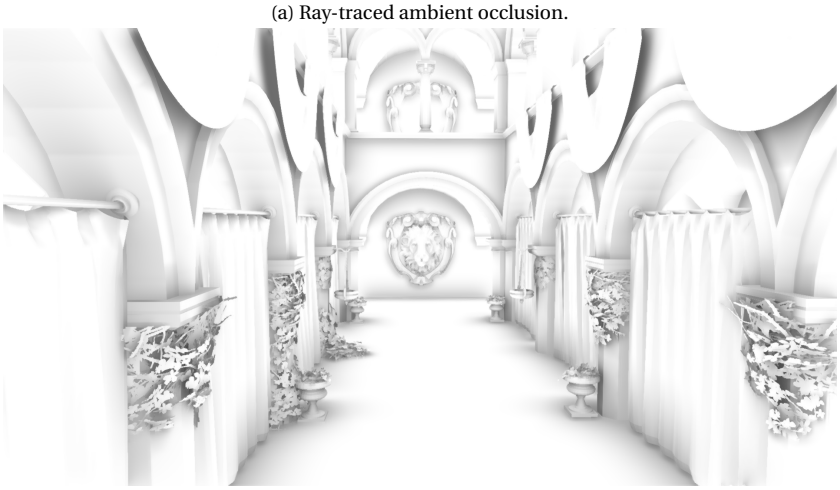



Figure 2.9: Integration over the normal-oriented hemisphere at point $p$ with normal $n$. The geometry surrounding $p$ is shown as the blue line. Rays indicated in red are under the geometry and thus deemed occluded, while the green rays are visible/unoccluded. The ambient occlusion is then computed as the fraction of unoccluded rays.

(a) Ray-traced ambient occlusion.



(b) Horizon-based ambient occlusion.

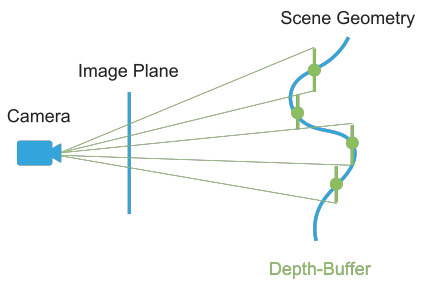
Figure 2.10: Ambient occlusion with the Sponza scene. The ray-traced image was rendered using Blender in around 2 minutes. The image with horizon-based ambient occlusion looks very similar, but only took around 1ms to render.

### Screen-Space Approximations

Casting rays to compute the ambient occlusion for each fragment is currently not feasible in real-time applications. Since ambient occlusion only depends on the geometry, it can be pre-baked into a texture for static geometry, but with dynamic geometry one needs to compute the ambient occlusion on-the-fly. Therefore ambient occlusion is often approximated in screen space (only taking the geometry currently shown on screen into account) as a post processing step, with techniques such as screen-space ambient occlusion (SSAO) [37] or horizon-based ambient occlusion (HBAO) [8]. These approximations deliver comparable results to ray-traced ambient occlusion at only a fraction of the cost (Figure 2.10), allowing them to be used in real-time applications.

These screen-space ambient occlusion algorithms work directly on the *depth buffer* (also known as $z$-buffer), which is a buffer/texture storing the depth information ($z$-value) per fragment (see Figure 2.11). The depth buffer has the same width and height as the color buffer (i.e., the image you see on the screen). When projecting the scene to screen space, each fragment will have an $X$ and $Y$ coordinate indicating its horizontal and vertical location on the screen and a $Z$ coordinate indicating its distance from the camera. The depth buffer stores in each pixel the lowest $z$-value (i.e., the depth of the fragment closest to the camera).

The downside of working in screen space is that these algorithms have no information about geometry not captured by the depth buffer, which can result in underestimated or missing occlusions.



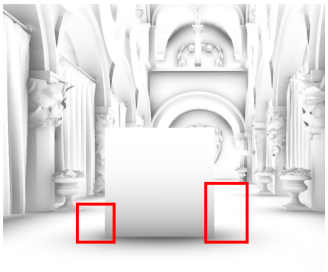

(a) Schematic overview of the depth buffer.

(b) Depth buffer from the game Borderlands 2.

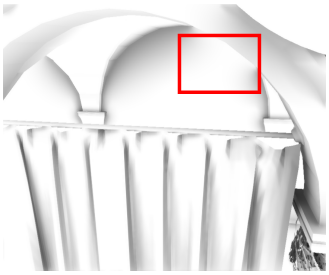
Figure 2.11: The depth buffer stores information about the geometry visible on screen. Each pixel stores the depth of the fragment closest to the camera (where black means close and white means far away from the camera).

There are three cases where this happens [3, 7] (shown in Figure 2.12): 1. The geometry is just outside of the view frustum, 2. The geometry is at a grazing angle with the camera or 3. Some geometry is hidden from view behind other geometry. In all these cases, we have incomplete information about the geometry surrounding point $p$, since it is hidden from view. The first case can be solved by slightly increasing the height and width of the depth buffer and the field of view during its rendering to capture the geometry just outside of the screen bounds [7, 35]. The second case requires the computation of screen-space ambient occlusion from additional viewpoints (using additional cameras) that are able to see the geometry under a non-grazing angle when compared to the main camera [42]. The main focus of this report is on the third case, which requires us to gather and use depth information of hidden/obstructed geometry, while still maintaining the real-time performance one would want from a screen-space ambient occlusion technique. For this we augment traditional screen-space ambient occlusion with transparency techniques. We will show that this approach can also be applied to solve the second case more efficiently, without requiring manual, per-scene placement of the additional cameras.

(a) Geometry is outside of the view frustum.



(b) Geometry is at a grazing angle.



(c) Geometry is obstructed by other geometry.

(d)

(e)

(f)

Figure 2.12: The three cases (a-c) where screen space does not provide enough information and results in missing occlusion (marked in red). Here d-f show the correct results (i.e., how it should look).

(a) Red on top. (b) Green on top.

Figure 2.13: The order between transparent objects has a large influence on the final blended color (i.e., where the squares overlap).

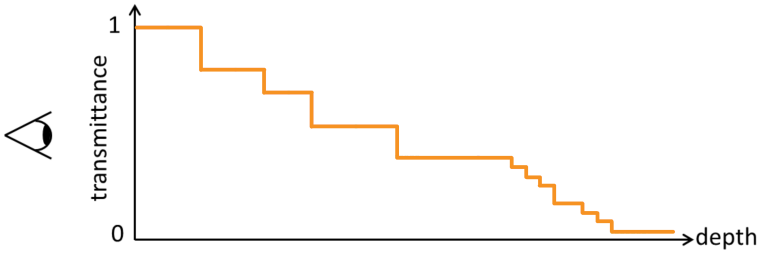



Figure 2.14: The visibility function $vis(z)$, where each fragments along a ray from the viewer into the scene represents a step. By sampling this function for a fragment with a certain depth, one can determine the transmittance of that fragment.

## 2.3. Transparency

Transparent objects allow light to pass through them, this enables us to see the geometry that is behind the transparent objects. Contrary to rendering opaque geometry, where only the closest fragment to the camera contributes to the final pixel color, multiple transparent fragments blend together to create the final pixel color. However, rendering accurate transparency efficiently with rasterization is quite a challenge. The main obstacle being the interaction between the transparent fragments, as the order between transparent objects has a large influence on the final the color (Figure 2.13).

Suppose that we have a transparent fragment $f_i$ with a depth $z_i$, a color $c_i$ and an alpha $\alpha_i$. Then given a visibility function $vis(z_i)$ that represents the total transmittance between $f_i$ and the viewer (Figure 2.14), one can compute the contribution of $f_i$ to the final color as:

$$c_i \alpha_i vis(z_i)$$

With the contribution of $n$ overlapping fragments given by:

$$\sum_{i=0}^{n} c_i \alpha_i vis(z_i) \tag{2.1}$$

Where the visibility function $vis(z)$ is defined as the product of the transmittance of every transparent fragment along a ray from the viewer to $z$:

$$vis(z) = \prod_{0<z_i<z} (1-\alpha_i) \tag{2.2}$$

This visibility function is often unknown during shading and depends on the depth-order of the transparent fragments. For this reason, Equation 2.1 is often computed recursively in back-to-front order using Porter and Duff's compositing (OVER) operator [40], also known as *alpha blending*:

$$\begin{aligned} C_0 &= \alpha_0 c_0 \\ C_n &= \alpha_n c_n + (1-\alpha_n) C_{n-1} \end{aligned} \tag{2.3}$$

Where $C_n$ is the final color. This technique gives accurate transparency without having to compute $vis(z)$, but requires the fragments to be sorted according to their depth value, which can be very expensive especially with the large amount of fragments in modern scenes.

For this reason, *order-independent transparency* (OIT) techniques are often used instead, as they produce accurate results regardless of the fragment submission order (i.e., fragments do not have to be sorted beforehand). Many popular OIT algorithms however, have highly varying performance costs and unbounded memory requirements, which are not desirable for real-time applications. Techniques such as *depth peeling* [18] or the *A-buffer* algorithm [10, 56] solve transparency using Equation 2.3, without requiring the fragments to be submitted in a back-to-front order, making them order-independent transparency techniques. In depth peeling, each transparency layer is rendered in a separate render pass, where in each pass the next nearest transparent fragment is determined using standard depth testing. For $n$ transparency layers, depth peeling requires $n$ rendering passes, making it unsuitable for real-time applications. The A-buffer algorithm stores all fragments in variable length per-pixel lists and sorts them after all geometry is rendered. This brings a large variance in performance depending on the scene's geometry and has unbounded memory requirements.

For real-time applications, we need efficient OIT algorithms that have a fixed number of render passes, and consistent and bounded memory requirements, independent of the scene's geometry. Some of these efficient OIT algorithms either apply or extend the OVER operator from Equation 2.3 in such a way that the fragments do not have to be sorted. Others try to approximate the visibility function as given by Equation 2.2 to solve Equation 2.1 directly. These techniques require 2 render passes, one to build a representation of the visibility function and one to shade the transparent fragments using this visibility representation. For a complete overview of the state of the art in efficient order independent transparency techniques, please refer to Appendix A.

A common representation of the visibility function is to store a bounded number of transparent fragments in a per-pixel list. Which fragments are stored in this list depends on the technique, using assumptions or heuristics to determine which samples would influence the final color the most. Such a per-pixel list stores the depth of multiple fragments contributing to the pixel color. These lists can be used to provide depth information about multiple depth layers to screen-space ambient occlusion techniques, similar to the A-buffer in [3] but with a bounded number of depth values. Efficient OIT techniques that use such a representation are: *adaptive transparency* [44], *multi-layer alpha blending* [43] and *stochastic transparency* [17].

Both adaptive transparency and multi-layer alpha blending store fragments in bounded per-pixel lists that are maintained in a front-to-back depth-order, where new fragments are inserted into the proper location. When rendering the transparent fragments, they are simply inserted into the list until the list is full. Then adaptive transparency will discard the fragment it deems to have the least impact on the visibility function, while multi-layer alpha blending merges the last entries in the list (i.e., the farthest fragments in the list) with the assumption that the farthest fragments have the least influence on the final color. To read/write to and from these per-pixel lists, these algorithms use techniques such as Intel's Pixel Synchronization or DirectX 12's Rasterizer Order Views that allow for write operations in primitive submission order, without incurring data races [13, 43]. The accuracy of these techniques depend on the size of the list and whether their assumptions or heuristics hold for the scene. If they do not, these algorithms can fail to accurately render the transparency.

In contrast, stochastic transparency uses *multi-sample anti-aliasing* (MSAA) to render and store the fragments that contribute to each pixel. For each fragment, an alpha-weighted coin is tossed to determine whether it is kept or discarded. Stochastic transparency uses this randomness to render, on average, correct alpha-blended colors at the cost of noise. Since no heuristics or assumptions are used, it is a very robust technique that works in every scene, but can require many samples to get a noise-free image. With only a few samples, we already get a quite convincing, albeit noisy, transparency effect with a very low cost in terms of performance. Other than traditional blending

operations on a multi-sampled depth buffer, stochastic transparency does not require any critical sections for read/write operations to its visibility representation. This robustness and performance are why we chose stochastic transparency to provide our ambient-occlusion algorithm with more depth information, we want it to work in every scenario and require as few samples as possible to reduce the performance impact.

# 3

# Related Work

Ambient occlusion is a global effect, meaning that the ambient occlusion at point $p$ depends on other geometry in the scene. Therefore, ray tracing is often used to compute accurate ambient occlusion. Ray tracing is however not a feasible option for real-time applications, so fast screen-space approximations are used instead. These can deliver comparable results in a fraction of the time. Even with the upcoming hardware support for real-time ray-tracing, the higher runtime and limited ray budget still make screen-space approximations or a hybrid solution preferable [9].

Many of the popular screen-space approximations are sample-based. They approximate the ambient occlusion at point $p$ in screen space, by projecting the hemisphere $\Omega$ to the image plane and sampling the geometry inside the hemisphere using the depth buffer. The differentiating factor between these techniques is their sampling scheme and their approximation of the ambient occlusion using these samples. The groundwork for these sample-based screen-space approximations was laid by [37] and [45]. They were later extended by [4, 8, 35, 36], with more physically-based derivations and more efficient sampling schemes. But, even though these screen-space approximations can come very close to their ray-traced counter part, they suffer from view-dependent artifacts as they do not have complete information about the scene's geometry, causing underestimation or sudden pop in/pop out of the ambient occlusion. For this reason, much research has been conducted on how to extend screen-space ambient occlusion in order to provide the missing information and remove these view dependent artifacts.

The approach taken by *hybrid ambient occlusion* [41] is to voxelize the scene on-the-fly, using a single-pass voxelization algorithm [16]. Then, for each point $p$ visible on the screen they compute the ambient occlusion by tracing rays in the voxelization, where a ray is deemed occluded if it intersects with a non-empty cell. The main downside of this approach is that ray tracing remains costly compared to the screen-space alternatives, even when using a voxelization of the scene [22]. To make it more performant for use in real-time applications, the resolution of the voxelization or the number of ray marching steps is lowered, which results in a loss of the high-frequency details.

Another volumetric approach is that of *ambient occlusion fields* [28], which precomputes a volume surrounding each object that encodes the occlusion caused by this object. *Ambient occlusion volumes* [29, 33] extends this approach for use with dynamic geometry, by borrowing techniques from *shadow volumes* [12]. Instead of precomputing a volume surrounding each object, they compute a volume surrounding each triangle. While these techniques can provide higher quality ambient occlusion than the screen-space algorithms, their speed is the limiting factor, making it undesirable for use in for example games. These techniques are mostly fill-rate limited, governed by the occlusion-volume overdraw.

*Multi-view ambient occlusion* [51] enhances traditional screen-space algorithms using information from multiple viewpoints. It uses the readily available shadow maps to get more information

regarding the geometry not visible from the main camera. For each point $p$ visible on the screen, the ambient occlusion is computed and averaged over the multiple viewpoints. To allow the algorithm to work in real-time, only a few additional viewpoints can be used. This technique works best in more open environments or scenes where the shadow maps are able to capture a significant portion of the geometry. If the shadow maps only capture a small region of the geometry, more viewpoints need to be used to accurately compute the ambient occlusion. In confined spaces or regions not present in the shadow maps, one would need to (manually) place additional cameras. The effectiveness of this approach is thus scene dependent, but its overall performance makes it suitable for real-time applications.

Another approach is to extend screen-space ambient occlusion using transparency techniques. The intuition is that transparency techniques allow us to "see through" any obstructing geometry. Instead of only regarding the first depth layer (i.e., the information stored in the depth buffer), these techniques store and use all depth layers from the main camera's viewpoint (in per-pixel lists) to compute the ambient occlusion. *Multi-layer dual-resolution screen-space ambient occlusion* [7] and *screen-space directional occlusion* [42] use *depth peeling* [18], while the approach from [3] uses an *A-buffer* [10, 56] to render and store all depth layers. To compute the ambient occlusion, these techniques iterate over all depth values stored in the per-pixel list and use the one that would provide the maximum occlusion. The downside of these techniques is that they are not suitable for real-time applications, since the transparency techniques used have highly varying performance costs and unbounded memory requirements [44]. Rendering and storing all depth layers can be very costly, and the performance and memory requirements largely depend on the scene's geometry and can vary widely per pixel. Furthermore, the ambient-occlusion computation itself becomes very expensive when iterating over many depth layers.

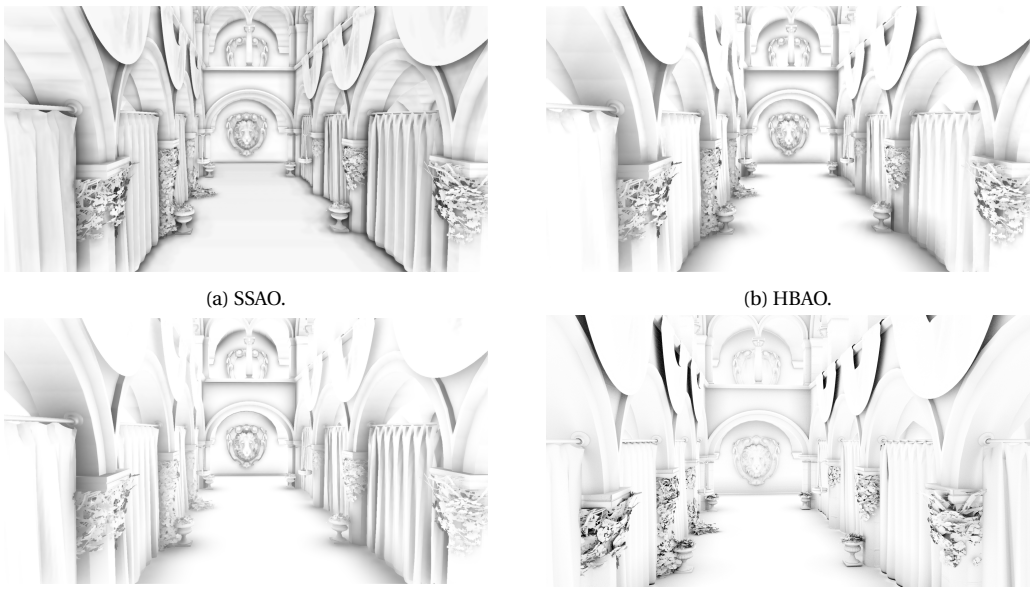
Since rendering and storing all depth layers is thus infeasible in real-time, the approach taken by the *deep G-buffer* technique [32] and NVIDIA's HBAO+ algorithm [48] is to use only two depth layers. They work under the assumption that only a few depth layers are necessary to remove most of the view-dependent artifacts. The deep G-buffer technique captures the first depth layer and the layer closest after that, with a certain minimum distance $\delta z$ (a scene dependent variable, specified by the user) from the first layer. The HBAO+ approach described by [48] uses the first layer of the static and dynamic geometry respectively, with the assumption that moving objects cause the most noticeable artifacts. The ambient-occlusion computation remains similar, it is computed for both depth layers and the maximum occlusion is used. While these techniques use two depth layers, they can easily be extended to an arbitrary number of layers. The disadvantage of using a fixed amount of depth layers is that it does not work for every scene, e.g., scenes with many obstructing objects could still lead to view-dependent artifacts.

Our proposal, stochastic-depth ambient occlusion, fits somewhere between the latter two approaches. We use stochastic transparency [17] to render and store $S$ (chosen by the user, between 1 and 16) randomly chosen depth layers, differing per pixel. The layers are rendered in a single rendering pass using MSAA and stored in a multi-sampled depth buffer. Similar to the other approaches, we compute the ambient occlusion for all $S$ depth layers and choose the maximum. Our approach has low and bounded memory requirements and a consistent runtime, similar to the deep G-buffer approach, since we only store a fixed number of depth layers. However, due to the stochastic nature, where the depth layers are different in each pixel, we are able to provide a greater amount of information with the $S$ samples. This allows it to even work well in more complex scenes with many obstructing depth layers, without negatively affecting the performance and memory costs.

# 4

# Screen-Space Ambient Occlusion

This project extends traditional screen-space ambient occlusion techniques by including information about multiple depth layers, to increase the accuracy and fill in the missing occlusion. The main focus was on horizon-based ambient occlusion (HBAO), but to compare how well our technique would generalize to other algorithms we also implemented an SSAO and HBAO+ version. These techniques use different approximations to compute the ambient occlusion in screen space, in turn producing vastly different results as seen in Figure 4.1. Let's first take a look at the different screen-space ambient occlusion algorithms and examine the failure case we aim to solve in more detail.



(a) SSAO.

(b) HBAO.

(c) HBAO+.

(d) Ray-traced ambient occlusion.

Figure 4.1: The different screen-space ambient occlusion techniques side by side. They all try to approximate the results from ray-traced ambient occlusion, but the more complex HBAO and HBAO+ techniques come much closer than the older and simpler SSAO algorithm. The results for HBAO and HBAO+ are similar, due to the similarities in the underlying algorithms, with the main difference being the overall less harsh ambient occlusion with HBAO+ compared to HBAO.

## 4.1. Screen-Space Ambient Occlusion (SSAO)

*Screen-space ambient occlusion* (SSAO) [37] was developed by Crytek and was the first screen-space ambient occlusion algorithm, and is to this day still popular. To avoid confusion, we will refer to this specific technique as SSAO. Between SSAO, HBAO and HBAO+, the SSAO algorithm is the simplest

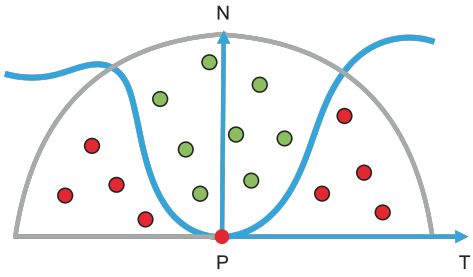



Figure 4.2: SSAO takes $N$ (in this case 16) sample points inside the normal-oriented hemisphere. We compare the depth of the sample positions with the depth stored in the depth buffer. If the sample point's depth is larger, it is obstructed by the geometry and deemed occluded (marked in red), otherwise is is unoccluded/visible (marked in green). The ambient occlusion is then the fraction of unoccluded samples.

**Algorithm 1** SSAO

Position $p \leftarrow$ reconstructed position (view space) from fragment's screen-space $X, Y$ position
Construct $TBN$ matrix and incorporate random rotation around the tangent-space $z$-axis
$AO \leftarrow 0$
**for** sample $s_i$ where $i = 0$ to $N$ **do**
    Sample $s_{view} \leftarrow$ view-space position of $s_i$ using $TBN$ matrix
    $s_{screen} \leftarrow$ screen-space position of $s_{view}$
    Depth $d \leftarrow$ view-space depth reconstructed using depth buffer with $X, Y$ position of $s_{screen}$
    **if** $d \geq s_{view}.z$ **then**
        $AO \leftarrow AO + smoothstep(0, 1, \frac{R}{||p.z-d||})$
$AO \leftarrow 1 - \frac{AO}{N}$
**return** $AO$

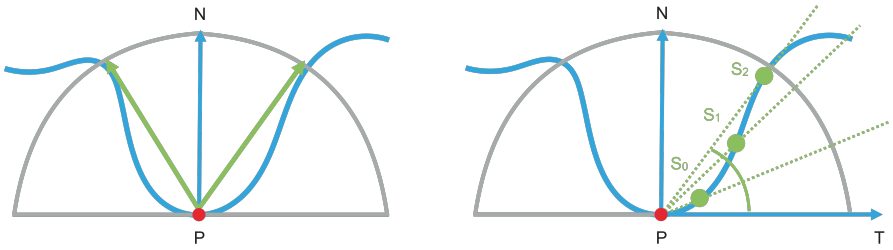
and follows the ray-traced ambient occlusion algorithm closely. It is rather different from HBAO and HBAO+, making it ideal to test how generalizable our proposed technique is.

SSAO takes $N$ sample points $s_i$ inside the normal-oriented hemisphere $\Omega$ with radius $R$ and determines whether these samples are occluded by geometry or not (Figure 4.2). It does this by comparing the depth of the sample points $s_i$ with the depth stored in the depth buffer at the same screen-space location, if the sample point's depth is larger than that of the depth buffer, it is occluded. The ambient occlusion is then the fraction of unoccluded samples. We only take the geometry inside a radius $R$ around point $p$ into account and ignore samples outside of this radius. To prevent large discontinuities in the ambient occlusion between neighbouring pixels where samples are just outside of the radius, SSAO uses a falloff term where samples far away from $p$ (in terms of depth) are penalized to contribute less to the overall ambient occlusion. This is done using OpenGL's smoothstep function interpolating between 0 and 1 with $\frac{R}{||p.z-d||}$, where $d$ is the view space depth reconstructed from the depth buffer.

The sample positions are stored in tangent space in a sample kernel, which is reused but rotated randomly for each pixel around the $z$-axis (i.e., the surface normal). This per-pixel randomness combined with a Gaussian blur allows SSAO to reduce the number of samples required for a convincing result, improving the performance. In Algorithm 1, the pseudocode for SSAO is described.

## 4.2. Horizon-Based Ambient Occlusion (HBAO)

*Horizon-based ambient occlusion* (HBAO) [8] aims to find the cone indicating the unobstructed region (Figure 4.3a). It does this by marching over the geometry inside the normal-oriented hemisphere $\Omega$ in multiple directions, where for each sample point $s_j$ it determines the angle between the vector




(a) The cone HBAO aims to find.

(b) Marching over the geometry.

Figure 4.3: Visualization of horizon-based ambient occlusion in a similar situation as that of Figure 2.9. HBAO aims to find the cone indicating the unobstructed region. It does this by marching over the geometry in $N_d$ directions and at each sample point $s_j$ it determines the angle between the vector from point $p$ to $s_j$ and the tangent vector $T$ of $p$. For each direction, we determine the vector with the largest angle with the tangent vector $T$ and these together form the cone of the unobstructed region.

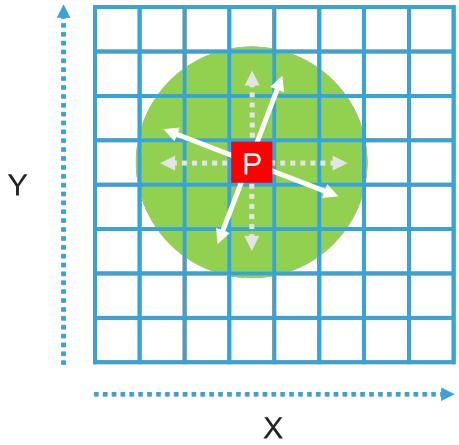



Figure 4.4: HBAO works directly on the depth buffer as visualized here. Point $p$ and the radius $R$ are projected onto the image plane, indicated by the red square and green circle respectively. We march in multiple directions (4 in this example) over the depth buffer to determine the ambient occlusion. The directions are uniformly distributed with a random per-pixel offset.

from point $p$ to $s_j$ and the tangent vector $T$ of $p$ in the direction of $s_j$ (as shown in Figure 4.3b). For each direction, we determine the vector with the largest angle with the tangent vector $T$ and these together form the cone indicating the unobstructed region.

Horizon-based ambient occlusion regards the depth buffer as a continuous heightfield over which we can march. We look at point $p$ and the geometry in a radius $R$ around $p$. Point $p$ and the radius $R$ are projected to the image plane, allowing us to march over the depth buffer in $N_d$ directions (a user parameter) from point $p$ inside the projected radius (Figure 4.4). The directions are uniformly distributed, with a random per-pixel offset.

In each direction $d_i$ that we march in, we take $N_s$ steps (a user parameter) at fixed intervals (Figure 4.5), with again a random per-pixel offset. For each sample $s_j$ in direction $d_i$, we determine the vector $v_j$ from $p$ to $s_j$. If $||v_i|| \geq R$ we ignore this sample, else we determine the angle $v_i$ with the $XY$-plane. The largest angle we find in direction $d_i$, is called the horizon angle $h$. For each direction $d_i$ we also compute the angle of the tangent vector of point $p$ with the $XY$-plane, giving us the tangent angle $t$. The ambient occlusion in this direction is then $A_{p,d_i} = sin(h) - sin(t)$. We average the ambient occlusion over all directions, giving us the final result.

Similar to SSAO, the contribution of each sample is weighted by a falloff term to reduce large discontinuities in the ambient occlusion between neighbouring pixels, in this case the radial function

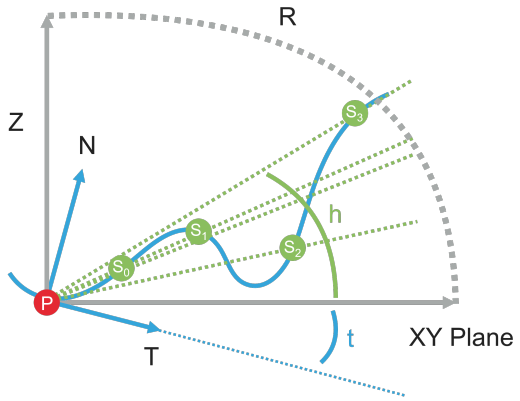
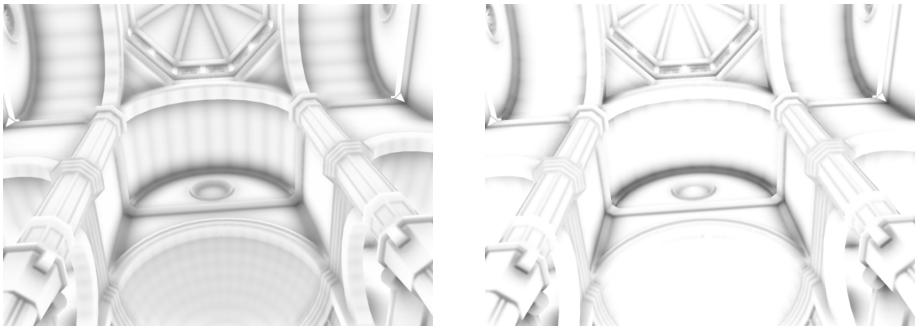



Figure 4.5: In each direction that we march in, we take $N_s$ steps (4 in this example). For each sample $s_j$, we determine the vector $v_j$ from $p$ to $s_j$ (ignoring samples outside of the radius $R$). We try to find the vector $v_j$ with the largest angle to the $XY$-plane, called the horizon angle $h$. We also determine the angle between the tangent vector $T$ and the $XY$-plane, i.e., the tangent angle $t$. Here the $Z$ axis points towards the camera.



(a) Without angle bias. (b) With angle bias (30 degrees).

Figure 4.6: Without the angle bias we see many of false occlusions in the creases of the arches on the ceiling. These false occlusions can be removed with the angle bias. Images from [8].

$W(r) = 1-(\frac{r}{R})^2$. We initialize $A_{p,d_i} = 0$ and for each sample we compute $A_{p,s_j} = sin(\phi_j) - sin(t)$ if $\phi_j > \phi_{j-1}$, where $\phi_j$ is the angle of $v_j$ with the $XY$-plane, and increment $A_{p,d_i}$ by $W(||v_i||)(A_{p,s_j} - A_{p,s_{j-1}})$. To remove false occlusion due to low tessellation, in for example arches (Figure 4.6), horizon-based ambient occlusion adds a small bias term ($bias$, a user parameter) to the tangent angle, making the algorithm ignore occlusion near the tangent plane. The horizon-based ambient occlusion algorithm is summarized by Algorithm 2.

## 4.3. Horizon-Based Ambient Occlusion Plus (HBAO+)

*HBAO+* [4] is largely based on the regular HBAO algorithm described above, with the key difference being that it uses a simpler ambient-occlusion approximation, similar to that of *Scalable Ambient Obscurance* [36]. This different way of computing the ambient occlusion eliminates the over-occlusion artifacts of HBAO. HBAO+ has support for two depth layers, one for dynamic and one for static geometry [48]. Our implementation aims to show this ambient-occlusion approximation can also be used in combination with our stochastic-depth buffer, showcasing that our technique is largely independent of the underlying ambient-occlusion computation.

Similar to horizon-based ambient occlusion we march over the depth buffer in $N_d$ directions, but instead of computing a horizon angle, each sample contributes to the ambient occlusion. For each sample $s_j$ in direction $d_i$, we compute the vector $v_j = s_j - p$, which we normalize and project onto the normal $n$ at point $p$ (Figure 4.7). The length of this projected vector $v'_j$, multiplied by

**Algorithm 2** HBAO

Position $p \leftarrow$ reconstructed position (view space) from fragment's screen-space $X, Y$ position
Normal $n \leftarrow$ reconstructed normal (view space) at $p$
$R' \leftarrow$ projected radius $R$ onto image plane
Determine stepsize as $R'/(N_s + 1)$
Determine directions with random offset
$AO \leftarrow 0$
**for** direction $d_i$ where $i = 0$ to $N_d$ **do**
    Determine tangent vector $T$
    Determine tangent angle $t$ of $T$ with $XY$-plane
    Horizon angle $h \leftarrow t + bias$
    Randomly offset first step
    **for** step $s_j$ where $j = 0$ to $N_s$ **do**
        Sample $s \leftarrow$ reconstructed position (view space) at step $j$ in direction $d_i$
        $v \leftarrow s - p$
        Determine angle $\phi$ of $v$ with $XY$-plane
        **if** $\phi > h$ and $||v|| < R$ **then**
            $AO \leftarrow AO + W(||v||)(sin(\phi) - sin(h))$
            $h \leftarrow \phi$
$AO \leftarrow 1 - \frac{AO}{N_d}$
**return** $AO$

the falloff term $W(||v_j||)$ (used to reduce large discontinuities in the ambient occlusion between neighbouring pixels), is then the contribution of this sample to the total ambient occlusion. We add up all the contributions of the samples and average them (i.e., divide by $N_d \cdot N_s$) to get the final ambient occlusion. This results in the pseudocode as described in Algorithm 3.

## 4.4. Algorithmic Details

The pseudocode provides a rough overview of how the screen-space ambient occlusion algorithms work, but does not provide all necessary details. in the following subsections we will explain some key steps shared by the algorithms in more detail.

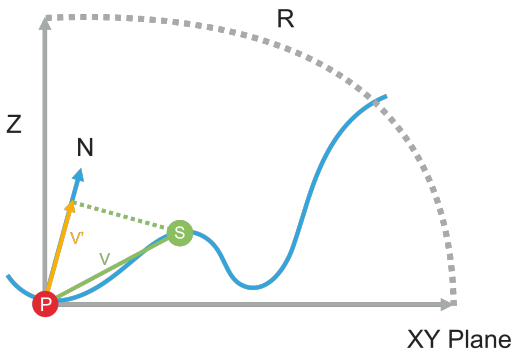



Figure 4.7: Similar to regular HBAO, HBAO+ marches over the depth buffer in each direction with $N_s$ steps. For each sample $s_j$, we determine the vector $v_j$ from $p$ to $s_j$ (ignoring samples outside of the radius $R$). We normalize and project this vector onto the normal $n$ at point $p$ to get the vector $v'_j$. Then the contribution of sample $s_j$ to the ambient occlusion is determined by the length of the projected vector $v'_j$.

**Algorithm 3** HBAO+

Position $p \leftarrow$ reconstructed position (view space) from fragment's screen-space $X, Y$ position
Normal $n \leftarrow$ reconstructed normal (view space) at $p$
$R' \leftarrow$ projected radius $R$ onto image plane
Determine step size as $R'/(N_s+1)$
Determine directions with random offset
$AO \leftarrow 0$
**for** direction $d_i$ where $i = 0$ to $N_d$ **do**
    Randomly offset first step
    **for** step $s$ where $j = 0$ to $N_s$ **do**
        Sample $s \leftarrow$ reconstructed position (view space) at step $j$ in direction $d_i$
        $v \leftarrow s - p$
        $AO \leftarrow AO + W(||v||)max(\frac{v \cdot n}{||v||} - bias, 0)$
$AO \leftarrow 1 - \frac{AO}{N_d \cdot N_s}$
**return** $AO$

### View-Space Positions Reconstruction

Screen-space ambient occlusion techniques often work with view-space positions, which are reconstructed using the depth buffer and the projection matrix $M_{proj}$. Using the projection matrix, positions in view space can be projected to clip space, with homogeneous coordinates ranging from $-1$ to 1. These are then converted from homogeneous coordinates to Cartesian coordinates and scaled to range from 0 to 1 to reach the screen-space positions. By doing the inverse of these steps, we can reconstruct the view-space positions using the screen-space positions and projection matrix.

To launch the ambient-occlusion computation for all pixels, we simply draw a screen-filling quad. The screen-space coordinates of this quad can then be used as UV coordinates in the depth buffer, to determine the screen-space $Z$ coordinate. With the $X$, $Y$ and the $Z$ value stored in the depth buffer we know the screen-space position:

$$p_{screen} = (X, Y, Z)$$

We can scale these to range from $-1$ to 1, by multiplying them by 2 and subtracting 1, and convert it to homogeneous coordinates to get the clip-space position:

$$p_{clip} = (2X - 1,\ 2Y - 1,\ 2Z - 1,\ 1)$$

Then we can multiply this clip-space position with the inverse of the projection matrix to get the view-space position as an homogeneous coordinate:

$$p_{viewH} = M_{proj}^{-1} p_{clip}$$

Then by dividing its $X$, $Y$, and $Z$ components with its $W$ coordinate, we get the view position as a Cartesian coordinate.

### View-Space Depth Reconstruction

In SSAO, we only require the view-space depth for each sample, this means that we can skip some parts of the complete reconstruction. The projection matrix has the form [23]:

$$M_{proj} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & d \\ 0 & 0 & e & 0 \end{bmatrix}$$

(a) Low sample count "banding". (b) Per-pixel randomness = noise. (c) + blur = acceptable.

Figure 4.8: With low sample counts we would get banding artifacts, but increasing the sample count would hurt performance. Instead we add per-pixel randomness (here the sample kernel is rotated randomly per-pixel, since SSAO is used), but this introduces noise. By blurring these results however, the noise and banding artifacts are removed. Image from [11].

and its inverse has the form:

$$M_{proj}^{-1} = \begin{bmatrix} \frac{1}{a} & 0 & 0 & 0 \\ 0 & \frac{1}{b} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{e} \\ 0 & 0 & \frac{1}{d} & -\frac{c}{de} \end{bmatrix}$$

The multiplication with $p_{clip}$ to get the view-space position then becomes:

$$p_{viewH} = M_{proj}^{-1} p_{clip} = \begin{bmatrix} \frac{1}{a} & 0 & 0 & 0 \\ 0 & \frac{1}{b} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{e} \\ 0 & 0 & \frac{1}{d} & -\frac{c}{de} \end{bmatrix} \begin{bmatrix} (2X-1) \\ (2Y-1) \\ (2Z-1) \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{a}(2X-1) \\ \frac{1}{b}(2Y-1) \\ \frac{1}{e} \\ \frac{1}{d}(2Z-1) - \frac{c}{de} \end{bmatrix}$$

The view-space depth $Z_{view}$ can thus be computed as:

$$Z_{view} = \frac{Z_{viewH}}{W_{viewH}} = \frac{\frac{1}{e}}{\frac{1}{d}(2Z-1) - \frac{c}{de}}$$

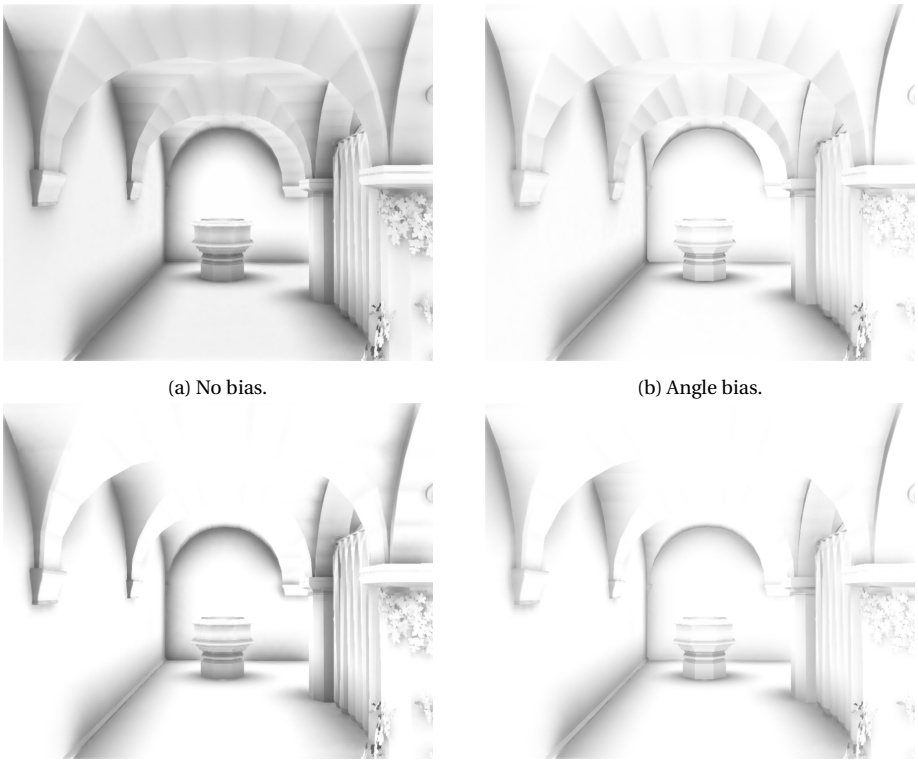
without the need of computing the whole matrix multiplication.

### Per-Pixel Randomness and Gaussian Blur

These screen-space ambient occlusion algorithms all take samples around the normal-oriented hemisphere to determine the ambient occlusion. If the sample count would be too low, we would get "banding" artifacts (Figure 4.8), but increasing the sample count comes at the cost of performance. We can trade these banding artifacts for noise, by changing the sample locations randomly per-pixel. Then to reduce the noise, a depth-dependent cross-bilateral Gaussian blur filter is applied, which blurs less across edges. This allows the screen-space ambient occlusion techniques to achieve good results, with only a small number of samples [8, 19, 37].

### Angle Bias with Normal Threshold

Horizon-based ambient occlusion makes use of an angle bias to reduce false occlusions, due to low tessellation. This adds a bias term to the tangent angle, making the algorithm ignore occlusion near the tangent plane. However, if we increase the tangent angle too much with such a bias term, the ambient occlusion will start to diminish, since it is computed as $A_{p,d_i} = sin(h) - sin(t)$, where $sin(t)$ increases, while $sin(h)$ stays the same. We found that ignoring samples with an angle larger than



(a) No bias. (b) Angle bias.

(c) Angle threshold. (d) Angle bias + threshold.

Figure 4.9: Without the angle bias we see lots of false occlusion in the creases of the arches on the ceiling. We can remove this false occlusion up to a certain extend with the angle bias, but increasing this bias anymore would make the true occlusion vanish. The angle threshold does a better job at removing this false occlusion, but is still not able to fully remove it. A combination of the angle bias and the angle threshold works best. Here we use an angle bias of 30 degrees and a angle threshold of around 66.5 degrees.

a certain threshold between the surface normal $n$ at $p$ and $v_j$ allows us to remove false occlusion, without impacting the true ambient occlusion. The best results were found when combining this angle threshold with the angle bias, allowing us to almost completely eliminate the false occlusion.

## 4.5. Failure Case

These screen-space approximations of ambient occlusion can get very close to ray-traced ambient occlusion, at a fraction of the cost. However, working in screen space comes with its downside: the algorithm only has the information that is currently in view. Geometry that is outside of the view frustum, at a grazing angle or hidden from view behind other geometry, is not visible in screen space and thus ignored by these algorithms. This leads to view-dependent artifacts, where the ambient occlusion suddenly pops in and out when moving the camera around the scene, as geometry becomes hidden from view.

Figure 4.10 shows an example where moving the camera slightly causes the ambient occlusion to disappear with horizon-based ambient occlusion, while the ray-traced algorithm is able to capture it correctly. In this case, rotating the camera causes the geometry casting the ambient occlusion to be hidden from view behind the pillar. Suppose that Figure 4.11a and 4.11b correspond to the geometry in Figure 4.10a and 4.10b respectively, then with horizon-based ambient occlusion we march along multiple directions to find the largest horizon angle. When we march over the geometry stored in the depth buffer (i.e., the frontmost depth layer), we take a sample $s_i$ at each step $i$ and determine the angle between the vector $s_i - p$ and the $XY$-plane, ignoring samples outside of the radius $R$. The screen-space algorithm only knows about the geometry in front, when we obstruct

a samples by placing an object (in this case a pillar) in front of it, the ambient-occlusion algorithm would not know it existed. In this failure case however, the object in front is outside of the radius, meaning that is should not contribute to the ambient occlusion and is thus ignored. Since we do not have information about the obstructed sample that would have been inside of the radius, we find a smaller horizon angle than we would find without having the object in front.

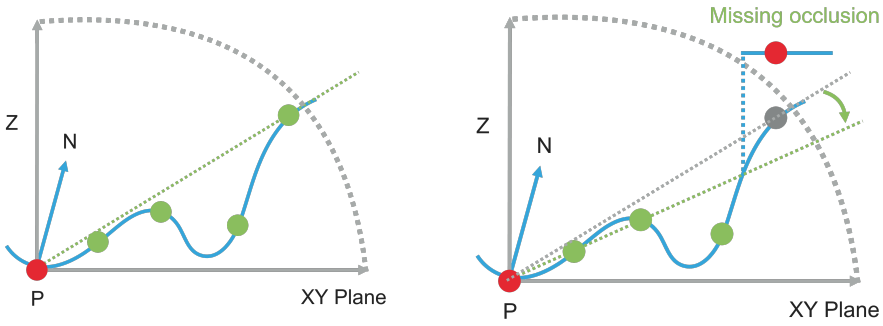
To solve this, we need to provide information about the obstructed depth layers. In the case that the frontmost sample is outside of the radius, we can look at the other depth layers to determine the correct maximum horizon angle. However gathering and storing multiple depth layers is quite a challenge in and of itself, we would somehow need to "see through" the obstructing geometry. With this intuition, we turn to transparency algorithms, which already use many depth layers in order to render accurate transparency. Using these transparency techniques, we can get information about otherwise obstructed objects in screen space.

(a) HBAO.

(b) HBAO from different viewpoint.

(c) Ray-traced ambient occlusion.

Figure 4.10: When rotating the camera around (from a to b), object casting occlusion can become hidden from view. In this case the geometry becomes hidden behind the pillar, causing missing occlusion (marked in red) when compared to the ray-traced ambient occlusion (c).




(a) Without obstructing geometry.

(b) With obstructing geometry.

Figure 4.11: Suppose that (a) and (b) correspond to Figure 4.10a and 4.10b respectively. The geometry that provides the largest horizon angle is suddenly occluded by the pillar. Since the sample on this pillar is outside of the radius (marked in red) and we have no information about the geometry behind the pillar (the sample marked in grey), we find a smaller horizon angle and thus have less occlusion than we should.

# 5

# Stochastic Transparency

Stochastic transparency [17] is based on traditional screen-door transparency [20], which creates a fake transparency effect by discarding fragments of a transparent surface in a stipple pattern (Figure 5.1). Stochastic transparency extends this approach with a random sub-pixel stipple pattern with the help of multi-sample anti-aliasing (MSAA). Using MSAA, stochastic transparency takes $S$ samples per pixel, storing only depth values. Stochastic transparency consists of the following render steps:

1. (Opaque shading pass) Render the opaque geometry into color buffer A
2. (Alpha accumulation pass) Render the transparent geometry, compute and store the total alpha for each pixel in a texture.
3. (stochastic-depth pass) Render the transparent geometry and build a representation of the visibility function: store depth in a multi-sampled depth buffer.
4. (Transparent shading pass) Render the transparent geometry, determine $vis(z)$ using the representation from step 3, and composite into color buffer B.
5. (Compositing pass) Composite color buffer A and B to create the final image, also apply the alpha correction using the total alpha from step 2.

Stochastic transparency requires a fixed number of render passes and a fixed amount of memory, and its performance scales with MSAA hardware improvements.

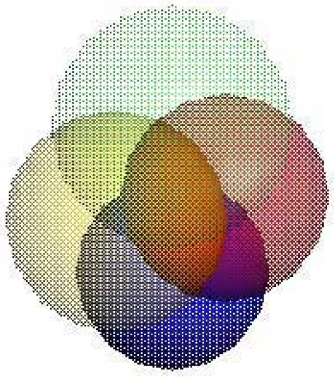


Figure 5.1: An example of screen-door transparency, where discarding fragments in a stipple pattern gives the illusion of transparency. From [38].

**Algorithm 4** Stochastic Depth

```
1: for each transparent fragment i that falls in current pixel do
2:     for each MSAA sample j do
3:         if z_i ≤ z_j then                ▷ where z_j is the depth currently stored at sample j
4:             r ← random number between 0 and 1
5:             if r > α_i then
6:                 discard fragment
7:             else
8:                 store fragment's depth z_i at sample j
9: return multi-sampled depth buffer, i.e., the stochastic-depth buffer
```

## 5.1. Stochastic Depth

The core of the stochastic transparency algorithm is the way it creates its visibility representation: the stochastic-depth pass. Here the transparent geometry is rendered to a multi-sampled depth buffer, which we will call the stochastic-depth buffer. When rendering a transparent fragment $f_i$, we throw an $\alpha_i$-weighted coin for each MSAA sample (up to 16 in our implementation). Depending on the outcome of this coin toss, we store or discard this fragment for that sample. Meanwhile, each sample does depth buffer comparisons to retain only the frontmost fragment. The resulting pseudocode of this stochastic-depth pass is summarized by Algorithm 4.

This process is very similar to Monte Carlo ray tracing, but instead of rays we use $S$ MSAA samples. A transparent fragment $f_i$ would thus be covered by a stochastic subset of $R$ samples, where $\alpha_i = R/S$. If fragment $f_i$ is not the frontmost fragment, its effective $\alpha$ becomes:

$$\alpha = \alpha_i \cdot \prod_{z_j < z_i} (1 - \alpha_j)$$

While this simple approach already works quite well, the results can be rather noisy. To reduce this noise, we can increase the number of samples, but this approach is plagued by the classic Monte Carlo problem of diminishing returns: halving the average error requires quadrupling the number of samples. For this reason, stochastic transparency makes use of stratified sampling to reduce the noise, where instead of flipping a coin toss for each sample individually, it sets a group of $R$ samples simultaneously. For a fragment $f_i$, stratified stochastic transparency sets a group of $R_i$ samples:

$$R_i = \lfloor \alpha_i S + \xi \rfloor$$

Here $\xi$ is a canonical random number between 0 and 1. We then choose a random subset of $R_i$ samples and let fragment $f_i$ be covered by these samples.

To do this efficiently we precompute and store all possible subsets/permutations of the 16 MSAA samples in a lookup table. Each subset is represented as a bitstring of 16 bits representing the (up to) 16 MSAA samples, where a 1 indicates that the sample is part of the group and 0 indicates that it is not. There are at most $2^{16} = 65536$ possible permutations/subsets, each requiring 16 bits or 2 bytes to store, coming to a total of only $65536 \cdot 2$ bytes = 131kB. This lookup table is send once to the GPU and is kept in memory as an `Shader Storage Buffer Object`. The lookup table is sorted by population count (i.e., the number of 1s in the bitstring) and a secondary array is used to store the index of the first subset with a certain population count. If we need to set a group of $R_i$ samples, we query this secondary index array to determine the index range in the lookup table for subsets with a population count of $R_i$ and pick one of them at random. This stratified approach is described by Algorithm 5.

**Algorithm 5** Stratified Stochastic Depth

**for** each transparent fragment $i$ that falls in current pixel **do**
  **for** each MSAA sample $j$ **do**
    **if** $z_i \leq z_j$ **then** ▷ where $z_j$ is the depth currently stored at sample $j$
      $\xi \leftarrow$ random number between 0 and 1
      $R_i = \lfloor \alpha_i S + \xi \rfloor$
      **if** $R_i = S$ **then** ▷ Trivial case, does not require lookup table
        store fragment's depth $z_i$ at sample $j$
      **else if** $R_i = 0$ **then** ▷ Trivial case, does not require lookup table
        discard fragment
      **else**
        index $\leftarrow$ random integer between indexArray[$R_i$] and indexArray[$R_i + 1$]
        mask $\leftarrow$ lookUpTable[Index]
        **if** $j$th bit of mask is 1 **then**
          store fragment's depth $z_i$ at sample $j$
        **else**
          discard fragment
**return** multi-sampled depth buffer, i.e., the stochastic-depth buffer

## 5.2. Transparency Using Stochastic Depth

In the stochastic-depth buffer, we thus store $S$ depth values per-pixel $z_0, z_1, ..., z_{S-1}$, corresponding to the $S$ MSAA samples. Using this stochastic-depth buffer, we can approximate the visibility function $vis(z)$ as:

$$vis(z) \approx \frac{\sum_{i=0}^{S} D(z, z_i)}{S}$$

where

$$D(z, z_i) \begin{cases} 1 & z \leq z_i \\ 0 & z > z_i \end{cases}$$

The visibility at depth $z$ is the fraction of samples that contain a depth value larger than $z$. With this visibility approximation, we can render the transparent geometry and determine their contribution to the overall pixel color using Equation 2.1. The pseudocode for this pass is given by Algorithm 6. Important to note is that we use the opaque depth buffer for depth testing the transparent fragments (only reading from, not writing to this buffer), this makes sure we do not shade transparent fragments behind opaque geometry.

**Algorithm 6** Stochastic Transparency Shading

Pixel color $C \leftarrow (0,0,0,0)$ ▷ color represented as (R, G, B, A)
**for** each transparent fragment $i$ that falls in current pixel **do**
  $vis \leftarrow 0$
  **for** each depth value $z_j$ in stochastic-depth buffer **do**
    **if** $z_i \leq z_j$ **then**
      $vis \leftarrow vis + 1$
  $vis \leftarrow \frac{vis}{S}$
  Shade fragment $i$, compute its color $c_i$, as (R, G, B), and $\alpha_i$
  $C \leftarrow C + (c_i \cdot \alpha_i \cdot vis, \alpha_i \cdot vis)$
**return** color buffer of transparent geometry

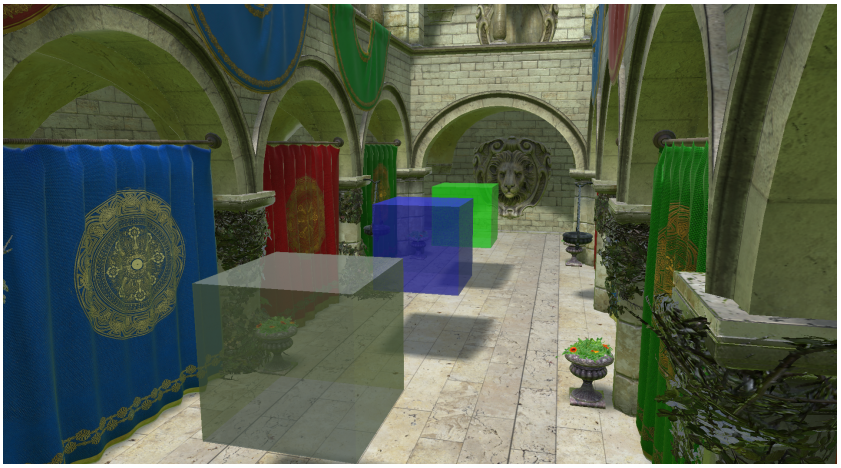


Figure 5.2: Our implementation of stochastic transparency in action, with 8 samples and stratified sampling. Here the cubes are half transparent ($\alpha = 0.5$), while the rest of the scene is opaque.

The accumulated $\alpha_{acc} = \sum_i vis(z_i)\alpha_i$ in a texel of the transparent color buffer B is not necessarily equal to the actual sum of all the fragment's alpha values, due to the stochastic approximation of the visibility. Instead the correct, non-stochastic total alpha of all contributing transparent fragments is defined as:

$$\alpha_{total} = 1 - \prod_i 1 - \alpha_i$$

which can be computed in one render pass over the transparent geometry (Step 2) as depth order does not influence this computation due to the product's commutative property. To get the correct intensity of the opaque and transparent geometry and reduce noise, an alpha correction factor is used during the compositing pass. Here, we blend the RGB color components of the opaque color buffer ($c_{opaque}$) and the transparent color buffer ($c_{transparent}$) to get the final pixel color $C$ in the following way:

$$C = (1 - \alpha_{total}) \cdot c_{opaque} + \frac{\alpha_{total}}{\alpha_{acc}} \cdot c_{transparent}$$

where $\alpha_{acc}$ is stored in the alpha channel of the transparent color buffer (color buffer B). Figure 5.2 shows our stochastic transparency implementation in action.

# 6

# Stochastic-Depth Ambient Occlusion

By combining screen-space ambient occlusion techniques with stochastic transparency, we can provide additional depth information to improve the accuracy of the fast screen space techniques, at a relatively low performance penalty. For this, we need to modify the screen-space ambient occlusion techniques to take geometry of other depth layers into account and use stochastic transparency to capture this information.

## 6.1. Stochastic Depth Rendering

Using stochastic transparency, we can render transparency in real-time, albeit with some noise. For our use case, we want to "see through" opaque geometry to get information about the obstructed geometry. To do this, we render the whole scene as if it is transparent, meaning that we give all opaque geometry an $\alpha$ value smaller than 1 (e.g., an $\alpha$ value of 0.2 was found to work well in most situations). This allows us to see through the opaque geometry, in order to gain information about the geometry otherwise hidden from view. With a high $\alpha$ value (very opaque), we are at risk of only capturing information about the depth layers most in front, while with a lower $\alpha$ (more transparent) we capture all depth layers more evenly. If the $\alpha$ value would be too low however, we are at risk of simply discarding many layers.

Stochastic transparency allows us to accurately compute the blended color of the now transparent geometry. However, with ambient occlusion we are not interested in color, but in the depth layers used to compose this colors. In the stochastic transparency algorithm this information is stored in its representation of the visibility function, the multi-sampled depth buffer, which we call the stochastic-depth buffer. In this buffer, a random selection of depth layers are stored to estimate the visibility function. While rendering a fully transparent scene with stochastic transparency would require 3 render passes over the transparent geometry, rendering the stochastic-depth buffer only requires a single pass (step 3 from the render steps described in Chapter 5). Also, because the stochastic-depth buffer is rendered as a depth buffer, we can use depth testing to automatically discard fragments with a depth value larger than what is currently stored. This makes it very fast to render the stochastic-depth buffer, making it a viable way of providing more information to the ambient-occlusion techniques.

Since we want to reconstruct view-space positions from the stochastic-depth buffer, it is important that the depth corresponds to geometry exactly in the middle of the texel, otherwise the depth does not correspond to the screen-space $X$ and $Y$ coordinates and the reconstructed positions would be incorrect. For this reason all MSAA samples are taken at the center of the pixel using the `ARB_sample_locations`[1] OpenGL extension, which allows us to program the sample locations.

[1] `https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_sample_locations.txt`

(a) The Sponza scene. (b) Rendered as transparent ($\alpha = 0.1$).

Figure 6.1: By rendering the whole scene as if it is transparent (using stochastic transparency), we are able to see geometry that would otherwise be hidden from view, while still remaining in screen space.

## 6.2. Ambient Occlusion

For *stochastic-depth ambient occlusion* (SDAO), we modify the screen-space ambient occlusion techniques discussed in Chapter 4 to iterate over the multiple depth layers. Stochastic-depth ambient occlusion takes as input both the regular and stochastic-depth buffer. The regular depth buffer captures what is actually visible from the camera, so we use it to determine the positions $p$ we want to compute the ambient occlusion for, but we march over the stochastic-depth buffer. Similar to [3, 7, 32], we use the $S$ depth samples in the stochastic-depth buffer to reconstruct $S$ positions from multiple depth layers. At each step, we compute the ambient-occlusion contribution of all these reconstructed positions and keep the maximum. This means that we still march in $N_d$ directions and take $N_s$ steps, but each step now checks $S$ positions corresponding to the $S$ samples in the stochastic-depth buffer. Each of these positions has the same screen-space $X$ and $Y$ coordinate as they correspond to the same texel, but has a different $Z$ coordinate.

If we take another look at the multiplication of the clip-space position and the inverse projection matrix from Section 4.4 and 4.4:

$$p_{viewH} = M_{proj}^{-1} p_{clip} = \begin{bmatrix} \frac{1}{a} & 0 & 0 & 0 \\ 0 & \frac{1}{b} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{e} \\ 0 & 0 & \frac{1}{d} & -\frac{c}{de} \end{bmatrix} \begin{bmatrix} (2X-1) \\ (2Y-1) \\ (2Z-1) \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{a}(2X-1) \\ \frac{1}{b}(2Y-1) \\ \frac{1}{e} \\ \frac{1}{d}(2Z-1) - \frac{c}{de} \end{bmatrix}$$

We can see that $X_{viewH}$, $Y_{viewH}$, $Z_{viewH}$ and part of $W_{viewH}$ do not depend on the clip or screen-space depth, only on the clip or screen-space $X$ and $Y$ coordinates. This means that we can reuse a part of this matrix multiplication that is similar for all depth samples:

$$m_{viewH} = (2X-1) \begin{bmatrix} \frac{1}{a} \\ 0 \\ 0 \\ 0 \end{bmatrix} + (2Y-1) \begin{bmatrix} 0 \\ \frac{1}{b} \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \frac{1}{e} \\ -\frac{c}{de} \end{bmatrix} = \begin{bmatrix} \frac{1}{a}(2X-1) \\ \frac{1}{b}(2Y-1) \\ \frac{1}{e} \\ -\frac{c}{de} \end{bmatrix}$$

And we can compute the view position as:

$$p_{viewH} = m_{viewH} + (2Z-1) \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{1}{d} \end{bmatrix} = \begin{bmatrix} \frac{1}{a}(2X-1) \\ \frac{1}{b}(2Y-1) \\ \frac{1}{e} \\ \frac{1}{d}(2Z-1) - \frac{c}{de} \end{bmatrix}$$

**Algorithm 7** Stochastic-depth HBAO

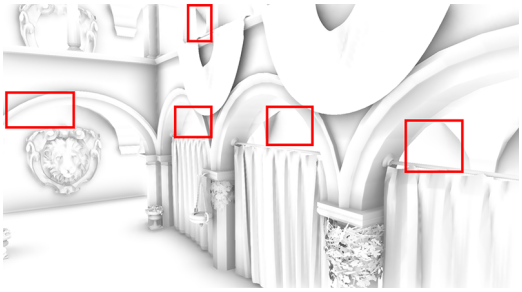
Position $p \leftarrow$ reconstructed position (view space) from fragment's screen-space $X, Y$ position
Normal $n \leftarrow$ reconstructed normal (view space) at $p$
$R' \leftarrow$ projected radius $R$ onto image plane
Determine stepsize as $R'/(N_s+1)$
Determine directions with random offset
$AO \leftarrow 0$
**for** direction $d_i$ where $i = 0$ to $N_d$ **do**
    Determine tangent vector $T$
    Determine tangent angle $t$ of $T$ with $XY$-plane
    Horizon angle $h \leftarrow t + bias$
    Randomly offset first step
    **for** step $s_j$ where $j = 0$ to $N_s$ **do**
        $AO_{max} \leftarrow 0$
        $h_{max} \leftarrow 0$
        $m_{viewH} \leftarrow$ partial view position reconstruction at step $j$ in direction $d_i$
        depths[] $\leftarrow$ all stochastic-depth values at step $j$ in direction $d_i$
        **for** each depth value $k$ in depths[] **do**
            Sample $s \leftarrow$ reconstructed view space position using $m_{viewH}$ and depth $k$
            $v \leftarrow s - p$
            **if** $\frac{v \cdot n}{||v||} >$ angle threshold **then**
                Determine angle $\phi$ of $v$ with $XY$-plane
                **if** $\phi > h$ and $||v|| < R$ **then**
                    $AO_k \leftarrow W(||v||)(sin(\phi) - sin(h))$
                    **if** $AO_k > AO_{max}$ **then**
                        $AO_{max} \leftarrow AO_k$
                        $h_{max} \leftarrow \phi$
        **if** $AO_{max} > 0$ **then**
            $AO \leftarrow AO + AO_{max}$
            $h \leftarrow h_{max}$
$AO \leftarrow 1 - \frac{AO}{N_d}$
**return** $AO$

This reduces the total work we need to perform per stochastic-depth sample.

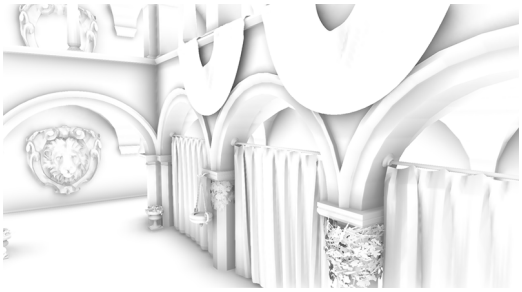
Furthermore, since we need to read multiple depth values, depending on the amount of depth samples in the stochastic-depth buffer, our stochastic-depth approach is mainly limited by bandwidth. To improve performance, we fetch all depth values of a texel in one batch, instead of reading the stochastic-depth buffer inside of a loop (similar to the Ray-Marching optimization example given by [5]). Because these texture instructions can be executed in parallel and the depth values of a single texel are stored directly after each other in memory, this can greatly improve performance. The pseudocode for stochastic-depth HBAO, HBAO+ and SSAO are given by Algorithm 7, 8 and 9 respectively (with their results shown in Figure 6.2).

## 6.3. Only Stochastic Depth If Outside of the Radius

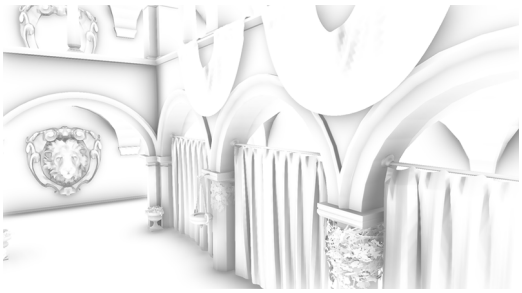
As explained in Section 4.5, screen-space ambient occlusion can miss occlusion when a sample, reconstructed via the depth buffer, lies outside of the radius $R$. It could be that there is geometry behind the first depth layer that is inside the radius $R$ that would cast occlusion. With this intuition, we experimented with only using the stochastic-depth buffer if the sample from the regular depth



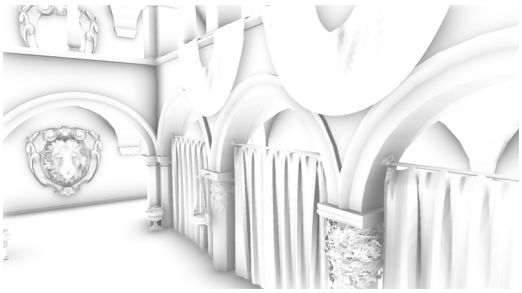
(a) HBAO.



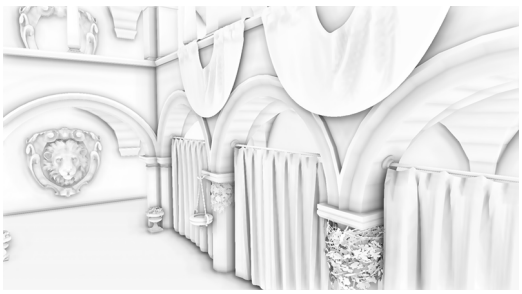
(b) Stochastic-depth HBAO.



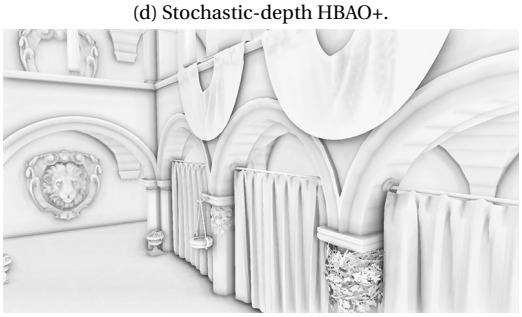
(c) HBAO+.



(d) Stochastic-depth HBAO+.



(e) SSAO.



(f) Stochastic-depth SSAO.

(g) Ray-traced ambient occlusion.

Figure 6.2: The results of stochastic-depth ambient occlusion with the different techniques. Notice how the missing occlusion (in the regions marked by the red squares) with the traditional screen-space ambient occlusion techniques is correctly captured by their stochastic-depth counterparts.

**Algorithm 8** Stochastic-Depth HBAO+

Position $p \leftarrow$ reconstructed position (view space) from fragment's screen-space $X, Y$ position
Normal $n \leftarrow$ reconstructed normal (view space) at $p$
$R' \leftarrow$ projected radius $R$ onto image plane
Determine step size as $R'/(N_s + 1)$
Determine directions with random offset
$AO \leftarrow 0$
**for** direction $d_i$ where $i = 0$ to $N_d$ **do**
    Randomly offset first step
    **for** step $s_j$ where $j = 0$ to $N_s$ **do**
        $AO_{max} \leftarrow 0$
        $m_{viewH} \leftarrow$ partial view position reconstruction at step $j$ in direction $d_i$
        depths[] $\leftarrow$ all stochastic-depth values at step $j$ in direction $d_i$
        **for** each depth value $k$ in depths[] **do**
            Sample $s \leftarrow$ reconstructed view-space position using $m_{viewH}$ and depth $k$
            $v \leftarrow s - p$
            $AO_{max} \leftarrow max(AO_{max},\ W(||v||)max(\frac{v \cdot n}{||v||} - bias, 0))$
        $AO \leftarrow AO + AO_{max}$
$AO \leftarrow 1 - \frac{AO}{N_d \cdot N_s}$
**return** $AO$

**Algorithm 9** Stochastic-Depth SSAO

Position $p \leftarrow$ reconstructed position (view space) from fragment's screen-space $X, Y$ position
Construct $TBN$ matrix and incorporate random rotation around the tangent-space $Z$-axis
$AO \leftarrow 0$
**for** sample $s_i$ where $i = 0$ to $N$ **do**
    Sample $s_{view} \leftarrow$ view-space position of $s_i$ using $TBN$ matrix
    $s_{screen} \leftarrow$ screen-space position of $s_{view}$
    $AO_{max} \leftarrow 0$
    depths[] $\leftarrow$ all stochastic-depth values at the screen-space $X, Y$ location of $S_{screen}$
    **for** each depth value $k$ in depths[] **do**
        Depth $d \leftarrow$ view-space depth reconstructed using $k$
        **if** $d \geq s_{view}.z$ **then**
            $AO_{max} \leftarrow max(AO_{max},\ smoothstep(0, 1, \frac{R}{||p.z - d||}))$
    $AO \leftarrow AO + AO_{max}$
$AO \leftarrow 1 - \frac{AO}{N}$
**return** $AO$

(a) Only use stochastic depth if sample is outside of the radius.

(b) Only use stochastic depth if sample is further than $0.3 \cdot R$.

(c) Always use stochastic depth.

Figure 6.3: Result when only using the stochastic-depth buffer (8 samples) when the sample from the regular depth buffer would be outside of the radius or further than $0.3 \cdot R$ from point $p$. Note the discontinuity artifacts where this technique switches from the regular depth buffer to the stochastic-depth buffer.

buffer would lie outside of the radius. Although this drastically sped up the ambient-occlusion computation (see Section 9.3 where the performance improvements are quantified), it brought many visual artifacts with it (as seen in Figure 6.3a). Due to the falloff term present in horizon-based ambient occlusion to penalize samples far away, in order to reduce large discontinuities in the ambient occlusion between neighbouring pixels, a sample $s_0$ with a smaller horizon angle than sample $s_1$ could actually result in more occlusion if it is much closer to point $p$. This meant that borders became visible where we switch from the regular depth buffer to the stochastic-depth buffer. The samples from the regular depth buffer were still in the radius, but their contribution to the ambient occlusion became minimal due to the falloff term, while the neighbouring pixel would use the stochastic-depth buffer.

We found that only using the stochastic-depth buffer when the sample from the regular depth buffer is further than $x \cdot R$ from point $p$ (with $x$ between 0 and 1, we found $x = 0.3$ to work well), can remove these border artifacts (Figure 6.3b), while still providing noticeable performance improvements. Given that samples from the regular depth buffer provide the largest horizon angle and are only slightly penalized by the falloff term when they are at most $0.3 \cdot R$ away from point $p$, this will almost always result in the correct ambient occlusion. Using only the regular depth buffer for samples farther than $0.3 \cdot R$ from $p$ introduces the risk of underestimating ambient occlusion compared to using the stochastic-depth buffer, with this risk increasing the farther away the sample is from $p$. Here $0.3 \cdot R$ was chosen conservatively, to preserve the visual quality and remove any transition artifacts (at the cost of the overall performance improvement), switching to the more accurate, but also more expensive, stochastic-depth samples at the slightest risk of underestimation. Linear interpolation (or other smooth transition schemes) to increase this range and improve the performance do not make sense, as they would require us to compute the ambient occlusion with the regular and stochastic-depth buffer, removing the performance benefit of skipping the ambient-occlusion computation with the stochastic-depth samples, without improving the visual quality over just using the more accurate stochastic-depth ambient-occlusion results.

Instead of only looking at the distance from $p$, we can also use the sample from the regular depth buffer to determine if we can skip iterating over the stochastic-depth samples. There are cases where we already know that none of the stochastic-depth samples would contribute to the final ambient occlusion, given the result of the from the regular depth buffer, making use of the fact that this sample is of the frontmost depth layer. In horizon-based ambient occlusion, we determine for each sample $s$ the angle between the vector $s-p$ and the $XY$-plane and only use the sample if the angle is larger than the current horizon angle $h$. Of all the depth samples in the same pixel, the one closest to the camera will have the largest angle with the $XY$-plane. If the frontmost sample (the one from the regular

(a) Only use stochastic depth if sample is further than $0.3 \cdot R$ with angle skip.

(b) Always use stochastic depth with angle skip.

(c) Always use stochastic depth without angle skip.

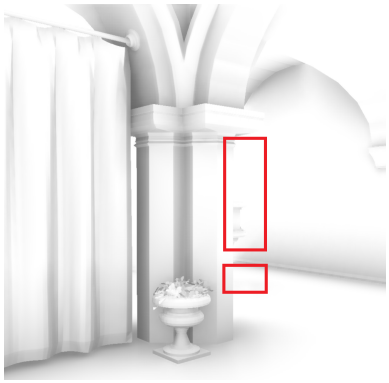
Figure 6.4: With the angle skip approach, performance is greatly improved without impacting the visual quality.

depth buffer) already has an angle smaller than $h$, we can skip iterating over all the stochastic-depth samples as none of them will contribute to the final ambient occlusion. This "angle skip" greatly improves performance without any noticeable impact in quality (Figure 6.4). While this approach is based on how horizon-based ambient occlusion computes the ambient occlusion, similar shortcuts could be used in HBAO+ and SSAO, where we know that none of the stochastic-depth samples would contribute to the ambient occlusion, if the sample from the regular depth buffer is already behind $p$.

## 6.4. Revisit Failure Case

In Section 4.5, we discussed a failure case of traditional horizon-based ambient occlusion with regards to obstructed geometry. In that scenario the geometry in front (i.e., the pillar) was outside of the radius and the geometry behind it was not known in screen space, leading to missing occlusion. When we use stochastic-depth ambient occlusion in this scenario (Figure 6.5), we can see that it is able to capture the previously missing occlusion, looking very similar to its ray-traced counterpart. With stochastic-depth ambient occlusion, moving the camera around does not cause occlusion to suddenly disappear, as was the case with traditional screen-space ambient occlusion techniques when the geometry causing the occlusion became hidden from view.

Figure 6.6 shows how we march over the geometry with stochastic-depth ambient occlusion, using a stochastic-depth buffer with two samples. Each texel of the stochastic-depth buffer stores two depth values from random depth layers (differing per texel), with the preference of closer depth layers. Instead of only sampling the geometry of the frontmost depth layer at each step, stochastic-depth ambient occlusion looks at all samples from the stochastic-depth buffer and chooses the one that provides the maximum ambient occlusion. This means that we now have information about the geometry behind the pillar and are able to determine the correct horizon angle. Since the depth layers that are stored in the stochastic-depth buffer are chosen randomly, it could be that we sometimes miss the occlusion in certain pixels, resulting in noise. This noise is not all that different from the per-pixel noise already present in screen-space ambient occlusion techniques and is smoothed out with the Gaussian blur already in place.



(a) HBAO.

(b) Our technique (stochastic-depth ambient occlusion).

(c) Ray-traced ambient occlusion.

Figure 6.5: Stochastic-depth ambient occlusion is able to capture the missing occlusion and looks visually very similar to its ray-traced counterpart.

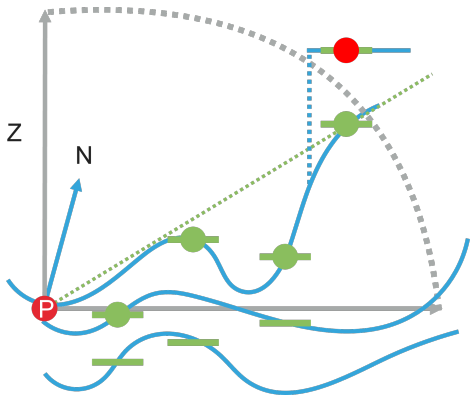



Figure 6.6: Stochastic-depth ambient occlusion stores $S$ depth samples of random depth layers per texel (in this example $S = 2$, visualized as green bars). When marching over the stochastic-depth buffer, we iterate over all depth samples in a texel and determine which one leads to the maximum ambient occlusion (marked with green dots). In this case, the sample of the obstructing geometry (i.e., the pillar) is ignored because it is outside of the radius $R$ (marked with a red dot), but we now have information regarding the geometry behind it. This allows us to correctly determine the maximum horizon angle.

# 7

# Extensions

In this chapter, we will highlight the extensions we made to the core stochastic-depth ambient occlusion algorithm discussed in Chapter 6. The extensions we discuss (bent normals and cones, and multi-view ambient occlusion), are popular extensions to traditional screen-space ambient occlusion techniques. However, since they build upon screen-space ambient occlusion, they have similar issues and view-dependent artifacts when information is missing in screen space. We show how these extensions can be generalized to work with stochastic-depth ambient occlusion, to improve upon their visual quality, stability and robustness.

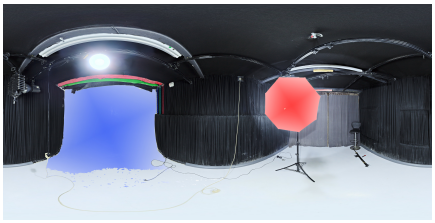
## 7.1. Bent Normals and Cones

Ambient occlusion modulates the amount of indirect light that is able to reach a certain point $p$, but it does not take the light's direction into account [27, 42]. While this works well in combination with ambient lighting, when using an environment map this approximation does not always result in realistic results, as light is still coming from occluded directions (see Figure 7.2). To take the directionality into account, one can compute a *bent normal* [30], which is a vector representing the average direction of the incoming light at point $p$ (as seen in Figure 7.3). We can use this bent normal instead of the regular surface normal to query the environment map (Figure 7.1), providing more natural looking indirect lighting. Furthermore, this can be extended to *bent cones* [27], a bent normal augmented with an angle. This allows a bent cone to capture the distribution of the unoccluded directions, by not only storing the average direction, but also the variance. This information is then used to remove the incoming light from occluded directions (i.e., directional occlusion).

A benefit of bent normals and bent cones is that they can be constructed during the ambient-occlusion computation, and that they easily integrate into the existing lighting pass. However, because they are constructed during the screen-space ambient occlusion pass, they will not be accurate when the underlying ambient-occlusion algorithm is underestimating or missing occlusions, due to missing information in screen space. This results in bent normals pointing in occluded directions or too large bent-cone angles that allow incoming light from occluded directions, when the occluding geometry is hidden from view. In these cases, stochastic-depth ambient occlusion is able to accurately compute the ambient occlusion, which in turn would allow for the construction of accurate bent normals and cones.

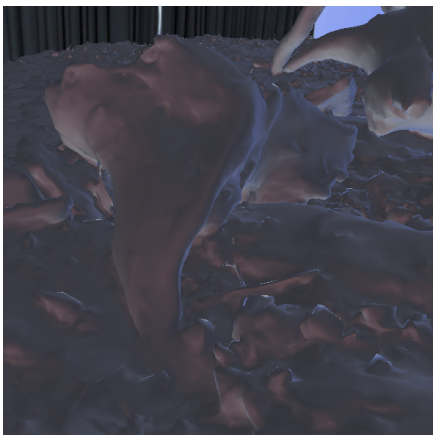
Our implementation of bent normals and cones is based on the approach given by [27], which mainly focuses on SSAO. To compute a bent normal $n'$ using SSAO, we determine a weighted average of the sample directions based on their visibility:

$$n' = \left( \sum_{i}^{N} V(s_i - p) \right)^{-1} \sum_{i}^{N} \frac{s_i - p}{||s_i - p||} V(s_i - p)$$

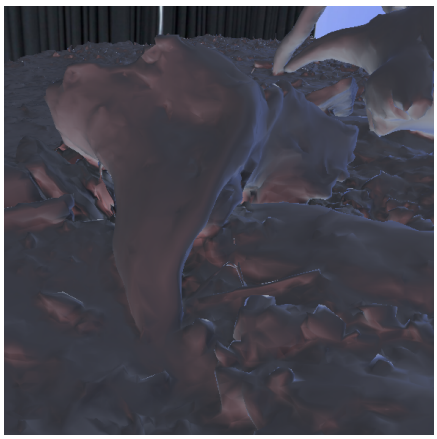


(a) The environment map.

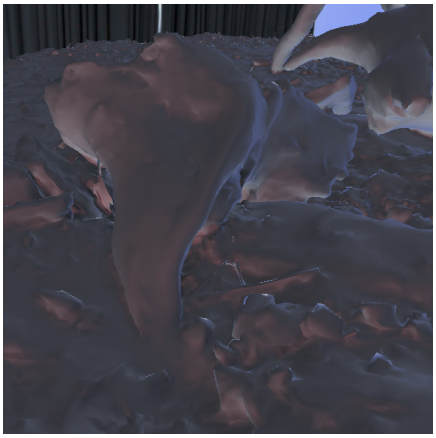
(b) Without bent normals.



(c) With bent normals (SDAO).



(d) With bent cones (SDAO).



(e) With bent cones (HBAO).

Figure 7.1: Using bent normals and cones, we are able to take the incoming light's direction into account, resulting in more natural looking indirect lighting. In this example we use the environment map seen in (a) with a red light behind and the blue wall towards the right of the camera, and a white floor. In the region marked with the red square (b), the normals point slightly downwards, resulting in the white incoming light. In contrast, the bent normals (c) point more toward the camera, resulting in less incoming white light and more red light. With bent cones (d), the regions with high occlusion only receive light in a narrow cone, further reducing the amount of white light reaching the marked region. Traditional HBAO (e) fails to capture the occlusion in the marked region, resulting in less accurate bent cones with more incoming white light. This scene was provided by Matousekfoto (SketchFab).

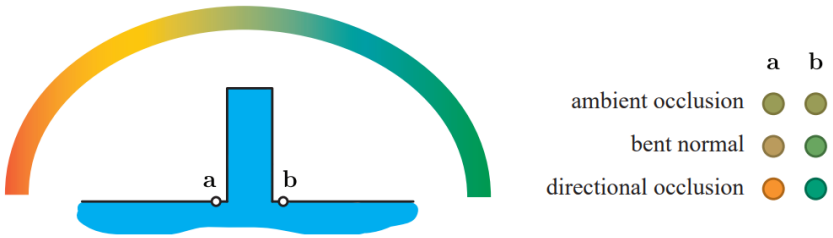



Figure 7.2: Ambient occlusion does not take the direction of the incoming light into account, it only modulates the intensity. Since $a$ and $b$ have a similar surface normal (pointing upwards), they will have the same color. A bent normal represents the average direction of the incoming light at point $p$, which can be used instead of the surface normal to query the environment map, resulting in a different color at $a$ and $b$. Directional occlusion techniques, such as bent cones, go one step further and eliminate light coming from occluded directions, creating a more realistic image. Image from [2].

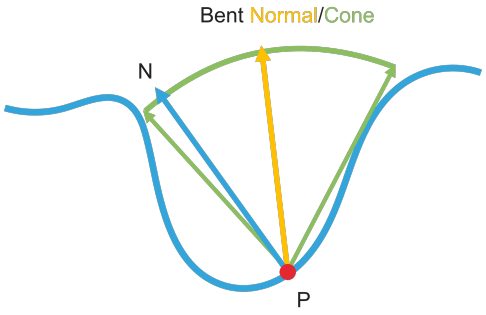



Figure 7.3: In comparison to the surface normal $n$, a bent normal (indicated in orange) points towards the average direction of the incoming light at point $p$. Bent cones (indicated in green) go one step further and capture the distribution of the unoccluded directions, which allows us to eliminate incoming light from occluded directions.

where $V(\omega) = 1 - AO(\omega)$ is the visibility in direction $\omega$, which is equal to 1 minus the occlusion in direction $\omega$. Here $n'$ is not normalized, but instead its length is used to determine the angle $c$ of the bent cone:

$$c = \frac{\pi}{2}\left(1 - max(2||n'|| - 1, 0)\right)$$

Furthermore, instead of storing the bent normals directly, we store the difference between the regular reconstructed normal and the bent normal: $n' - n$. This difference is then added to the high-frequency normal from the normal texture used in the lighting pass to preserve details. For stochastic-depth ambient occlusion, the only part that changes is the visibility function $V(\omega)$, the rest of the bent normal and cone computation remains the same. Algorithm 10 shows how this bent normal/cone computation integrates into the ambient-occlusion pass for stochastic-depth SSAO.

While [27] mentions an HBAO implementation, they do not provide any details. Our implementation for HBAO uses the horizon angles found for each direction that we march in to determine the bent normal and cone. For each direction $d_i$ that we march in, we compute a vector $n_r$, which is the reconstructed normal rotated around the axis $d_i \times (0,\ 0,\ -1)$ by the horizon angle $h$ (to rotate the normal away from the vector giving us the horizon angle $h$, in the plane given by the direction $d_i$ and the negative $Z$-axis pointing from the camera into the scene, as seen in Figure 7.4). Here we pad the two dimensional vector $d_i$ with an 0 in the $Z$ dimension, to make it three dimensional. Since $n_r$ indicates the average direction of incoming light for the marched direction, we can average the $n_r$ vectors from all directions to determine the bent normal. In a similar fashion, we compute the bent cone angle for each direction and average them together as:

$$\frac{1}{N_d}\sum_{i}^{N_d}\frac{\pi}{2} - h_i$$

**Algorithm 10** Stochastic-Depth SSAO with bent cones

Position $p \leftarrow$ reconstructed position (view space) from fragment's screen-space $X, Y$ position
Normal $n \leftarrow$ reconstructed normal (view space) at $p$
Construct $TBN$ matrix and incorporate random rotation around the tangent-space $Z$-axis
$AO \leftarrow 0$
Bent normal $n' \leftarrow (0,\ 0,\ 0)$
$U \leftarrow 0$
**for** sample $s_i$ where $i = 0$ to $N$ **do**
    Sample $s_{view} \leftarrow$ view-space position of $s_i$ using $TBN$ matrix
    $s_{screen} \leftarrow$ screen-space position of $s_{view}$
    $AO_{max} \leftarrow 0$
    depths[] $\leftarrow$ all stochastic-depth values at the screen-space $X, Y$ location of $S_{screen}$
    **for** each depth value $k$ in depths[] **do**
        Depth $d \leftarrow$ view-space depth reconstructed using $k$
        **if** $d \geq s_{view}.z$ **then**
            $AO_{max} \leftarrow max(AO_{max},\ smoothstep(0, 1, \frac{R}{||p.z-d||}))$
    $AO \leftarrow AO + AO_{max}$
    $V \leftarrow 1 - AO_{max}$
    $n' \leftarrow n' + normalize(s_{view} - p) \cdot V$
    $U \leftarrow U + V$
$AO \leftarrow 1 - \frac{AO}{N}$
$n' \leftarrow \frac{n'}{U} - n$ where $n'$ and $n$ are both in world space
Bent cone angle $c \leftarrow \frac{\pi}{2}\left(1 - max(2||n'|| - 1, 0)\right)$
**return** $AO$, $n'$ and $c$

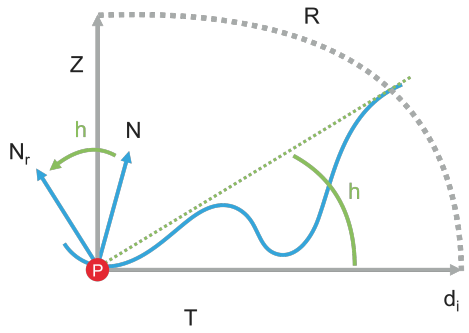
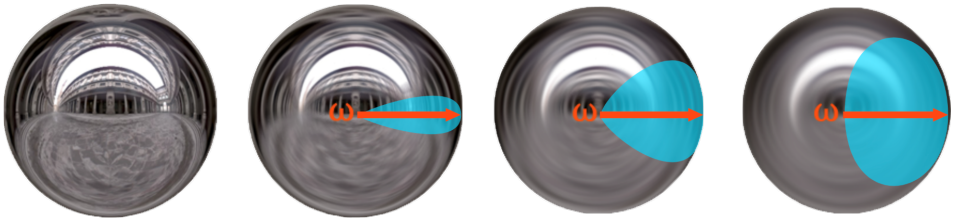



Figure 7.4: With HBAO, we compute for each direction the vector $n_r$ by rotating the normal by the horizon angle $h$ and we average these together to form the bent normal.



(a) Environment map. (b) Irradiance map with $\alpha = 10$. (c) Irradiance map with $\alpha = 45$. (d) Irradiance map with $\alpha = 90$.

Figure 7.5: We generate multiple irradiance maps, were we only look at light inside a cone with angle $\alpha$ during the pre-convolution step. Image from [27].

where $h_i$ is used to denote the horizon angle in direction $d_i$. The process is similar for HBAO+, where the $\frac{v_j \cdot n}{||v_j||} - bias$ term can be used to determine the angle with the $XY$-plane. We keep track of the largest value $\lambda = max_j(\frac{v_j \cdot n}{||v_j||} - bias)$ we find and compute the angle with the $XY$-plane as:

$$\frac{pi}{2} - \cos^{-1}(\lambda)$$

which we use similarly to the horizon angle $h$. Since the algorithms are very similar, we will only show the pseudocode for HBAO (Algorithm 11).

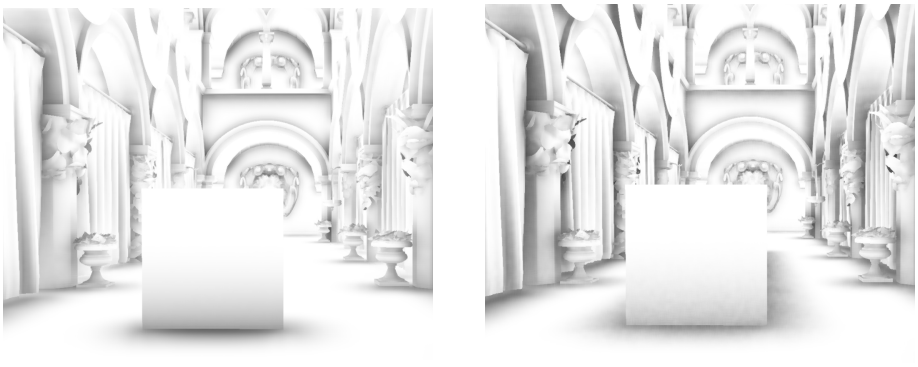
As described by [27], the bent normal and cone can be used during the existing lighting pass. The bent normal is used, instead of the regular surface normal, to query the irradiance map. This gives us the light coming from the correct direction, but does not yet take the cone angle into account. For this, the pre-convolution step is modified to generate multiple irradiance maps, each with a different sized cone of incoming light, with an angle $\alpha$ up to 90 degrees (as seen in Figure 7.5). Up to $K$ irradiance maps are generated, by discretizing $\alpha$ into $K$ steps (we use $K = 8$). During the lighting pass, the irradiance corresponding to the bent cone angle $c$ can be determined, by linearly interpolating between the different irradiance maps.

## 7.2. Multiple Viewpoints

With traditional screen-space ambient occlusion algorithms, there were three issues resulting in underestimated or missing occlusion: 1. objects are outside of the view frustum, 2. objects are viewed under a grazing angle and are thus not visible to the camera and 3. objects are behind other geometry and hidden from the camera. Of these three problems, stochastic-depth ambient occlusion solves the issue with obstructed geometry. By using slightly larger depth buffers that includes the geometry just outside of the view frustum, the first problem can also be solved. But the problem related to grazing angles remains.

**Algorithm 11** Stochastic-depth HBAO with bent cones

Position $p \leftarrow$ reconstructed position (view space) from fragment's screen-space $X, Y$ position
Normal $n \leftarrow$ reconstructed normal (view space) at $p$
$R' \leftarrow$ projected radius $R$ onto image plane
Determine stepsize as $R'/(N_s+1)$
Determine directions with random offset
$AO \leftarrow 0$
Bent normal $n' \leftarrow (0,\ 0,\ 0)$
Bent cone angle $c \leftarrow 0$
**for** direction $d_i$ where $i = 0$ to $N_d$ **do**
    Determine tangent vector $T$
    Determine tangent angle $t$ of $T$ with $XY$-plane
    Horizon angle $h \leftarrow t + bias$
    Randomly offset first step
    **for** step $s_j$ where $j = 0$ to $N_s$ **do**
        $AO_{max} \leftarrow 0$
        $h_{max} \leftarrow 0$
        $m_{viewH} \leftarrow$ partial view position reconstruction at step $j$ in direction $d_i$
        depths[] $\leftarrow$ all stochastic-depth values at step $j$ in direction $d_i$
        **for** each depth value $k$ in depths[] **do**
            Sample $s \leftarrow$ reconstructed view-space position using $m_{viewH}$ and depth $k$
            $v \leftarrow s - p$
            **if** $\frac{v \cdot n}{||v||} >$ angle threshold **then**
                Determine angle $\phi$ of $v$ with $XY$-plane
                **if** $\phi > h$ and $||v|| < R$ **then**
                    $AO_k \leftarrow W(||v||)(sin(\phi) - sin(h))$
                    **if** $AO_k > AO_{max}$ **then**
                        $AO_{max} \leftarrow AO_k$
                        $h_{max} \leftarrow \phi$
        **if** $AO_{max} > 0$ **then**
            $AO \leftarrow AO + AO_{max}$
            $h \leftarrow h_{max}$
    $h \leftarrow h - (t + bias)$ ▷ We initialized $h$ with the tangent angle $t$
    $n_r \leftarrow rotate(n,\ h,\ vec3(d_i, 0) \times (0,\ 0,\ -1))$
    $n' \leftarrow n' + n_r$
    $c \leftarrow c + \frac{\pi}{2} - h$
$AO \leftarrow 1 - \frac{AO}{N_d}$
$n' \leftarrow \frac{n'}{N_d} - n$ where $n'$ and $n$ are both in world space
$c \leftarrow \frac{c}{N_d}$
**return** $AO$, $n'$ and $c$



(a) HBAO+.

(b) Multi-view SDAO.

Figure 7.6: With traditional HBAO+, the sides of the rectangle are under a grazing angle with the main camera, resulting in missing ambient occlusion. Using multi-view stochastic-depth ambient occlusion, we can correctly capture this missing occlusion.

To solve the issue with grazing angles, an additional camera that is able to see this geometry is used [42, 51]. By computing the ambient occlusion with both cameras and again choosing the maximum, we are able to capture the occlusion from geometry under a grazing angle with the main camera. Ideally one would use a camera with a completely different viewpoint, for example a side view. However, this comes with the problem of occlusion: in many scenes we can not simply place a camera and expect to see the same geometry as the main camera, oftentimes there is other geometry obstructing the visibility. For this reason, multiple additional cameras are used and special care needs to be taken on where to place them (often placed manually per scene). If however, we would have a camera that is able to "see through" the geometry, we could use a single additional camera that is always at the same position relative to the main camera.

This is where stochastic-depth ambient occlusion comes in. By rendering an additional stochastic-depth buffer at the secondary camera position, we can accurately compute the ambient occlusion even if geometry would normally obstruct the camera's view. This means that we can place the secondary camera exactly where we would want it and use the same location for every scene, without having to take visibility into account. In this way, stochastic-depth ambient occlusion could thus also help with the third and final problem of screen-space ambient occlusion techniques.

We have implemented support for a secondary camera using stochastic-depth ambient occlusion with the HBAO+ algorithm (as shown in Figure 7.6). An additional stochastic-depth buffer is rendered from the viewpoint of the secondary camera, which is positioned relative to the main camera. In the stochastic-depth ambient occlusion algorithm, we use the regular depth buffer (from the main camera) to determine the positions $p$ we want to compute the ambient occlusion for. We then compute the ambient occlusion with both stochastic-depth buffers and use the maximum. To compute the ambient occlusion with the secondary stochastic-depth buffer, we first transform point $p$ and its normal $n$ to the view space of the secondary camera and then project them to determine their screen-space position on the secondary camera. This means that we know the location of $p$ in the secondary stochastic-depth buffer and we can march over the geometry surrounding it. We allow the user to render the secondary stochastic-depth buffer at half the resolution (a quarter of the pixels) to reduce the performance impact, while still providing similar visual quality.

We use HBAO+ for our implementation of multi-view stochastic-depth ambient occlusion, since it only requires a position $p$ and a normal $n$, and not an additional tangent vector like HBAO. With HBAO a tangent vector is constructed following the screen-space direction that we march into, using the geometry surrounding $p$ (i.e., the neighbouring texels around $p$ in the regular depth buffer). We can not transform this tangent vector from the view space of the main camera to that of the secondary camera, as then it would not match the screen-space direction that we march in with the secondary

camera. It can also be quite difficult to construct such a tangent vector for the secondary camera, as it only uses a stochastic-depth buffer, where neighbouring texels do not necessarily contain the same geometry. By using HBAO+ we circumvent these issues.

To transform position $p$ from the main camera's view space ($p^0_{view}$) to the secondary camera's view space ($p^1_{view}$), we first transform it into world space ($p_{world}$):

$$p_{world} = \left( (M^0_{view})^{-1} \cdot \begin{bmatrix} p^0_{view}.x \\ p^0_{view}.y \\ p^0_{view}.z \\ 1 \end{bmatrix} \right).xyz$$

where $(M^0_{view})^{-1}$ is the inverse view matrix of the main camera. Then we multiply $p_{world}$ with the view matrix of the secondary camera ($M^1_{view}$) to get the view-space position $p^1_{view}$:

$$p^1_{view} = \left( M^1_{view} \cdot \begin{bmatrix} p_{world}.x \\ p_{world}.y \\ p_{world}.z \\ 1 \end{bmatrix} \right).xyz$$

Then to get $p$'s screen-space position for the secondary camera, we project $p^1_{view}$ using its projection matrix $M^1_{proj}$ to get the clip-space position $p^1_{clip}$:

$$p^1_{clip} = M^1_{proj} \cdot \begin{bmatrix} p^1_{view} \\ p^1_{view} \\ p^1_{view} \\ 1 \end{bmatrix}$$

that can easily be transformed into the screen-space position $p^1_{screen}$:

$$p^1_{screen} = \left( \frac{\frac{p^1_{clip}.x}{p^1_{clip}.w} + 1}{2}, \quad \frac{\frac{p^1_{clip}.y}{p^1_{clip}.w} + 1}{2} \right)$$

The process to transform normals is similar, except that for the multiplications with the view matrix and the inverse view matrix, we set the $w$ component to 0 instead of 1 to avoid translating the normal vector. This results in the pseudocode given by Algorithm 12.

**Algorithm 12** Multi-view stochastic-depth HBAO+

Position $p \leftarrow$ reconstructed position (view space) from fragment's screen-space $X, Y$ position
Normal $n \leftarrow$ reconstructed normal (view space) at $p$
$AO \leftarrow 0$
**for** each camera $l$ **do**
    $AO \leftarrow max(A0, ComputeAO(p, n, l))$
$AO \leftarrow 1 - \frac{AO}{N_d \cdot N_s}$
**return** $AO$

**function** COMPUTEAO($p, n, l$)
    Position $p_k \leftarrow p$ transformed to the view space of camera $l$
    Screen-space position $p_{screen} \leftarrow p_k$ projected to the screen space of camera $l$
    Normal $n_k \leftarrow n$ transformed to the view space of camera $l$
    Determine step size as $R'/(N_s + 1)$
    Determine directions with random offset
    $AO \leftarrow 0$
    **for** direction $d_i$ where $i = 0$ to $N_d$ **do**
        Randomly offset first step
        **for** step $s_j$ where $j = 0$ to $N_s$ **do**
            $AO_{max} \leftarrow 0$
            $m_{viewH} \leftarrow$ partial view position reconstruction (for camera $l$) at step $j$ in direction $d_i$
            depths[] $\leftarrow$ all stochastic-depth values at step $j$ in direction $d_i$ from stochastic-depth texture $l$
            **for** each depth value $k$ in depths[] **do**
                Sample $s \leftarrow$ reconstructed view-space position using $m_{viewH}$ and depth $k$
                $v \leftarrow s - p$
                $AO_{max} \leftarrow max(AO_{max},\ W(||v||)max(\frac{v \cdot n}{||v||} - bias, 0))$
            $AO \leftarrow AO + AO_{max}$
    **return** $AO$

# 8

# Implementation

This chapter discusses our the implementation details of the techniques and algorithms from the previous chapters. We have implemented the three screen-space ambient occlusion techniques (SSAO, HBAO and HBAO+), stochastic transparency and, of course, stochastic-depth ambient occlusion with its extensions bent normals/cones and multi-view stochastic-depth ambient occlusion.

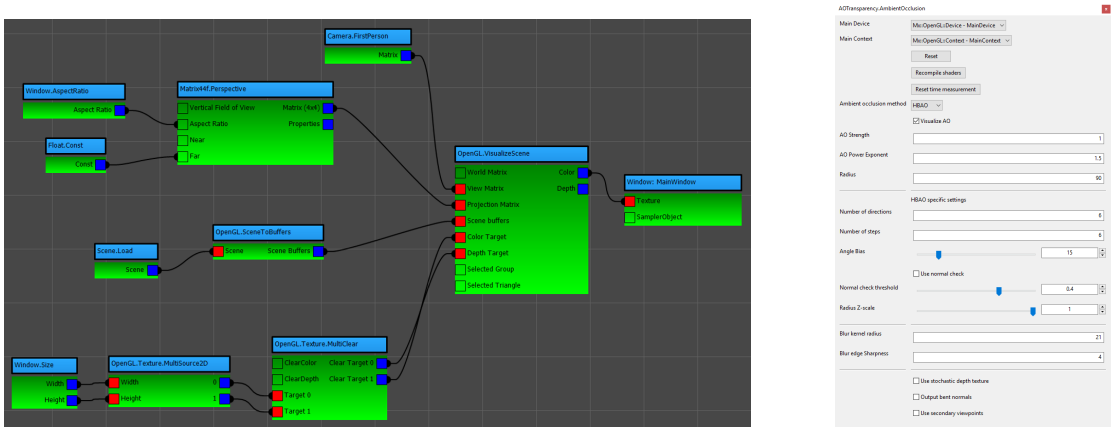
For our implementations, we used the *MxEngine*, a rendering engine developed by the Computer Graphics and Visualization group at the Delft University of Technology. The MxEngine and our implementations are written in *C++* and *OpenGL*. The MxEngine features a node-based editor (Figure 8.1a), where each step of the rendering pipeline is represented as a node, called a *filter*. A filter can have input and output ports to transfer data to other filters. Each filter also has their own UI to expose tweakable parameters to the user (Figure 8.1b), allowing for on-the-fly adjustments. The MxEngine already has many filters for different light sources, shadow maps, PBR shading, etc., giving a nice foundation to work on.

## 8.1. Screen-Space Ambient Occlusion

We implemented the three screen-space ambient occlusion techniques discussed in Chapter 4 (i.e., SSAO, HBAO and HBAO+) into a single MxEngine filter. The screen-space ambient occlusion techniques are implemented as fragment shaders, following the respective pseudocode. This filter allows the user to easily switch between the different ambient-occlusion algorithms and exposes all the tweakable parameters to the user. The filter takes as input the depth buffer (used to determine the screen-space position) and the projection matrix (used to reconstruct the view positions from the screen-space positions or to project view-space positions to screen space in SSAO).

Screen-space ambient occlusion works best in a deferred rendering pipeline, where we first render a G-buffer (i.e., a collection of textures containing all the kinds of information about the geometry, relevant to later lighting passes: positions, normals, albedo, depth, etc) and use these textures during later passes, without rendering the geometry again. For ambient occlusion this means that we can use the depth buffer already available to us. In a forward rendering pipeline (like the MxEngine, when using the PBR lighting filter *visualizeScenePBR*), where we do not use a G-buffer, but render the geometry and use the information from the vertex shader directly, screen-space ambient occlusion still works, but we need to render a depth texture first. For this reason we created an additional depth filter that renders a depth buffer, which we can then be used in the ambient-occlusion filter.

Since ambient occlusion is not a physically-accurate technique, there are a number of parameters to tweak the result more to your liking. First there is the radius parameter $R$, indicating how large the normal-oriented hemisphere is. Increasing $R$ means looking at geometry in a larger radius around $p$, which can come at the cost of performance due to more sparse depth buffer reads. Furthermore,




(a) MxEngine's editor.

(b) MxEngine's filter UI.

Figure 8.1: The MxEngine features a node-based editor, where each step of the rendering pipeline is represented as a filter (i.e., a node). Each filter can expose several tweakable parameters, making on-the-fly adjustments possible.

these algorithms also have a multiplier (the ambient occlusion strength) and exponent (the ambient occlusion exponent) that the user can tweak to increase or decrease the final ambient occlusion:

$$AO = (AO \cdot AOstrength)^{AOexponent}$$

### Per-Pixel Randomness and Gaussian Blur

To add per-pixel randomness, we use the hashing function described by [52], which we seed by the the screen-space $XY$-coordinates. For the blur, we use a 1D cross-bilateral Gaussian filter, that is applied separately in the $X$ and $Y$ direction similar to [6, 8]. Such a cross-bilateral filter is a weighted average of the ambient occlusion in a kernel, where the weights depend on the pixel location in the kernel and also on their corresponding depth value:

$$AO_{blur} = \frac{\sum_{i=-k}^{k} AO_i \, w(i, d_i, d_0)}{\sum_{i=-k}^{k} w(i, d_i, d_0)}$$

where $k$ is the kernel size, $w()$ is the weight function and $d_i$ the depth value corresponding to pixel $i$ in the kernel. We use a cross-bilateral weight similar to that of [6], where the depth of the current pixel is compared to the depth value of the centermost pixel in the kernel. This approach preserves edges during the blur, with the assumption that different surfaces will have a different depth.

In addition we slightly offset the depth values along the tangent direction with a small bias according to the depth slope at the kernel's center, similar to NVIDIA's blur for their HBAO+ algorithm [4]. We compute the depth slope of as $s = d_{-1} - d_0$ or $s = d_1 - d_0$ (used for the left and right half of the kernel respectively) and decrease the depth for each pixel $i$ in the kernel by $s \cdot |i|$. This depth slope bias term greatly reduces the amount of noise in the image, while still preserving the real edges.

## 8.2. Stochastic Transparency

We have implemented a stochastic transparency filter that is based on the `VisualizeScenePBR` filter. This filter allows us to test our implementation of stochastic transparency, which we will use for our stochastic-depth ambient occlusion algorithm. It renders the opaque geometry and transparent geometry to separate textures, that are composited at the end. The opaque geometry is rendered just like in the `VisualizeScenePBR` filter. The transparent geometry is rendered using the stochastic transparency algorithm as described in Chapter 5, where the color of the transparent fragments is computed in a similar fashion as the opaque geometry. We allow the user to select the number of MSAA samples to use (from 1 to 16 samples).

(a) Without alpha testing. (b) With alpha testing. (c) Corresponding alpha texture.

Figure 8.2: Leaves and plants are often made up of simple geometry (e.g., planes) that are cut out using their alpha texture with alpha testing. In this alpha texture the color value represents the alpha value, i.e., black means $\alpha = 0$, while white means $\alpha = 1$.

Special care has to be taken with the alpha testing, used to render objects such as leaves accurately. Alpha testing allows us to discard fragments based on the alpha value stored in their alpha texture. This makes it possible to use very simple geometry (simple planes) for leaves, and cut out the actual leaf using the alpha texture (Figure 8.2). Without alpha testing the leaves would render as opaque planes (as was the case previously with the MxEngine since the alpha textures were not stored correctly into the materials, this work fixes that), but we do not want to discard actual transparent geometry. For this reason we chose to disable alpha testing if the object was fully transparent (e.g., not an opaque object with an alpha texture). In this way transparent objects with an alpha value below the alpha testing threshold (0.1) are not discarded, while opaque fragments with an alpha value below this threshold (from its alpha texture) are still discarded.

For the stochastic depth we implemented both the simple and stratified sampling approach and we allow the user to switch between them. We use the same random number generator as described in [52], which we seed by the fragment's screen-space $X, Y$-position multiplied by its $Z$ coordinate, since the fragments falling in the sample pixel should have a different hash value. We multiply this by a random seed based on the current CPU time to change the randomness each frame (we allow the user to fix this time seed to reduce flickering). In addition, we add the ID of the MSAA sample to this time seed, so the that outcomes for each sample are independent of each other.

Furthermore, we implemented the more sophisticated hashing/sampling approaches discussed in [54] and allow the user to switch between them. This includes a 4D hash (based on their 3D hash, but hashed again with the time seed and MSAA sample ID), the isotropic alpha threshold and the anisotropic alpha threshold. The 3D and 4D hash is similar to the 2D hashing function from [52], but applied recursively:

$$hash3D(X,Y,Z) = hash2D(hash2D(X,Y),Z)$$
$$hash4D(X,Y,Z,M) = hash2D(hash3D(X,Y,Z),M)$$

for the screen-space coordinates $X, Y, Z$ and the MSAA sample ID $M$. The more sophisticated isotropic alpha threshold and anisotropic alpha threshold focus on providing better spatial and temporal stability, rather than trying to improve upon the visual quality. They ensure that the randomness is anchored to the geometry and does not change when moving the camera around, by basing the hash on discretized coordinate derivatives instead of the coordinates themselves. The anisotropic alpha threshold is an extension to the isotropic alpha threshold, which discretizing the coordinate derivations independently per axis, in order to reduce the anisotropy when a surface is viewed obliquely (which otherwise reintroducing temporal instability or results in elongated regions with a similar hash value). As suggested by the authors, our implementation uses world-space

coordinates instead of screen-space coordinates for these hashes (but this can easily be modified to object-space coordinates).

## 8.3. Stochastic-Depth Ambient Occlusion

For stochastic-depth ambient occlusion we implemented a filter that render and outputs the stochastic-depth buffer. This stochastic-depth filter is largely based on the stochastic transparency filter described in Section 8.2, but is purely focused on rendering the stochastic-depth buffer. It uses the same shader as the stochastic transparency filter for the stochastic-depth buffer, but now uses the $\alpha$ value (chosen by the user) to render opaque geometry as if it is transparent. The filter can easily be extended to output more information about the geometry (we already support the output of normals) in separate multi-sampled color buffers.

Furthermore, we modify the ambient-occlusion filter discussed in Section 8.1 to take this stochastic-depth buffer as input. We allow the user to easily toggle between the regular screen-space techniques and their stochastic-depth counterparts. Since only the ambient-occlusion computation itself changes, the same tweakable parameters are exposed and the rest of the pipeline stays the same (except that we now have an additional pass to render the stochastic-depth texture).

The ambient-occlusion filter is further extended to output the bent normals (more specifically, the difference between the bent normal and the reconstructed surface normal) and bent-cone angles to a separate texture. This works in combination with the traditional screen-space ambient occlusion techniques and their stochastic-depth counterparts. We also modified the pre-convolution step and the lighting pass as mentioned in Section 7.1. However during this implementation, we noticed that the physically-based rendering (PBR) indirect lighting pass of the MxEngine was only half finished. We fixed and finished this implementation by computing and using a BRDF lookup texture, used to compute the specular component [26]. We now also determine the amount of refracted (diffuse) light, based on the amount of light that is not reflected (specular) and the material's metallic value, instead of only on the metallic value, to conserve the incoming light's energy [14].

For multi-view stochastic-depth ambient occlusion, we render and output a secondary stochastic-depth buffer in the stochastic-depth filter, which we then use in the ambient-occlusion filter. The secondary camera is positioned relative to the main camera, slightly to the right and in front of the main camera, angled towards the center of the main camera's view frustum. For simplicity, we use a secondary camera with the same settings as the main camera (e.g., the same focal length, viewport height/width and near/far plane distances), so that we only need to provide a new view matrix and can reuse the projection matrix of the main camera during the ambient-occlusion computation.

# 9 Evaluation

To evaluate our proposed method, stochastic-depth ambient occlusion (SDAO), we compare it to its traditional screen-space ambient occlusion counterparts and ray-traced ambient occlusion. We focus on three main aspect: accuracy, performance and robustness. The aim of stochastic-depth ambient occlusion is to improve upon traditional screen-space ambient occlusion techniques in terms of accurately capturing the occlusion of hidden geometry, to more closely match ray-traced ambient occlusion. However, we want to keep the main advantage of screen-space ambient occlusion techniques: their performance that allows them to be used in real-time applications. By testing our method in multiple scenarios, we determine how robust stochastic-depth ambient occlusion is or whether there are any failure cases.

The images for the traditional screen-space ambient occlusion techniques or the stochastic-depth counterparts are rendered using the MxEngine. Unfortunately the MxEngine did not contain a ray tracing pipeline, therefore we used Blender[1] (version 2.82), a very popular open source 3D modeling program, for the ray-traced ambient occlusion renders. We use Blender's Cycles renderer, which is a physically-based path tracer that is able to render ray-traced ambient occlusion[2] as described in Chapter 2. The view point from the blender images do not exactly match those from the MxEngine (due to slightly different camera settings and placement), but are close enough to make direct comparisons possible.

## 9.1. Impact of the Number of Stochastic-Depth Samples

The main difference between stochastic-depth ambient occlusion and traditional screen-space ambient occlusion is that we have $S$ depth samples, instead of only 1. The depth layers that we store for each texel are decided stochastically and differ per texel, two neighbouring texels do not necessarily store the depth values of the same depth layers. The more depth values we store, the more likely it is that we store the depth layer with the maximum ambient occlusion. However, increasing the number of samples increases the memory and bandwidth requirements to read/write/store the depth values, the amount of MSAA samples we need to render the stochastic-depth buffer, and the amount of depth samples that we need to iterate over in the ambient-occlusion shader. With fewer depth samples, the performance penalty of stochastic depth is reduced but the chance that we capture the depth layer with the maximum ambient occlusion goes down.

The depth layers captured in the stochastic-depth buffer differ per texel, which means that neighbouring positions in the ambient-occlusion computation can have very different results. This is visible in the form of noise, with some pixels in the ambient-occlusion texture capturing the missing

[1] `https://www.blender.org/`

[2] `https://docs.blender.org/manual/en/latest/render/shader_nodes/input/ao.html`

occlusion, while others do not. By applying a Gaussian blur to the ambient-occlusion texture, this noise is removed and the occlusion is spread out across the pixels. If only a small number of pixels capture the correct ambient occlusion, the ambient occlusion gets washed out.

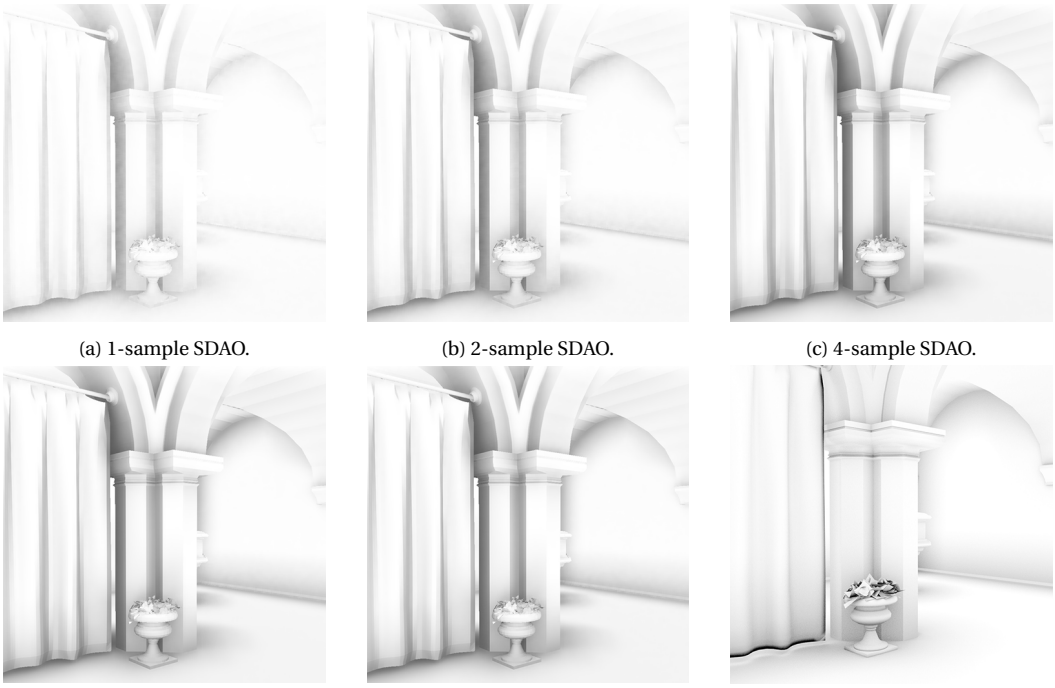
Figure 9.1 showcases stochastic-depth ambient occlusion with a different amount of stochastic samples. In this example we march in 4 directions with 4 steps, use stratified sampling and use $\alpha = 0.2$ for the stochastic depth (we will use these same settings throughout our evaluation, unless stated otherwise). Here we can see that even with only 1 sample, we are able to capture the otherwise missing occlusion, but the whole image looks washed out. With more samples, we are not able to capture more missing occlusions, but we find the occlusions more often, resulting in a far stronger occlusion estimate. An important observation is that all ambient occlusion becomes more washed out, not only the otherwise missing occlusion. This is because we compute the ambient occlusion using only the stochastic-depth buffer, which can sometimes miss the frontmost depth layer. This could be resolved by including the regular depth samples in the ambient-occlusion computation, but in this case it results in some nice properties: the strength of the occlusion is consistent across the image, allowing us to increase the overall ambient-occlusion strength (using the $AO_{strength}$ and $AO_{exponent}$ terms described in Section 8.1) to get a visually similar looking image to one with more depth samples. Figure 9.2 shows this in action, where the images for lower sample counts are now very similar to those with a higher sample count, albeit with more noise.

This consistency in occlusion strength is related to the Monte Carlo nature of the problem, where repeated sampling makes the probability of finding the depth layer giving us the maximum occlusion similar for layers at a different depth order, eventually converging to the correct result. The goal of stochastic-depth ambient occlusion is to find the largest horizon angle in each direction that we march in, but it could be that the depth layer giving the largest horizon angle is missing from certain pixels. The probability $P_j$ that a depth layer $j$ is captured in pixel of the stochastic depth buffer depends on its depth order (i.e., the number of depth layers in front of layer $j$), the $\alpha$ at which we render the stochastic depth texture and the number of stochastic depth samples $S$:

$$P_j = \alpha(1-\alpha)^{j-1}S$$

The lower the $\alpha$, the more similar this probability becomes for depth layers at a different depth order, but also the more often we miss capturing the correct ambient occlusion, resulting in the washed out ambient occlusion after the blur. The probability of finding the depth layer with the maximum occlusion (layer $j$) during the ambient-occlusion computation can almost be modeled like a binomial distribution, where the probability of success in each trial is $P_j$ and the number of trials is based on the number of steps and directions. With enough trials, this probability goes up and becomes similar for finding depth layers at a different depth order. Furthermore, due to the per-pixel noise and the Gaussian blur, pixels spread out their ambient occlusion to neighbouring pixels, further increasing the consistency (steps from neighbouring pixels can almost be seen as additional Bernoulli trials). When we greatly increase the number of steps we take and the directions that we march in (i.e., the number of trials), we can see the Monte Carlo nature in action, where the lower sample count image can be almost indistinguishable from the higher sample count image (Figure 9.3). This is analogous to Monte Carlo ray tracing, where given enough rays, we eventually converge to the correct result and the noise disappears. This also indicates that a more sophisticated blur or temporal filtering could improve the quality for lower sample counts, allowing us to increase the number of trials without increasing the frame time.

By increasing the sample count, we reduce the noise, which also reduces flickering when moving the camera around. With 1 or 2 samples, the flickering is noticeable, with 4 samples it is still present but much less apparent and with 8 or more samples it is almost completely gone. When the noise itself is reduced, by for example increasing the number of directions, the flickering is also reduced. Again temporal filtering should be able to reduce the noise and in turn the flickering substantially.



(a) 1-sample SDAO. (b) 2-sample SDAO. (c) 4-sample SDAO.

(d) 8-sample SDAO. (e) 16-sample SDAO. (f) Ray-traced ground truth.

Figure 9.1: Stochastic-depth ambient occlusion with a different amount of stochastic-depth samples. With 1 or 2 samples the occlusion becomes more washed out, while the results for 4, 8 or 16 samples are very similar to one another.

Table 9.1 shows the runtime of SDAO for the different amount of stochastic samples, obtained using an Intel Core i7-4770k at 4.5Ghz, a NVIDIA RTX 2070 and 16GB ram (used throughout this evaluation). We tested the performance of SDAO with the Sponza, Sibenik, San Miguel and Hairball scenes [34], where the first two show a more realistic setting, while the latter two show the worst case performance. Since screen-space ambient occlusion is often computed at half resolution [6, 8, 25] (i.e., a quarter of the pixels), we have included results for both 1920x1080 and 960x540. We only look at sample counts that are powers of two, since rendering the stochastic-depth buffer takes exactly the same time for a non power of two as it would for the next power of two. This is due to the fact that we use MSAA for our stochastic-depth samples, which only works with a sample count that is a power of two. When rendering a 3-sample stochastic-depth buffer for example, we would still require 4x MSAA, but throw away the results of one of the MSAA samples.

What we see is that the ambient-occlusion computation for the full resolution takes roughly four

(a) 1-sample SDAO (tweaked). (b) 2-sample SDAO (tweaked). (c) 16-sample SDAO.

Figure 9.2: By tweaking the $AO_{strength}$ and $AO_{exponent}$ parameters, we can make 1 or 2 sample SDAO look more similar to 16-sample SDAO. We do however introduce noise and lose finer details.

(a) 1-sample SDAO with $N_s = N_d = 20$. (b) 16-sample SDAO.

Figure 9.3: By increasing the number of steps and directions, the results with 1-sample SDAO look indistinguishable from 16-sample SDAO.

times longer than for the half resolution, which makes sense as we have four times the number of pixels. When doubling the number of samples, the ambient-occlusion runtime also roughly doubles, except when going from one to two samples, which is likely caused by our implementation not being optimized for using only a single stochastic-depth sample. The runtime of the full resolution ambient-occlusion computation for the more complex scenes (San Miguel and Hairball) is higher than that of the simpler scenes for lower sample counts. For higher sample counts, they perform very similar (or better, in the case of the Hairball scene) than the simpler scenes. Thus the runtime for the ambient-occlusion computation is roughly linear with the number of stochastic-depth samples.

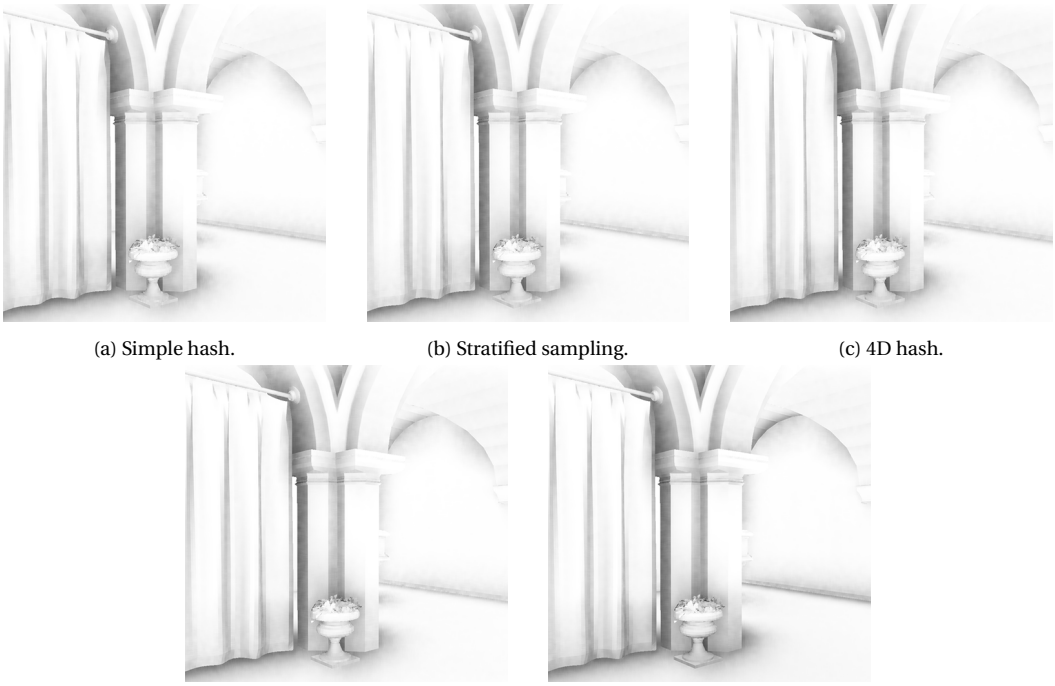
The stochastic-depth runtime scales differently. The runtime difference of the stochastic-depth pass is very small between one and two stochastic-depth samples and between the different resolutions. From two to four stochastic-depth samples, the jump in runtime is larger, especially for the full resolution, creating a larger performance gap between the resolutions. From four to eight samples, the runtime for the full resolution stochastic depth increases by a factor of 2.5, while the half resolution stochastic depth roughly doubles in runtime. From eight to sixteen samples, the runtime increase for the stochastic depth is more scene dependent. We also see that the runtime for the stochastic-depth pass often increases very little when doubling the number of samples (especially apparent with half the resolution), but suddenly increases rapidly for higher samples counts. The runtime of the stochastic-depth pass is thus less sensitive to the number of samples (or resolution for that matter), mostly seeing large increases with the higher sample counts.

When we compare the runtime of the stochastic-depth pass between the different scenes, we see that it largely depends on the scene's complexity. The runtime of the stochastic-depth pass for the more complex scenes is significantly higher than for the simpler scenes. The San Miguel scene has 6 million vertices and 10 million triangles, but due to its large size, most triangles fall outside of the view frustum. This is reflected by the stochastic-depth runtime, where we see very similar performance regardless of sample count or resolution, the limiting factor here is viewport culling (which is confirmed using NVIDIA Nsight's profiler). With the Hairball scene that has around 1.5 million vertices and 3 million triangles that all fall within the view frustum, we see higher runtimes, but similar behaviour with respect to the sample count as to the simpler scenes.

When comparing both quality and performance, we see a sweet spot with one to four samples after which the increase in visual quality does not warrant the performance cost. Especially going from eight to sixteen samples makes very little sense, with almost no perceptual difference. What is interesting to see is that using only one stochastic-depth sample is already enough to get usable results, with only a small additional cost in terms of runtime (0.4 ms, i.e., the cost of rendering the stochastic depth, assuming an ambient-occlusion implementation optimised for a single stochastic-depth sample).

Table 9.1: Runtime (AO + stochastic-depth pass, in milliseconds) of stochastic-depth ambient occlusion with a different amount of stochastic-depth samples on both full and half resolution (1920x1080 and 960x540 respectively).

| | Sponza | | Sibenik | | San Miguel | | Hairball | |
|---|---|---|---|---|---|---|---|---|
| | Full | Half | Full | Half | Full | Half | Full | Half |
| Regular HBAO | 1.22 | 0.41 | 1.46 | 0.41 | 1.78 | 0.33 | 2.22 | 0.28 |
| 1-sample SDAO | 1.82 + 0.43 | 0.59 + 0.34 | 2.11 + 0.25 | 0.59 + 0.15 | 3.46 + 8.06 | 0.50 + 7.77 | 3.44 + 3.72 | 0.50 + 2.61 |
| 2-sample SDAO | 2.51 + 0.55 | 0.76 + 0.36 | 2.78 + 0.33 | 0.77 + 0.18 | 3.99 + 7.91 | 0.64 + 7.45 | 3.70 + 4.58 | 0.60 + 2.66 |
| 4-sample SDAO | 4.62 + 0.95 | 1.28 + 0.50 | 4.84 + 0.59 | 1.26 + 0.28 | 6.32 + 8.48 | 1.07 + 7.58 | 4.85 + 8.87 | 0.94 + 3.37 |
| 8-sample SDAO | 9.87 + 2.40 | 2.19 + 0.68 | 9.50 + 1.45 | 2.66 + 0.57 | 9.95 + 9.99 | 2.28 + 7.81 | 7.39 + 22.31 | 1.72 + 7.81 |
| 16-sample SDAO | 17.78 + 4.06 | 4.34 + 1.06 | 17.46 + 2.24 | 4.25 + 0.65 | 16.79 + 12.56 | 4.33 + 8.56 | 13.11 + 29.53 | 3.14 + 9.84 |



(a) Simple hash. (b) Stratified sampling. (c) 4D hash.

(d) Isotropic alpha threshold. (e) Anisotropic alpha threshold.

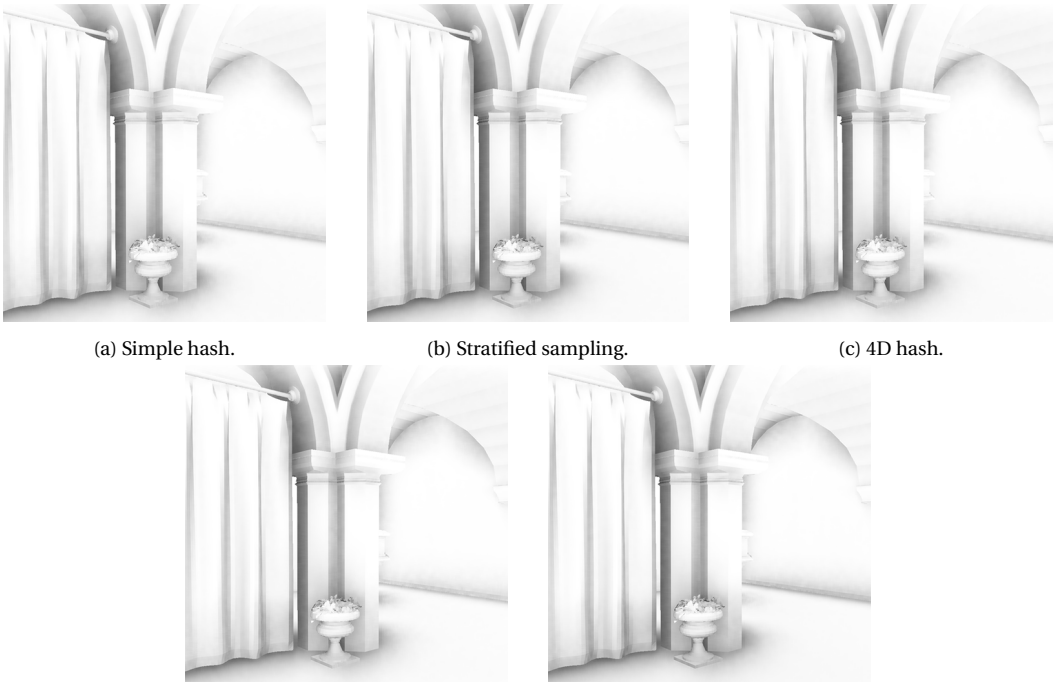
Figure 9.4: Results with the different sampling methods with 1-sample SDAO. The simple hash and 4D hash look very similar. The isotropic alpha threshold produces slightly less noise, but the improvement is not enough to warrant the additional performance cost. The anisoptropic alpha threshold produces results with the least amount of noise, but is less able to capture the otherwise missing occlusion. Stratified sampling produces the overall best results, showing less noise than the simpler hashes, while still able to capture the otherwise missing occlusion.

## 9.2. Impact of the Stochastic-Depth Sampling Method

There are two main ways of improving the quality of stochastic-depth ambient occlusion: increasing the number of samples or improving the sample quality. We have just determined how increasing the number of samples affects the quality and performance, seeing that this way of scaling can be somewhat expensive. Just like regular stochastic transparency, we can improve the sample quality by using stratified sampling [17] or by using a more sophisticated hashing/alpha testing function [54] (i.e., the isotropic and anisotropic alpha threshold, as mentioned in Section 8.2). The isotropic and anisotropic alpha threshold focuses on improving the spatial and temporal stability, by anchoring the noise to the geometry. Changing the sampling method does not affect the runtime of the ambient-occlusion computation (as the number of samples remains the same), only the runtime for the stochastic-depth pass is affected.

If we already have a large number of samples (four or more samples), improving the hashing function or sampling scheme does not influence the overall quality by much. For this reason, we will focus on the effects on lower sample counts. Using the different hashing/alpha testing functions in combination with stratified sampling, gave very similar results (or in the case of the anisotropic alpha threshold worse results) to the simple hash at a higher performance cost, therefore we only use stratified sampling with the simple hashing function. Figure 9.4 and 9.5, and Table 9.2 show the results for the different sampling methods in terms of quality and performance respectively (with the same settings as before and at full resolution).

The simple hash, 4D hash and the isotropic alpha threshold are visually very similar, but the simple hash is the cheapest to compute, making it the better option. Stratified sampling improves upon the simple hash, giving a result with overall less noise, at a small performance penalty. The anisotropic alpha threshold further reduces the noise, but captures less of the missing occlusion. The



(a) Simple hash. (b) Stratified sampling. (c) 4D hash.

(d) Isotropic alpha threshold. (e) Anisotropic alpha threshold.

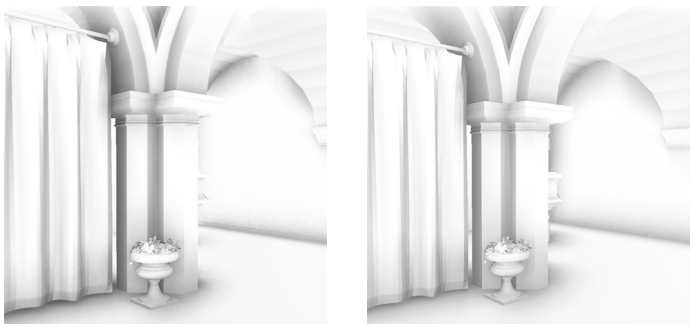
Figure 9.5: Results with the different sampling methods with 2-sample SDAO. The same observations as with 1-sample SDAO hold here. The simple hash, 4D hash and the isotropic alpha threshold look visually very similar. The anisotropic alpha threshold again shows the least amount of noise, but the otherwise missing occlusion is significantly less strong than with the other sampling methods. Stratified sampling still produces the best overall results.

Table 9.2: Runtime (in milliseconds) of the stochastic-depth pass with different sampling methods.

| | 1-sample SDAO | 2-sample SDAO |
|---|---|---|
| Simple hash | 0.43 | 0.47 |
| Stratified sampling | 0.45 | 0.57 |
| 4D hash | 0.51 | 0.51 |
| Isotropic alpha threshold | 0.97 | 0.97 |
| Anisotropic alpha threshold | 1.18 | 1.21 |

anisotropic alpha threshold shows a clear preference to capture the frontmost depth layer, resulting in an uneven strength of the ambient occlusion throughout the image. We also see that the more sophisticated methods (i.e., the isotropic and anisotropic alpha threshold) are much more expensive to compute. For the cost of 1-sample SDAO (the stochastic-depth buffer generation + the ambient-occlusion computation) with the anisotropic alpha threshold, we can also compute SDAO with two simpler samples which provides superior image quality.

The advantage of the isotropic and anisotropic alpha threshold are that they are more spatially and temporally stable than regular hashes. This is indeed true when used for regular stochastic transparency, but with the additional randomness from the ambient-occlusion pass this does not hold for stochastic-depth ambient occlusion. There is no noticeable reduction in flickering when moving the camera around, compared to the simple hash or stratified sampling.



(a) 1-sample SDAO. (b) 8-sample SDAO.

Figure 9.6: The result when only using stochastic-depth if sample is further than $0.3 \cdot R$. The occlusion from the regular depth buffer is far stronger than that of 1-sample SDAO, resulting in inconsistent occlusion strength with the otherwise missing ambient occlusion. This approach works best for higher sample counts, where it provides the best performance improvements without a reduction in visual quality.

## 9.3. Only Using Stochastic Depth If Outside of the Radius

In Section 6.3, we described how only iterating over the stochastic-depth samples if the sample from the regular depth buffer would be outside of the radius could improve the overall performance. As shown in Table 9.3, this approach can cut the runtime in half, but is unusable in practice due to visual artifacts it introduces. Instead, only using the stochastic-depth buffer when the sample from the regular depth buffer is further than $0.3 \cdot R$ from point $p$, removes these artifacts, while still providing some of the performance benefits (0.3 was empirically found to work well on the Sponza scene, chosen conservatively to preserve the visual quality). However, compared to always using the stochastic-depth buffer, we do slightly more work when we do use the stochastic-depth buffer, as we first reconstruct the sample position using the regular depth buffer. If this extra work is small compared to the effort we save if we can skip using the stochastic-depth buffer, this can improve the overall performance. This means that the higher the sample count (and thus the more expensive using the stochastic-depth buffer is), the more the performance is improved with this technique. For lower sample counts (less than four samples), the additional work can result in similar or even decreased performance. In addition, the occlusion strength with lower sample counts is not consistent anymore, the otherwise missing occlusion is much more washed out than the occlusion from the regular depth buffer (Figure 9.6). This approach thus only makes sense for higher sample counts.

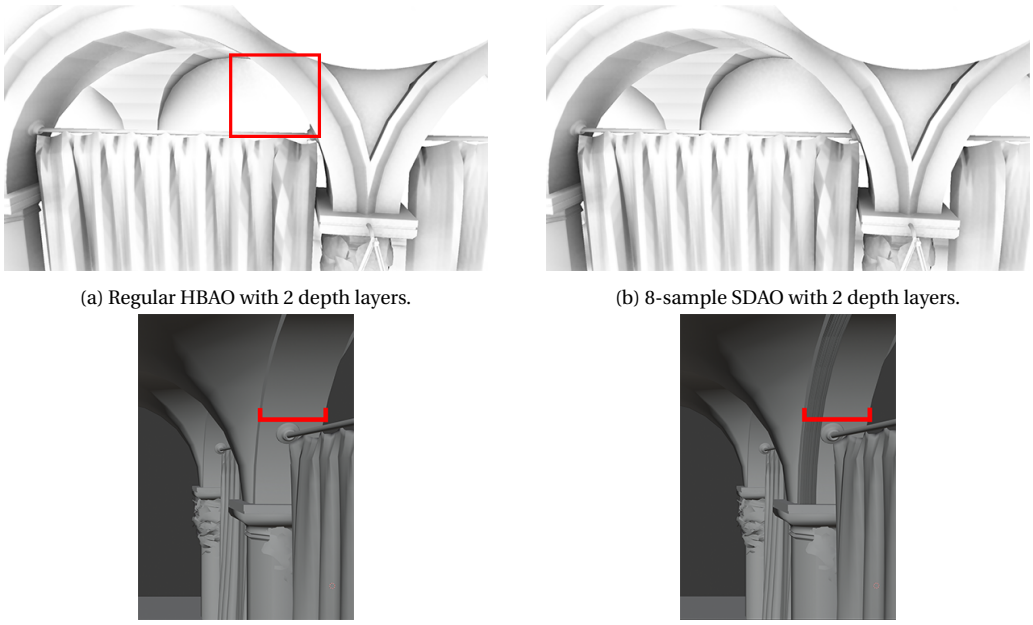
The other performance improvement described in Section 6.3, the "angle skip", checks if the angle of the sample from the regular depth buffer is larger than the current horizon angle $h$. If not, the angle found with the stochastic-depth buffer would only be similar or smaller, meaning that we can skip iterating over all the stochastic-depth samples. This angle skip approach shows a performance improvement similar to only using the stochastic-depth buffer if the sample is further than $0.3 \cdot R$ from point $p$, but both can be combined to come close to the performance one would get when only using the stochastic-depth buffer outside of the radius, without any of the visual artifacts.

## 9.4. Impact of the Number of Depth Layers

With the stochastic-depth buffer, we provide information about obstructed depth layers to the screen-space ambient occlusion algorithm. This stochastic-depth buffer can store up to 16 layers per pixel, but due to its origins as an transparency technique it prioritises layers closer to the camera. To determine what this means for occlusion coming from geometry obstructed by many depth layers, we used the scenario as seen in Figure 9.7. Here we have a wall, consisting of 2 depth layers that obstruct the geometry that casts the occlusion. Using Blender, we added additional depth layers to this wall (Figure 9.7d) to determine how this influences the ambient occlusion.

Table 9.3: Runtime (in milliseconds) of stochastic-depth ambient occlusion (only the ambient-occlusion part), when only partially using the stochastic-depth buffer.

| | 2-sample SDAO | 4-sample SDAO | 8-sample SDAO |
|---|---|---|---|
| Always stochastic depth | 2.51 | 4.62 | 9.87 |
| Only outside of radius | 1.86 | 2.38 | 3.68 |
| When further than $0.3 \cdot R$ | 2.70 | 4.06 | 6.89 |
| Always stochastic depth with Angle skip | 2.78 | 3.84 | 6.32 |
| When further than $0.3 \cdot R$ with angle skip | 2.01 | 2.64 | 4.23 |



(a) Regular HBAO with 2 depth layers.

(b) 8-sample SDAO with 2 depth layers.

(c) Side view from blender with 2 depth layers.

(d) Side view from blender with 32 depth layers.

Figure 9.7: In this scenario, we have a wall consisting of 2 depth layers that obstructs the geometry behind it, causing missing occlusion. We added additional depth layers to this wall to determine how this influences the results with stochastic-depth ambient occlusion.

Figure 9.8 shows the results, where we see that the occlusion brought back with SDAO gradually disappears, until it is completely gone with 16 layers. But there are two ways that we can deal with this: we can increase the number of samples to capture more layers and/or we can lower the alpha value at which we render the scene to make it more transparent. If the ambient occlusion is still present but faded (e.g.„ with 8 depth layers), increasing the number of samples can bring it fully back (Figure 9.9). If the ambient occlusion is completely gone (with 16 or 32 depth layers), increasing the number of samples has almost no impact. Instead we need to render the scene more transparently to bring this occlusion back.

Lowering the alpha value at which we render the scene allows us to capture further depth layers more often, but decreases the chance that we capture one of the frontmost depth layers. This leads to similar behaviour as decreasing the number of samples, the occlusion gets more washed out, but again by tweaking the parameters we can get a visually similar looking image albeit with noise. Figure 9.10 shows how we can bring back the occlusion with 32 depth layers by lowering the $\alpha$ value from 0.2 to 0.01, at the cost of significant noise. By greatly increasing the number of directions (similar to the example with 1-sample SDAO shown in Figure 9.3), we can remove most of this noise. This does however lead to over-occlusion in certain regions of the image (e.g., the arch), instead increasing the number of steps can provide better results, as it removes the noise without the over-occlusion.

(a) 2 depth layers.

(b) 4 depth layers.

(c) 8 depth layers.

(d) 16 depth layers.

(e) 32 depth layers.

Figure 9.8: Results of 8-sample SDAO with an alpha of 0.2 for a different amount of depth layers. We see that the occlusion brought back using stochastic-depth ambient occlusion gradually disappears.

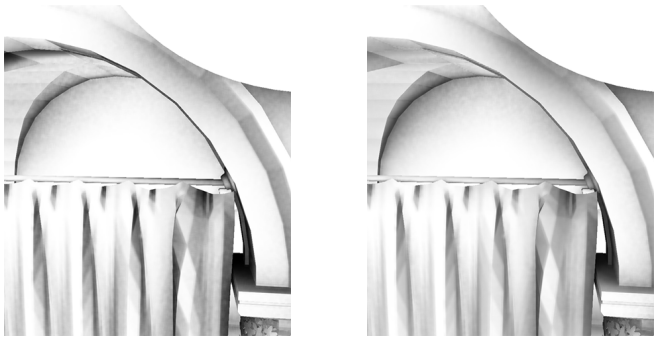
(a) 8 depth layers.

(b) 16 depth layers.

Figure 9.9: Results with 16-sample SDAO with an alpha of 0.2. With 8 depth layers, increasing the number of samples can bring the occlusion back. With 16 depth layers, there is almost no difference compared to using only 8 stochastic-depth samples.

(a) $\alpha = 0.2$. (b) $\alpha = 0.01$. (c) $\alpha = 0.01$ (tweaked).

Figure 9.10: By lowering the $alpha$ to 0.01 and tweaking the ambient-occlusion parameters, we can bring back the missing occlusion with 32 depth layers using 8-sample SDAO, at the cost of significant noise.



(a) $N_d = 40$. (b) $N_s = 40$.

Figure 9.11: The noise with 8-sample SDAO and $alpha = 0.01$ for 32 depth layers can be removed by greatly increasing the number of steps and directions. Increasing the number of directions can lead to over-occlusion in certain regions (e.g., the arch), which does not happen when only increasing the number of steps.

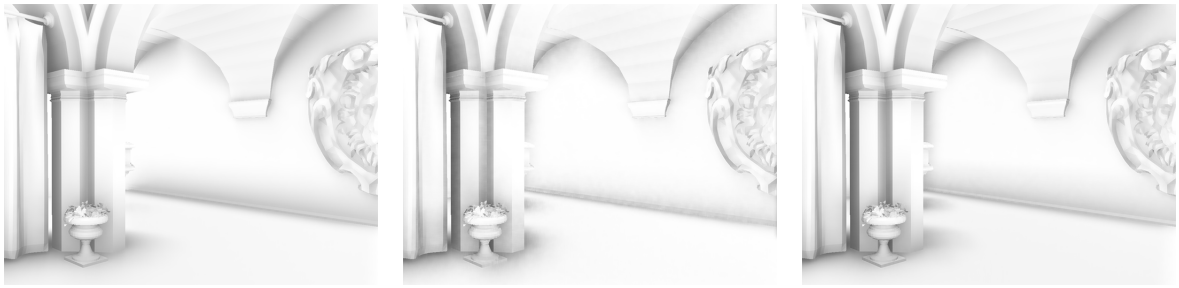
By amortizing these steps over time, we could remove the noise while still maintaining real-time performance.

## 9.5. Comparison of Stochastic-Depth HBAO, HBAO+ and SSAO

We have implemented stochastic-depth ambient occlusion using three different screen-space ambient occlusion techniques (HBAO, HBAO+ and SSAO) to compare how generalizable our technique is. Implementing stochastic-depth ambient occlusion is very similar for the three techniques, adding an additional loop around the ambient-occlusion computation to iterate over all the depth samples from the stochastic-depth buffer. Figure 9.12, 9.13 and 9.14 show the stochastic-depth variant of different ambient-occlusion techniques with a different amount of stochastic-depth samples. For HBAO and HBAO+ we use similar settings (4 directions, 4 steps), but to make the comparison fair for SSAO we use 42 samples to get a similar runtime to its the horizon-based counterparts.

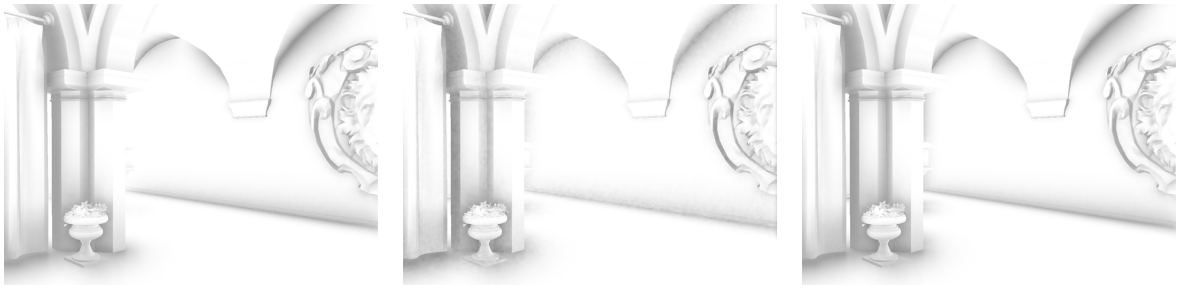
We see that stochastic-depth HBAO and HBAO+ provide very similar looking results. They both capture the same missing occlusion and behave similarly to the number of stochastic-depth samples. We do however notice that HBAO+ seems more noisy than HBAO for lower sample counts, which is caused by the softer and more nuanced ambient occlusion HBAO+ provides, compared to the harsher (over-occluded) ambient occlusion of HBAO. With too much noise, these nuances get lost and turn into more noise.

For SSAO the results look quite different to those of HBAO and HBAO+, with the occlusion more focused around edges and creases. We also see some haloing artifacts surrounding the objects, which become worse the larger the radius is (Figure 9.15). These halo artifacts, surrounding the pillar and



(a) Regular HBAO. (b) 2-sample stochastic-depth HBAO. (c) 8-sample stochastic-depth HBAO.

Figure 9.12: Results of stochastic-depth ambient occlusion with HBAO.

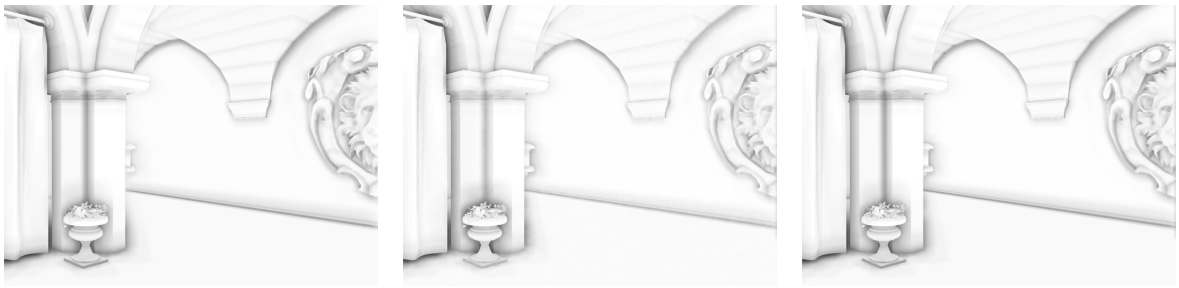


(a) Regular HBAO+. (b) 2-sample stochastic-depth HBAO+. (c) 8-sample stochastic-depth HBAO+.

Figure 9.13: Results of stochastic-depth ambient occlusion with HBAO+.

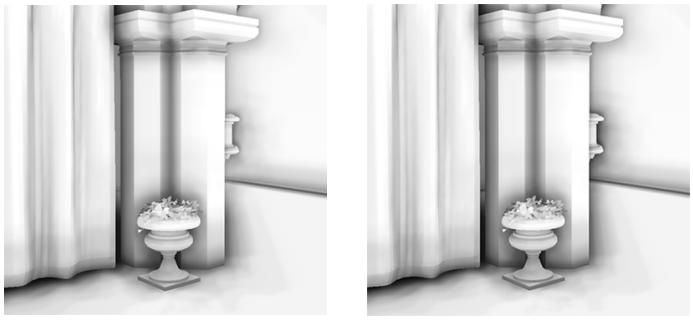
the object behind it, hide a large part of the missing occlusion that the stochastic-depth variant is able to recover. The increase in visual quality is thus very subtle and on its own would not warrant the additional performance cost for still images. When moving the camera around however, these small differences become much more noticeable. Figure 9.16 shows how the stochastic-depth SSAO variant is able to capture the missing occlusion behind the various arches and pillars. Without stochastic depth, small changes to the camera's position or rotation could lead to a sudden change in occlusion strength, and pop in and pop out of occlusion. stochastic-depth SSAO is far more spatially stable and does not suffer from these issues.

When changing the number of stochastic-depth samples with SSAO, we see similar behaviour to that of HBAO and HBAO+, where the occlusion gets washed out but remains consistent across the image. Then by tweaking the occlusion strength parameters, we can get a visually similar looking image to one with more stochastic-depth samples. When comparing a low sample image from SSAO with one from HBAO or HBAO+, we see that the noise is much less perceptible with SSAO. This is due to the fact that the occlusion with SSAO is harsher and more focused, the noise is therefore better



(a) Regular SSAO. (b) 2-sample stochastic-depth SSAO. (c) 8-sample stochastic-depth SSAO.

Figure 9.14: Results of stochastic-depth ambient occlusion with SSAO.



(a) Regular SSAO. (b) 8-sample stochastic-depth SSAO.

Figure 9.15: When increasing the radius, SSAO's haloing artifacts become worse. In many cases, these halos hide the missing occlusion.

(a) Regular SSAO. (b) 8-sample stochastic-depth SSAO.

Figure 9.16: Stochastic-depth SSAO is able to capture the missing occlusion behind various arches and pillars (marked by the red circles). Even though these results are hard to see in a still image, this greatly improves the spatial stability as the occlusion does not suddenly pop in and pop out when moving the camera around.

contained which allows the Gaussian blur to remove it more effectively.

In terms of performance, the three algorithms lie very close together as seen in Table 9.4. We see that the runtime gap between HBAO and HBAO+ seems to be consistent (on average 0.35 ms) and does not increase with the number of samples (except for 16 samples). We set up SSAO so it would have a similar runtime to HBAO, but when enabling stochastic depth we see a performance gap of around 0.20 ms that increases drastically when using eight or more stochastic-depth samples. SSAO has a simpler ambient-occlusion computation, meaning that per sample/step, SSAO has to do less work than HBAO or HBAO+. SSAO does however take 42 samples, instead of the 4x4 steps HBAO and HBAO+ take, meaning that we use the stochastic-depth buffer more often (2.625 times as much) and become even more bandwidth limited.

Table 9.4: Runtime (in milliseconds) of stochastic-depth ambient occlusion (only the ambient-occlusion part) with the different screen-space ambient occlusion algorithms.

| | HBAO | HBAO+ | SSAO |
|---|---|---|---|
| Regular | 1.22 | 1.08 | 1.23 |
| 1-sample SDAO | 1.82 | 1.68 | 2.03 |
| 2-sample SDAO | 2.51 | 2.26 | 2.73 |
| 4-sample SDAO | 4.62 | 4.30 | 5.12 |
| 8-sample SDAO | 9.87 | 9.37 | 12.04 |
| 16-sample SDAO | 17.78 | 17.23 | 34.16 |

## 9.6. Bent Normals and Cones With Stochastic-Depth Ambient Occlusion

Bent normals and cones (Section 7.1) can be used to render a more accurate image when using an environment map, since the direction of the indirect light is then taken into account. The main benefit of bent normals and cones are that they can be computed during the ambient-occlusion pass, with only a slight modification to the algorithm. There are no extra modifications necessary to make it work for stochastic-depth ambient occlusion.

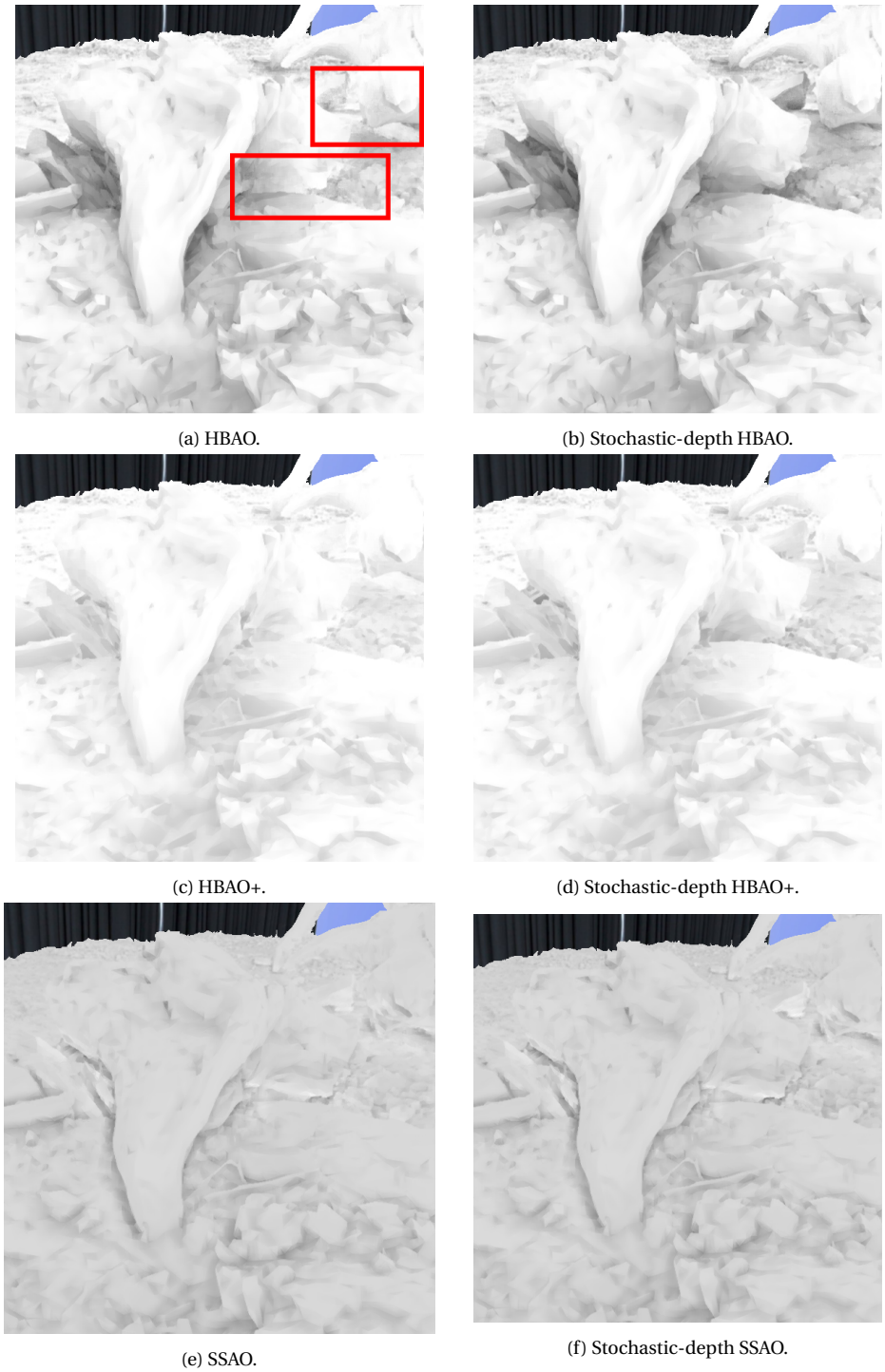
To compute the bent normals and cones, we do slightly more work per direction (for HBAO and HBAO+) or per sample (for SSAO), without increasing the workload per stochastic-depth sample. This means that the performance impact should be independent of the number of stochastic-depth samples. Table 9.5 shows this is indeed the case, where compared to Table 9.4 the runtimes are increased by roughly 0.1 ms irregardless of the ambient-occlusion technique or the number of stochastic-depth samples.

In terms of quality, stochastic-depth ambient occlusion generates more accurate bent normals and cones in regions where traditional screen-space ambient occlusion algorithms fail to capture the ambient occlusion. This can be seen in Figure 9.17, where these regions have significantly larger bent-cone angles than they should. Stochastic-depth ambient occlusion improves the accuracy of the indirect lighting in these regions and improves the overall spatial stability, removing large changes in indirect light when moving the camera around (as seen in Figure 9.18). Overall we see a larger improvement to the bent cones using stochastic-depth ambient occlusion with the HBAO and HBAO+ algorithm, than when using the SSAO algorithm. Figure 9.19 shows how the quality of the bent normals and cones differ with the number of stochastic-depth sample. The quality of the bent normals is less dependent on the number of samples than the bent cones, having no noticeable difference in quality from four samples onwards. The bent cones still show a slight difference between four and eight samples, and the difference between one and four samples is more pronounced. Even with one or two stochastic-depth samples, the accuracy improves in regions where traditional screen-space ambient occlusion techniques fail to capture the ambient occlusion due to

Table 9.5: Runtime (in milliseconds) of stochastic-depth ambient occlusion when computing bent normals/cones (only the ambient-occlusion part) with the Sponza scene.

| | HBAO | HBAO+ | SSAO |
|---|---|---|---|
| Regular | 1.28 | 1.20 | 1.38 |
| 1-sample SDAO | 1.93 | 1.76 | 2.18 |
| 2-sample SDAO | 2.58 | 2.34 | 2.87 |
| 4-sample SDAO | 4.71 | 4.36 | 5.17 |
| 8-sample SDAO | 9.98 | 9.52 | 12.10 |
| 16-sample SDAO | 17.89 | 17.35 | 34.26 |

obstructing geometry. However, at least 4-sample SDAO are required to achieve similar accuracy to these traditional screen-space techniques in non-obstructed regions.



(a) HBAO.

(b) Stochastic-depth HBAO.

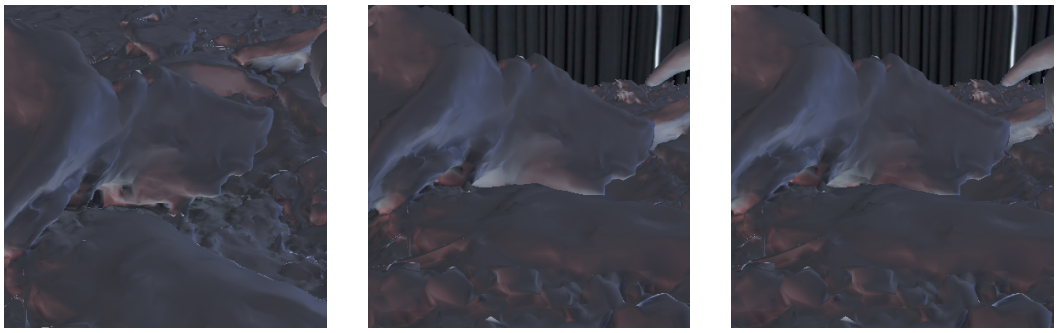
(c) HBAO+.

(d) Stochastic-depth HBAO+.

(e) SSAO.

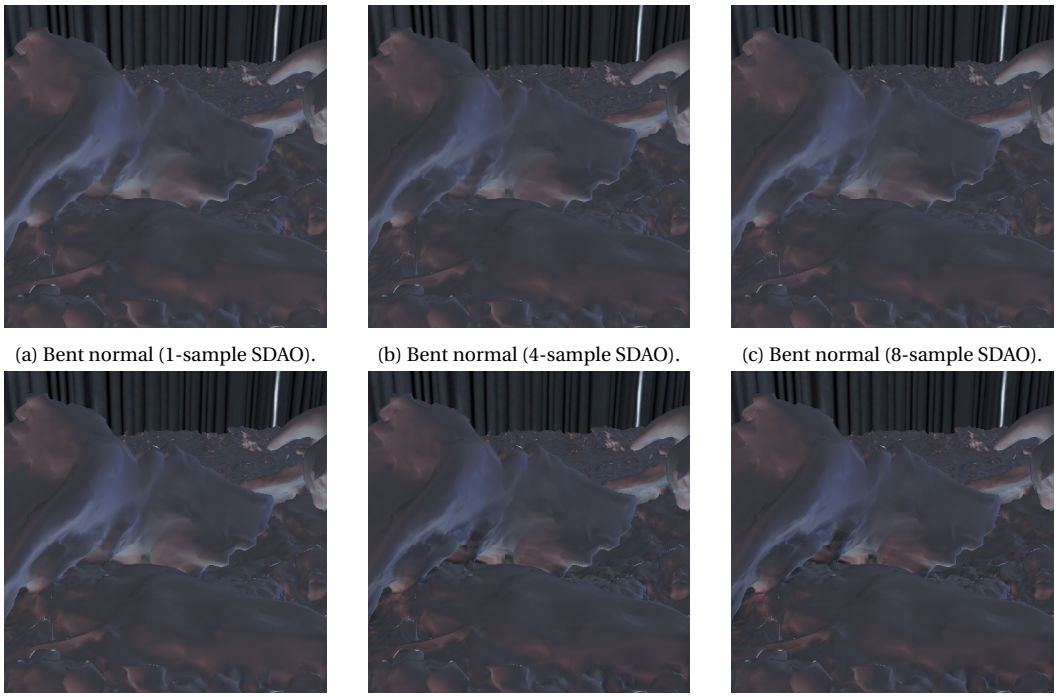
(f) Stochastic-depth SSAO.

Figure 9.17: Visualization of the bent cone angles, where black means 0 degrees and white means 90 degrees, resembling the ambient-occlusion images. We see that in regions where traditional screen-space algorithms fail to capture the occlusion due to obstruction (marked by the red squares), the bent cone angle is significantly larger than it should be. Stochastic-depth ambient occlusion (8-samples) is able to more accurately compute the bent cones in these regions. With SSAO the difference in a still image is more subtle, however stochastic-depth ambient occlusion provides better spatial stability.



(a) Bent cones using HBAO. (b) Bent cones using HBAO (slightly rotated). (c) Bent cones using SDAO.

Figure 9.18: When we rotate the camera, the color of the indirect light changes, because the bent normals and cones are not accurate anymore, due to the surrounding geometry being obstructed. When using stochastic-depth ambient occlusion (8-samples) the indirect light remains more stable under these camera movements.



(a) Bent normal (1-sample SDAO). (b) Bent normal (4-sample SDAO). (c) Bent normal (8-sample SDAO).

(d) Bent cone (1-sample SDAO). (e) Bent cone (4-sample SDAO). (f) Bent cone (8-sample SDAO).

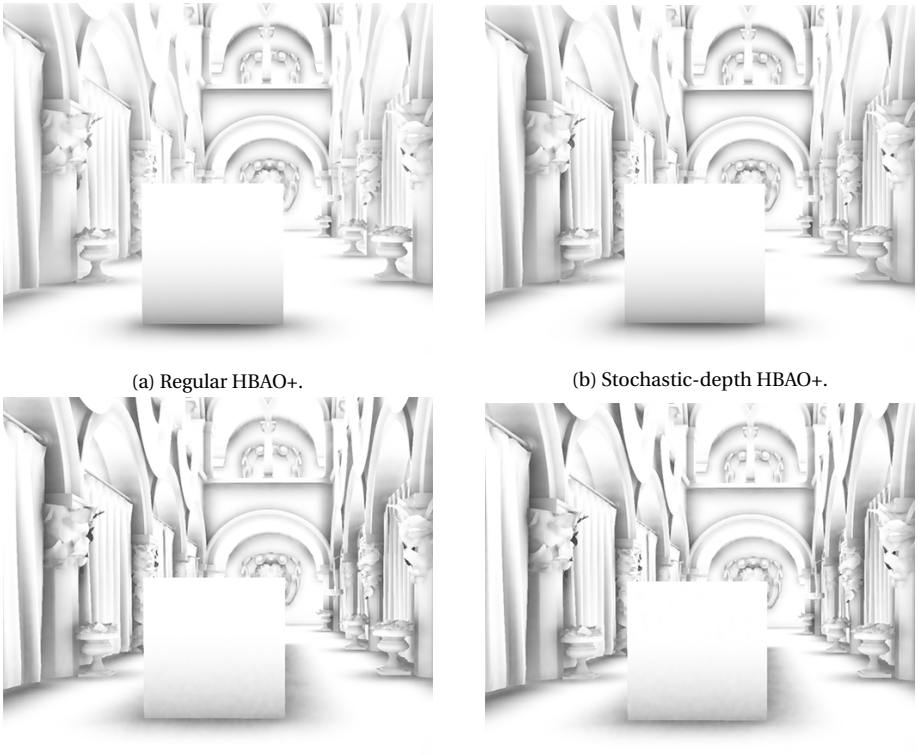
Figure 9.19: Increasing the number of stochastic-depth samples, improves the accuracy of the bent normal and cones. With bent normals, we see much more incorrect white light with 1 or 2 stochastic-depth samples, but from 4 samples and onwards there is no perceptible difference. With bent cones, we see overall lower bent cone angles with lower sample counts, due to lower occlusion. Here, we still see a subtle difference between 4 and 8 samples (the 8-sample result is a slightly darker red), after which there is again no perceptible difference.

## 9.7. Evaluation of Multi-View Stochastic-Depth Ambient Occlusion

With multi-view stochastic-depth ambient occlusion (Section 7.2), we use two stochastic-depth buffers (rendered from different cameras) to compute the ambient occlusion. This allows us to capture occlusion from geometry under a grazing angle from the main camera. The main benefit of using a stochastic-depth buffer is that we do no have to take the scene's geometry and visibility into account when placing the second camera, since it can simply "see through" any obstructing geometry.

Figure 9.20 shows how multi-view stochastic-depth ambient occlusion is able to capture the occlusion from this rectangle that regular HBAO+ or stochastic-depth HBAO+ could not. When comparing the results of a full and half resolution stochastic-depth buffer for the secondary camera, we see almost no perceptual difference in terms of quality and stability. For multi-view stochastic-depth ambient occlusion to work well, we need to use a relatively low alpha value (e.g., $\alpha = 0.05$), otherwise the occlusion strength becomes less consistent (Figure 9.21). Also the occlusion from the secondary camera is slightly noisier and shows more flickering when moving around than the ambient occlusion from the main camera. This is due to the secondary camera being obstructed by multiple depth layers, while the primary camera is not. This behaviour is similar to stochastic-depth ambient occlusion when many occluding depth layers are present.

A possible solution would be to cull fragments that do not contribute to the ambient occlusion of the main camera. Only fragments surrounding the geometry from the main camera (i.e., fragments inside of the radius $R$ from the geometry visible via the main camera) potentially contribute to the ambient occlusion. This means that we could cull all fragment outside of an extended frustum around the main camera's view. A more advanced approach could be based on the selective rendering and uncertainty buffer found in adaptive multi-view screen-space ambient obscurance [55], which only



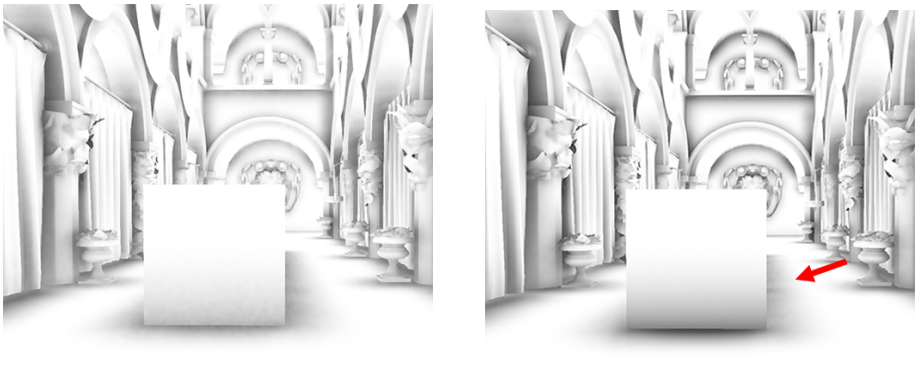
(a) Regular HBAO+.

(b) Stochastic-depth HBAO+.

(c) Multi-view SDAO.

(d) Multi-view SDAO (half res).

Figure 9.20: Using multi-view stochastic-depth ambient occlusion, we are able to correctly capture the occlusion from the sides of the rectangle, which regular HBAO+ and SDAO fail to capture. In terms of visual quality we see almost no difference between a full or half resolution secondary stochastic-depth buffer.



(a) $\alpha = 0.05$.

(b) $\alpha = 0.25$.

Figure 9.21: Multi-view SDAO needs a relatively low alpha, otherwise the additional occlusion becomes uneven in strength, due to geometry obstructing the secondary camera. However lowering the alpha value does mean introducing noise.

(a) Stochastic-depth HBAO+.

(b) Stochastic-depth HBAO+ slightly rotated.

(c) Multi-view SDAO.

Figure 9.22: When we rotate the camera, the occlusion in the corner gradually disappears with regular HBAO+ or stochastic-depth HBAO+. With multi-view ambient occlusion, the occlusion remains consistent.

renders the fragments in the regions where the main camera potentially misses information. This would reduce the number of irrelevant, but occluding depth layers when rendering the secondary stochastic-depth buffer, in turn improving the visual quality and stability.

Figure 9.22 shows another example where the occlusion gradually disappears when rotating the camera, as the wall casting the occlusion is slowly coming under a grazing angle. With multi-view stochastic-depth ambient occlusion, the ambient occlusion remains consistent. So even if the geometry is still visible from the main camera, multi-view stochastic-depth ambient occlusion can provide better spatial stability and coherence when moving around the scene.

As expected, by computing the ambient occlusion two times, the runtime of the ambient occlusion pass roughly doubles (Table 9.6). In a similar manner, the runtime of the stochastic-depth pass is also roughly doubled, as we now render two stochastic-depth buffers. By rendering the secondary stochastic-depth buffer at half the resolution, we can drastically reduce the runtime of the stochastic-depth pass, now only taking approximately 1.5 times as long as with a single camera. When reducing the resolution of the secondary stochastic-depth buffer, we also see a small performance improvement in the ambient-occlusion pass. This is likely caused by the improved texture and cache efficiency, as there are less texels in the stochastic-depth buffer in total and we more often use the same or near by texels to compute the ambient occlusion for adjacent pixels.

Table 9.6: Runtime (in milliseconds) of multi-view stochastic-depth ambient occlusion.

| | Single Camera | Multi-view | Multi-view (with half res additional camera) |
|---|---|---|---|
| 1-sample SDAO | 1.60 (AO) + 0.54 (depth) | 2.98 (AO) + 1.34 (depth) | 2.82 (AO) + 0.87 (depth) |
| 2 sample SDAO | 2.17 (AO) + 0.70 (depth) | 4.13 (AO) + 1.60 (depth) | 3.82 (AO) + 1.11 (depth) |
| 4 sample SDAO | 3.76 (AO) + 1.22 (depth) | 7.63 (AO) + 3.05 (depth) | 6.86 (AO) + 1.83 (depth) |
| 8 sample SDAO | 8.66 (AO) + 3.29 (depth) | 16.63 (AO) + 7.93 (depth) | 14.83 (AO) + 4.63 (depth) |

# 10 Conclusion

Traditional screen-space ambient occlusion techniques use the information stored in the depth buffer to approximate the ambient occlusion. While these techniques are very fast compared to their ray-traced counterparts, they have no information regarding geometry not captured by the depth buffer. This can result in underestimated or missing occlusion when: 1. The geometry is just outside of the view frustum, 2. The geometry is at a grazing angle with the camera or 3. The geometry is hidden from view behind other geometry. The first case is already solved efficiently by previous work, but the second and third case are not.

Our proposal, stochastic-depth ambient occlusion, improves upon traditional screen-space ambient occlusion techniques by augmenting them with depth information of hidden/obstructed geometry, using a stochastic transparency based approach. In contrast to previous techniques, which either use transparency techniques unsuitable for real-time applications (e.g., depth peeling or an A-buffer) or only look at two fixed depth layers, stochastic-depth ambient occlusion maintains the real-time performance one would want from a screen-space ambient occlusion technique, while being more accurate and robust than other techniques that use fixed number of depth layers. Furthermore, multi-view stochastic-depth ambient occlusion removes the need for manual (per scene) camera placement to capture geometry at a grazing angle with the main camera. With previous approaches, one would need to carefully place these cameras on a per scene basis in order to take the visibility in the scene into account. Stochastic-depth ambient occlusion allows us to compute the ambient occlusion with hidden geometry, removing this need for manual camera placement. In conclusion, stochastic-depth ambient occlusion greatly improves upon the quality and spatial stability of traditional screen-space approaches, while maintaining the real-time performance. Stochastic-depth ambient occlusion is robust and works well for most scenes. It generalizes to ambient occlusion extensions such as bent normals and cones, and it integrates well into existing rendering pipelines.

## 10.1. Future Work

Due to the scope of this project, several interesting directions were left unexplored. In this section we will highlight potential directions for future work to extend or improve upon stochastic-depth ambient occlusion.

**Temporal Amortization** As we have seen in the evaluation (Chapter 9), using many directions and/or steps in the ambient occlusion pass often leads to results of much higher quality. While these settings are infeasible in real-time applications, it does indicate that higher quality results can be achieved with the data already present. This computation could be temporally-amortized, where the

resulting ambient occlusion becomes more accurate each frame. This could increase the quality of stochastic-depth ambient occlusion, while keeping the runtime suitable for real-time applications. Temporal accumulation can also be used to improve the ambient occlusion with many depth layers, by using a low alpha value and changing the random seed for the stochastic-depth pass each frame. To reduce ghosting and trailing artifacts such temporal-amortizations often produce, a temporal filtering step with reprojection similar to [6, 21] can be used.

**Global Illumination** As shown in [32, 42], screen-space ambient occlusion techniques can also be used to compute one bounce of global/indirect illumination (by iterating this process, multiple bounces can be simulated). These screen-space global illumination techniques (SSGI) only allow light to bounce from objects visible on the screen. This means that light does not bounce from objects that are behind the camera, but also not from objects obstructed from view behind other geometry. A stochastic-depth ambient occlusion based approach could improve upon this latter scenario, as it allows obstructed geometry to be taken into account. What makes SSGI efficient, is that we can lookup the incoming light's color and intensity from the color buffer after the direct lighting pass. However, we cannot use this same approach for hidden geometry, as it is not included in the color buffer. This means that with a stochastic-depth approach, we would need to compute the direct light for hidden geometry before we can determine the global illumination. To maintain real-time performance, this is only feasible with a small number of stochastic-depth samples. Temporal accumulation and filtering can also be used here to improve upon the quality and performance.

**Virtual Reality** While this work mostly focuses on traditional rendering, stochastic-depth ambient occlusion could be a good fit for use in stereoscopic rendering or virtual reality (VR). With stereoscopic rendering, the scene is rendered with two cameras, slightly moved apart to create the stereoscopic effect. Ambient occlusion could greatly improve the sense of depth and the overall visual quality of the image, but rendering ambient occlusion in this scenario is quite challenging. These virtual reality applications have very low frame time budgets, as lower frame rates may induce nausea and/or a feeling of discomfort [24, 53]. Because of their performance, traditional screen-space ambient occlusion techniques may seem like a good fit. However, due to their aforementioned issues, their results are often view dependent, resulting in inconsistent ambient occlusion between the eyes [47]. Stochastic-depth ambient occlusion does not suffer from these view dependent artifacts, while delivering comparable performance.

**Deep Learning** In recent years, deep learning has become very prevalent, even in 3D computer graphics. As shown in [39], convolutional neural networks can be trained to perform screen-space shading techniques, including ambient occlusion, at interactive frame rates. Neural networks can achieve better results than contemporary screen-space ambient occlusion approximations in a similar time budget, and their quality will only increase with more (diverse) training data. These neural networks take screen-space information as input, using per-pixel attributes such as view-space positions, normals, depth, etc. For this reason, they suffers from the typical screen space limitation: missing shading/occlusion from objects hidden from view. A stochastic-depth approach could be used to provide this missing information and further improve the quality of these techniques, with only a small performance cost. We know that the stochastic-depth buffer provides all the necessary information for high quality ambient occlusion, even with only a few stochastic-depth samples or in difficult situations (e.g., when there are many obstructing depth layers), but getting the most out of this information can be very expensive with human-programmed shaders. If a neural network can use the information in the stochastic-depth buffer more efficiently, this could open up the possibility for an ambient occlusion technique with the quality of ray-traced ambient occlusion and the runtime of screen-space techniques.

# Bibliography


[1] Scratchapixel 2.0. 3D Viewing: the Pinhole Camera Model, n.d. URL https://www.scratchapixel.com/lessons/3d-basic-rendering/3d-viewing-pinhole-camera. Accessed: 2020-08-17.

[2] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michat Iwanicki, and Sébastien Hillaire. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd., USA, 4th edition, 2018. ISBN 9781138627000.

[3] Fabian Bauer, Martin Knuth, Arjan Kuijper, and Jan Bender. Screen-space ambient occlusion using a-buffer techniques. In *2013 International Conference on Computer-Aided Design and Computer Graphics*, pages 140–147. IEEE, 2013.

[4] Louis Bavoil. HBAOPlus, 2018. URL https://github.com/NVIDIAGameWorks/HBAOPlus. Accessed: 2020-07-28.

[5] Louis Bavoil. The Peak-Performance-Percentage Analysis Method for Optimizing Any GPU Workload, 2019. URL https://developer.nvidia.com/blog/the-peak-performance-analysis-method-for-optimizing-any-gpu-workload/. Accessed: 2020-08-01.

[6] Louis Bavoil and Johan Andersson. Stable SSAO in Battlefield 3 with Selective Temporal Filtering. Game Developers Conference, 2012. URL https://developer.nvidia.com/sites/default/files/akamai/gamedev/files/gdc12/GDC12_Bavoil_Stable_SSAO_In_BF3_With_STF.pdf. Accessed: 2020-07-26.

[7] Louis Bavoil and Miguel Sainz. Multi-Layer Dual-Resolution Screen-Space Ambient Occlusion. In *SIGGRAPH 2009: Talks*, SIGGRAPH '09, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605588346. doi: 10.1145/1597990.1598035. URL https://doi.org/10.1145/1597990.1598035.

[8] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. Image-Space Horizon-Based Ambient Occlusion. In *ACM SIGGRAPH 2008 Talks*, SIGGRAPH '08, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605583433. doi: 10.1145/1401032.1401061. URL https://doi.org/10.1145/1401032.1401061.

[9] Louis Bavoil, Edward Liu, Peter Shirley, and Morgan McGuire. Hybrid Ray-Traced Ambient Occlusion, August 2018. URL https://casual-effects.com/research/Bavoil2018AO/index.html. Poster at HPG18 ACM SIGGRAPH / Eurographics High Performance Graphics.

[10] Loren Carpenter. The A-buffer, an antialiased hidden surface method. *ACM Siggraph Computer Graphics*, 18(3):103–108, 1984.

[11] John Chapman. SSAO Tutorial, 2011. URL http://john-chapman-graphics.blogspot.com/2013/01/ssao-tutorial.html. Accessed: 2020-07-27.

[12] Franklin C. Crow. Shadow Algorithms for Computer Graphics. In *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '77, page 242–248, New York, NY, USA, 1977. Association for Computing Machinery. ISBN 9781450373555. doi: 10.1145/563858.563901. URL https://doi.org/10.1145/563858.563901.

[13] Leigh Davies. Order-Independent Transparency Approximation with Raster Order Views (Update 2017), 2017. URL https://software.intel.com/content/www/us/en/develop/articles/oit-approximation-with-pixel-synchronization.html. Accessed: 2020-07-27.

[14] Joey de Vries. LearnOpenGL, n.d. URL https://learnopengl.com/. Accessed: 2020-08-15.

[15] Johannes Deligiannis, Jan Schmid, and Yasin Uludag. "It Just Works": Ray-Traced Reflections in 'Battlefield V'. Game Developers Conference, 2019. URL https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s91023-it-just-works-ray-traced-reflections-in-battlefield-v.pdf. Accessed: 2020-07-25.

[16] Elmar Eisemann and Xavier Décoret. Fast Scene Voxelization and Applications. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, page 71–78, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 159593295X. doi: 10.1145/1111411.1111424. URL https://doi.org/10.1145/1111411.1111424.

[17] Eric Enderton, Erik Sintorn, Peter Shirley, and David Luebke. Stochastic transparency. *IEEE transactions on visualization and computer graphics*, 17(8):1036–1047, 2010.

[18] Cass Everitt. Interactive Order-Independent Transparency, 2001.

[19] Dominic Filion and Rob McNaughton. Effects & Techniques. In *ACM SIGGRAPH 2008 Games*, SIGGRAPH '08, page 133–164, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781450378499. doi: 10.1145/1404435.1404441. URL https://doi.org/10.1145/1404435.1404441.

[20] Henry Fuchs, Jack Goldfeather, Jeff P. Hultquist, Susan Spach, John D. Austin, Frederick P. Brooks, John G. Eyles, and John Poulton. Fast Spheres, Shadows, Textures, Transparencies, and Imgage Enhancements in Pixel-Planes. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, page 111–120, New York, NY, USA, 1985. Association for Computing Machinery. ISBN 0897911660. doi: 10.1145/325334.325205. URL https://doi.org/10.1145/325334.325205.

[21] Robert Herzog, Elmar Eisemann, Karol Myszkowski, and H.-P. Seidel. Spatio-Temporal Upsampling on the GPU. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, page 91–98, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589398. doi: 10.1145/1730804.1730819. URL https://doi.org/10.1145/1730804.1730819.

[22] Thai-Duong Hoang and Kok-Lim Low. Efficient screen-space approach to high-quality multiscale ambient occlusion. *The Visual Computer*, 28(3):289–304, 2012.

[23] Brian Hook. Mouse Ray Picking Explained, 2005. URL http://bookofhook.com/mousepick.pdf. Accessed: 2020-08-17.

[24] IrisVR. The Importance of Frame Rates, n.d. URL https://help.irisvr.com/hc/en-us/articles/215884547-The-Importance-of-Frame-Rates. Accessed: 2020-08-18.

[25] Jorge Jiménez, Xianchun Wu, Angelo Pesce, and Adrian Jarabo. Practical real-time strategies for accurate indirect occlusion. *SIGGRAPH 2016 Courses: Physically Based Shading in Theory and Practice*, 2016. URL https://www.activision.com/cdn/research/PracticalRealtimeStrategiesTRfinal.pdf. Accessed: 2020-07-26.

[26] Brian Karis and Epic Games. Real shading in unreal engine 4. *SIGGRAPH 2013 Courses: Physically Based Shading in Theory and Practice*, 2013. URL https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf. Accessed: 2020-09-16.

[27] Oliver Klehm, Tobias Ritschel, Elmar Eisemann, and Hans-Peter Seidel. *Screen-space Bent Cones: A Practical Approach*, chapter Global Illumination Effects, pages 191–207. GPU Pro. CRC Press, 2012. URL http://graphics.tudelft.nl/Publications-new/2012/KRES12. ISBN: 978-1439887820.

[28] Janne Kontkanen and Samuli Laine. Ambient occlusion fields. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 41–48, 2005.

[29] Samuli Laine and Tero Karras. Two Methods for Fast Ray-Cast Ambient Occlusion. In *Proceedings of the 21st Eurographics Conference on Rendering*, EGSR'10, page 1325–1333, Goslar, DEU, 2010. Eurographics Association. doi: 10.1111/j.1467-8659.2010.01728.x. URL https://doi.org/10.1111/j.1467-8659.2010.01728.x.

[30] Hayden Landis. Production-ready global illumination. *SIGGRAPH Courses notes*, 2002.

[31] Zander Majercik, Jean-Philippe Guertin, Derek Nowrouzezahrai, and Morgan McGuire. Dynamic diffuse global illumination with ray-traced irradiance fields. *Journal of Computer Graphics Techniques Vol*, 8(2), 2019.

[32] M. Mara, M. McGuire, D. Nowrouzezahrai, and D. Luebke. Deep G-Buffers for Stable Global Illumination Approximation. In *Proceedings of High Performance Graphics*, HPG '16, page 87–98, Goslar, DEU, 2016. Eurographics Association. ISBN 9783038680086.

[33] Morgan McGuire. Ambient occlusion volumes. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 1–1, 2010.

[34] Morgan McGuire. Computer Graphics Archive, July 2017. URL https://casual-effects.com/data. Accessed: 2020-08-01.

[35] 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*, HPG '11, page 25–32, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308960. doi: 10.1145/2018323.2018327. URL https://doi.org/10.1145/2018323.2018327.

[36] 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.

[37] Martin Mittring. Finding next Gen: CryEngine 2. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, page 97–121, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781450318235. doi: 10.1145/1281500.1281671. URL https://doi.org/10.1145/1281500.1281671.

[38] Jurriaan D. Mulder, Frans C. A. Groen, and Jarke J. van Wijk. Pixel Masks for Screen-Door Transparency. In *Proceedings of the Conference on Visualization '98*, VIS '98, page 351–358, Washington, DC, USA, 1998. IEEE Computer Society Press. ISBN 1581131062.

[39] Oliver Nalbach, Elena Arabadzhiyska, Dushyant Mehta, Hans-Peter Seidel, and Tobias Ritschel. Deep Shading: Convolutional Neural Networks for Screen Space Shading. *Computer Graphics Forum*, 36(4):65–78, 2017. doi: 10.1111/cgf.13225. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13225.

[40] Thomas Porter and Tom Duff. Compositing Digital Images. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, page 253–259, New York, NY, USA, 1984. Association for Computing Machinery. ISBN 0897911385. doi: 10.1145/800031.808606. URL https://doi.org/10.1145/800031.808606.

[41] Christoph K Reinbothe, Tamy Boubekeur, and Marc Alexa. Hybrid Ambient Occlusion. In *Eurographics (Areas Papers)*, pages 51–57, 2009.

[42] 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*, I3D '09, page 75–82, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584294. doi: 10.1145/1507149.1507161. URL https://doi.org/10.1145/1507149.1507161.

[43] Marco Salvi and Karthik Vaidyanathan. Multi-Layer Alpha Blending. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '14, page 151–158, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327176. doi: 10.1145/2556700.2556705. URL https://doi.org/10.1145/2556700.2556705.

[44] Marco Salvi, Jefferson Montgomery, and Aaron Lefohn. Adaptive Transparency. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, page 119–126, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308960. doi: 10.1145/2018323.2018342. URL https://doi.org/10.1145/2018323.2018342.

[45] Perumaal Shanmugam and Okan Arikan. Hardware Accelerated Ambient Occlusion Techniques on GPUs. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, page 73–80, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936288. doi: 10.1145/1230100.1230113. URL https://doi.org/10.1145/1230100.1230113.

[46] Tomasz Stachowiak and Yasin Uludag. Stochastic screen-space reflections. *ACM SIGGRAPH Courses 2015: Advances in Real-Time Renderingin Games*, 2015.

[47] John E Stone, William R Sherman, and Klaus Schulten. Immersive molecular visualization with omnidirectional stereoscopic ray tracing and remote rendering. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1048–1057. IEEE, 2016.

[48] Andrei Tatarinov and Alexey Panteleev. Advanced Ambient Occlusion Methods for Modern Games. Game Developers Conference, 2016. URL http://developer.download.nvidia.com/gameworks/events/GDC2016/atatarinov_alpanteleev_advanced_ao.pdf. Accessed: 2020-07-15.

[49] Mariano Trebino. Rasterization (I): Overview, 2017. URL https://mtrebi.github.io/2017/02/01/rasterization-i.html. Accessed: 2020-08-15.

[50] Michal Valient. Killzone Shadow Fall Demo Postmortem. Sony Devstation, 2013. URL https://www.guerrilla-games.com/read/killzone-shadow-fall-demo-postmortem. Accessed: 2020-07-25.

[51] Kostas Vardis, Georgios Papaioannou, and Athanasios Gaitatzes. Multi-View Ambient Occlusion with Importance Sampling. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '13, page 111–118, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319560. doi: 10.1145/2448196.2448214. URL https://doi.org/10.1145/2448196.2448214.

[52] Patricio Gonzalez Vivo and Jen Lowe. The Book of Shaders: Random, 2015. URL https://thebookofshaders.com/10/. Accessed: 2020-08-01.

[53] Oculus VR. Oculus Best Practices, 2017. URL https://static.oculus.com/documentation/pdfs/intro-vr/latest/bp.pdf. Accessed: 2020-08-18.

[54] Chris Wyman and Morgan McGuire. Improved Alpha Testing Using Hashed Sampling. *IEEE transactions on visualization and computer graphics*, 25(2):1309–1320, 2017.

[55] Felix Yang. Adaptive Multi-view Screen-space Ambient Obscurance. Master's thesis, Delft University of Technology, the Netherlands, 2019.

[56] Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-Time Concurrent Linked List Construction on the GPU. *Computer Graphics Forum*, 29(4):1297–1304, 2010. doi: 10.1111/j.1467-8659.2010.01725.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2010.01725.x.

[57] S. Zhukov, A. Iones, and G. Kronin. An ambient light illumination model. In *Rendering Techniques '98*, pages 45–55, Vienna, 1998. Springer Vienna. ISBN 978-3-7091-6453-2.

# A

# The State of the Art in Efficient Order-Independent Transparency Techniques

# The State of the Art in Efficient Order-Independent Transparency Techniques

Jop Vermeer, 4462734

30 January, 2020

**Abstract**

Traditional transparency rendering techniques require the fragments to be sorted based on their depth value, which brings high computational costs. To render transparent geometry more efficiently, order-independent transparency techniques are used. Order-independent transparency techniques always render transparent geometry identically, regardless of the fragments' order. However, conventional order-independent transparency techniques have unbounded, highly varying memory requirements and non-deterministic computational cost. This report focuses on efficient order-independent transparency techniques, that have a bound on their memory usage and work in a fixed amount of render passes. These techniques can be split into two categories: the techniques based on alpha blending, that trade accuracy in favour of performance, and the techniques that approximate the visibility function, that are often slower but more accurate. The different techniques are discussed and compared with each other, giving recommendations for their use case.

## 1 Introduction

Rendering accurate transparency efficiently with rasterization is quite a challenge. Contrary to rendering opaque geometry, where only the closest fragment to the camera contributes to the final pixel color, multiple transparent fragments blend together to create the final pixel color. This interaction between transparent fragments is the main obstacle, as the order between transparent objects has a large influence on the final the color. Traditional transparency techniques therefore require the user to sort the fragments before submission, but with complex scenes this can become very costly or even impossible. Sorting transparent geometry at object level is another solution that is computationally much cheaper, but results in artifacts when objects overlap in depth and is therefore often not an option.

For this reason, *order-independent transparency* (OIT) techniques are highly desired, as fragments can be submitted in arbitrary order, while always producing accurate results. Conventional order-independent transparency techniques however, have non-deterministic memory requirements and computational costs. In this report we discuss and compare efficient order-independent transparency techniques that work in real-time and can handle an arbitrary amount of transparency layers with a fixed number of render passes and bounded memory requirements. The faster techniques are based on recursive alpha blending, while the more accurate techniques try to approximate the visibility function. While these OIT algorithms are designed for rasterizers, they also apply to ray tracing when the acceleration structure used does not guarantee ordered ray intersections (e.g. bounding volume hierarchies) [1].

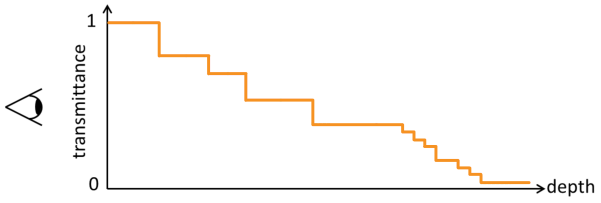



Figure 1: The visibility function $vis(z)$, where each fragments along a ray from the viewer into the scene represents a step. By sampling this function for a fragment with a certain depth, one can determine the transmittance of that fragment.

## 2 Background Theory

Suppose that we have a transparent fragment $f_i$ with a depth $z_i$, a color $c_i$ and an alpha $\alpha_i$. Then given a visibility function $vis(z_i)$ that represents the total transmittance between $f_i$ and the viewer (Figure 1), one can compute the contribution of $f_i$ to the final color as:

$$c_i \alpha_i vis(z_i)$$

With the contribution of $n$ overlapping fragments given by:

$$\sum_{i=0}^{n} c_i \alpha_i vis(z_i) \tag{1}$$

Where the visibility function $vis(z)$ is defined as the product of the transmittance of every transparent fragment along a ray from the viewer to $z$:

$$vis(z) = \prod_{0<z_i<z} (1 - \alpha_i) \tag{2}$$

This visibility function is often unknown during shading and depends on the depth-order of the transparent fragments. For this reason, Equation 1 is often computed recursively in back-to-front order using Porter and Duff's compositing (OVER) operator [2], also known as *alpha blending*:

$$\begin{aligned} C_0 &= \alpha_0 c_0 \\ C_n &= \alpha_n c_n + (1 - \alpha_n) C_{n-1} \end{aligned} \tag{3}$$

Where $C_n$ is the final color. This technique gives accurate transparency without having to compute $vis(z)$, but requires the fragments to be sorted according to their depth value, which can be very expensive especially with the large amount of fragments in modern scenes.

Techniques such as depth peeling [3] or the A-buffer algorithm [4, 5] solve transparency using Equation 3, without requiring the fragments to be sorted, making them order-independent transparency techniques. In depth peeling, each transparency layer is rendered in a separate render pass, where in each pass the next nearest transparent fragment is determined using standard depth testing. For $n$ transparency layers, depth peeling requires $n$ rendering passes, making it unsuitable for real-time applications. The A-buffer algorithm stores all fragments in variable length per pixel lists and sorts them after all geometry is rendered. This brings a large variance in performance depending on the scene's geometry and has unbounded memory requirements.

Other algorithms explicitly compute the visibility function $vis(z)$ as given by Equation 2 to solve Equation 1 directly. These algorithm often use the following rendering steps:

1. Render opaque geometry into color buffer A
2. Render transparent geometry and build a representation of the visibility function
3. Render transparent geometry, shade fragments by sampling visibility from representation build in step 1 and composite into color buffer B.
4. Composite color buffer A and B to create final image

By first rendering the opaque geometry, one can reuse the depth buffers of this step to discard transparent fragments that lie behind opaque geometry.

This report will cover OIT algorithms that have bounded memory requirements and a fixed number of render passes that either solve transparency using alpha blending (Section 3) or that explicitly compute/approximate the visibility function (Section 4).

# 3 Techniques Based on Alpha Blending

The algorithms discussed in this section are based on Porter and Duff's OVER operator [2], which is also known as alpha blending:

$$c = \alpha_A c_A + \alpha_B(1 - \alpha_A)c_B$$

By recursively applying this OVER operator in back-to-front order, one can accurately compute the final color (Equation 3). The techniques discussed in this section apply or extend this OVER operator in such a way that the fragments do not have to be sorted. They trade accuracy in favour of performance, requiring only a single rendering pass.

## 3.1 Multi-Layer Alpha Blending

Alpha blending can be used to produce accurate transparency if the fragments are sorted in the correct order. Multi-layer alpha blending relaxes this requirement by approximating the recursive alpha blending equation with an arbitrary but bounded amount of terms [6]. The result is a semi order-independent transparency algorithm that only requires a single pass over the transparent geometry.

Multi-layer alpha blending stores a blending array with $m$ rows. Each row $j$ corresponds to one of the terms for the OVER operator and stores a color $c_j$, transmittance $t_j = 1 - \alpha_j$ and distance to the camera $z_j$. If for each row $j$, we have that $z_j < z_{j+1}$ then the final color can be computed as follows:

$$c = \sum_{j=0}^{m-1} c_j \prod_{i=0}^{j-1} t_i$$

Upon rendering the transparent geometry, each fragment $i$ contributing to a pixel will be inserted in the corresponding blending array as the new $j$th row, such that $z_{j-1} < z_i \leq z_j$. If a fragment is to be inserted while the blending array already contains $m$ entries, the last two rows of the blending array $m-2$ and $m-1$ are merged together:

$$\begin{aligned} c_{merge} &= c_{m-2} + c_{m-1}t_{m-2} \\ t_{merge} &= t_{m-2}t_{m-1} \\ z_{merge} &= z_{m-2} \end{aligned}$$

This merging approach produces accurate results if no fragments $i$ is inserted, while row $k$ and $l$ are merged together where $z_k < z_i < z_l$, else it becomes an approximation. This is why this technique is not fully order-independent, as submitting the fragments in different order produces different results, with in some cases visible artifacts.

To make sure the results stay temporally consistent, the implementation uses Intel's `Pixel Synchronization` extension which ensures that the fragments are inserted into the blending array in the same order as the primitives are submitted. The final algorithm then becomes:

1. Render transparent geometry, construct a blending array for each pixel with $m$ terms.
2. Compute color using the blending arrays from the previous step.

Multi-layer alpha blending has performance that is significantly faster than most visibility-function based techniques and almost on par with regular alpha blending (without sorting), while still delivering accurate results. When compared to adaptive or hybrid transparency, multi-layer alpha blending requires significantly fewer layers to produce similar looking images. multi-layer alpha blending is however not fully order independent and can produce different result if the fragments are rendered in a different order, which can lead to visible artifacts that can usually not be removed by increasing the number of layers. This makes multi-layer alpha blending the go-to OIT algorithm when performance is the most important factor and the semi order-independence is less of a problem, which can be the case in for example games.

## 3.2 Weighted Blended Order-Independent Transparency

The OVER operator is not commutative, meaning that one would have to sort the fragments to be able to

determine the correct final color. Weighted blended order-independent transparency tries to redefine this OVER operator such that its arguments commute, which makes the compositing operator order independent [7]. This method allows for rendering transparency in a single pass over the transparent geometry, with the trade-off being that it does not produce completely accurate results.

The method builds forth on the OVER operators as defined by [8, 9] and extends them by computing the exact coverage of the background (similar to the alpha correction step of occupancy maps [Section 4.1] and stochastic transparency [Section 4.3]), and weighting the contribution of a fragment with its distance from the camera. The exact coverage of the background is computed as:

$$coverage = \prod_{i=1}^{n}(1-\alpha_i)$$

The resulting commutative OVER operator is then defined as:

$$c = \frac{\sum_{i=1}^{n} c_i \cdot w(z_i, \alpha_i)}{\sum_{i=1}^{n} \alpha_i \cdot w(z_i, \alpha_i)}(1-\prod_{i=1}^{n}(1-\alpha_i)) + c_0 \prod_{i=1}^{n}(1-\alpha_i)$$

where $c_0$ is the background color, $z_i$ is the distance of fragment $i$ to the camera and $w(z_i, \alpha_i)$ is the weight function. The weight function is based on $z_i$ and $\alpha_i$ to make sure that fragments very close to the camera with a very low alpha, that should be almost imperceptible, won't have a large contribution to the final color.

This weight function can be seen as an heuristic approximation to the visibility function. For the best results the weight function needs to be tuned specifically for the scene, to able to discriminate between transparent objects at different distances. There are however multiple generic weight functions that should be appropriate for arbitrary scenes with large depth ranges [7].

The resulting algorithm then proceeds as follows:

1. Render transparent geometry, accumulate color and alpha multiplied by the respective weight and write to texture. Simultaneously compute exact coverage of the background and write to separate texture.
2. Compute color using the textures from the previous step.

Since weighted blended OIT redefines the OVER operator to be commutative, it is very fast and very easy to implement. It can achieve acceptable accuracy if the weight function is tuned well for the scene, otherwise distant objects may blend together or silhouette artifacts can appear. Weighted blended works best when the transparent fragments are distributed rather uniformly or have similar colors.

While weighted blended is order-independent, it is not translation-invariant along the depth axis. This means that moving the entire scene to a different depth range can change its color, making it unsuitable for certain applications, such as CAD-like software. In very resource-constrained applications weighted blended OIT can be a good fit, however additional time is required to correctly tune the algorithm for the specific application.

# 4 Techniques Approximating the Visibility Function

The techniques in this section all explicitly approximate/compute a representation of the visibility function (Equation 2). These techniques all roughly follow the rendering steps as described in Section 2. They are often more accurate than the alpha blending based algorithms, at the cost of performance (i.e. requiring multiple render passes and higher memory usage).

## 4.1 Occupancy Maps

Occupancy maps can be used to approximate the depth-order of the transparent fragments [10]. This depth-order approximation can then be used in combination with alpha blending to achieve accurate transparency without sorting the fragments, with the assumption that the alpha values of the transparent objects are all approximately equal to each other, which is the case in for example hair.

The occupancy map discretizes the visibility in a scene using a 3D-grid, consisting of a certain amount of slices (2D textures). Each texel contains a bit indicating whether the slice is occupied by at least one

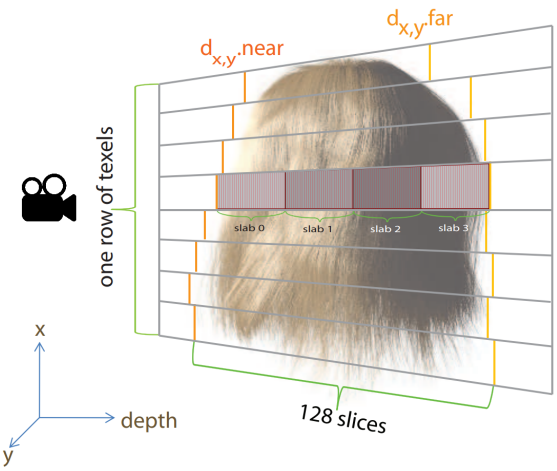



Figure 2: Visualization of an occupancy map of 128 slices, with 4 slabs. Here $d_{x,y}.near$ and $d_{x,y}.far$ correspond to the depths of the nearest and furthest transparent fragment from the depth-range map. From [10].

fragment at that location. To approximate the number of transparent fragments that lie within a certain texel of this 3D-grid, the algorithm uses a slab-map, where a slab is a group of slices. This slab-map stores at each texel $F_i$, which is the number of transparent fragments that lie within slab $i$ divided by the amount of occupied slices corresponding to slab $i$. The original implementation of occupancy maps [10] used 128 slices in their 3D-grid, divided into four slabs of 32 slices each (Figure 2), but can be of up to $N \cdot 128$ slices, where $N$ is the amount of simultaneous render targets supported.

To maximize the precision of the occupancy and slab-map, a depth-range map is first generated, which contains the depth of the nearest and furthest transparent fragment from the camera for each texel. This depth-range map is generated by rendering the scene twice with depth-testing enabled while changing the depth-test function.

Using these maps, we can estimate the depth order as the sum of slab-sizes in front of the current slab plus the number of transparent fragments preceding the current fragment within the its slab. The first we know and the latter is approximated as $F_i$ times the amount of occupied slices before the current slice. This depth order approximation assumes that (1.) there is an equal amount of fragments in each of the occupied slices within one slab and (2.) the fragments within one slice are uniformly distributed over that slice.

Now to use this depth order with alpha blending, let us first take a look at an example with three transparent fragments:

$$c = \alpha_2 c_2 + (1-\alpha_2)\alpha_1 c_1 + (1-\alpha_2)(1-\alpha_1)\alpha_0 c_0$$

With the assumption that all alpha values are approximately equal, then we can rewrite this equation as:

$$c = (1-\alpha)^0 \alpha c_2 + (1-\alpha)^1 \alpha c_1 + (1-\alpha)^2 \alpha c_0$$

which can be generalized for $n$ transparent fragments as:

$$c = \sum_{i=0}^{n-1} (1-\alpha)^i \alpha c_{n-1-i}$$

Thus for each transparent fragment, the following color should be outputted:

$$c = (1-\alpha)^o \alpha c_i$$

with $o$ as the estimated depth order of the fragment.

To reduce the artifacts when lots of fragments fall into the same pixel, an alpha correction is applied. In a single pass the exact sum of all alphas, the total alpha ($\alpha_{total}$), is computed for each pixel:

$$\alpha_{total} = 1 - (1-\alpha)^n$$

The accumulated alpha of a certain pixel ($\alpha_{sum}$) is computed as:

$$\alpha_{sum} = (1-\alpha)^o \cdot \alpha$$

The final color is then multiplied by the ratio of the exact total alpha to the accumulated alpha:

$$c = c \cdot \frac{\alpha_{sum}}{\alpha_{total}}$$

The algorithm then proceeds as follows:

1. Render transparent geometry to generate depth buffer using a greater than operator and determine total alpha.

2. Render transparent geometry to generate depth buffer using a less than operator.

3. Render transparent geometry, construct occupancy and slab-map.

4. Render transparent geometry, determine depth order and output color.

5. Apply alpha correction.

Here step 1 and 2 could be combined into a single pass, but the two pass approach was faster as a large amount of fragments can be discarded using culling.

Occupancy maps produces images that often deviates significantly from A-buffer baseline images due to the, often false, assumption that all fragments have equal alpha values. Even when the assumption does hold, it often produces inferior images compared to other OIT algorithms due to its discretization of the visibility in a scene using a 3D-grid. This means that the visibility is less accurate the larger the depth range is between the closest and furthest transparent fragment. This makes occupancy maps often the suboptimal choice of OIT algorithm.

## 4.2 Fourier Opacity Mapping

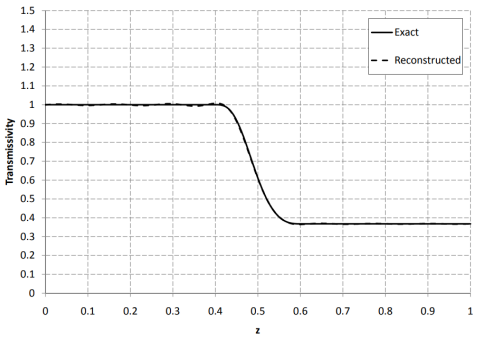
Fourier opacity mapping is a technique to render volumetric shadows [11], but since it approximates the visibility function, it can also be used to render transparency. Inspired by Convolution Shadow Maps [12], Fourier opacity mapping approximates the visibility function as a Fourier series (Figure 3). Due to the smooth nature of Fourier series, visibility functions with sharp steps can require many coefficients. Fourier opacity mapping is therefore mostly geared towards media with smooth visibility functions such as fog and smoke.

For translucent media with constant absorption $\sigma$, Beer's Law can be used to approximate $vis(z)$ as:

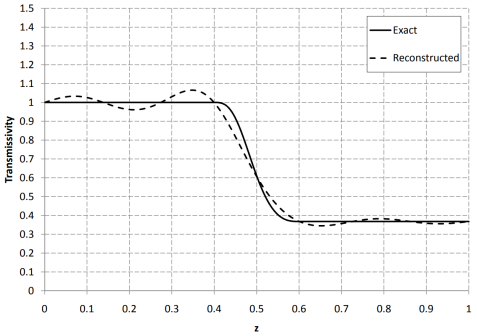
$$vis(z) = e^{-\sigma z}$$

This can be generalized to variable absorption using the integral over the absorption function $\sigma(z)$:

$$vis(z) = e^{-\int_0^z \sigma(x)dx}$$




(a) Fourier Opacity Mapping with 15 coefficients.




(b) Fourier Opacity Mapping with 7 coefficients.

Figure 3: Reconstruction of the visibility function using Fourier opacity mapping with different amount of coefficients. Notice the small amount of ringing with 7 coefficients. From [11].

Here the absorption function $\sigma(z)$ can be represented as a Fourier series, such that the integral of $\sigma(z)$ can be formulated as follows:

$$\int_0^z \sigma(x)dx \approx \frac{a'_0}{2}z + \sum_{k=1}^{n} \frac{a'_k}{2\pi k}\sin(2\pi kz) + \sum_{k=1}^{n} \frac{b'_k}{2\pi k}(1-\cos(2\pi kz))$$

$$a'_k = 2\int_0^1 \sigma(z)\cos(2\pi kz)dz$$

$$b'_k = 2\int_0^1 \sigma(z)\sin(2\pi kz)dz$$

Unfortunately we don't have access to the $\sigma(z)$ function to directly compute $a'_k$ and $b'_k$, instead we have to determine the absorption properties of the translucent medium as the opacity of its fragments. In the case of a single fragment with opacity $\alpha_i$ at depth $z_i$ we have that:

$$vis(z) = \begin{cases} 1 & z < z_i \\ 1-\alpha_i & z \geq z_i \end{cases}$$

This can be reformulated in terms of Beer's Law using the Dirac delta function:

$$vis(z) = e^{\int_0^z \ln(1-\alpha)\delta(x-z_i)dx}$$

Which can be generalized to multiple fragments as:

$$vis(z) = e^{\int_0^z \sum_i \ln(1-\alpha_i)\delta(x-z_i)dx}$$

where $\alpha_i$ and $d_i$ are the opacity and depth of the $i$th fragment. Thus in this discretized setting we get that:

$$\sigma(z) = -\sum_i \ln(1-\alpha_i)\delta(z-z_i)$$

Substituting this into $a'_k$ and $b'_k$ whilst making use of the Dirac delta's sifting property, we obtain:

$$a'_k = -2\sum_i \ln(1-\alpha_i)\cos(2\pi kz_i)$$

$$b'_k = -2\sum_i \ln(1-\alpha_i)\sin(2\pi kz_i)$$

The texture(s) with the $a'_k$ and $b'_k$ coefficients for each pixel is called the Fourier opacity map. Since the formulation of $a'_k$ and $b'_k$ sum all the contributions of the individual fragment, they can be generated in a single render pass over the transparent geometry by outputting a fragment's contributions to these coefficients without requiring any specific order.

The Fourier Opacity Map algorithm then proceeds as follows:

1. Render transparent geometry to generate the Fourier opacity map (i.e. texture(s) with coefficients $a'_k$ and $b'_k$ for each pixel).

2. Render transparent geometry, determine visibility by sampling $vis(z)$ and composite into a color buffer.

These steps correspond to the render steps described in section 2.

Fourier opacity mapping is geared towards rendering and shadowing smooth volumetric media with low opacity, such as smoke. Its approximation of the visibility function as a Fourier series works well for a smooth visibility function, but fails when clear steps are present, such as with thin objects with high opacity. If the approximation with a Fourier series fails, it can results in very noticeable ringing artifacts (Figure 3). This means that Fourier opacity mapping is appropriate for rendering volumetric media such as smoke and gas, but not for general transparent scenes.

### 4.3 Stochastic Transparency

Stochastic transparency [13] is based on traditional screen-door transparency [14], which creates a fake transparency effect by discarding fragments of a transparent surface in a certain stipple pattern (Figure 4). Stochastic transparency extends this with a random sub-pixel stipple pattern.

Stochastic transparency takes $S$ samples per pixel. A transparent fragment $i$ will cover a stochastic subset of $R$ samples, where $\alpha_i = R/S$. Each sample does z-buffer comparisons, retaining the front-most fragment that covers it. This means that for a fragment $i$ that is not in front, its effective $\alpha$ becomes:

$$\alpha = \alpha_i \cdot \prod_{z_j < z_i} (1-\alpha_j)$$

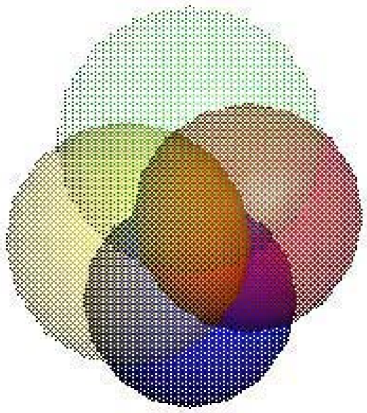


Figure 4: An example of screen-door transparency, where discarding fragments in a stipple pattern gives the illusion of transparency. From [15].

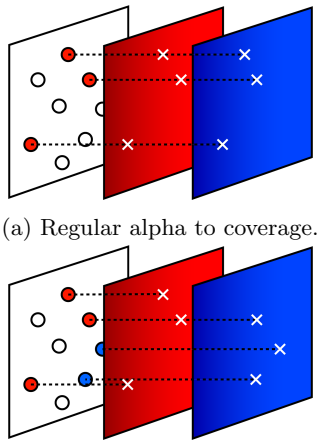
where $z_i$ is the depth value of fragment $i$. By blending the colors of the samples, we get the final pixel color. Stochastic transparency can be seen as tracing a ray for each sample, where each time a ray hits a transparent fragment, an $\alpha$-weighted coin is tossed whether to keep the fragment and stop or to discard the fragment and continue.

To reduce the noise the number of samples $S$ can be increased, but this will have the classic Monte Carlo problem of diminishing returns, where halving the average error requires quadrupling $S$. To combat this, stratified sampling is used instead. For a fragment $i$, instead of naively flipping an $\alpha$-weighted coin for each sample, stratified sampling set a group of $R_i$ samples, where $R_i$ is chosen as:

$$R_i = \lfloor \alpha_i S + \xi \rfloor$$

Here $\xi$ is a canonical random number between 0 and 1.

To efficiently take multiple samples, MSAA is used, where each pixel contains several samples at different positions within the area of the pixel. Alpha to coverage can then be used to determine which samples a fragment covers, using a coverage masks. Stochastic transparency uses a stochastic variant of alpha to coverage, where the mask is uncorrelated between samples in the same pixel (Figure 5).



(a) Regular alpha to coverage.

(b) Stochastic alpha to coverage.

Figure 5: regular and stochastic alpha to coverage with 8 samples and a red and blue fragment, both with $\alpha = \frac{3}{8}$. With regular alpha to coverage (a), fragments with similar alpha will cover the same samples. With stochastic alpha to coverage (b) the mask is chosen randomly. From [16].

In a multi-sampled buffer called the stochastic transparency buffer, the z-value of the front-most fragment that covered the corresponding sample is stored. For each pixel we thus store $S$ depth values $z_0$, $z_1$, ...$z_{S-1}$. Using this buffer, the visibility function $vis(z)$ is approximated as follows:

$$vis(z) \approx \frac{\sum_{i=0}^{S} D(z, z_i)}{S}$$

where

$$D(z, z_i) \begin{cases} 1 & z \leq z_i \\ 0 & z > z_i \end{cases}$$

Thus the visibility at depth $z$ is the fraction of samples that contain a $z$-value lower than $z$.

To further reduce the noise in the image, the alpha correction pass of the occupancy map algorithm is used, where the final color is multiplied by the ratio of the exact total alpha to the accumulated alpha. The total alpha ($\alpha_{total}$) for each pixel is computed in a single pass over all transparent fragments, but since stochastic transparency does not assume all alphas to

be equal it instead computes:

$$\alpha_{total} = 1 - \prod_i 1 - \alpha_i$$

The accumulated alpha of a certain pixel ($\alpha_{acc}$) is computed as the following sum over each fragment contributing to the pixel:

$$\alpha_{sum} = \sum_i vis(z_i)\alpha_i$$

The final color of the pixel is then multiplied by $\frac{\alpha_{total}}{\alpha_{sum}}$.

The resulting algorithm then proceeds as follows:

1. Render transparent geometry, compute total alpha and store into a multi-sampled buffer.
2. Render transparent geometry into the opaque z-buffer. Discard samples using stochastic alpha-to-coverage.
3. Render transparent primitives determine $vis(z)$ and composite into a multi-sampled color buffer.
4. Apply alpha correction.

Note that this will automatically give an anti-aliased output, since MSAA is used for each color buffer. Stochastic transparency takes 3 render passes, but if the user wants to use more samples than the $s$ MSAA samples supported by the hardware, the algorithm needs $1 + 2(\frac{S}{s})$ passes (the total alpha is independent to the number of samples, but step 2 and 3 will need to be repeated). To reduce noise and improve the temporal coherence, one could apply temporal filtering, such as TAA [17].

Stochastic transparency is a simple and very robust OIT algorithm, it has no underlying assumptions and will always produce accurate, albeit noisy, images. Stochastic transparency has large similarities with ray traced transparency. The quality of stochastic transparency can be scaled by increasing the number of samples, reducing noise at the cost of performance. The noise however depends on the opacity of the fragments. If a surface is very opaque, the chance is high that fragments behind it are discarded and not taken into account, which introduces noise [18]. Stochastic transparency also has very high memory bandwidth requirements and requires additional render passes if more than 16 samples are used (the common maximum amount of MSAA samples supported by the hardware). For these reasons, stochastic transparency is currently not recommended for most interactive applications, but its stochastic approach could become more prevalent in the future due to its robustness.

## 4.4 Adaptive Transparency

In adaptive transparency the visibility function, $vis(z)$, is adaptively compressed to fit into bounded memory [19]. The compression scheme used is largely inspired by that used in Adaptive Volumetric Shadow Maps [20], and is based on a heuristic that aims to discard fragments with the smallest contributions to the final color, in turn reducing the overall error of the approximation.

The visibility function is represented/approximated using a step function basis:

$$vis(z) = \prod_{i=0}^{n}(1 - \alpha_i H(z - z_i))$$

where

$$H(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

Which can then be rewritten as:

$$vis(z) = \prod_{i=0}^{n} H_{z_i,\alpha_i}(z))$$

where

$$H_{z,\alpha}(x) = \begin{cases} 1 & x \leq z \\ 1 - \alpha & x > z \end{cases}$$

Using this $H_{z,\alpha}(x)$ as basis function, the visibility function can be represented as a sequence of depth and transmittance pairs ($z_i$, $1 - \alpha_i$), called nodes (see Figure 6).

A fixed amount of nodes are stored, to allow this technique to run with bounded memory. The list of nodes is maintained in a front-to-back depth-order, where new nodes are inserted into the proper location. When the list is full, the node which on removal produces the smallest change to the current visibility

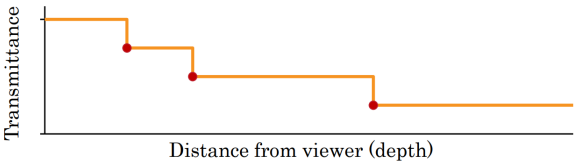



Figure 6: Step function representation of visibility function. The red dots represent the nodes (i.e. the depth and transmittance pairs that parameterize the $H_{z,\alpha}$ basis functions). From [19].

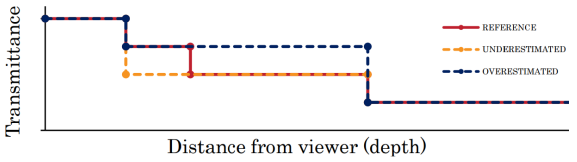



Figure 7: When a node is removed, the visibility function is either over- (blue) or underestimated (orange). From [19].

is removed. This means that the node that generates the smallest error to the integration over the visibility function is removed, which due to the step function basis, is simply a rectangle area comparison. Selecting this node $i$ can be formulated as:

$$\arg\min_i \{(vis(z_{i-1}) - vis(z_i))(z_i - z_{i-1}))\}$$

Where node $i-1$ is the next closest node from node $i$.

There are two ways to remove a node, either over- or underestimating the real visibility function (Figure 7). Underestimation was found to produce a more accurate image, with less artifacts. Selecting the node removal technique on a node-by-node basis minimized the error in the represented visibility function, but introduced undesirable high-frequency artifacts.

The resulting algorithm then proceeds as follows:

1. Render transparent geometry to build per-pixel visibility function, $vis(z)$
2. Render transparent geometry, determine visibility by sampling $vis(z)$ and composite into a color buffer

These steps correspond to the render steps described in section 2.

Unfortunately the implementation given in [19] does not yet operate in bounded memory, as critical sections were not supported in fragment shaders at the time. A more modern OpenGL implementation could make use of the `NV_fragment_shader_interlock`[1] extension to add support for critical sections. Intel has provided an updated implementation of adaptive transparency that does run in bounded memory[2]. This implementation originally made use of Intel's `Pixel Synchronization` extension, which allows for write operations in primitive submission order, without incurring data races, but was later updated with DirectX 12's very similar `Rasterizer Order Views`.

In terms of performance, the unbounded memory implementation is faster than for example stochastic transparency, but not as fast as hybrid transparency or multi-layer alpha blending. With enough nodes, adaptive transparency can produce very accurate images. However, the compression scheme used has a tendency to underestimate the transmittance function, which can lead to darkening artifacts. Compared to hybrid transparency, adaptive transparency often needs far more nodes to produce similar looking results. Unlike hybrid transparency however, there is no real failure case. This makes adaptive transparency a more robust technique, at the cost of performance and memory requirements.

## 4.5 Hybrid Transparency

Hybrid transparency [21] uses the assumption that the front-most transparency layers contribute more to the final color than further layers, motivated by the monotonically-decreasing nature of the transmittance function along depth. For this reason, hybrid transparency splits the visibility function, $vis(z)$, into two parts: a core and a tail, where the core represents the first $k$ transparency layers and the tail the rest (Figure 8). Hybrid transparency makes makes use of a slower, accurate method for the $k$ core layers and an approximate but fast method for the tail layers, where $k$ is chosen according to the memory budget.

[1] `https://www.khronos.org/registry/OpenGL/extensions/NV/NV_fragment_shader_interlock.txt`

[2] `https://software.intel.com/en-us/articles/oit-approximation-with-pixel-synchronization`

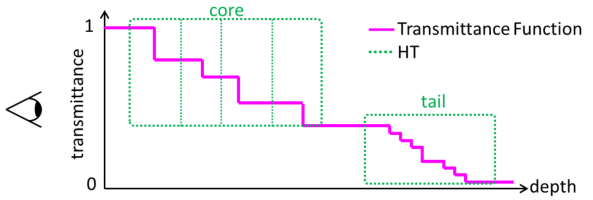



Figure 8: In hybrid transparency (HT), the $k$ transparency layers closest to the viewer correspond to core, the remaining layers to the tail. From [21].

For the core a truncated version of the A-buffer algorithm [4] is used, which allows to accurately blend a fixed number of fragments using bounded memory. For the $n - k$ fragments that compose the tail, the weighted averages algorithm is used [8], but could be interchanged with the weighted blended algorithm (Section 3.2) which is a refinement of the weighted averages algorithm.

The resulting algorithm then proceeds as follows:

1. Render transparent geometry and store $k$ frontmost fragments (core)

2. Compute visibility at each depth for the core and store in visibility buffer

3. Render transparent geometry, shade fragments by sampling visibility buffer for the core fragments or using weighted averages for the tail fragments.

For the core fragments, these steps correspond to the render steps described in section 2. For the tail fragments, only a single rendering pass is needed. Hybrid transparency can also be implemented as a slight modification of the multi-layer alpha blending algorithm (Section 3.1). This approach allows for an hybrid transparency implementation that requires only a single rendering pass [6].

Hybrid transparency combines the accurate visibility function based techniques with the fast alpha blending based techniques. Its single pass implementation has similar performance as multi-layer alpha blending, while being fully order independent. Hybrid transparency makes the assumption that transparent fragments further away contribute less to the final color than nearer fragments. When this assumption is wrong, hybrid transparency would need to store many fragments to be able to produce accurate results. However more often than not, this assumption is true and allows hybrid transparency to produce very accurate images with only a few layers in its core. This makes hybrid transparency currently the best choice in many applications.

## 4.6 Stochastic Layered Alpha Blending

Stochastic layered alpha blending [1] provides a link between stochastic transparency and $k$-buffering transparency techniques, more specifically hybrid transparency, and shows how they are related in a non-obvious way. Stochastic layered alpha blending has an explicit user-determined parameter, the number of coverage mask bits $b$ per layer, that changes its results from being identical to stochastic transparency, hybrid transparency or anywhere in between.

Similar to hybrid transparency and other $k$-buffer techniques, stochastic layered alpha blending stores $k$ transparency layers in a per-pixel list, each layer containing a fragment's depth and coverage. The layers are ordered based on their depth. If a new fragment is inserted while the list is full, the furthest layer is discarded.

A fragment $f_i$ is inserted into the layer list if its random coverage mask, of $b$ bits where each bit is turned on with probability $\alpha_i$, is not fully covered by all layers in front of it (i.e. it is visible). Instead of computing a new random coverage mask per fragment, stochastic layered alpha blending instead derives a probability function, based only on the number of coverage bits, to determine whether a fragment is visible and should thus be inserted or not.

Using stratified sampling, a fragment $f_i$ with alpha value $\alpha_i$ will have $\mathbb{b}_i$ of its $b$ bit coverage mask set, computed by discretizing $\alpha_i b$, which we will notate as $\mathbb{b}_i = \lfloor \alpha_i b \rceil$. The amount of bits $\mathbb{b}_{occl}$ set in the coverage mask of the occluding layers is computed as $\mathbb{b}_{occl} = \lfloor (1 - \prod_{t=0}^{i-1}(1 - \alpha_t))b \rceil$. Then the stratified probability of fragment $f_i$ being visible is:

$$P_b(\mathbb{b}_{occl}, \mathbb{b}_i) = \begin{cases} 1 - \frac{\mathbb{b}_{occl}!(b-\mathbb{b}_i)}{b!(\mathbb{b}_{occl}-\mathbb{b}_i)!} & \mathbb{b}_i \leq \mathbb{b}_{occl} \\ 1 & \mathbb{b}_i > \mathbb{b}_{occl} \end{cases}$$

The fragment is inserted into the layer list if $\xi < P_b(\mathbb{b}_{occl}, \mathbb{b}_i)$, where $\xi$ is a random number between 0 and 1.

With this per-pixel layer list, we can determine the final color using Equation 1, where $c_i$ and $\alpha_i$ come directly from the fragment and $vis(z_i) = P_b(\mathbb{b}_{occl}, \mathbb{b}_i)$. Stochastic layered alpha blending also takes exactly the same alpha correction step as stochastic transparency where we compute $\alpha_{total}$ and $\alpha_{sum}$, and multiply the final color by $\frac{\alpha_{total}}{\alpha_{sum}}$.

Instead of storing coverage, we can also increase the number of (virtual) bits $b$ to "infinity" to be able to store and use continuous alpha values. Once we store continuous alpha values, $b$ becomes the simply becomes the precision of the discretization. If the lists store $k$ layers, then by choosing $b = k$ the algorithm becomes similar to regular stochastic transparency. When $b \to \infty$ however, the probability function $P_b$ approaches 1, meaning that fragments are always inserted. This makes stochastic layered alpha blending converge to $k$ layer hybrid transparency.

The resulting algorithm then proceeds as follows:

1. Render transparent geometry, generate per-pixel layer list and compute total alpha $\alpha_{total}$.

2. Render transparent geometry, determine visibility using layer list and composite into color buffer.

3. Apply alpha correction.

However due to the large resemblance to hybrid transparency, a single pass implementation of stochastic layered alpha blending is also possible. Furthermore by incorporating temporal anti-aliasing (e.g. [17]) one can not only remove aliasing, but also greatly reduce the noise introduced by the stochasticity.

Stochastic layered alpha blending was designed to explore how different OIT algorithms are related. It connects hybrid and stochastic transparency, which at first sight seemed fundamentally different. Its performance is similar to that of the single pass hybrid transparency implementation and gives rise to the question whether this single pass optimization may carry through regular to stochastic transparency. Stochastic layered alpha blending provides a single parameter to decrease bias in exchange for performance. Its can give results identical to both stochastic and hybrid transparency and anywhere in between. Stochastic layered alpha blending can be a good choice in applications like CAD software, as it can change between an accurate but slow/noisy algorithm and a fast but biased one.

# 5 Discussion

There is currently no perfect solution for order-independent transparency. The techniques discussed in Section 3 and 4 all have their own trade-offs and underlying assumptions. An OIT algorithm should be chosen based on the application's needs, depending on performance, accuracy, robustness and potential visual artifacts, where robustness describes if a technique is applicable to all kinds of scenes, or whether it fails in certain scenarios due to underlying assumptions or heuristics.

Overall the methods based on alpha blending are significantly faster than those that explicitly approximate the visibility function, with the trade-off that they are also less accurate. Their speed is related to the fact that they require only a single rendering pass and don't explicitly compute and store a visibility function, saving memory and bandwidth. These techniques are very much heuristic driven or not fully order-independent. Depending on the use case, these algorithms can be a good enough approximation. They can be a good choice if performance is more important than accuracy. An example application would be gaming especially on lower-end hardware such as phones or gaming consoles, where only a small fraction of the frame time budget is allotted to transparency rendering [22] to allow for more time to spend on other techniques.

In comparison, techniques that approximate the visibility function are often more accurate, but require multiple render passes. Since they build a representation of the visibility function they require more peak memory, more memory bandwidth and some sort of atomic operations or synchronization to insert fragment data into their data structures, greatly impacting the performance. Their robustness varies, depending on the assumptions made by the algorithm. Their accuracy, performance and robustness can often be tuned

by changing their memory budget, which is not always possible with alpha blending based techniques. These techniques are recommended when more accurate and reliable results are desired. Hybrid transparency would be the recommended OIT technique for real-time applications, for example games on higher-end hardware, due the good accuracy it delivers while still maintaining great competitive performance with alpha blending based techniques. Stochastic transparency or stochastic layered alpha blending would be a good fit for CAD-like applications, where performance is less important than unbiased and accurate transparency.

In the future, stochastic or ray traced transparency techniques may become more prevalent [22], due to their great accuracy, robustness and scalability combined with their simplicity. These techniques make no implicit assumptions about the transparent geometry, making them very robust and applicable in all kinds of applications. The main downside currently is that they require high memory bandwidth, better hardware support (e.g. higher MSAA for stochastic transparency) and that they can be noisy. A lot of work is put into hardware support and denoising techniques for ray tracing, which could pave the way for real-time ray traced transparency [23].

# 6 Conclusion and Future Work

This report presented the state-of-the-art of order-independent transparency techniques that have bounded memory requirements and a fixed number of render passes. Some techniques are based on alpha blending, while others explicitly compute a visibility function. The main trade-off being performance versus accuracy and robustness. There is currently no one size fits all solution, each technique fits a specific purpose with their own advantages, disadvantages and assumptions. In current real-time applications or in applications where accuracy/bias is less important than runtime, the alpha blending based techniques can be a good choice given their excellent performance. In the future stochastic transparency or ray tracing may become more prevalent, due to their robustness, scalability and accuracy. In the mean time, hybrid techniques such as hybrid transparency can provide a middle ground between the alpha blending based techniques and the visibility function approximating techniques, striking a good balance between performance, accuracy and robustness. For this reason, hybrid transparency remains the recommended OIT technique for many applications in the near future.

Future work should investigate how to optimize the more general and robust techniques, such as stochastic or ray traced transparency, to achieve competitive performance and reduce the noise. Furthermore, the current techniques all require a separate forward rendering pipeline for the transparent geometry. Future research should aim to better fit transparency into existing (deferred) pipelines, instead of treating transparency as a special case. In the same spirit, future work should investigate how to incorporate transparency phenomena such as diffusion and refraction into order-independent transparency techniques. Ray tracing would allow for this, but techniques such as phenomenological transparency [24, 25] show that this could also be possible using rasterization.

# References


[1] C. Wyman, "Exploring and expanding the continuum of oit algorithms.," in *High Performance Graphics*, pp. 1–11, 2016.

[2] T. Porter and T. Duff, "Compositing digital images," in *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pp. 253–259, 1984.

[3] C. Everitt, "Interactive order-independent transparency," 2001.

[4] L. Carpenter, "The A-buffer, an antialiased hidden surface method," *ACM Siggraph Computer Graphics*, vol. 18, no. 3, pp. 103–108, 1984.

[5] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz, "Real-time concurrent linked list construction on the gpu," *Computer Graphics Forum*, vol. 29, no. 4, pp. 1297–1304, 2010.

[6] M. Salvi and K. Vaidyanathan, "Multi-layer alpha blending," in *Proceedings of the 18th meet-*

*ing of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 151–158, ACM, 2014.

[7] M. McGuire and L. Bavoil, "Weighted blended order-independent transparency," *Journal of Computer Graphics Techniques*, 2013.

[8] L. Bavoil and K. Myers, "Order independent transparency with dual depth peeling," *NVIDIA OpenGL SDK*, pp. 1–12, 2008.

[9] H. Meshkin, "Sort-independent alpha blending," *GDC Talk*, 2007.

[10] E. Sintorn and U. Assarsson, "Hair self shadowing and transparency depth ordering using occupancy maps," in *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pp. 67–74, ACM, 2009.

[11] J. Jansen and L. Bavoil, "Fourier opacity mapping," in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pp. 165–172, ACM, 2010.

[12] T. Annen, T. Mertens, P. Bekaert, H.-P. Seidel, and J. Kautz, "Convolution shadow maps," in *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pp. 51–60, Eurographics Association, 2007.

[13] E. Enderton, E. Sintorn, P. Shirley, and D. Luebke, "Stochastic transparency," *IEEE transactions on visualization and computer graphics*, vol. 17, no. 8, pp. 1036–1047, 2010.

[14] H. Fuchs, J. Goldfeather, J. P. Hultquist, S. Spach, J. D. Austin, F. P. Brooks Jr, J. G. Eyles, and J. Poulton, "Fast spheres, shadows, textures, transparencies, and imgage enhancements in pixel-planes," *ACM SIGGRAPH Computer Graphics*, vol. 19, no. 3, pp. 111–120, 1985.

[15] J. D. Mulder, F. C. Groen, and J. J. van Wijk, *Pixel masks for screen-door transparency.* IEEE, 1998.

[16] E. Enderton, E. Sintorn, P. Shirley, and D. Luebke, "Stochastic transparency." `http://luebke.us/publications/StochTransp-slides.pdf`, 2010. Visited on 2019-11-18.

[17] B. Karis, "High-quality temporal supersampling," *Advances in Real-Time Rendering in Games, ACM SIGGRAPH Courses*, 2014.

[18] C. Wyman, "Exploring and expanding the continuum of OIT algorithms." `https://research.nvidia.com/sites/default/files/publications/2016_HPG_ExpandingOITContinuum.pdf`, Jun 2016. Last accessed on 2020-1-27.

[19] M. Salvi, J. Montgomery, and A. Lefohn, "Adaptive transparency," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pp. 119–126, ACM, 2011.

[20] M. Salvi, K. Vidimče, A. Lauritzen, and A. Lefohn, "Adaptive volumetric shadow maps," *Computer Graphics Forum*, vol. 29, no. 4, pp. 1289–1296, 2010.

[21] M. Maule, J. Comba, R. Torchelsen, and R. Bastos, "Hybrid transparency," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 103–118, ACM, 2013.

[22] M. McGuire and L. Bavoil, "Weighted, blended order independent transparency." `http://jcgt.org/published/0002/02/09/presentation.pptx`, Mar 2014. Last accessed on 2020-1-27.

[23] M. Stich, "Real-time raytracing with nvidia rtx." `http://on-demand.gputechconf.com/gtc-eu/2018/pdf/e8527-real-time-ray-tracing-with-nvidia-rtx.pdf`, 2018. Last accessed on 2020-1-27.

[24] M. McGuire and M. Mara, "A phenomenological scattering model for order-independent transparency," in *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 149–158, 2016.

[25] M. McGuire and M. Mara, "Phenomenological transparency," *IEEE transactions on visualization and computer graphics*, vol. 23, no. 5, pp. 1465–1478, 2017.